DISSERTATION

STRUCTURE IN COMBINATORIAL OPTIMIZATION AND ITS EFFECT ON HEURISTIC PERFORMANCE

Submitted by

Doug Hains

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2013

Doctoral Committee:

Advisor: Darrell Whitley
Co-Advisor: Adele Howe

Wim Bohm
Edwin Chong

ABSTRACT

STRUCTURE IN COMBINATORIAL OPTIMIZATION
AND ITS EFFECT ON HEURISTIC PERFORMANCE

The goal in combinatorial optimization is to find a good solution among a finite set of solutions. In many combinatorial problems, the set of solutions scales at an exponential or greater rate with the instance size. The maximum boolean satisfiability (MAX-SAT) is one such problem that has many important theoretical and practical applications. Due to the exponential growth of the search space, sufficiently large instances of MAX-SAT are intractable for complete solvers. Incomplete solvers, such as stochastic local search (SLS) algorithms are necessary to find solutions in these cases. Many SLS algorithms for MAX-SAT have been developed on randomly generated benchmarks using a uniform distribution. As a result, SLS algorithms for MAX-SAT perform exceptionally well on uniform random instances. However, instances from real-world applications of MAX-SAT have a structure that is not captured in expectation by uniform random problems. The same SLS algorithms that perform well on uniform instances have a drastic drop in performance on structured instances.

To better understand the performance drop on structured instances, we examine three characteristics commonly found in real-world applications of MAX-SAT: a power-law distribution of variables, clause lengths following a power-law distribution, and a community structure similar to that found in small-world models. We find that those instances with a community structure and clause lengths following a power-law distribution have a significantly more rugged search space and larger backbones than uniform random instances. These search space properties make it more difficult for SLS algorithms to find good solutions and in part explains the performance drop on industrial instances.

In light of these findings, we examine two ways of improving the performance of SLS algorithms on industrial instances. First, we present a method of tractably computing the average evaluation of solutions in a subspace that we call a hyperplane. These averages can be

used to estimate the correct setting of the backbone variables, with as high as 90% accuracy on industrial-like instances. By initializing SLS algorithms with these solutions, the search is able to find significantly better solutions than using standard initialization methods. Second, we re-examine the trade-offs between first and best improving search. We find that in many cases, the evaluation of solutions found by SLS algorithms using first improving search are no worse, and sometimes better, than those found by best improving. First improving search is significantly faster; using first improving search with AdaptG2WSAT, a state-of-the-art SLS algorithm for MAX-SAT, gives us more than a 1,000x speedup on large industrial instances.

Finally, we use our hyperplane averages to improve the performance of complete solvers of the satisfiability problem (SAT), the decision version of MAX-SAT. We use the averages to heuristically select a promising hyperplane and perform a reduction of the original problem based on the chosen hyperplane. This significantly reduces the size of the space that must be searched by the complete solver. Using hyperplane reduction as a preprocessing step, a standard complete SAT solver is able to outperform many state-of-the-art complete solvers. Our contributions have advanced the understanding of structured instances and the performance of both SLS algorithms and complete solvers on these instances. We also hope that this work will serve as a foundation for developing better heuristics and complete solvers for real-world applications of SAT and MAX-SAT.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisors, Darrell Whitley and Adele Howe. I would not have made it this far without their guidance. Under their mentoring, I have grown as both a scientist and a person and I will always be grateful for having the opportunity to do so. I would like to thank the Air Force Office of Scientific Research for funding this work. Much of my research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-11-1-0088.

I would also like to thank my parents, Lynn and Barbara Hains, for adopting me and giving me the best childhood a kid could ask for. Their unconditional support for my every endeavor has meant the world to me. Last but certainly not least, I would like to thank my wife, Bryanne Hains, for her enduring support, unwavering love, and plentiful cookies throughout my graduate career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

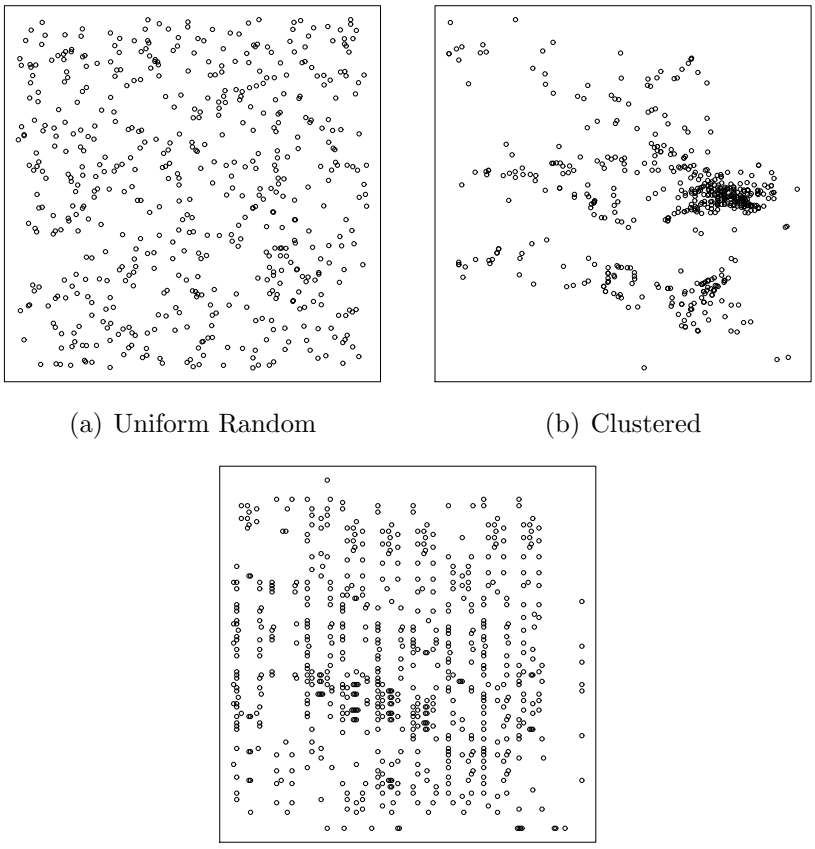
Chapter 1

Introduction

The goal of a combinatorial optimization problem is to find an optimal solution, or simply a good solution, among a set of finite solutions. In two important combinatorial optimization problems, the Traveling Salesman Problem (TSP) and Maximum Satisfiability (MAX-SAT), the set of possible solutions respectively grows factorially and exponentially in relation to the size of the problem. Complete solvers, those that use branch and bound or other techniques to provably find an optimal solution, are intractable for sufficiently large problems. This necessitates the use of incomplete solvers, algorithms that are not guaranteed to return the optimal solution [38].

Many of the best incomplete solvers for the TSP perform exceptionally well on *uniform* instances, those generated by a uniform random generator, but perform significantly worse on *structured* instances [42, 35, 47]. We define structured instances as those instances with an inherent structure that is not generally found in instances generated using a uniform random distribution. Two examples of structured instances for the TSP are routing problems through clusters of cities and the grid-like patterns that arise from circuit-board printing problems (see Figure 1.1). We have performed our own analyses of the TSP search space [32] and have shown that our crossover operator, Generalized Partition Crossover, can improve the performance of incomplete solvers on structured instances of the TSP [84, 85, 30].

Structured problems in MAX-SAT are found in industrial problems, such as circuit debugging and analysis [66, 13]. Like the TSP, incomplete solvers perform worse on structured instances than uniform random problems [47]. In the MAX-SAT 2012 challenge, the submitted incomplete solvers were able to solve all 378 uniform instances in the random uniform track, however they were only able to solve 2 out of the 55 industrial problems in the application track [52]. The underlying features of the search space that cause this difference in performance are not well understood [47, 61, 3]. This is a cause for concern as structured instances mainly arise from real-world applications of MAX-SAT.



(a) Uniform Random

(b) Clustered

(c) Grid-Like

Figure 1.1: Examples of three Euclidean TSP problems: (a) a 550 city problem generated with a uniform random generator, (b) a 535 city problem from TSPLIB constructed using airport locations and (c) a 662 city circuit board drilling problem.

Ansotegui et al [3, 5] have observed that the vast majority of industrial instances of MAX-SAT share three characteristics: a variable frequency that follows a power-law distribution, variable length clauses that also follow a power-law distribution and a community structure that is characterized by families of variables that have many interactions between variables of the same family but few interactions between variables of different families. Although these characteristics have been observed in industrial instances of MAX-SAT, it is not known if these characteristics represent an underlying structure that makes industrial instances more difficult or what that structure might be.

This gap in knowledge of the search space on structured instances and the consequences to local search behavior motivates the following questions that constitute the majority of this dissertation:

- How do the three characteristics observed by Ansotegui et al. in structured instances of MAX-SAT influence the behavior of local search?
- How is the underlying search space affected by these characteristics?
- Can we improve the performance of incomplete and complete solvers on structured instances?

We find that instances with a community structure and variable clause length are more difficult for incomplete solvers. Furthermore, we will present evidence that the underlying structure of the search space is, at least in part, the cause for this difficulty. In light of these results, we are able to develop several improvements for both incomplete and complete solvers that increase the performance of state-of-the-art algorithms on industrial instances.

1.0.1 Structure in Instances of MAX-SAT

To address the first question posed above, we begin by developing instance generators that are able to construct problems with the three characteristics in question: the power-law variable frequency, clause lengths following a variable frequency and a community structure. By using our generated instances, we are able to control for these three characteristics. We

first examine how the performance of AdaptG2WSAT, a state-of-the-art stochastic local search (SLS) algorithm [48, 49], is influenced by these characteristics. We find that those instances with a community structure and variable clause length are significantly more difficult to solve than uniform random instances. Instances with a power-law variable frequency are significantly easier for AdaptG2WSAT.

To better understand why these characteristics influence the behavior of SLS algorithms, we analyze the search space on our generated instances. We base our experimental methodology on that developed for previous analyses conducted on search space features of uniform instances, specifically plateaus [21, 76, 33] and backbones [89].

Plateaus are subsets of connected solutions in the search space with equal evaluations [21]. A large number of small plateaus indicates a rugged space, where the ruggedness of a space is determined by the variability in evaluations of neighboring solutions [40, 38]. A rugged landscape can be difficult to navigate for SLS algorithms [54], leading to a decrease in the quality of solutions that are found.

We find that on those instances with a variable clause length following a power-law distribution and a community structure that there are significantly more plateaus of a smaller average size than on uniform random instances. Thus, the search space is most rugged on those instances that are the most difficult for SLS algorithms.

The *backbone* is the set of variables that are consistently set to the same truth assignment across all global optima. Zhang has shown that on uniform random instances of MAX-3SAT that the size of the backbone set is correlated to the difficulty of the instances [89]. This is conjectured to be due to the fact that SLS algorithms have a higher probability of incorrectly assigning a backbone variable when there are more of them. Furthermore, the backbone size sets an upper bound on the size of the subspace that can contain the optimal solution. The upper bound grows smaller as the backbone size grows larger. We find in our analysis of industrial-like instances that that the backbone size is significantly larger in those instances with a community structure and clause length following a power-law distribution than other instances.

Our analysis of the search space reveals that instances with these characteristics have a more rugged space and a larger backbone, two search space properties that can make finding good solutions difficulty. Therefore, we provide not only the first documented link between the characteristics found in industrial instances and SLS performance, but our search space analysis provides a fundamental first step towards understanding why problems with these characteristics are more difficult.

1.0.2 Improving SLS Algorithms

In our analysis of the search space, we observed that the search space is more rugged with a larger backbone in industrial-like instances. We conjecture that estimating the correct settings of the backbone variables could improve the performance of SLS algorithms by using this information to initialize the search closer to good solutions. We empirically show that initializing an SLS algorithm with the correct settings of the backbone variables can improve the quality of solutions found at the end of the search.

Although this tells us that the backbone information can be useful to SLS algorithms, determining the correct settings requires prior knowledge of the global optimal solutions. Thus, any initialization method that must first find the correct backbone settings is impractical for search. We therefore propose a method of using the average evaluation of solutions in a subspace to estimate the correct setting of the backbone variables. This estimation requires no prior knowledge of the globally optimal solution and can be efficiently computed directly from the problem instance.

We refer to a subset of solutions in the search space that share a common truth assignment over some number of variables less than n as a hyperplane. Using the *Walsh transformation*, a discrete transform of the objective function to the Walsh basis, we are able to tractably compute the average evaluation of the solutions contained within an arbitrary hyperplane [83, 82, 31]. With this technique, we compute the average evaluations of the hyperplanes defined by the clauses of a given instance of MAX-SAT. These averages are then used to construct a probability distribution that is used to set each of the n variables initial truth assignment.

Using the probability distributions constructed from our hyperplane averages, we are able to estimate the correct setting of the backbone variables with over 90% accuracy, on average, on industrial-like instances. We use this estimation to initialize an SLS algorithm and find that on real-world applications of MAX-SAT, hyperplane initialization can significantly improve the quality of solutions found on the vast majority of our benchmark problems.

In addition to our hyperplane initialization method, we also re-examine the trade-off between two types of local search: first and next improving search. Best improving search selects the move yielding the best improvement in evaluation in the Hamming distance 1 neighborhood. In contrast, first improving search selects an arbitrary improving move in the neighborhood. Many implementations of best improving search have a $O(n)$ computational complexity per move in the worst case as a result of a scan to find the best move in a buffer of improving moves. Our implementation of first improving search has only a $O(1)$ worst case complexity per move as it can simply choose a move at random from the improving move buffer.

In 1992, Gent et al. conducted a study on the trade-off between using best and improving search in small, random uniform instances of SAT [23] and found that best improving local search was no better than first improving search. However, many modern SLS algorithms continue to use best improving search despite the higher computational cost [71, 49, 75].

We examine the difference between best and first improving search in the first phase of AdaptG2WSAT on industrial instances. The average evaluations of the local optimum found by best improving search are significantly better than those found by first improving search. However, SLS algorithms do not stop at the first local optimum encountered but enter a second phase of search. The average evaluations of the solutions found at the end of the second phase using first improving search are not significantly different, and often better, than those found by AdaptG2WSAT using best improving search.

We conjecture that this is due to "critical variables" being set early by best improving search. We empirically show that critical variables are flipped very early in the search by best improving search. This is due to the greedy bias in best improving search; the biggest change

in evaluation will come from flipping the variables that appear in the most clauses. This is not the case in first improving search as it has no such bias. Because the critical variables are not fixed early on by first improving search, AdaptG2WSAT has a higher probability of correctly setting the variables in the second phase.

We examine the evaluation change over time over two runs of 5,000,000 bit flips from AdaptG2WSAT with best improving search and AdaptG2WSAT with first improving search. The version using best improving search has a higher rate of improvement in the first phase of search. However, after entering the second phase of search, the version of AdaptG2WSAT using first improving search finds a better solution than AdaptG2WSAT with best improving search. At this point, the rate of improvement is greater in the version using first improving search. It is our belief that this occurs because the critical variables are not fixed in the first phase by AdaptG2WSAT with first improving search and it therefore has more opportunities to improve the evaluation of the candidate solution in the second phase.

In addition to leading the search to better solutions, first improving search is much faster. On the largest instances in our benchmark set, we see a speedup of over 1,000 times in AdaptG2WSAT by replacing best improving search with first improving search. Using both first improving initial descent and hyperplane initialization significantly improves the evaluations of solutions found on large industrial instances. Thus, we can significantly improve not only the quality of solutions found by SLS algorithms on structured instances, but the time required to do so.

1.0.3 Improving Complete SAT Solvers

Our hyperplane initialization method uses hyperplane averages to construct a probability distribution that can provide a remarkably good estimate of the backbone variables. The hyperplane averages appear to be a very powerful tool that can give us information about where good solutions are located in the search space. Given our promising results using hyperplane averages to improve incomplete solvers, we next look at how we can use the averages to improve the performance of complete SAT solvers.

The satisfiability problem (SAT) is the decision version of MAX-SAT. Instead of finding the solution that satisfies the greatest number of clauses, the goal is to determine if a satisfying solution to a given formula exists. If one does exist, we call the problem satisfiable. If not, the problem is unsatisfiable. Complete SAT solvers are guaranteed to return a satisfying solution if one exists, otherwise they will return "unsatisfiable". However, large, industrial applications of SAT are so large they can be intractable for SAT solvers. We therefore propose a search space reduction based on our hyperplane averaging method.

When defining a hyperplane, we fix the truth settings of k variables, and the remaining 'free' variables form the solutions in the hyperplane. By using all the possible combinations of truth assignments to the k variables, we can partition the solutions in the search space of a given instance into 2^k mutually exclusive subspaces, each containing 2^{n-k} solutions. Each subspace contains a reduced number of solutions that can be searched independently for a satisfying solution to the original instance.

We use this partitioning in a preprocessing step to reduce a given instance before passing it to a complete solver. Our preprocessing step works as follows. We use the first four most frequent variables in a given instance to partition the search space into 16 subspaces. Each hyperplane represents a reduced subspace that can then be searched by a complete solver. If the original problem is satisfiable, at least one of the 16 hyperplanes must contain a satisfying solution.

We next compute the averages of these subspaces using the Walsh coefficients as we do in hyperplane initialization. The hyperplanes are then ranked in order based on their evaluation; the rank 1 hyperplane has the best average evaluation while the rank 16 hyperplane has the worst. We then run a complete solver on the reduced problem corresponding to the rank 1 hyperplane. If a satisfying solution is found in the reduced space, the truth settings for the fixed variables are merged with the reduced solution, and the result is a satisfying solution for the original problem. If the rank 1 hyperplane does not contain a satisfying solution, we can run a complete solver on the next highest ranking hyperplane. Theoretically, this

process could be repeated (or done in parallel) until either a satisfying solution is found or all 16 hyperplanes have been determined to be unsatisfiable.

We combined our hyperplane reduction strategy with the MiniSAT complete solver [19] and submitted our solver to the 2013 SAT competition. In the Application SAT track, we ranked in 3rd place out of a field of 31 solvers, many of which are considered state-of-the-art and placed within the top three of previous competitions [68]. In one instance (003e.cnf), our entry was the only solver that was able to solve the instance. This demonstrates that not only is our strategy an effective one, but in some cases the hyperplane reduction can greatly simplify difficult industrial instances.

1.0.4 Summary of Contributions

We have analyzed the structure in instances of MAX-SAT, focusing on three characteristics found in industrial instances: variable frequency following a power-law distribution, variable length clauses that also follow a power-law distribution and a community structure. We have developed instance generators capable of producing instances with these structures and used them to analyze the impact of these characteristics on the performance of SLS algorithms. We found that instances with clause lengths following a power-law distribution and a community structure are significantly more difficult for SLS algorithms to solve.

Further analysis of the search space of these instances reveals several properties that can explain this drop in performance. Plateaus on industrial-like instances are smaller in size and greater in frequency than on other instances. This indicates a rugged search space that can be difficult for SLS algorithms to navigate. Industrial-like instances also have significantly larger backbones. This increases the probability in expectation that search will incorrectly set a backbone variable and decreases the upper bound on the size of the subspace containing the globally optimal solution. These results represent the first documented empirical evidence of significant differences in the search space of industrial-like instances and can, in part, explain why industrial instances are more difficult for SLS algorithms.

We present two methods of increasing the performance of SLS algorithms on industrial instances. The first is a method of efficiently computing hyperplane averages and using these averages to estimate the correct setting of the backbone variables. We also evaluate the trade-off between first and best improving search in the first phase of SLS algorithms. We find that first improving search does not significantly decrease the average evaluation of solutions found by the search over those found by best improving search, and in some cases can improve them. Combined, our improvements are able to significantly increase the evaluation of solutions found on industrial instances of MAX-SAT on the majority of our benchmark instances.

Finally, we improve the performance of complete SAT solvers by using our hyperplane averages to heuristically select a promising subspace of the search space. Empirical evidence shows that on industrial instances the subspace containing solutions with the best average evaluation contains a globally optimal solution in the majority of cases. By reducing the original formula based on the fixed variables that define the subspace, we can significantly reduce both the number of variables and clauses. Using the MiniSAT complete solver on these reduced problems, we were able to solve 113 of the 150 industrial instances in the Application SAT track of the 2013 SAT competition [68].

Developing theory through analyses of uniform random instances is a common practice in SAT, MAX-SAT and other combinatorial optimization problems. The resulting theory can potentially influence the development of algorithms, thus tailoring algorithms to uniform random instances. However, instances from real-world applications often have structure that is not captured in expectation by uniform random problems. This results in a gap both in our understanding of the theoretical implications of structure and the practical importance of developing algorithms for real-world applications. Our contributions advance the understanding of structured instances of SAT and MAX-SAT and in part explain why they are more difficult than uniform random instances for SLS algorithms. Furthermore, we advance the state-of-the-art in both incomplete solvers for MAX-SAT and complete solvers

for SAT. It is our hope that this work will inspire others and serve as a foundation for future research in structured instances of combinatorial optimization.

1.1 Document Structure

Chapter 2 covers much of the background information relevant to this dissertation. It covers concepts such as the search space, local search, and SLS algorithms. We also discuss previous work that is related to our original work presented in the following chapters.

Chapter 3 is our search space analysis of structured instances in MAX-SAT. We present our instance generators and the algorithms used to construct instances with three characteristics discussed above. We then show that our generator is capable of producing instances that are significantly more difficult for SLS algorithms. We then perform an in-depth analysis of the search space of these instances, finding many significant differences that can help explain the difficulty encountered by SLS algorithms in industrial instances.

Chapter 4 covers our improvements to SLS algorithms for MAX-SAT. We first present empirical evidence showing that correctly setting the backbone variables can improve the performance of SLS algorithms on industrial-like instances. We then present the Walsh transform and describe how the Walsh coefficients can be used to efficiently compute hyperplane averages. We then show how these averages can be used to find a remarkably good estimation of the correct setting of the backbone variables. We next revisit the trade-off between first improving and best improving search and find that first improving search can greatly reduce the computational time on large industrial instances without sacrificing solution quality.

Chapter 5 details our improvements to complete solvers for SAT. We again utilize the hyperplane averaging technique first described in Chapter 4 to heuristically choose a hyperplane that potentially contains a globally optimal solution. We then describe how the original problem can be significantly reduced based on this hyperplane. By running a complete solver on our hyperplane-reduced problem, we are able to outperform a large number of state-of-the-art complete solvers on industrial instances.

Chapter 6 is the conclusion of this dissertation. We present a summary of our results and contributions. As we hope that this work will also serve as inspiration for continued research into structure in combinatorial optimization problems, we outline several open questions that arise from our work. We believe that these questions form a solid foundation for future work.

Chapter 2

Anatomy of Stochastic Local Search for MAX-SAT

The focus of this dissertation is on the maximum satisfiability problem (MAX-SAT), an NP-hard combinatorial optimization problem [22]. However, at times we will also reference its decision counterpart, the satisfiability problem (SAT).

An instance of MAX-SAT is a boolean formula, which we will assume is in conjunctive normal form, with n variables and m clauses. The goal is to find the assignment of truth values to variables that maximizes the number of satisfied clauses. MAX-kSAT instances have exactly k variables in each clause. Instances are represented in the same manner, only the goal is simply to decide if there exists a truth assignment that can satisfy all clauses of the given instance. Similar to MAX-kSAT, kSAT refers to an instance of SAT with exactly k variables per clause.

A solution, x, to an instance of MAX-SAT or SAT is an assignment of truth values (either true or false) to each of the n variables in a given instance. We will represent a solution using a bit string. A value of 0 at bit p in x represents an assignment of false to variable p, likewise a 1 represents an assignment of true to variable p.

The following is an example instance in conjunctive normal form with four variables and three clauses:

$$(x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

We define our MAX-SAT objective function, f, as the number of clauses that remain unsatisfied under x. Our definition of f is due to convention; because f counts the number of unsatisfied clauses, this makes MAX-SAT a problem of minimization. In the above example, $f(1011) = 1$ as the clause $(\neg x_1 \vee \neg x_3 \vee \neg x_4)$ is unsatisfied. If the above formula were an instance of SAT, it would be satisfiable as the solution 1001 satisfies all three clauses. An instance of SAT is unsatisfiable if there are no satisfying solutions to that instance.

Given an instance with n variables, there are 2^n possible solutions representing all possible truth assignments. We use X to denote the set of all possible solutions. We define a neighborhood relation N on X such that $N : X \mapsto 2^X$. X and N form the space of solutions that a local search algorithm will search as well as the connections between these solutions. In the search algorithms that we will study for MAX-SAT and SAT, the neighborhood operator is a single bit flip. Thus $N(x)$ is the set of all solutions which differ by a single bit from x. Using the example above,

$$N(1011) = \{0011, 1111, 1001, 1010\}$$

Taken together, $L = (X, N, f)$ is referred to as a *combinatorial landscape* [65].

The aim of local search algorithms is to traverse the solution space of X, using the connections defined by N, in an attempt to find the best possible solution as evaluated by f. We will refer to the current solution of a local search as the *candidate solution* and denote this solution with x. By deciding to flip bit i in x, the local search moves in the search space from x to a new candidate solution y_i. We therefore often refer to a bit flip as a *move*. An *improving move* is one in which $f(y_i) < f(x)$, an *equal move* is one in which $f(y_i) = f(x)$ and a *disimproving move* is one in which $f(y_i) > f(x)$.

A *local optimum* is a solution x' such that $f(x') < y$ for all $y \in N(x)$. A *global optimum*, x^*, is a solution that $f(x^*) \leq y$ for all $y \in X$. The goal in MAX-SAT is to find a global optimum, or at least to get as close to global optimum as possible. In SAT, local search cannot determine if an instance is unsatisfiable. However, it can determine an instance is satisfiable by finding a solution that satisfies the instance. Therefore, local search for both SAT and MAX-SAT performs essentially the same function: to satisfy as many clauses as possible. Local search will therefore follow the gradient of the objective function in the local neighborhood. As a result, it will become stuck at the first local optimum it finds. If $f(x') > f(x^*)$, then the local search will be unable to find the global optimum.

To address this issue, an element of stochasticity is added to the local search. For example, the search could take a random walk after it encounters a local optimum by flipping some number of random bits in the local optimum. The algorithm could then continue with the

standard local search until another local optima is reached, and repeat. Such algorithms are known as stochastic local search (SLS) algorithms.

The best performing incomplete solvers for both MAX-SAT and SAT are SLS algorithms [38, 47, 53, 71, 49]. Even though SLS algorithms represent the state-of-the-art in incomplete solvers, Kroc et al. have shown that many SLS algorithms for SAT and MAX-SAT have difficulty on industrial instances [47]. In the 2011 SAT Competition, SLS algorithms were the top four algorithms overall (thus beating over a hundred incomplete and complete solvers) in the uniform random SAT track. These same solvers fell to below 40th place in the industrial instance category. In fact, no incomplete solvers were above rank 40 in this category [67]. Similarly, in the MAX-SAT 2012 competition, only 2 of the 55 industrial instances were solved by incomplete solvers, while SLS algorithms were able to solve all of the instances in the uniform random track [52].

The remainder of this chapter will discuss the anatomy of SLS algorithms for MAX-SAT and SAT that we break down into three major components: Initialization, Exploitation, and Exploration. This will provide a useful framework for discussing our analysis and improvements that follow in the next few chapters, as well as cover much of the prior work in SLS algorithms for MAX-SAT.

2.1 Initialization

SLS algorithms maintain a candidate solution x and incrementally move through the search space by flipping bits in x. While SLS algorithms differ in the heuristics they use to determine which bit to flip, they all must first choose an initial candidate solution x.

The initialization phase is the first step in any SLS algorithm and is perhaps the most widely overlooked step in the literature on SLS algorithms for MAX-SAT [24, 89, 62]. The most common method by far of initializing SLS is a uniform probability distribution. In uniform random initialization, each of the n variables in a given formula are set to 0 or 1 with equal probability.

Literature on initialization of SLS algorithms is sparse. Gent et al. performed a study on several deterministic methods of setting an initial string [24]. These methods did not use any information about the given instance, but rather tried different deterministic bit settings, e.g., setting all bits to 0, setting all bits to 1, setting half the bits to 0, etc. The deterministic initialization methods found similar results as uniform random initialization [24]. This suggests that randomness is not necessary in initialization of SLS algorithms, i.e., a uniform random solution is just as good as setting all bits to 1, given that multiple runs of the search on the same problem use different deterministic settings.

However, the study of Gent et al. does not address initialization methods which use information about the search space of the given instance to initialize the search. We are aware of only two initialization methods that have been reported in the literature that use information from the problem instance to construct an initial solution, which we discuss in detail.

2.1.1 Estimating Backbone Frequencies

The first initialization method is one that claims to estimate the backbone of a given instance. The *backbone* of an instance of MAX-SAT is the set of variables that have a consistent truth assignment across all globally optimal solutions for that instance [74]. For example, if an instance of MAX-SAT with four variables, x_1, x_2, x_3, x_4, has the global optima, $0010, 0011, 0110, 0111$, the backbone would be x_1, x_3. If an instance has a single global optimum, the backbone is the set of all n variables.

The *backbone frequency* is an extension of the backbone to all n variables of a given instance that counts the frequency that each variable-value pair $l = (x_i, v_i)$, where x_i is the variable and v_i is the truth value of that variable, occurs in the globally optimal solutions [89]. In the above example, the variable-value pair $l' = (x_2, 1)$ occurs twice. The backbone frequency of l' is then $\frac{2}{4} = .5$. Each variable will be associated with two variable-value pairs. If that variable is in the backbone, one of these pairs will have a frequency of 1 while the other will have a frequency of 0.

Computing the backbone frequency requires all global optima to be known for a particular instance. Therefore, the backbone frequency must be estimated for it to be useful in local search. Zhang presents a method of using a set of local optima to construct a probability distribution over the n variables [88]. This distribution is then used to construct a new solution for an SLS algorithm. Zhang's claim is that this distribution, called the pseudo-backbone frequency, approximates the backbone frequency.

The *pseudo-backbone frequency* is an estimate of the backbone frequency using the variable-value pairs found in a set S of local optima rather than the set of global optima. Because S is a set of locally optimal solutions they are mostly likely not of equal quality. To account for the variation in solution quality, the following *cost reciprocal average counting* (CRAC) method is used to weight the frequency of each variable-value pair l by the cost of the solutions in which it is found [89]:

$$p(l) = \frac{\sum_{\forall s_i \in S, l \in s_i} 1/c_i}{\sum_{\forall s_i \in S} 1/c_i} \tag{2.1}$$

where c_i is the cost of solution s_i. A new solution is then constructed by setting each variable x_i to 1 with probability $p(l = (x_i, 1))$, or to 0 with probability $p(l = (x_i, 0))$.

2.1.2 Estimating Locations of Optima

Qasem et al. take a similar approach to initializing SLS in that they use a set of local optima to determine a probability distribution [62]. This method is based on the hypothesis that solutions of good evaluation tend to cluster around other good solutions, with the best solutions being at the center of a cluster. Furthermore, the search space typically has multiple clusters. Parkes' hypothesizes that some clusters are centered around local optima while other are centered around global optima [59]. Therefore, instead of using the entire set of local optima, Qasem et al. first apply k-means clustering [34] using the Hamming distance between local optima as a distance metric. The centroid of the clusters found by k-means are then computed and the lowest evaluation is used as the initial solution to an SLS algorithm.

2.1.3 Similarity between Initialization Methods

In the cluster method of Qasem et al., once k-means determines a partitioning of the local optima into a set of clusters, C, the centroid of each cluster, c_i, is computed using the following equation:

$$c_i = \text{argmin}_{c_i} \frac{1}{|C_i|} \sum_{s \in C_i} d(c_i, s) \tag{2.2}$$

where c_i is the binary string that minimizes the Hamming distance d from the centroid to the solutions $s \in C_i$. The most efficient way to compute this centroid is by simply taking the mean of the solutions in the cluster, treating each solution as a real-valued vector in base 10:

$$c_i = \frac{\sum_{s \in C_i} s}{|C_i|} \tag{2.3}$$

In this case c_i will be a vector of n elements in the interval $[0, 1]$. By rounding the elements to the nearest integer, we find the bit vector that satisfies equation 2.2.

In Zhang's method, equation 2.1 is used to compute the probability of setting variable l to 1. This can be simplified to

$$p(l) = \frac{\sum_{\forall s_i \in S, l \in s_i} 1}{|S|} \tag{2.4}$$

where S is the set of local optima. Note that for this initialization method we require a probability for each variable-value pair l and $p(l = (x_i, 1)) = 1 - p(l = (x_i, 0))$. We need not compute all the variable-value pairs. Instead, we can more efficiently compute just the p values for variable-value pairs with a value of 1. Let q be a vector where element $q_i = p(l = (x_i, 1))$. We can then compute q using the following equation:

$$q = \frac{\sum_{s \in S} s}{|S|} \tag{2.5}$$

Equations 2.3 and 2.5 both find the average truth assignments in a set of local optima. Therefore, both methods utilize an average of local optima to initialize the search. Although

the computation is identical, there are two differences between the methods. The first is that in Zhang's method the entire set of local optima is used to determine the average. Qasem et al. first partition the set with K-means clustering. The second is that Zhang uses the averages directly as a probability to set each bit. Qasem et al. round the average to find the closest bit string and use this as an initial solution.

Nevertheless, both methods are very similar. They both require an upfront cost of running local search to find a large number of local optima (> 100 in the empirical trials in both studies [62, 89]). Furthermore, these are the only two methods known to us that use information about a given instance to initialize the search. Determining if better initialization methods can impact the performance of SLS will discussed in detail in Chapter 5.

2.2 The Exploitation Component

The exploitation component is the means by which an SLS algorithm guides the search towards a good solution. Typically, this resembles a standard local search in which gradient information provided by the evaluation function is used to guide the next move. Variations on the evaluation function can relax the greediness as we discuss in the next section.

GSAT is perhaps one of the most well-studied stochastic local search algorithms for SAT and MAX-SAT that makes use of a strict greedy strategy [71, 70, 38]. GSAT will determine the evaluation of all neighbors of the current solution, $y \in N(x)$ and will choose the neighbor y_i that maximizes $f(x) - f(y_i)$, where y_i is the solution found by flipping bit i in x and f is the standard MAX-SAT evaluation function returning the number of unsatisfied clauses.

GSAT does not require that a move be improving. If x is a local optimum, GSAT will make an equal move, otherwise it will make a disimproving move that maximizes $f(x)-f(y_i)$. It will always take a solution yielding the most favorable change to the current evaluation; it simply does not require that move to be an improving move, allowing it to escape local optima. In this sense it is one of the most exploitative SLS algorithms for MAX-SAT.

The most straightforward variation to this strategy is in how an improving move is chosen. We will refer to the strategy outlined above as *best improving search*. That is at

each step, the search flips bit i that maximizes $f(x) - f(y_i)$. If there are multiple solutions with evaluation equal to $f(y_i)$, best improving search breaks ties at random.

A variation on this strategy is *first improving search*. First improving search randomly selects a move from all improving moves, regardless of the change to evaluation. Let I be a subset of $N(x)$ such that $y_i \in I$ if and only if $f(y_i) < f(x)$. If I is non-empty, then the next move is chosen by randomly selecting a move from I. Otherwise, a random equal move is taken if one exists, otherwise a random disimproving move is taken.

Gent et al. studied the differences in best improving and first improving search with respect to the performance in terms of quality of solution on six problems: three SAT encodings of the n-queens problem ($n = 6, 8, 16$) and three random 3SAT problems with clause to variable ratio of 4.3 (for 50, 70 and 100 variables) [23]. This study found that there was no evidence suggesting that best improving search is better than first improving.

The issue of time complexity and differences between first improving and best improving has not been thoroughly addressed. This may be due to the fact that the instances used in prior studies were small and thus the difference in timings was not significant. However, modern industrial instances can have well over a million variables. As a result, implementations of GSAT and some state-of-the-art SLS algorithms, such as iterated robust tabu search [75] and AdaptG2WSAT [49], use a slow implementation of best improving search that can severely impact their performance on large instances. This issue is discussed at length in Chapter 4.

2.3 The Exploration Component

The last component of an SLS algorithm that facilitates exploration of the search space. In this case, we do not necessarily care if the move is improving or not. The exploration component allows the search to continue moving around the space and prevents it from becoming stuck at local optima. While GSAT allows the search to explore the space by allowing equal and disimproving moves, it does so randomly. This can be improved by adding a bias to a random walk when no improving moves are in the neighborhood of the

candidate solution [53]. Much of the research in SLS algorithms over the past twenty years has been in how to bias a random walk in lieu of gradient information.

One of the first SLS algorithms to make use of a biased walk is WalkSAT [70]. The WalkSAT algorithm has two major differences from GSAT. The first is that, given a candidate solution x, WalkSAT does not choose a move based on the objective function evaluation of the solutions in $N(x)$. Instead, it examines a variable's break-count. The *break-count* of variable x_i is the number of clauses that will become unsatisfied if x_i is flipped. WalkSAT uses the break-count to heuristically select which variable to flip, but also has a probability of randomly flipping a variable.

The second difference is that WalkSAT does not consider all n variables when making a move. WalkSAT first chooses a random clause, c_j, which is unsatisfied under the current candidate solution x. With probability p, it then examines the break count of all variables in c_j and chooses the variable with the least break count. With probability $1-p$, it chooses a random variable in clause c_j.

The WalkSAT family of heuristics represents perhaps the most widely-used and best-performing heuristics for SAT and MAX-SAT [70, 53, 49, 38, 46]. The majority of improvements to WalkSAT are in the choice of which variable to flip after an unsatisfied clause is chosen [70, 53, 49]. The Novelty heuristic considers history in this choice, by biasing the choice towards those variables that have not been recently flipped [53]. Novelty+ adds a lookahead into the variable choice by also considering the evaluation of solutions in $N(y_p)$ when considering a flip of variable p in the candidate solution [48].

While WalkSAT emphasizes exploration of the space, its exploitation component is much more relaxed than that of GSAT. Consider an instance of MAX-3SAT and a candidate solution x with an evaluation of $f(x) > f(x^*)$ where x^* is the global optimum. WalkSAT will first choose clause c_j which is unsatisfied under x. Because $k = 3$, c_j contains three variables: i, j, and k. Let $bc(i)$ be the break-count of variable i and $mc(i)$ be the *make-count*, that is the number of currently unsatisfied clauses that will become satisfied if we flip variable i. Note that $f(y_i) = f(x) + bc(i) - mc(i)$. It is quite possible that $bc(i) < bc(k)$ and

$bc(i) - mc(i) > bc(k) - mc(k)$. Thus with probability p, WalkSAT will flip variable i even though flipping variable k will produce a better solution.

AdaptG2WSAT is an SLS algorithm that combines the search space exploration of WalkSAT and the greediness of GSAT [49]. At each step of the search, if there exists $y_i \in N(x)$ such that $f(y_i) < f(x)$, AdaptG2WSAT will flip bit i that minimizes $f(y_i)$. If only equal or disimproving moves exist, AdaptG2WSAT will employ the Novelty+ heuristic to determine which bit to flip.

This strategy works well. Kroc et al. find that AdaptG2WSAT is the best performing SLS algorithm on structured instances of MAX-SAT of those algorithms in the UBCSAT suite of SLS algorithms [79, 47]. AdaptG2WSAT and other SLS algorithms based on it have routinely been among the best performing SLS algorithms at SAT competitions [67, 48, 49] on random instances.

We will focus on AdaptG2WSAT for our empirical studies as it represents a well-studied, top performing SLS algorithm that combines both a greedy exploitation component and an exploration component. Furthermore, the code is freely available as part of the UBCSAT collection of incomplete solvers for MAX-SAT [79].

Although AdaptG2WSAT performs exceptionally well on uniform random instances, its performance decreases dramatically on structured instances from real-world applications [47]. The focus of the next chapter will be to better understand why AdaptG2WSAT and other SLS algorithms have such difficulty with structured instances.

Chapter 3

Structured Instances of MAX-SAT

Stochastic Local Search (SLS) algorithms for SAT and MAX-SAT have historically been developed using uniform random instances as benchmark problems [40, 70, 49, 71]. These instances have several attractive properties: they are easy to generate and the expectation of uniformity allows for fairly straightforward analytical analyses. However, several studies have shown that instances of MAX-SAT with properties that are not found in expectation in uniform random instances are difficult for SLS algorithms [47, 38, 39, 12, 72].

Ansotegui et al. have found that difficult instances of SAT and MAX-SAT have three characteristics in common: a distribution of variable frequency that follows a power-law probability density function [3], a distribution of clause lengths that also follows a power-law distribution [4] and a community structure [5]. The *community structure* arises in graph representations of SAT and MAX-SAT instances and is characterized by subsets of nodes that share many connections between other nodes in the same subset, but few connections to nodes outside of the subset. These characteristics are very similar to those found in graph representations of social and communication networks.

These characteristics are associated with difficult problems, but what is not known is why these problems are difficult and how the characteristics observed by Ansotegui et al. relate to the search space of the instance. In the following analyses, we will examine these characteristics to determine how they influence properties of the search space. To address the issue of why problems with these characteristics are difficult, we first examine several features of the search space related to the ruggedness of the space.

The *ruggedness* of a search space is a measure of how the fitness differs between neighboring points [54]. Problems with more rugged landscapes are typically more difficult for stochastic local search [38]. We find that the search space is more rugged on instances with a community structure and variable clause length. We find an inverse correlation between

this ruggedness and AdaptG2WSAT's ability to find global optima. The variable distribution does not seem to be an important factor in increasing ruggedness of the space or in decreasing the performance of AdaptG2WSAT. Indeed, as our following analysis will reveal, a power-law distribution specifically seems to make problems easier rather than harder for SLS. It is also important to note that these characteristics are identifiable in a tractable way prior to running any search.

To begin our analysis, we will first review the three characteristics of difficult instances. We next discuss a new instance generator that we developed to construct MAX-SAT instances with a community structure similar to that found in industrial instances. We then show that this instance generator produces highly rugged instances that are difficult for AdaptG2WSAT to solve. Specifically, AdaptG2WSAT fails to solve instances from our generator with variable counts as low as $n = 100$ after 10,000 bit flips. In contrast, similar sized instances from uniform or power-law generators are solved with a probability approaching 1 in less than half of the same number of bit flips. We then present an analysis of the search space of these instances which reveals certain features of the search space are significantly different on instances with a community structure than those without.

3.1 Characteristics of Structured MAX-SAT Instances

To examine the characteristics of community structure, variable distribution and clause distribution, it is helpful to first transform the instance of MAX-SAT or SAT to a graph. This allows us to not only visualize the instance, but also to employ algorithms developed for graphs to compute statistics about the instance.

MAX-SAT and SAT instances can be represented by a number of graph representations [73]. Two of the most common representations are the variable interaction graph (VIG) and the clause-variable interaction graph (CIG) [5]. The *variable interaction graph* (VIG) is a graph $G = (V, E)$ where each vertex $v_i \in V$ corresponds to variable i in the SAT or MAX-SAT instance. The edge set E contains edges between vertices v_i and v_j if and only if variable i and variable j appear in a clause in the instance.

The *clause-variable interaction graph* (CIG) is a bipartite-graph $G = ((V_1, V_2), E)$ where V_1 is a set of vertices corresponding to variables as in the VIG and V_2 is a set of vertices which map one-to-one to the clauses of the SAT or MAX-SAT instance. Edge (i, j) appears in E if and only if variable i appears in clause j.

Ansotegui et al. have shown that both VIGs and CIGs constructed from industrial instances have a *scale-free* structure that is not present in uniform random instances [3, 5]. This structure is characterized by the distribution of vertex degrees which can be described by the power-law probability density function [3],

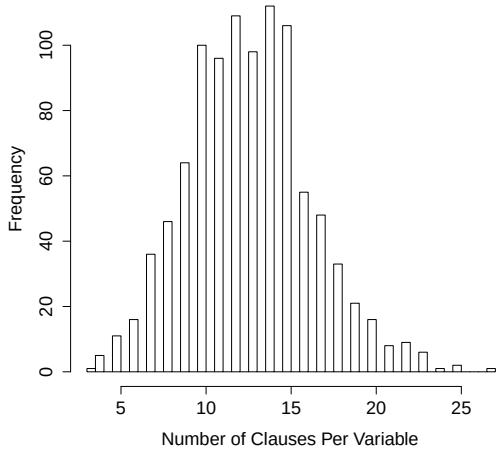
$$\theta^{pow}(x; \beta) = (1 - \beta)x^{-\beta} \tag{3.1}$$

As examples, Figure 3.1 shows the histograms of variable frequency for three different instances of MAX-SAT with 50 variables and 214 clauses: an instance generated using a uniform distribution of variable frequency (Figure 3.1a), an instance generated using a power-law distribution (Figure 3.1b), and the b15 industrial instance from the MAX-SAT 2012 competition (Figure 3.1c) [52]. These histograms show the number of degrees at each node of a VIG representation of the instances. The distributions shown in Figure 3.1b and Figure 3.1c are typical of problems with a scale-free structure [3, 80], while Figure 3.1a is clearly from a uniform distribution.

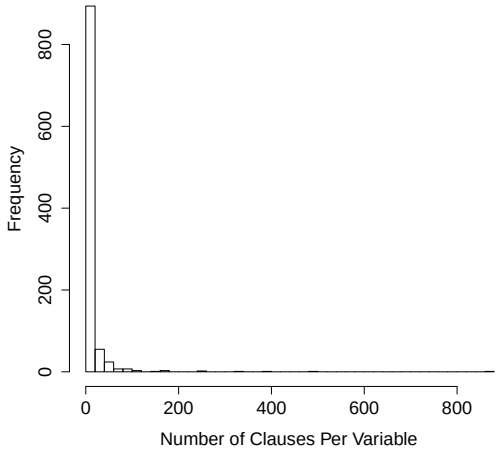
We analysized of 380 industrial instances collected from the MAX-SAT 2012 competition [52] and the 2011 SAT competition [67]. The instance names along with the number of variables and clauses can be found in Table 1 of Appendix A. We found that the variable distribution in all but four of these instances followed a power-law distribution. We found that in all but ten of these instances that the distribution of clause lengths also follows a power-law distribution. These analyses are described in detail in Section 6 of this chapter.

3.1.1 Community Structure in SAT and MAX-SAT instances

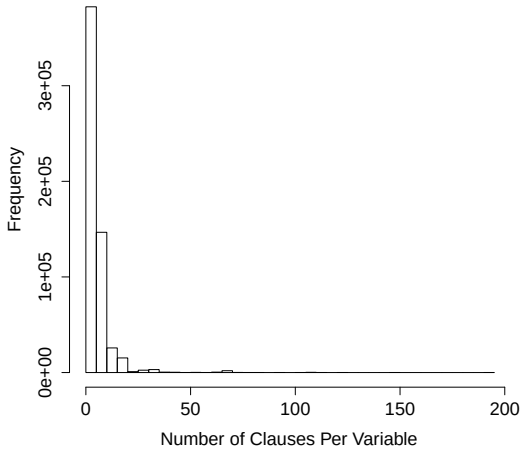
Another feature common to industrial instances is the presence of a *community structure* in their VIG and CIG representations [5]. Often found in complex networks such as communication networks or social links, a community structure refers to a clustering of nodes into



(a) Generated Random Uniform



(b) Generated Random Power-Law



(c) Industrial

Figure 3.1: Histograms of the number of clauses in which each variable appears for three types of problems: an instance randomly generated with a uniform distribution (a), an instance generated randomly with a power-law distribution (b), and an industrial instance from a circuit debugging application (c).

families. Nodes within the same family share more edges than they do with those outside of the family. The small world networks of Watts and Strogatz [81] were proposed as a model of complex networks that can capture community structure. As a way of quantifying the community structure in such models, Newman et al. have proposed the following modularity metric [58].

Given a graph $G = (V, E)$ and a partition of V into j families, $P = P_1, P_2, ...P_j$, the *modularity* of G is defined as

$$Q = \sum_i (A_{ii} - a_i^2) \tag{3.2}$$

where A is a jxj matrix such that entry a_{ij} is the number of edges shared between nodes in P_i and P_j and a_i is defined as the sum of row i in E, $a_i = \sum_j a_{ij}$. Note that Q depends on the partition P. The modularity of a graph is defined as the Q value under the partition yielding the maximum modularity score. A partition with a single subset, i.e. $P = P_1 = V$ will return a Q value of 0. Thus the minimum modularity of any graph is 0 and the maximum possible modularity is 1. Random graphs have an expected modularity of 0 [58].

In complex networks based on communication and social networks, graphs with a high community structure typically have a Q value in the range of .4 to .7 with values over .7 being rare [58]. Ansotegui et al. [5] studied on over 100 industrial instances from the SAT 2011 competition and found that the VIG representations of all but one class of industrial problems have a modularity score $> .4$, with some instances having a modularity high as .9. We found similar results in instances produced by our community structure generator described in the next section. The mean modularity was over $> .4$ on instances generated with a community structure, while the mean modularity of uniform instances with no community structure was 0.

3.1.2 Summary of Structure in MAX-SAT

We have discussed three characteristics that relate to the structure of MAX-SAT and SAT instances: the power-law distribution of variable frequency and clause lengths and the community structure. These characteristics will form the basis for our analysis in the following chapter for several reasons. First, we know that the overwhelming majority of industrial instances have these characteristics due to the work of Ansotegui et al. [3, 5] and our own analyses (See Section 6 for details). Second, we have methods to determine if an instance has these characteristics: The goodness of fit test for power-law distributions and the modularity metric for community structure. Not only does this allow us to quantify the

degree to which instances have these characteristics, but it may also be useful for future work in the design of enhanced portfolio solvers. Finally, as we will discuss in the next section, we have developed methods of generating instances with these characteristics.

3.2 Generating Structured Instances of MAX-SAT

Uniform random instances of MAX-SAT are generated by choosing k variables per clause with a uniform distribution over each variable with the constraint that a particular variable can only appear once per clause. Thus in expectation, each variable will appear in the same number of clauses. Uniform instances serve as standard benchmarks for developing incomplete solvers for SAT and MAX-SAT. It is perhaps not surprising that there is a significant decrease in the performance of SLS algorithms on industrial instances [47, 3].

We have identified three characteristics of industrial instances that are not present in uniform random instances. However, the industrial instances at our disposal are much too large for a complete analysis. We therefore wish to generate small instances with similar characteristics. Smaller instances will allow us to perform a more thorough analysis of the search space in our effort to determine why instances with these characteristics are more difficult for SLS algorithms. The power-law generator of Ansotegui et al. [4] will serve as our starting point. We alter this generator so that it constructs instances with community structure. We will then show that instances created with our generator are difficult for state-of-the art SLS algorithms.

3.2.1 Generating Power-Law Instances

To construct a problem generator that can create instances with the characteristics we wish to examine, we begin with a problem generator due to Ansotegui et al. that is capable of producing instances with a variable frequency that follows a power-law distribution [4].

The power-law continuous probability distribution is given by the density function in Equation 1 [4]. Ansotegui et al. derive a discrete probability function based on this function by dividing the domain $[0, 1]$ into n intervals. A continuous probability distribution θ over

the domain $[0, 1]$ can be broken into n intervals with end points $1/n$,$2/n$,...,n [4]. Normalizing for the n intervals yields the following family of discrete probability distributions

$$P(X = i; n) = \frac{\theta(i/n)}{\sum_{j=1}^{n} \theta(j/n)}$$

As the power-law distribution is not defined at $x = 0$, the term ϵ is introduced, making the density function

$$\theta^{pow}(x; \beta) = \frac{1 - \beta}{(1 + \epsilon)^{1-\beta} - \epsilon^{1-\beta}}(x + \epsilon)^{-\beta}$$

This function is then discretized using the above method, giving

$$P(X = i, \beta, n) = \frac{(i + \epsilon n)^{-\beta}}{\sum_{j=1}^{n}(j + \epsilon n)^{-\beta}}$$

This distribution can be used to generate instances that have a power-law distribution of variables as follows. To generate a formula of m clauses, we must choose k variables for each clause. These variables are chosen using the following algorithm This will generate

Algorithm 1: Variable Selection in Power-Law Instance Generator

```
1 Let C = ∅;
2 while Number of elements in C = 1 do
3     Let p =Random number in [0 + ε, 1 + ε];
4     Let q = 0;
5     Let i = 0;
6     while q < 0 do
7         i = i + 1;
8         q = q + P(i/n);
9     Repeat previous loop to select a new i
```

a set of k variable indices from the power-law distribution; variables with the lowest index will have the highest probability of being chosen. Once the variables are chosen, they are negated with a .5 probability. The algorithm is repeated for each of the m clauses in the instance [4]. To construct a formula with variable length clauses, we use a similar algorithm to first select the length of each clause prior to generating the variables.

In practice, we define our function P by letting n be the number of variables and $\beta = .082$. This value of β was found to generate problems with similar characteristics as industrial

instances [3]. To generate instances with a uniform distribution we replace the discrete power-law function in Algorithm 1 with $P(X = i; n) = 1/n$.

3.2.2 Adding Community Structure to Generated Instances

Our generator allows us to create problems with a uniform or power-law variable frequency and fixed or variable clause lengths. We now wish to extend the generator to construct problems with the third characteristic we wish to examine: community structure.

To add community structure to the instances generated with Algorithm 1, we want to generate families of variables that co-occur with one another with high frequency, but co-occur with variables outside the family with low frequency. To accomplish this task, we add an additional parameter l to specify the number of families to construct. We then partition the variables into $l + 1$ ordered sets based on the variable indices, such that the first set contains $n/(l + 1)$ variables and the remaining sets have $n/(l + 1) + 1$ variables.

For example, given the n variable indices $1, 2, 3, ..., n$, set 1 consists of variables

$$\{1, 2, .., n/(l + 1)\}$$

set 2 are the variables

$$\{n/(l + 1) + 1, n/(l + 1) + 2, ..., 2(n/(l + 1)), 2(n/(l + 1)) + 1\}$$

set 3 is

$$\{2n/(l + 1) + 1, n/(l + 1) + 2, ..., 3(n/(l + 1)), 3(n/(l + 1)) + 1\}$$

and so on for $l + 1$ sets. The final variable in the last set will be the first variable in set 2. Thus set 1 is mutually exclusive of all other sets, but the remaining sets will overlap in one variable.

The first set is special and we refer to it as *connector* variables, $D = d_1, d_2, ..., d_{n/(l+1)}$. This set does not correspond to a family of variables, but will be distributed within the remaining sets to connect them to one another. The remaining sets correspond to the partition of families,

$$F = \{F_1 = \{v_1^1, v_2^1, ..., v_{n/(l+1)}^1\}, F_2, ..., F_l = \{v_1^l, v_2^l, ..., v_{n/(l+1)}^l\}\}$$

Table 3.1: Description of the five instance generators and the shorthand notation used to refer to each generator.

Notation	Variable Occurrence	Clause Length	Modular
pl/k3	Power-law	Fixed Length, $k = 3$	No
uni/k3	Uniform	Power-law	No
pl/pl	Power-law	Fixed Length, $k = 3$	No
pl/k3	Uniform	Power-law	No
pl/k3 (mod)	Power-law	Fixed Length, $k = 3$	Yes
uni/k3 (mod)	Uniform	Power-law	Yes
pl/pl (mod)	Power-law	Fixed Length, $k = 3$	Yes
pl/k3 (mod)	Uniform	Power-law	Yes

We call algorithm 1 on each family of variables to generate m/l clauses from each family. This gives us m clauses composed of l families. The variables within each family have a high rate of co-occurrence, which was one of our objectives, but there is still a high rate of co-occurrence between families due to overlapping variables. The last variable maps to the tail-end of the power-law distribution, so in expectation it will occur least frequently. However, this same variable is at the front-end of the power-law distribution in the next family and it will be the most frequently occurring variable in expectation. Therefore, without further modification, there will still be a high number of connections between families.

To adjust for this, we map the last variable of each family to a connector variable in D. Every appearance of the variable $v^i_{n/(l+1)}$ in family F_i is replaced with variable d_i or variable d_{i+1} with equal probability. Thus our generator creates instances with a 'ring' structure such that family F_i will likely be connected to F_{i-1} and F_{i+1}. We will show that instances generated in this manner have a modularity similar to that in industrial instances and are difficult for SLS.

3.2.3 Comparison of Instance Generators

With our community structure extension, we have eight generator types. Table 3.1 summarizes our generators and defines the notation we will use to refer to them.

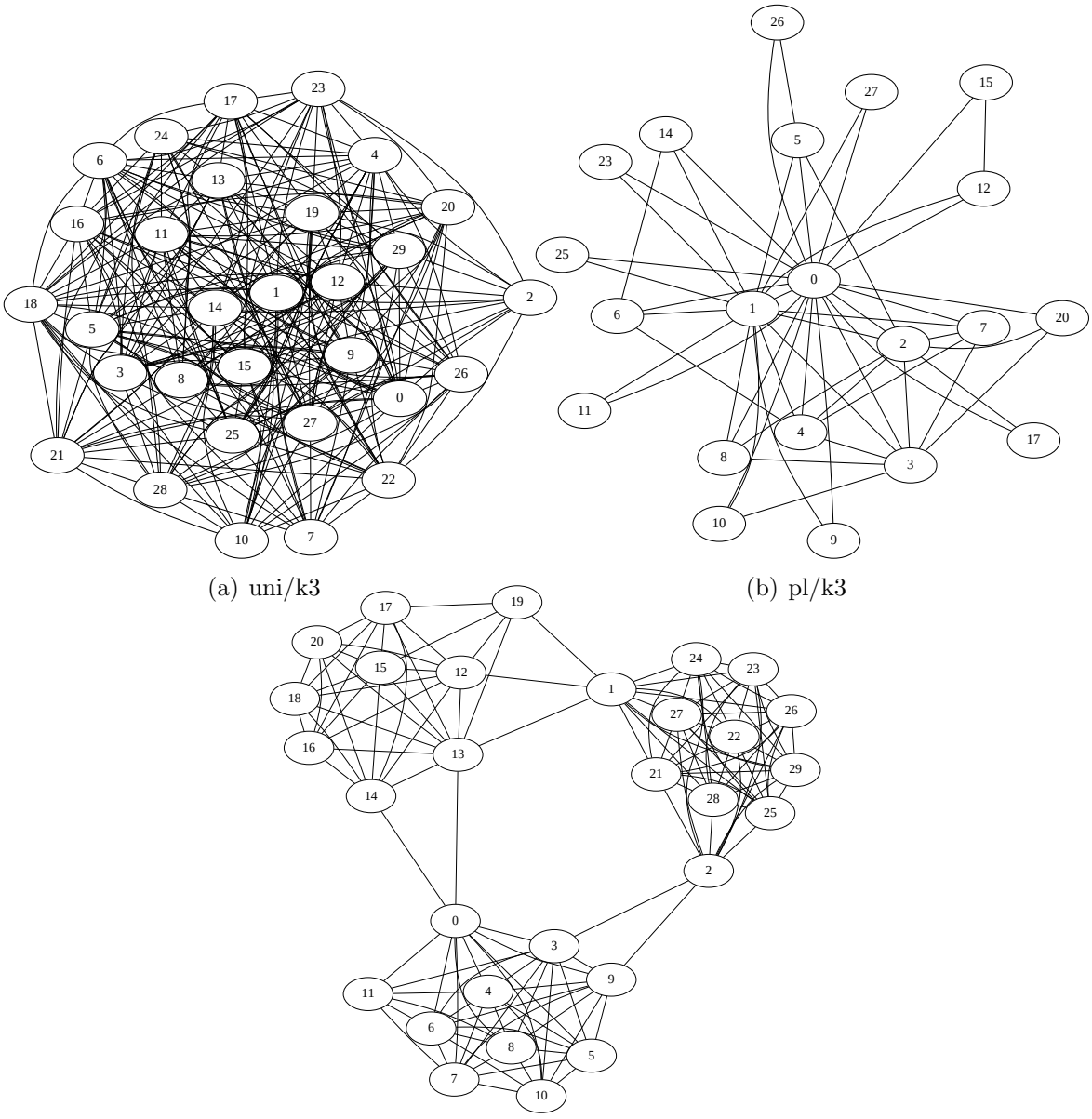
To visualize the difference in variable connectivity between our generators, we generated one instance from each of the pl/k3, uni/k3 and pl/k3 (mod) generators with $n = 30$ variables

and $m = 150$ clauses. We used a $\beta = .82$ for the power-law distribution and set the number of clusters to 3 for our modular generator. We then constructed a VIG for each instance. The resulting graphs are shown in Figure 3.2. It is clear that the uniform instance in Figure 3.2(a) resembles a random graph. In the non-modular instance with a power-law distribution in variable frequency, we see that there are a few variables with many connections and many variables with few connections. The modular instance in Figure 3.2(c) shows the community structure that we wanted to capture: three clusters of nodes with many interior connections, but few exterior connections.

To confirm our hypothesis that uniform and power-law generated instances do not have community structure we need to measure the community structures of instances created by our generators. The modularity metric of Newman et al. [58] described in Equation 2 provides such a measure. Computing the modularity of a graph requires a partition of the vertices of the graph. Finding the optimal partition is NP-hard [11, 5]. We therefore use the graph folding algorithm of Newman et al. [58], to approximate the optimal partition as it has been shown to find partitions close to optimal and is tractable even on larger graphs [5].

Given a VIG representation of an instance of SAT or MAX-SAT, the *graph folding algorithm* (GFA) works as follows. First, each variable is assigned to one family such that if there are n variables, there are n families each containing a single variable. The modularity of VIG under this initial partition is computed using Equation 2. The algorithm then examines each vertex and computes the change to modularity that would occur from removing the vertex from its current partition and adding it to each other partition. The change that results in the maximum increase to modularity is carried out. This process is continued until no improvement to modularity can be found, at which point the algorithm has reached a local optimum.

Once a local optimum is reached, GFA constructs a new graph where the vertices $V = 1, 2, ..., l$ correspond to the partitions in the locally optimal partition $P = P_1, P_2, ..., P_l$ in the local optimum. Edges are made such that two vertices (i, j) are connected if and only if there is an edge between the partitions P_i and P_j. The algorithm is then restarted on this new



(a) uni/k3

(b) pl/k3

(c) pl/k3 (mod)

Figure 3.2: Variable interaction graphs of instances with $n = 30$ and $m = 150$ clauses made by three generators: (a) the standard uniform distribution with $k =$ clauses, (b) Ansotegui's power-law generator with $k = 3$, and (c) our modular generator using a power-law distribution on variable frequency and $k = 3$.

Table 3.2: Mean and standard deviations of modularity on our generated instances. See Table 3.1 for the description of our generators.

pl/k3	uni/k3	pl/pl	uni/pl
0.00 ± 0.01	0.07 ± 0.02	0.00 ± 0.00	0.02 ± 0.02
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
0.72 ± 0.02	0.72 ± 0.03	0.75 ± 0.04	0.67 ± 0.01

graph. It continues recursively until no further improvements to modularity can be found. Although GFA is not guaranteed to return the optimal partition, it does find partitions with higher modularity than those found by other heuristic algorithms that stop at the first local optimum, such as the Label Propagation Algorithm (LPA) [14, 5].

We generated a set of 50 instances with $n = 50$ variables and $m = 214$ clauses with each of the instance generators. For the modular generator we choose $l = 6$ families as empirically we found this parameter setting to generate the highest modularity. We then ran GFA on each of the instances and recorded the modularity. The modularity of a graph is the maximum modularity under the optimal partition. Because GFA is a non-deterministic algorithm, we at best can say the Q found is a lower bound on the modularity. Therefore, we ran GFA 50 times for each instance and recorded the maximum Q found for each instance. Table 3.2 reports the mean and standard deviation of the maximum Q values recorded for each instance.

The (mod) generators are the only generators which produce instances with a modularity similar to those reported by Ansotegui et al. on industrial instances [5]. This gives us a set of instances that have similar characteristics to industrial instances. The rationale for analyzing these these instances is to determine how these factors influence the performance of SLS algorithms and what impact they have on the underlying search space. SLS algorithms perform poorly on industrial instances as compared to their performance on uniform random instances [47]. Therefore, we expect that SLS algorithms will perform worse on those instances that share characteristics with industrial instances.

To determine if this is indeed the case, we will use the run length distribution (RLD) as defined by Hoos et al. [41] to examine the performance of AdaptG2WSAT on our generated

instances [49, 48, 71, 70, 53]. The RLD estimates the probability distribution that an SLS algorithm will find the overall best solution as a function of the number of bit flips. We used the same 50 instances from each generator with $n = 50$ variables and $m = 214$ clauses that were used to compute the modularity in Table 3.2.

To compute the RLD we ran AdaptG2WSAT on each of our generated instances for 50 runs with each run starting from a different randomly generated solution. Each run was allowed 10,000 bit flips. For each run, we used an indicator variable for each bit flip. This variable was set to 1 if a globally optimal solution was found, or set to 0 if not. We then computed the mean of the resulting binary vectors over the 50 runs from each of the 50 instances from the eight generators. This gave us the probability of finding the optimal solution at each bit flip for each instance type. We then repeated this experiment using 50 more instances with each generator with $n = 100$ variables and $m = 427$ clauses.

Figure 3.3 shows the RLDs of each instance type for the $N = 50$ instances;Figure 3.4 shows the RLDs for the $n = 100$ instances. AdaptG2WSAT has the most difficulty finding the globally optimal solutions on the modular instances with variable clause length distributions. Surprisingly, on the $n = 100$ instances, less than 20% of the modular instances with variable clause lengths were solved after 10,000 bit flips.

Figure 3.3 also shows that AdaptG2WSAT can find the optimal solutions more quickly on those instances with a power-law distribution over the variables than those with a uniform distribution. This would seem to indicate that the difficulty in hard industrial instances is related more to the community structure and clause length than the variable frequency. Having developed generators capable of producing difficult instances of MAX-SAT, we now wish to perform a deeper analysis of these instances to better understand the differences in SLS performance noted in Figures 3.3 and 3.4.

3.3 Plateaus in Structured Instances

Industrial instances come from real-world problems which are transformed to an instance of MAX-SAT, e.g., circuit debugging and verification [66] or planning [44, 45]. There may be

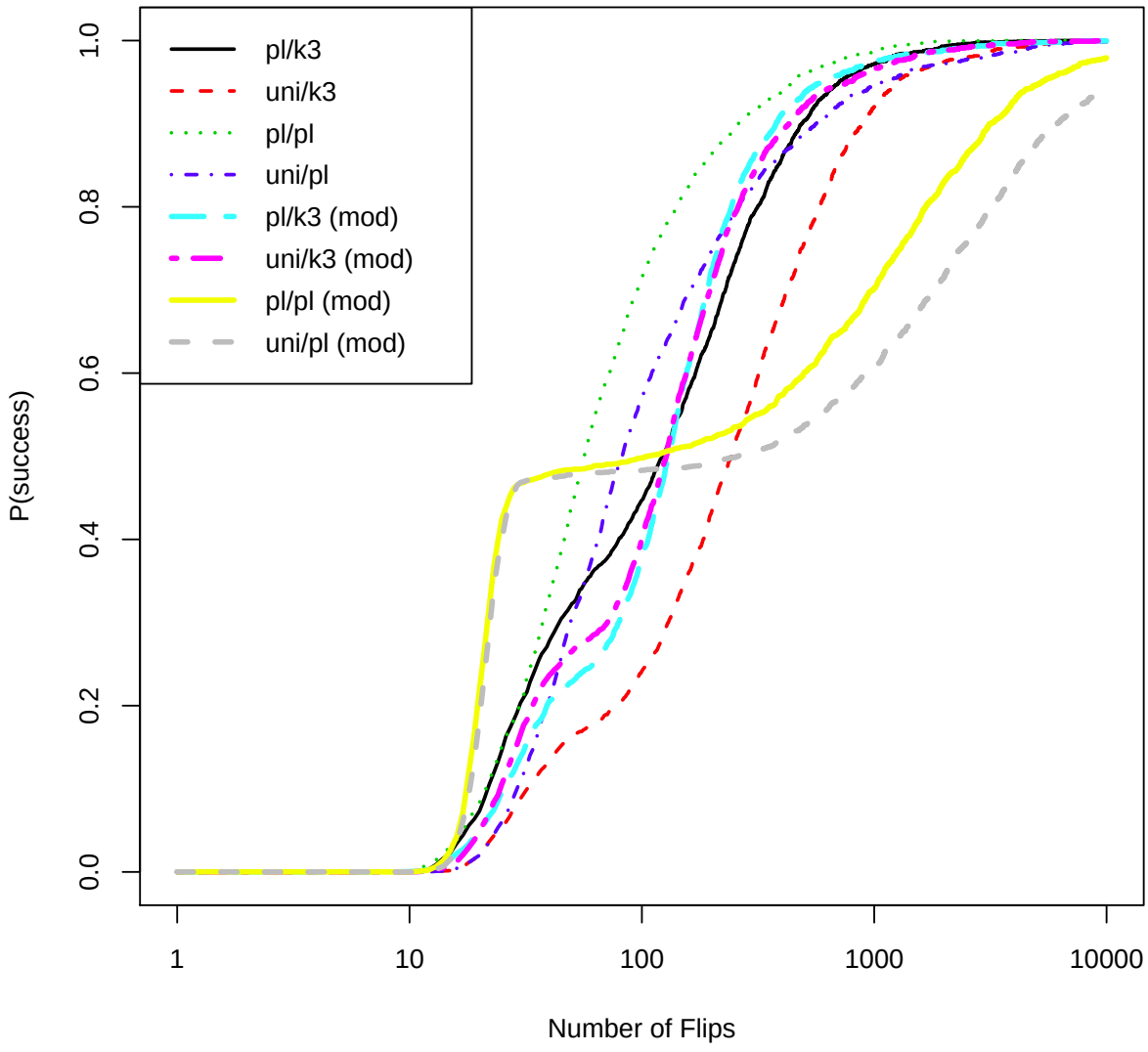


Figure 3.3: The Run-Length Distribution of 10,000 bit flips of AdaptG2WSAT on 50 instances constructed by each of the eight generator types with size $n = 50$.The legends denote the generator type used to constructed the instances using the notation from Table 3.1.

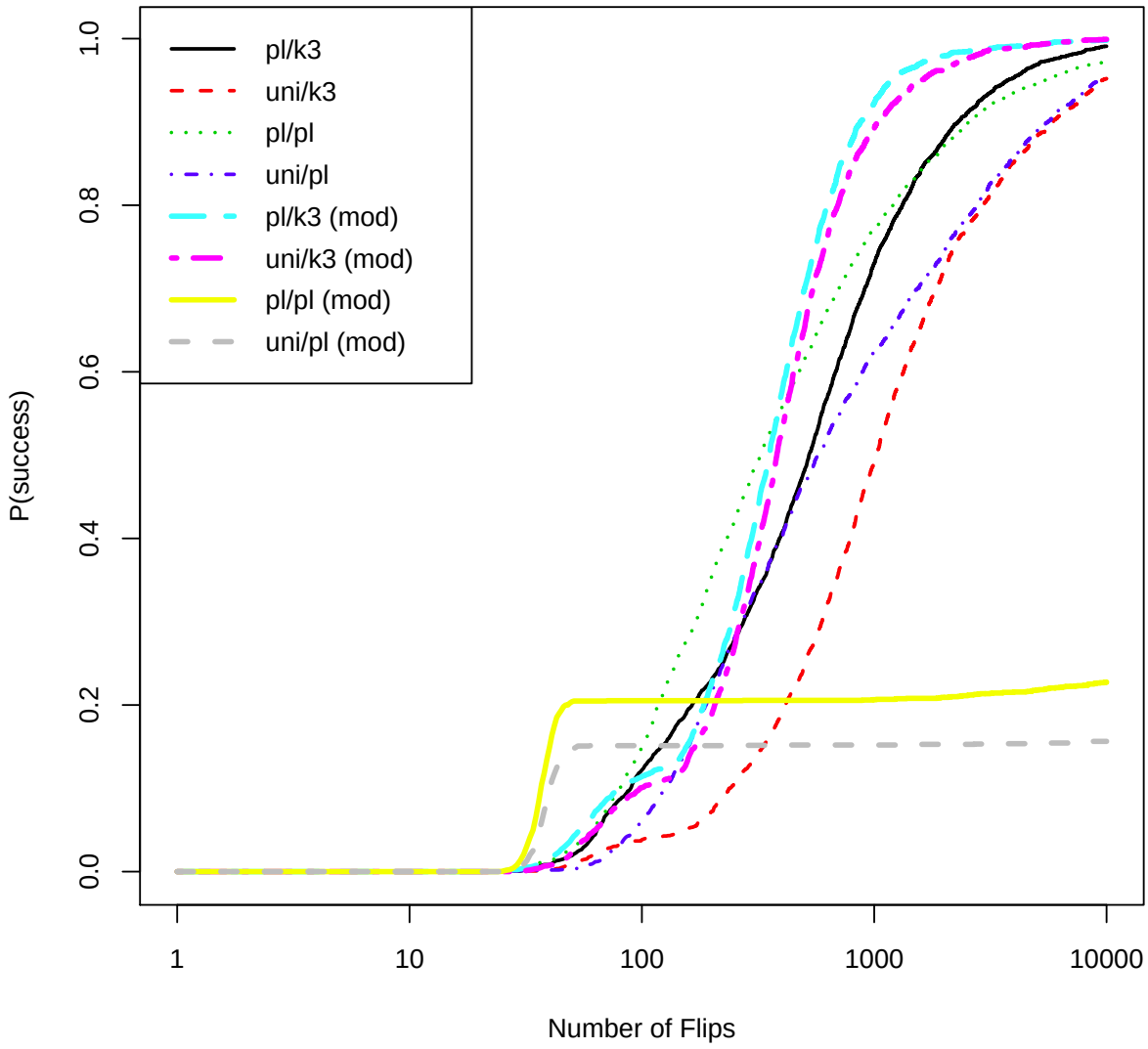


Figure 3.4: The Run-Length Distribution of 10,000 bit flips of AdaptG2WSAT on 50 instances constructed by each of the eight generator types with size $n = 100$. The legends denote the generator type used to constructed the instances using the notation from Table 3.1.

a deeper structure inherent to these instances as a result of the specific application. However, instances with a scale-free and community structure offer a good starting point for our analysis as many industrial instances have been observed to have these characteristics [4, 3, 5]. Furthermore, we have shown that generated instances with these properties are difficult for AdaptG2WSAT. By using our generators we can directly control for these characteristics in order to investigate both their individual effects and interaction effects on features of the search space. Before we discuss our new results, we will first review previous work on these features in uniform instances of MAX-SAT.

3.3.1 Previous Analysis of Plateaus in Uniform Instances

One of the most well-studied features of the MAX-SAT search space is plateaus. Given the landscape $L = (X, N, f)$ of an instance of MAX-SAT, the search graph of L is defined as $G_n = (X, N)$ [76]. A *plateau* P is a maximal connected subgraph of G_n such that all solutions in P have an evaluation of l [21]. We refer to l as the *level* of P, e.g., a plateau of level 5 is a maximal set of connected solutions that have 5 unsatisfied clauses. A *level-set* is all solutions in X that have the same evaluation. The lower the level of a plateau, the closer that plateau is to the global optimum in terms of evaluation.

The *size* of a plateau is the number of solutions on that plateau. An *exit* is a solution, x, on a plateau such that $y \in N(x)$ and $f(y) < f(x)$. Any plateau with at least one exit is an *open* plateau and its *escape density* is the ratio of number of exits on the plateau to its size. A *closed* plateau is one with no exits.

Hampson and Kibler investigated several characteristics of plateaus on instances of uniformly generated 3SAT with a *cv*-ratio of 4.3 [33]. They sampled the space using GSAT, an early stochastic local search algorithm for SAT [71]. From each solution, they ran a breadth-first search limited to 100,000 nodes on neighboring solutions with equal evaluation. Hampson and Kibler used these results to estimate characteristics of plateaus and found that as the level of a plateau improves, so does the size and escape density of that plateau [33].

This implies that plateaus will become more of an issue as the search finds solutions closer in evaluation to the optimal solution.

Frank et al. performed a similar study of plateaus on uniform instances of 3SAT with varying cv-ratios and $n = 100$ [21]. On each instance, they used GSAT to sample solutions with levels of 1,2,3,4 and 5. They then used a breadth-first search to exhaustively expand the plateaus containing the sampled solutions and collect statistics such as the size and escape density of the plateaus. They found that plateaus were smaller with a higher number of closed plateaus and a lower escape density on open plateaus at levels closer to the global optimum. However, Smyth attempted to recreate the experiments of Frank et al. and found that plateau sizes were much larger than those reported in the original work [76], indicating that there was a difference in their methodology and the one employed by Frank.

On small uniform instances of 3-SAT ($n \leq 45$), Smyth found by exhaustively exploring the low level space that the number of plateaus and size of plateaus decreases with the level. Furthermore, the majority of high level plateaus are open; closed plateaus occur mostly in the lowest levels of the space, i.e., close to the optimal solution. Smyth also sampled the space of larger instances ($n = 100$) and found the same trends as his results on smaller instances. These results also match those found by Hampson and Kibler. As the results of Smyth's sampling methodology coincide with both Smyth's results from exhaustive enumeration of plateaus on smaller instances and Hampon and Kibler's results, this lends credence to Smyth's results.

Smyth's results suggest that the search space becomes more rugged at levels closer to the global optima. Ruggedness refers to the variability in evaluations of neighboring solutions and can impact the reachability of solutions to local search [40, 38]. Landscapes that are more rugged are more difficult to search [40, 38]. Smyth has shown that plateaus on higher levels are larger than those on lower levels. As both the plateau size and number of plateaus decrease on levels closer to the optima, the search space will become more rugged as there will be more variability in neighboring evaluations and the reachability between solutions will decrease. Because more of these plateaus are closed, a local search algorithm will have

to take disimproving moves in order to escape. The findings of Smyth show that plateaus become more problematic to search at levels closer to optimal solutions.

Interestingly, Smyth also observed that the solutions on high levels are contained within a single large plateau, the so-called "one-big-plateau" phenomenon [76]. One implication of this is that the level-sets lower than the large plateau are connected in the sense that search need not have to travel higher in the space than the one-big-plateau in order move around in the lower level space. Thus the one-big-plateau phenomenon may enable search to more easily traverse the space by increasing the reachability of the lower levels.

3.3.2 Plateau Characteristics of Structured Instances

We will first look at several characteristics of the plateaus in our generated instances, the number of plateaus and the size of plateaus, using a methodology that follows closely to that in Smyth [76]. Our results indicate a more rugged landscape characterized by a higher number of small plateaus on modular instances with variable clause lengths. An instance with a small number of large plateaus will be less rugged as neighboring solutions on a plateau have the same evaluation. A more rugged landscape is associated with a decrease in the performance of stochastic local search [54]. We conjecture that at least part of the reason that the instances generated with the pl/pl(mod) and uni/pl(mod) generator are more difficult for AdaptG2WSAT (see Figure 3.3) is that these instances are more rugged.

We will use the same terminology as introduced in our review of previous analyses of plateaus with one exception regarding the term 'level'. As we are not filtering our instances for satisfiability, the globally optimal solutions may not have an evaluation of 0. We define the *normalized level* of a solution x as $f(x) - f(x^*)$ where x^* is a global optimum. The normalized levels of a satisfiable instance are the same as the levels referred to in previous analyses, but will be normalized on unsatisfiable instances. Note that the prior analyses filtered instances for satisfiability [76, 33, 21]. As there is no difference between the normalized level and the prior definition of level in the prior analysis on satisfiable instances, we will simply use the term level to refer to our revised definition of normalized level.

3.3.2.1 Number of Plateaus in Structured Instances

To analyze plateaus we follow a similar methodology as Smyth in that we first wish to enumerate the space. This limits the size of instances as enumeration of the entire space is only tractable for small n. We choose $n = 15$ and $m = 64$ as we found the enumeration to be tractable and the instances are large enough to notice significant differences in the search space structure. We generated 50 such instances with each of the eitght generators listed in Table 3.1, enumerated every solution and computed the normalized level of each solution. We then performed a modified breadth first search of solutions to group them into plateaus by our previous definition.

Figure 3.5 shows the mean number of plateaus as a function of normalized level. For completeness we give all data up to level 33 as this was the highest level containing solutions in any of the instances, although we will focus on lower levels due to the fact that search will likely never encounter the higher levels as even a random initial solution will start at the mean level of the entire space in expectation. Furthermore, we ignore level 0 for now as it represents a special case: the set of global optima. We will present results specifically about the global optima in Section 4.

The most striking difference in Figure 3.5 is in the modular instances, especially those with a power-law distribution of clause lengths. The fixed length modular instances have more plateaus but the rise, fall, and center of the peak corresponds more closely to the non-modular instances than those modular instances with a power-law distribution.

We hypothesize that the peaks seen in Figure 3.5 are centered around the mean evaluation of all solutions in the space. To verify this hypothesis, we computed the mean normalized evaluation of each instance. The mean normalized evaluation is the average of the normalized level of each solution in the search space for the instance. Table 3.3 lists the mean and standard deviation of the mean normalized evaluation of each instance type as well as the number of of peaks. While the peak level of plateaus and average solution seem to be quite close on most instance types, the peak number of plateaus in the two modular instances with

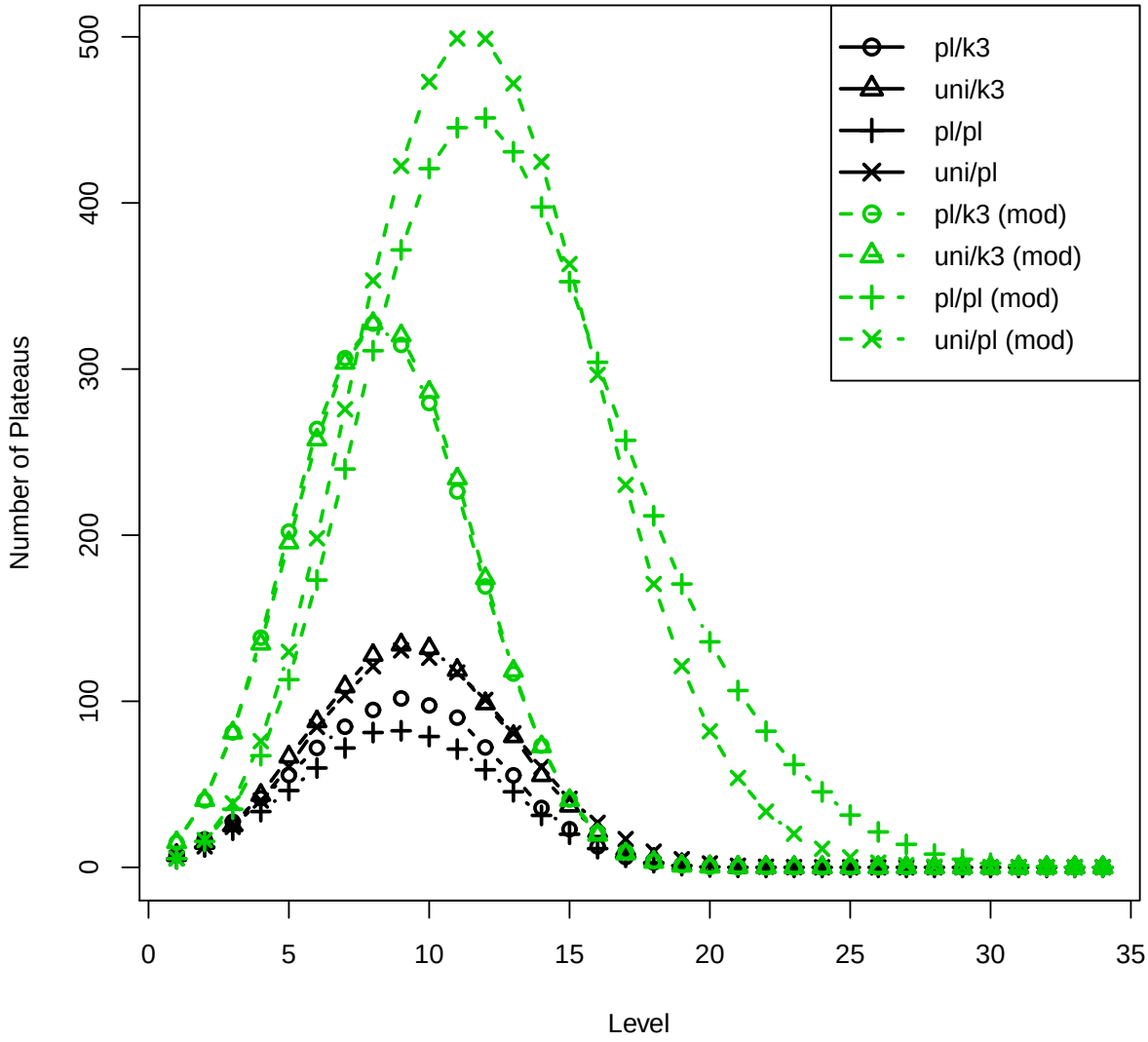


Figure 3.5: The mean number of plateaus over 50 instances with $n = 15$ and $m = 64$ grouped by generator type (See Table 3.1) for levels 1 through 33. None of the instances had solutions after level 33.

Table 3.3: The mean and standard deviation of the average evaluation of all solutions in the 50 instances constructed by each generator type (Avg. Sol.) and the mean and standard deviation of the evaluation at which the highest number of plateaus occurred across all instances of each type (Plat. Peak).

Measure	pl/k3	uni/k3	pl/pl	uni/pl
Avg. Sol.	7.42 ± 0.67	7.82 ± 0.39	7.64 ± 1.04	8.32 ± 1.16
Plat. Peak	9.04 ± 1.40	9.08 ± 1.12	8.72 ± 1.70	9.20 ± 1.28
Measure	pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
Avg. Sol.	7.52 ± 0.61	7.60 ± 0.53	11.60 ± 2.37	11.38 ± 1.81
Plat. Peak	7.98 ± 1.04	8.30 ± 0.91	11.56 ± 2.32	11.36 ± 1.90

power-law distributions match almost exactly with the average evaluation of the solutions in the space.

Figure 3.6 plots the level of the highest number of plateaus on the x-axis versus the mean normalized evaluation of solutions in the space on the y-axis for each instance. We performed a correlation test on these data across all problems which gave us a correlation coefficient of 0.80 with a p-value $< 2.2e-16$. Indeed, there is a correlation between the level at which the highest number of plateaus appears and the average of the space. Furthermore, as Table 3.3 shows, the level at which the peak number of plateaus appears tends to be very close to the average evaluation of the entire space.

Figure 3.5 clearly shows that there are differences in the number of plateaus that are dependent on the instance type. To analyze the effect of our problem types on the number of plateaus, we ran an analysis of variance test using the variable distribution (vd), clause distribution (cd) and modularity (mod) as factors. Table 3.4 lists the results of this test for the first 21 levels. There were no significant factors at $\alpha < .001$ at levels greater than 21.

We use a significance of $\alpha = .001$ to highlight the fact that the clause distribution, modularity and the interaction between modularity and clause distribution are the most significant factors. However, the variable distribution is also significant on levels greater than 7 with $\alpha < .05$. Interestingly, Figure 3.5 shows that there are *less* plateaus than those instances generated with a uniform variable distribution and fixed clause length. Under our conjecture that less plateaus indicate a less rugged space and will thus make finding optimal

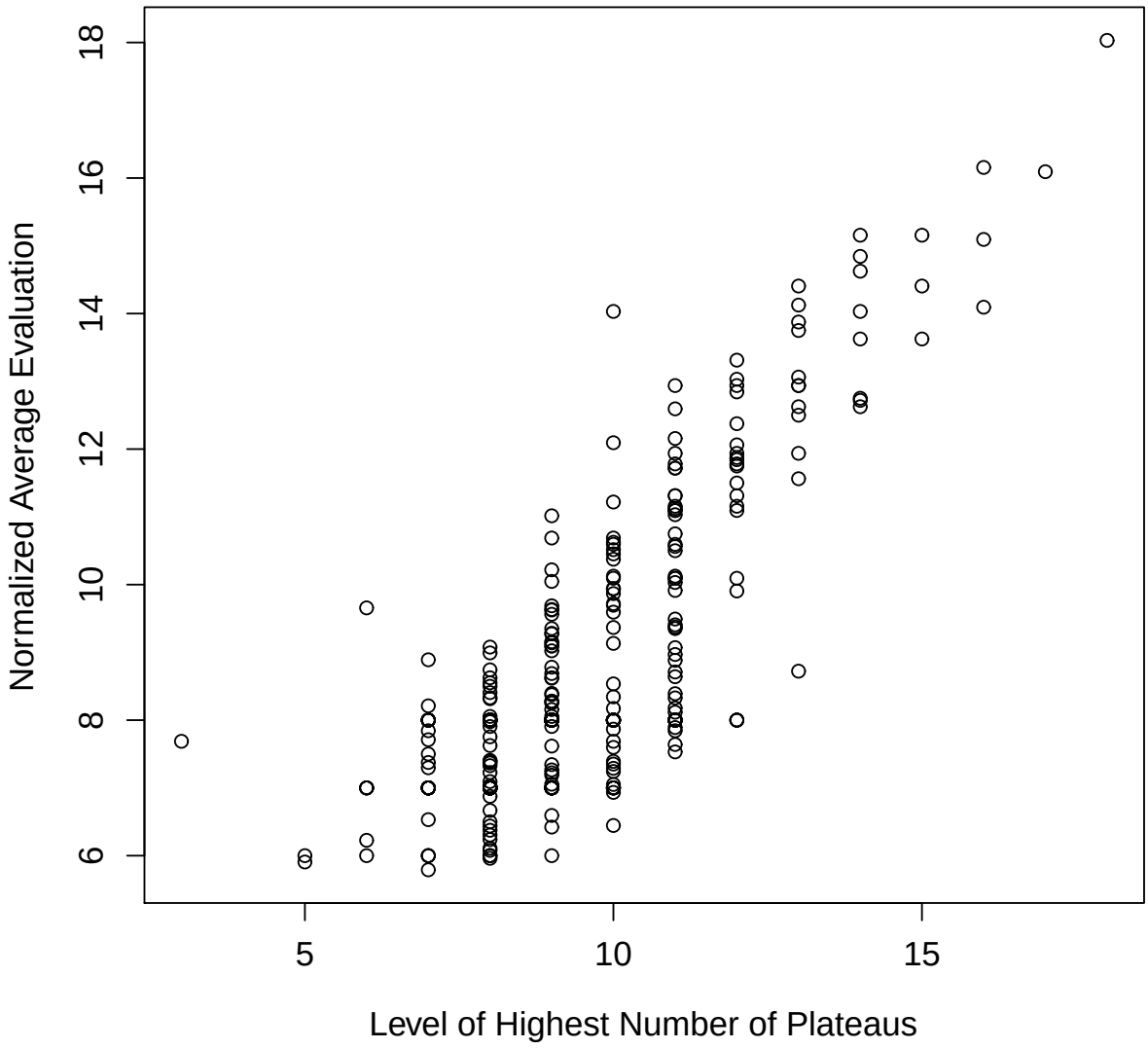


Figure 3.6: The mean normalized level of the space as a function of the normalized level containing the highest mean number of plateaus. The means were taken over 50 instances with $n = 15$ and $m = 64$ from each generator type (See Table 3.1)

Table 3.4: P-values from an analysis of variance test on the mean number of plateaus on the first 21 levels from 50 instances from each of our eight generators with an alternative hypothesis that there is a difference in means. There were three factors in the test: variable distribution (vd), taking values of power-law and uniform, and clause distribution (cd), taking values of fixed length with $k = 3$ and power-law, and modular (mod) taking values of true or false. The vd:cd, vd:mod, cd:mod, and vd:cd:mod are the p-values of interaction effects between the respective factors. Any factors that are significant at the 0.001 level are marked with a '*'.

Level	vd	cd	mod	vd:cd	vd:mod	cd:mod	vd:cd:mod
0	0.479	<.001(*)	0.001	0.860	0.316	<.001(*)	1.000
1	0.855	<.001(*)	<.001(*)	0.814	0.965	<.001(*)	0.747
2	0.872	<.001(*)	<.001(*)	0.548	0.609	<.001(*)	0.723
3	0.738	<.001(*)	<.001(*)	0.412	0.707	<.001(*)	0.967
4	0.373	<.001(*)	<.001(*)	0.259	0.817	<.001(*)	0.615
5	0.113	<.001(*)	<.001(*)	0.230	0.497	<.001(*)	0.372
6	0.051	<.001(*)	<.001(*)	0.184	0.471	<.001(*)	0.455
7	0.020	0.003	<.001(*)	0.222	0.549	0.045	0.412
8	0.012	0.812	<.001(*)	0.285	0.507	0.515	0.441
9	0.009	0.009	<.001(*)	0.243	0.637	<.001(*)	0.580
10	0.014	<.001(*)	<.001(*)	0.307	0.690	<.001(*)	0.570
11	0.024	<.001(*)	<.001(*)	0.292	0.826	<.001(*)	0.636
12	0.052	<.001(*)	<.001(*)	0.347	0.796	<.001(*)	0.663
13	0.101	<.001(*)	<.001(*)	0.407	0.797	<.001(*)	0.653
14	0.218	<.001(*)	<.001(*)	0.542	0.719	<.001(*)	0.765
15	0.443	<.001(*)	<.001(*)	0.759	0.672	<.001(*)	0.949
16	0.787	<.001(*)	<.001(*)	0.994	0.560	<.001(*)	0.826
17	0.827	<.001(*)	<.001(*)	0.731	0.440	<.001(*)	0.569
18	0.540	<.001(*)	<.001(*)	0.505	0.350	<.001(*)	0.402
19	0.367	<.001(*)	<.001(*)	0.367	0.286	<.001(*)	0.298
20	0.265	<.001(*)	<.001(*)	0.269	0.232	<.001(*)	0.229
21	0.216	<.001(*)	<.001(*)	0.215	0.200	<.001(*)	0.198

solutions harder for SLS algorithms, this would indicate that the power-law distribution actually makes instances easier. Indeed, this is the trend seen in the run-length distributions in Figure 3.3.

3.3.2.2 Size of Plateaus in Structured Instances

A second characteristic we wish to examine is the size of plateaus. As our enumerable instances all have $n = 15$ variables, the search space contains 2^{15} solutions. Given the higher number of plateaus in the modular instances, we can infer that the size of those plateaus must be smaller. To determine exactly how much smaller, we measured the size of the plateaus found in each instance by level. Figure 3.7 shows the mean size of plateaus as a function of level for each generator.

We again wish to examine the effect of the variable distribution, clause length distribution and modularity on the size of plateaus in our generated instances. Table 3.5 lists the p-values of an analysis of variance (ANOVA) test on the size of plateaus for the first 26 levels. No significant factors were found at greater levels at $\alpha < .001$. With the exception of levels 15-17, where we see a crossover between the modular and non-modular instances in Figure 3.7, the modularity is significant at all levels. In contrast to the ANOVA results on the number of plateaus, the variable distribution is a factor at lower levels while the clause distribution is not. Although this may seem strange given the data in Figure 3.7, we note that there is an interaction effect between clause distribution and modularity at these lower levels, albeit at a slightly higher α than .001.

At levels 11 and above the clause distribution becomes a significant factor and we once again see interaction affects between modularity and the clause distribution. Figure 3.7 shows that the plateaus are larger for those non-modular instances generated with a power-law variable distribution than those with a uniform variable distribution. Again, this would indicate that the power-law variable distribution generators create instances with less rugged landscapes.

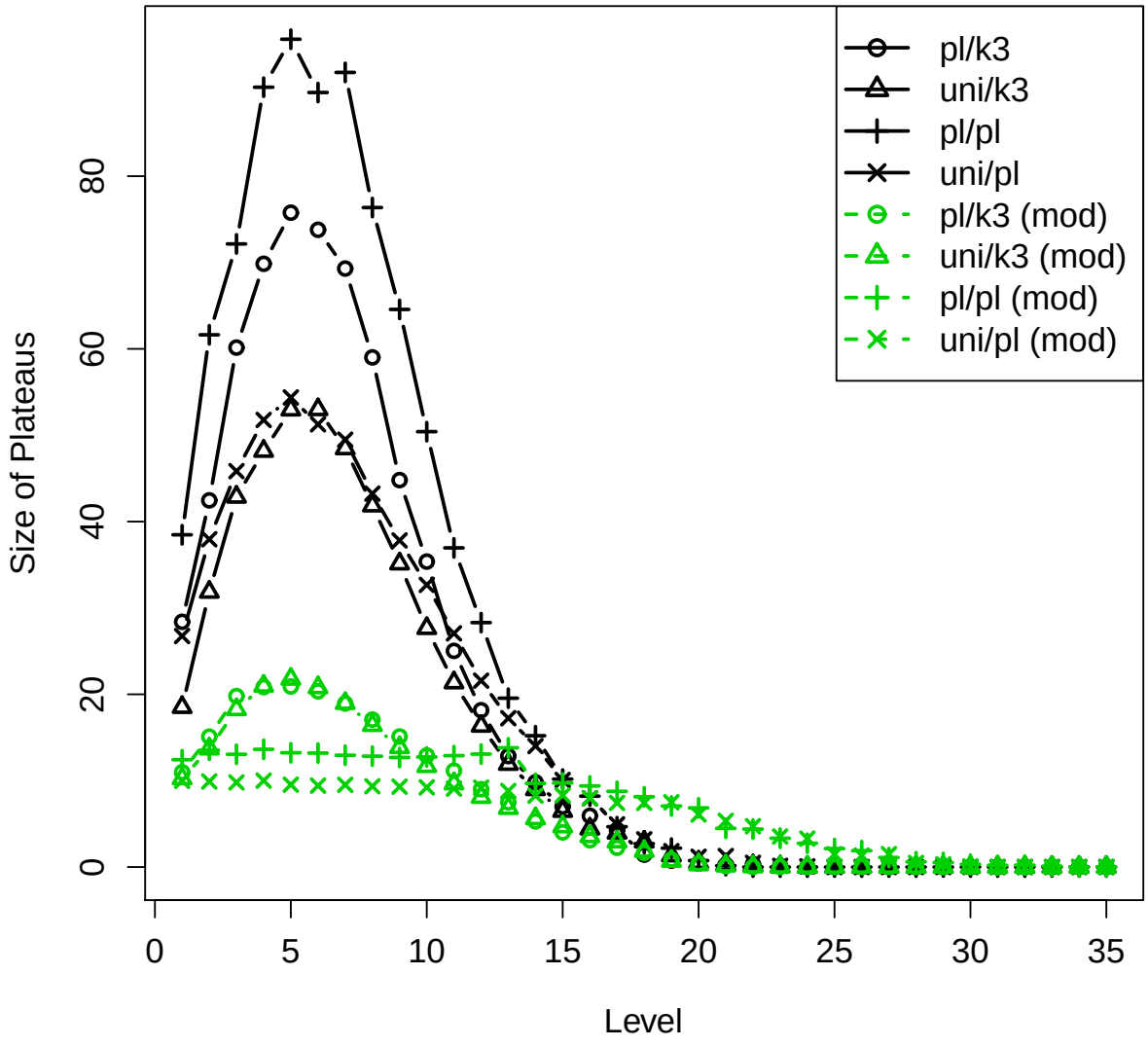


Figure 3.7: The mean plateau size over 50 instances grouped by generator type (See Table 3.1) as a function of level. Levels 1 through 33 are shown as none of the instances had solutions after level 33.

Table 3.5: P-values from an analysis of variance test on the size of plateaus in non-modular instance generators. There were tree factors in the test: variable distribution (vd), clause distribution (cd), and modularity (mod). The vd:cd, vd:mod, cd:mod and vd:cd:mod columns are the interaction between factors. Any factors that are significant at the 0.05 level are marked with a '*'.

Level	vd	cd	mod	vd:cd	vd:mod	cd:mod	vd:cd:mod
1	0.008	0.036	<.001(*)	0.703	0.048	0.065	0.989
2	0.004	0.138	<.001(*)	0.254	0.029	0.023	0.418
3	<.001(*)	0.988	<.001(*)	0.408	0.003	0.020	0.573
4	<.001(*)	0.671	<.001(*)	0.138	<.001(*)	0.002	0.346
5	<.001(*)	0.908	<.001(*)	0.093	<.001(*)	0.003	0.307
6	<.001(*)	0.690	<.001(*)	0.045	<.001(*)	0.003	0.218
7	<.001(*)	0.604	<.001(*)	0.116	<.001(*)	0.015	0.254
8	<.001(*)	0.549	<.001(*)	0.131	<.001(*)	0.016	0.292
9	<.001(*)	0.112	<.001(*)	0.049	0.001	0.003	0.125
10	<.001(*)	0.009	<.001(*)	0.065	0.002	<.001(*)	0.244
11	<.001(*)	<.001(*)	<.001(*)	0.112	0.124	0.002	0.466
12	0.004	<.001(*)	<.001(*)	0.091	0.421	0.028	0.678
13	0.069	<.001(*)	<.001(*)	0.244	0.620	0.448	0.549
14	0.311	<.001(*)	<.001(*)	0.489	0.766	0.275	0.654
15	0.588	<.001(*)	0.007	0.521	0.966	0.322	0.328
16	0.259	<.001(*)	0.295	0.800	0.752	0.044	0.196
17	0.744	<.001(*)	0.105	0.503	0.794	<.001(*)	0.219
18	0.460	<.001(*)	<.001(*)	0.345	0.216	<.001(*)	0.978
19	0.809	<.001(*)	<.001(*)	0.879	0.987	<.001(*)	0.451
20	0.888	<.001(*)	<.001(*)	0.885	0.523	<.001(*)	0.660
21	0.130	<.001(*)	<.001(*)	0.190	0.834	<.001(*)	0.897
22	0.557	<.001(*)	<.001(*)	0.438	0.976	<.001(*)	0.772
23	0.788	<.001(*)	<.001(*)	0.702	0.878	<.001(*)	0.970
24	0.527	<.001(*)	<.001(*)	0.527	0.639	<.001(*)	0.639
25	0.306	<.001(*)	<.001(*)	0.306	0.306	<.001(*)	0.306
26	0.494	<.001(*)	<.001(*)	0.494	0.494	<.001(*)	0.494

3.3.3 The One Big Plateau Hypothesis

Modular instances with a power-law distribution over clause length have more plateaus containing a smaller number of plateaus than other instances. This is indicative of a more rugged landscape. If an SLS algorithm follows the gradient to a plateau at normalized level other than 0 it must escape that plateau by taking either disimproving moves to a higher level or equal moves to other solutions on the plateau until an escape is found. The one big plateau hypothesis states that in uniform random instances, there is one large plateau that connects the lower levels of the space. This allows the search to easily move through the space without having to move to levels higher than the level containing the 'one big plateau' [76]. It is our conjecture that in the rugged landscapes of the modular instances that a single large plateau does not exist.

The one big plateau hypothesis was first put forth by Smyth [76]. Smyth enumerated all solutions on uniform random instances with $n = 15$ variables and $m = 64$ clauses and computed the same variable characteristics as we did for our generated instances. Smyth observed that in levels 3 through 5 on these instances, the majority of solutions were contained in a single, large plateau. The one big plateau hypothesis is this single large plateau connects the lower levels by enabling SLS to traverse the one big plateau in order to descend to different parts of the lower level space. Under this hypothesis, the presence of the plateau might make it easier for SLS to explore the space and find a globally optimal solution.

We conjecture that SLS will have a more difficult time finding the global optimum on instances where the large plateau occurs at higher levels or not at all. We examined each of the enumerable instances from our generators and recorded the size of the largest plateau at each level. We then computed the fraction of total solutions on each level that were contained in the level's largest plateau. Figure 3.8 shows the mean fraction of each set of instances as a function of level.

We can see from Figure 3.8 that the majority of solutions in the non-modular instances are contained in a single large plateau roughly in the level range of 3 to 10, coinciding with Smyth's findings [76]. However, there are no such plateaus on the non-modular instances.

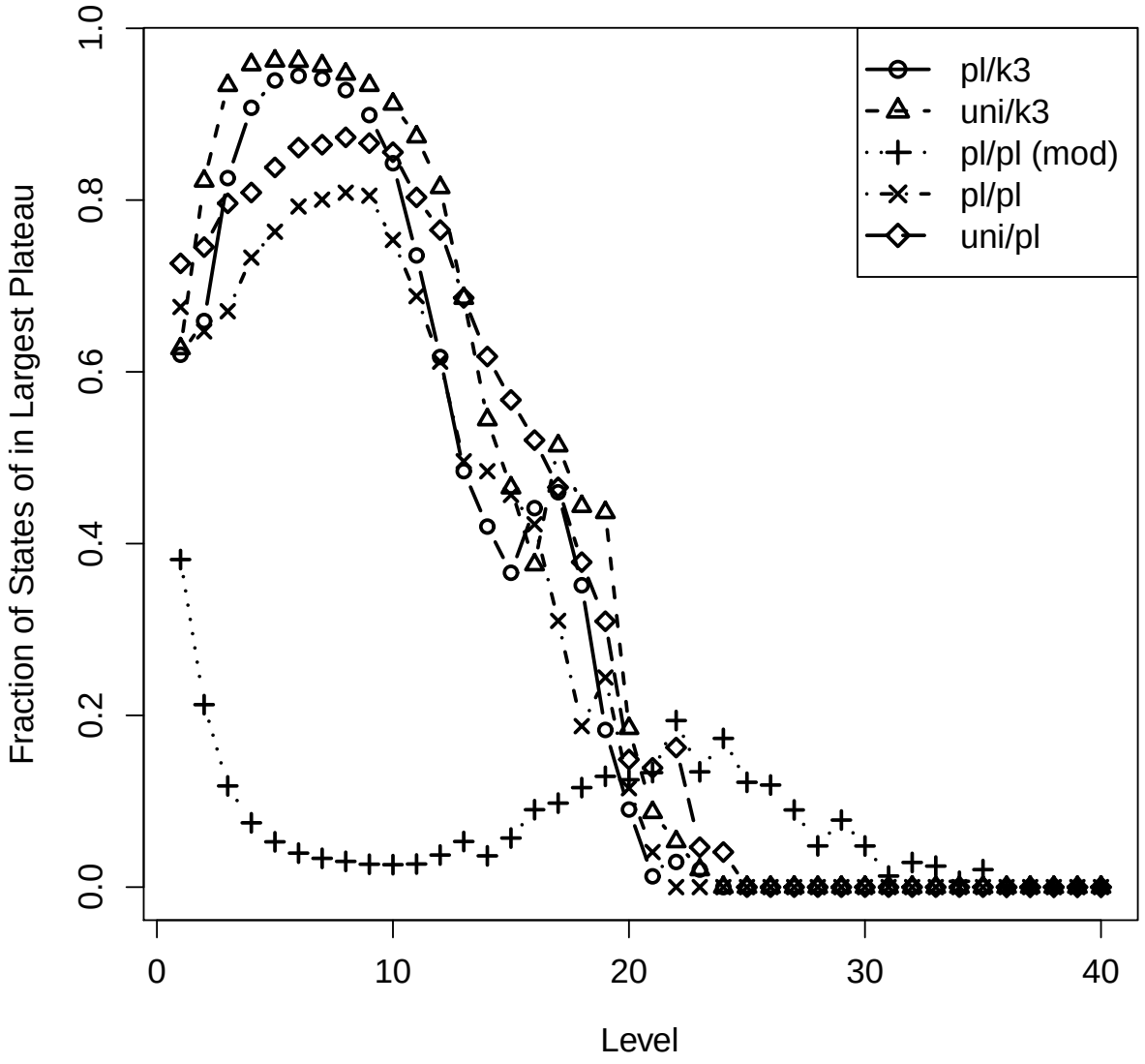


Figure 3.8: The fraction of total solutions on each level contained in the largest plateau found on that level. We observe that large plateaus contain the majority of solutions on the non-modular instances at levels close to the global optimum. There seems to be no single large plateau in modular instances. Under the one big plateau hypothesis, this would suggest modular instances are more difficult.

This would suggest that the space has less connectivity. If SLS were to follow a gradient leading to an area of the space that does not contain a global optimum, we conjecture that it would have to travel farther against the gradient to recover from this mistake due to the lack of an overarching large plateau. Verifying this conjecture, or other possible explanations, we leave for future work. We can, however, conclude that the one big plateau conjecture does not hold on our instances with a community structure.

3.3.4 Summary of Plateau Characteristics

These experiments have revealed several significant differences in the plateau characteristics in our generated instances. First, using a power-law variable distribution and clause distribution is a significant factor in the number of plateaus. However, using the power-law distribution for the variable distribution creates instances with less plateaus than a uniform variable distribution. Using a power-law distribution rather than a fixed clause length creates instances with more plateaus.

The variable distribution and clause distributions are also significant factors in the size of plateaus. Again, the effect of using a power-law distribution is split. Instances with power-law distributed variables have larger plateaus than those with uniform distributions. Instances with clause lengths following a power-law distribution have smaller plateaus than those with fixed clause lengths.

The modularity of an instance impacts both the size and number of plateaus and has interaction effects with the clause and variable distributions. Those instances created with a modular generator have a greater number of smaller plateaus than their non-modular counterparts. These plateau properties are associated with rugged landscapes that are difficult for SLS algorithms [54]. The most rugged landscapes appear to be those generated using a modular generator with a power-law distribution of clause lengths.

We found that there is no large plateau containing the majority of solutions in modular instances. We do, however, observe this phenomenon in those instances that are non-modular, regardless of the variable or clause distribution. This confirms the one big plateau hypothesis

of Smyth [76] on uniform random instances and extends it to those non-modular power-law distributions. However, we have found that the one big plateau structure is not present in modular instances, refuting the conjecture on modular instances.

The sum of our plateau analyses strongly point to the fact that modular instances are more rugged than non-modular instances, variable clause lengths are more rugged than fixed clause lengths and uniform variable distributions are more rugged than power-law distributions. Our run-length distributions on $n = 50$ and $n = 100$ instances in Figure 3.3 show that indeed AdaptG2WSAT can find the optimal solutions more easily on those landscapes which our analyses show to be less rugged. These observations suggest that the difficulty of SLS on structured instances is in part due to the ruggedness of the search space imposed by a community structure and power-law distributed clause lengths.

3.4 Characteristics of Global Optima

We will look at two characteristics of the global optima in the space: the number of global optima and the backbone. The *backbone* of an instance of MAX-SAT is the set of variables that have a consistent truth assignment across all globally optimal solutions for that instance [74]. If an instance has a single global optimum, the backbone is the set of all n variables. In the case of multiple global optima, the backbone is the set of variables that have the same truth assignment.

3.4.1 Prior Analysis of Backbones in Uniform Instances

Zhang conducted a study on the backbone size in uniform instances of MAX-3SAT [89] with $n = 25$ and the ratio of clauses to variables (cv-ratio) ranging from 1 to 20. For each instance, Zhang used a complete MAX-SAT solver to find all optimal solutions. From these, he computed the size of the backbone on each instance. On average, he found that the backbone size increases with the cv-ratio. Zhang has also shown that as the cv-ratio of uniform MAX-3SAT problems increases, so does the difficulty of the problem for both complete and incomplete solvers [87, 89], thus correlating the backbone size with the difficulty

Table 3.6: Mean and standard deviations of backbone size on 50 generated instances with $n = 15$ and $m = 64$ per problem type. See Table 3.1 for the description of our generators.

pl/k3	uni/k3	pl/pl	uni/pl
7.58±4.54	8.08±5.11	8.86±3.61	9.20±4.59
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
7.68±4.23	7.44±4.24	11.80±1.98	11.88±1.91

of the instance. However, it is unknown how the structure of a MAX-SAT instance can influence the backbone size and the implications of this influence to SLS performance.

We examined the same set of instances with $n = 15$ used in the plateau analysis. As we have enumerated the entire space of these instances, we simply gathered the solutions at the normalized level of 0 for each instance. These are the global optima. We then counted the number of solutions in each set and the backbone size for each set by determining which variables were set consistently in every solution. This gave us the number of global optima and the backbone sizes for all of our generated instances.

3.4.2 Backbone Analysis of Generated Instances

Table 3.6 lists the mean and standard deviation of the backbone sizes grouped by instance type. Two trends appear in the data. The first is that those instances with power-law distribution over clause length have a larger backbone then their fixed clause length counterparts. Second, the modular instances with a power-law distribution over clause length have a higher average backbone size than the other instances. The larger backbone sizes correspond with the increased difficulty to find the global optima found in our run-length distributions. This also matches Zhang's findings that instances with larger backbones are more difficult for SLS algorithms [87].

Table 3.7 lists the results of an analysis of variance test on the mean size of backbones in these instances. The alternative hypothesis is that there is a difference in means. As we previously have seen, the modularity and clause distribution are the most significant factors, with the interaction between the two also being significant. This verifies that the clause distribution and modularity are significant factors in backbone size and, along with

Table 3.7: P-values from an analysis of variance test on the mean size of backbones in 400 generated instances with $n = 15$ variables. There were three factors in the test: variable distribution (vd), clause distribution (cd) and modularity (mod). The vd:cd, vd:mod, cd:mod, and vd:cd:mod columns are the p-values of interaction effects between the respective factors. Any factors that are significant at the 0.001 level are marked with a '*'.

vd	cd	mod	vd:cd	vd:mod	cd:mod	vd:cd:mod
0.666	<.001(*)	<.001(*)	0.919	0.526	<.001(*)	0.761

Table 3.6, that modular instances with variable clause distribution have larger backbones than the other instance types tested.

To understand why a larger backbone is correlated with more difficult instances, recall that the backbone size is the number of variables set consistently across all global optima. Let the backbone size of an instance be b, then the number of variables that are 'free' in the sense that they can take either true or false values is $n - b$. We can now define a subset of 2^{n-b} solutions by enumerating all possible combinations of the free variables. We will refer to this subset as a *hyperplane*, a topic that will be discussed at length in the next chapter.

The hyperplane defined by the backbone as described above is guaranteed to contain all the globally optimal solutions. The hyperplane can be thought of as an upper bound on the area of the space containing the globally optimal solution. If the goal of SLS is to find a globally optimal solution, then it must find a solution within this hyperplane. As the backbone size grows, the size of this hyperplane shrinks, thus making a smaller target subspace for SLS algorithms trying to find the global optima. Additionally, as stochasticity is an inherent element of all SLS algorithms, the probability of incorrectly setting a backbone variable on average also increases as the backbone size grows. As the backbone variables are set consistently within this hyperplane, any such mistakes would drive the search away from the target area.

3.4.3 Number of Global Optima

We next wish to look at the number of global optima in our instances. Table 3.8 reports the mean and standard deviation of the number of global optima found in our instances.

Table 3.8: Means and standard deviations of the number of global optima over 50 instances for each of our generated instance types. See Table 3.1 for the description of our generators.

pl/k3	uni/k3	pl/pl	uni/pl
28.46±33.37	11.62±17.54	27.46±31.20	16.70±22.38
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
20.54±24.31	17.74±22.22	13.46±20.78	11.36±11.83

Table 3.9: P-values from an analysis of variance test on the number of global optima in 400 generated instances with $n = 15$ variables. The alternative hypothesis is that the mean number of optima are different. The three factors in the test: variable distribution (vd), clause distribution (cd), and modularity (mod). The "vd:cd", "vd:mod", "cd:mod", and "vd:cd:mod" are the p-values of interaction effects between the respective factors. Any factors that are significant at the 0.001 level are marked with a '*'.

vd	cd	mod	vd:cd	vd:mod	cd:mod	vd:cd:mod
<.001(*)	0.326	0.027	0.478	0.018	0.067	0.573

Interestingly, there are more global optima in those instances with power-law variable distributions than those with uniform distributions. This trend is consistent across the various settings of clause distribution and modularity. There do not appear to be any trends related to the modularity of the instances, e.g. the uni/k3 instance have less global optima on average than the uni/k3 (mod) instances but the uni/pl instances have more global optima than the uni/pl (mod) instances.

Table 3.9 reports the results of an analysis of variance test on the number global optima across our instance factors. This verifies our observation from Table 3.8: variable distribution is the only significant factor. Instances with a power-law distribution of variable frequency have more global optima than those with a uniform distribution as shown in Table 3.8.

Finally, we wish to examine the density of global optima contained in the hyperplane defined by the backbone. As we previously stated, we conjecture that a larger backbone creates a smaller target area for SLS algorithms that search for the globally optimal solution. We know the number of solutions contained in the hyperplane is 2^{n-b}. Given that we have the number of global optima and backbone size, we can compute the density by dividing the number of global optima by the size of the hyperplane defined by the backbone for each

Table 3.10: Mean and standard deviations of backbone density on our generated instances. See Table 3.1 for the description of our generators.

pl/k3	uni/k3	pl/pl	uni/pl
0.37±0.40	0.36±0.43	0.46±0.38	0.42±0.40
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
0.28±0.37	0.25±0.33	0.80±0.25	0.85±0.24

instance. Table 3.10 reports the means and standard deviations of the backbone density for the 50 instances of each type.

Interestingly the most difficult instances, those with a community structure and variable clause length have the highest density of solutions. Over 80%, on average, of the solutions in the hyperplane defined by the backbone set are globally optimal. This would suggest that if SLS were to find the hyperplane defined by the backbone, it would not be difficult to find the globally optimal solution. Indeed, a large portion of the next chapter is devoted to a method that we developed of estimating the backbone and using this information to initialize the search.

3.4.4 Summary of Global Optima Characteristics

We have found that clause distribution and modularity are significant factors in the size of backbones. Those instances with a community structure and variable clause length have a larger backbone than non-modular instances with fixed clause lengths. These instances are also associated with a higher difficulty of finding the global optima (see Figure 3.3). This agrees with the results of Zhang's study showing that backbone size and difficulty of instances are correlated.

We show that variable distribution is the only significant factor in the number of global optima. There are more global optima in those instances with a power-law distribution than those with a uniform distribution. The run-length distributions in Figure 3.3 indicate that these instances are slightly easier than their uniform distribution counterparts when fixing the other factors. This is not surprising as we expect SLS algorithms will have an easier time finding the global optima as the number of them increases.

Finally, we see that the density of global optima contained in the hyperplane defined by the backbone is highest on those problems with the largest backbones, namely the instances with community structure and power-law distribution of clause lengths. This suggests that finding this hyperplane would greatly increase the success of SLS algorithms in finding the global optima on those instances. This result in part motivates one of our improvements discussed in the next chapter, a method of estimating the backbone using hyperplane averages.

3.5 Backbones and Plateaus

We have seen that the number and size of plateaus indicate a more rugged space in modular instances. Furthermore, the backbone size in these same instances are larger on average. The ruggedness of the space and backbone size are both related to the difficulty of instances for SLS algorithms. It is a natural next step to determine if there is any correlation between these features.

To determine if there is any correlation between the plateau size and the backbone size, we first grouped the instances by the generator that constructed them. This gave us eight groups of 50 instances. For each group we then found the correlation coefficient using Pearson's method between the average plateau size at each level and the backbone size for each instance. Although we looked at all levels, we found significant correlations on levels 1 through 13. We therefore only report these levels. Table 3.11 lists the correlation coefficient and the p-value for each level.

Interestingly, we see in Table 3.11 that the only correlations significant at the .001 level, with the exception of the uni/pl instances at level 1, are the modular instances with variable clause distribution. There is an inverse correlation, indicating that the average plateau sizes are smaller on those instances with larger backbones. We have discussed how smaller plateaus indicate a more rugged space and thus more difficult instances. Similarly, larger backbones are also correlated to instance difficulty. Thus as one measure of hardness increases, so does the other.

Table 3.11: Correlation coefficient (left of comma) and p-value (right of comma) found by Pearson's method testing the correlation of mean plateau size at each level and backbone size for the 50 instances from each generator. We report the first 13 levels, after this no correlation was significant at the .001 level. Significant correlations are marked with (*).

	pl/k3	uni/k3	pl/pl	uni/pl
1	-0.15,0.310	-0.30,0.034	-0.43,0.002	-0.48,< .001(*)
2	-0.29,0.043	-0.23,0.104	-0.18,0.214	-0.38,0.006
3	-0.21,0.143	-0.15,0.310	-0.26,0.064	-0.34,0.015
4	-0.06,0.664	0.13,0.352	-0.27,0.058	-0.16,0.278
5	-0.04,0.781	0.26,0.066	-0.27,0.055	-0.12,0.426
6	0.11,0.440	0.31,0.029	-0.08,0.577	-0.10,0.479
7	0.20,0.158	0.39,0.005	-0.05,0.745	0.03,0.861
8	0.20,0.172	0.38,0.007	-0.06,0.690	0.06,0.657
9	0.25,0.085	0.36,0.010	-0.02,0.870	0.12,0.404
10	0.24,0.094	0.35,0.013	0.05,0.754	0.17,0.242
11	0.32,0.025	0.36,0.010	0.11,0.436	0.28,0.052
12	0.38,0.007	0.35,0.013	0.13,0.371	0.31,0.027
13	0.45,0.001	0.24,0.100	0.22,0.122	0.30,0.032

	pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
1	-0.15,0.304	-0.18,0.200	-0.70,< .001(*)	-0.68,< .001(*)
2	0.12,0.404	-0.11,0.435	-0.58,< .001(*)	-0.72,< .001(*)
3	0.20,0.166	0.01,0.922	-0.57,< .001(*)	-0.68,< .001(*)
4	0.22,0.122	0.09,0.556	-0.53,< .001(*)	-0.64,< .001(*)
5	0.17,0.243	0.11,0.467	-0.54,< .001(*)	-0.66,< .001(*)
6	0.17,0.237	0.13,0.385	-0.53,< .001(*)	-0.65,< .001(*)
7	0.13,0.376	0.19,0.188	-0.54,< .001(*)	-0.60,< .001(*)
8	0.05,0.730	0.22,0.117	-0.54,< .001(*)	-0.59,< .001(*)
9	-0.00,0.977	0.26,0.069	-0.53,< .001(*)	-0.56,< .001(*)
10	-0.09,0.545	0.27,0.054	-0.51,< .001(*)	-0.53,< .001(*)
11	-0.18,0.223	0.32,0.025	-0.48,< .001(*)	-0.52,< .001(*)
12	-0.18,0.200	0.28,0.049	-0.47,< .001(*)	-0.50,< .001(*)
13	-0.13,0.381	0.16,0.262	-0.42,0.002	-0.47,< .001(*)

It is also interesting to note that this inverse correlation is only significant on those modular instances with a power-law distribution of clause lengths. These results again highlight the fact that there are significant differences in problems with these characteristics than the other problem types.

We performed the same analysis on the average number of plateaus, again for each group of 50 instances and for each level. In this case, we found significant correlations at levels 1 through 17. These are reported in Table 3.12.

Although there are a number of significant inverse correlations at the lower levels, we once again see that the modular instances with variable clause distribution separate from the rest of instances. At levels 12 through 17 and higher, both instance groups have significant positive correlations between backbone size and number of plateaus, with the correlation in the pl/pl (mod) instances starting at level 8. Again, similar to the correlation with plateau size, a large number of plateaus is indicative of a more rugged space. Those instances with larger backbones tend to have more plateaus at these levels.

3.5.1 Summary of Correlation between Backbone and Plateaus

We have examined three characteristics related to problem difficulty: the backbone size, the number of plateaus and the size of plateaus. At certain levels in modular instances with variable distribution there is an inverse correlation between size of plateaus and backbone size. Likewise, there is positive correlation between number of plateaus and backbone size. Thus as a backbone size grows, it would seem that the plateaus in these instance are also more problematic.

We conjecture that this is again due to the ruggedness of the landscape. As the ruggedness grows, the size of plateaus gets lower and the number of them increases. This reduces the connectedness of the landscape in the sense that to move from one plateau to another on the same level, the search must also travel to at least one additional level. While this is true for any landscape, the more rugged the space, the more often this happens.

Table 3.12: Correlation coefficient(left of comma) and p-value (right of comma) found by Pearson's method testing the correlation of mean number of plateaus at each level and backbone size for the 50 instances of each generator. We report the first 17 levels, after this no correlation was significant at the .001 level. Significant correlations are marked with (*).

Level	pl/k3	uni/k3	pl/pl	uni/pl
1	-0.67,< .001(*)	-0.42,0.002	-0.44,0.001	-0.46,< .001(*)
2	-0.42,0.003	-0.36,0.011	-0.33,0.020	-0.39,0.005
3	-0.39,0.005	-0.53,< .001(*)	-0.28,0.048	-0.33,0.019
4	-0.43,0.002	-0.59,< .001(*)	-0.13,0.367	-0.37,0.008
5	-0.39,0.005	-0.47,< .001(*)	-0.03,0.823	-0.28,0.051
6	-0.36,0.011	-0.40,0.004	0.01,0.954	-0.17,0.250
7	-0.28,0.047	-0.41,0.003	0.06,0.691	-0.10,0.491
8	-0.12,0.402	-0.31,0.031	0.13,0.383	0.03,0.844
9	-0.01,0.943	-0.26,0.072	0.18,0.214	0.12,0.408
10	0.07,0.626	-0.19,0.197	0.21,0.136	0.19,0.176
11	0.13,0.373	-0.16,0.279	0.24,0.096	0.22,0.127
12	0.25,0.086	-0.15,0.315	0.23,0.109	0.30,0.037
13	0.25,0.077	-0.07,0.609	0.24,0.099	0.34,0.014
14	0.30,0.037	-0.04,0.780	0.24,0.090	0.38,0.007
15	0.23,0.113	-0.03,0.855	0.21,0.144	0.39,0.005
16	0.17,0.239	0.00,0.977	0.18,0.219	0.40,0.004
17	0.09,0.536	0.05,0.710	0.12,0.413	0.42,0.003

Level	pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
1	-0.70,< .001(*)	-0.62,< .001(*)	-0.21,0.151	-0.35,0.012
2	-0.69,< .001(*)	-0.52,< .001(*)	-0.07,0.624	-0.14,0.321
3	-0.64,< .001(*)	-0.42,0.002	0.07,0.613	-0.01,0.961
4	-0.53,< .001(*)	-0.36,0.011	0.21,0.135	0.09,0.519
5	-0.39,0.005	-0.30,0.032	0.28,0.046	0.15,0.313
6	-0.26,0.071	-0.26,0.071	0.37,0.009	0.21,0.149
7	-0.14,0.345	-0.22,0.133	0.41,0.003	0.25,0.075
8	-0.02,0.885	-0.16,0.253	0.46,< .001(*)	0.31,0.030
9	0.04,0.791	-0.09,0.514	0.49,< .001(*)	0.36,0.011
10	0.09,0.520	-0.03,0.845	0.54,< .001(*)	0.40,0.004
11	0.12,0.410	0.02,0.898	0.57,< .001(*)	0.43,0.002
12	0.13,0.386	0.06,0.654	0.60,< .001(*)	0.46,< .001(*)
13	0.10,0.483	0.14,0.328	0.61,< .001(*)	0.49,< .001(*)
14	0.08,0.590	0.16,0.260	0.59,< .001(*)	0.50,< .001(*)
15	0.04,0.773	0.16,0.273	0.56,< .001(*)	0.51,< .001(*)
16	0.00,0.983	0.19,0.197	0.52,< .001(*)	0.50,< .001(*)
17	-0.03,0.840	0.13,0.384	0.47,< .001(*)	0.47,< .001(*)

3.6 Generalizing to Industrial Instances

We cannot perform a similar analysis on industrial instances as it would require enumerating the space. At the very least, we would need to be able to generate all globally optimal solutions for backbone analysis and sample close to the local optima for a plateau analysis. This is simply not tractable for industrial instances given their size and the difficulty SLS algorithms have in getting even relatively close to the global optima [47]. In lieu of applying our analysis to these instances, we will show that industrial instances have the same characteristics studied in some of our generated instances.

We took all the industrial instances from the 2011 SAT Competition and the industrial instances from the 2012 MAX-SAT competition for a total of 322 industrial instances. To examine the variable distribution of these instances we counted the number of times each variable appeared in a clause. We then used the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [26] for the maximum likelihood estimation of β in the power-law distribution to fit the distribution to the sample [2, 57, 15].

To determine the goodness of fit between the distribution and our sample, we used a nonparametric test known as the Kolmogorov-Smirnov test [51]. This gave us a p-value for the goodness of fit of the variable frequencies of each instance to a power-law distribution fit to each instance using maximum likelihood estimation.

We then examined these p-values and found all but 4 instances had a p-value $< .05$. These four instances were homer14.shuffled, homer.16.shuffled, homer17.shuffled and bart17.shuffled. Each variable in these instances appeared exactly the same number of times, thus had a perfect uniform distribution. No information could be found about the source of these instances and we consider them anomalies in our set of industrial instances.

We repeated the same experiment but this time counted the clause lengths of each instance. We applied the same estimation to fit a power-law distribution and the Kolmogorov-Smirnov test to provide a goodness of fit statistic. We again found that the majority of instances had a p-value $< .05$ except for the four previously mentioned instances and 6

more: c10idw-i, manol-pipe-c10nidw, manol-pipe-c6bidw-i, manol-pipe-f7nidw, post-c32s-gcdm16-23, and vda-gr-rcs-w9.shuffled.

Although a power-law distribution could not be fit to the clause lengths of these instances, the clause lengths were not a fixed length. Figure 3.9 shows the histogram of clause lengths for the port-c32s-gcdm16-23 instance. The other instances had very similar distributions of clauses with lengths 2 and 3.

This raises the question: Must the clause distribution be a power-law distribution or s any non-fixed clause length distribution is sufficient to create difficult instances? We leave this question open for future work, as our analysis has shown that the majority of industrial instances do have a power-law distribution over clause length.

3.7 Summary

We have developed a new instance generator that is capable of producing instances with a community structure. Using the run length distribution method of Hoos et al. to examine the performance of SLS algorithms, we found that AdaptG2WSAT has more difficulty finding the global optima on those problems with a power-law distribution of clause length. We have shown that these instances have characteristics that are common to industrial instances and we will refer to them as industrial-like.

We analyzed the search space on instances created with our generators and found that industrial-like instances have a larger number of small plateaus than the other instances. This indicates a more rugged landscape. Furthermore, in the industrial-like instances there was no one big plateau, a feature found in other instances that is conjectured to enable the search to easily traverse the space [76].

Our analysis on backbones revealed a similar pattern. The size of backbones has been correlated with problem difficulty [89]. Industrial-like instances had a larger backbone, thus decreasing the upper bound of the area of the space that contains the globally optimal solution. However, we also found that in industrial-like instances the density of solutions contained in this area is greater. Finally, we found correlations on industrial-like instances

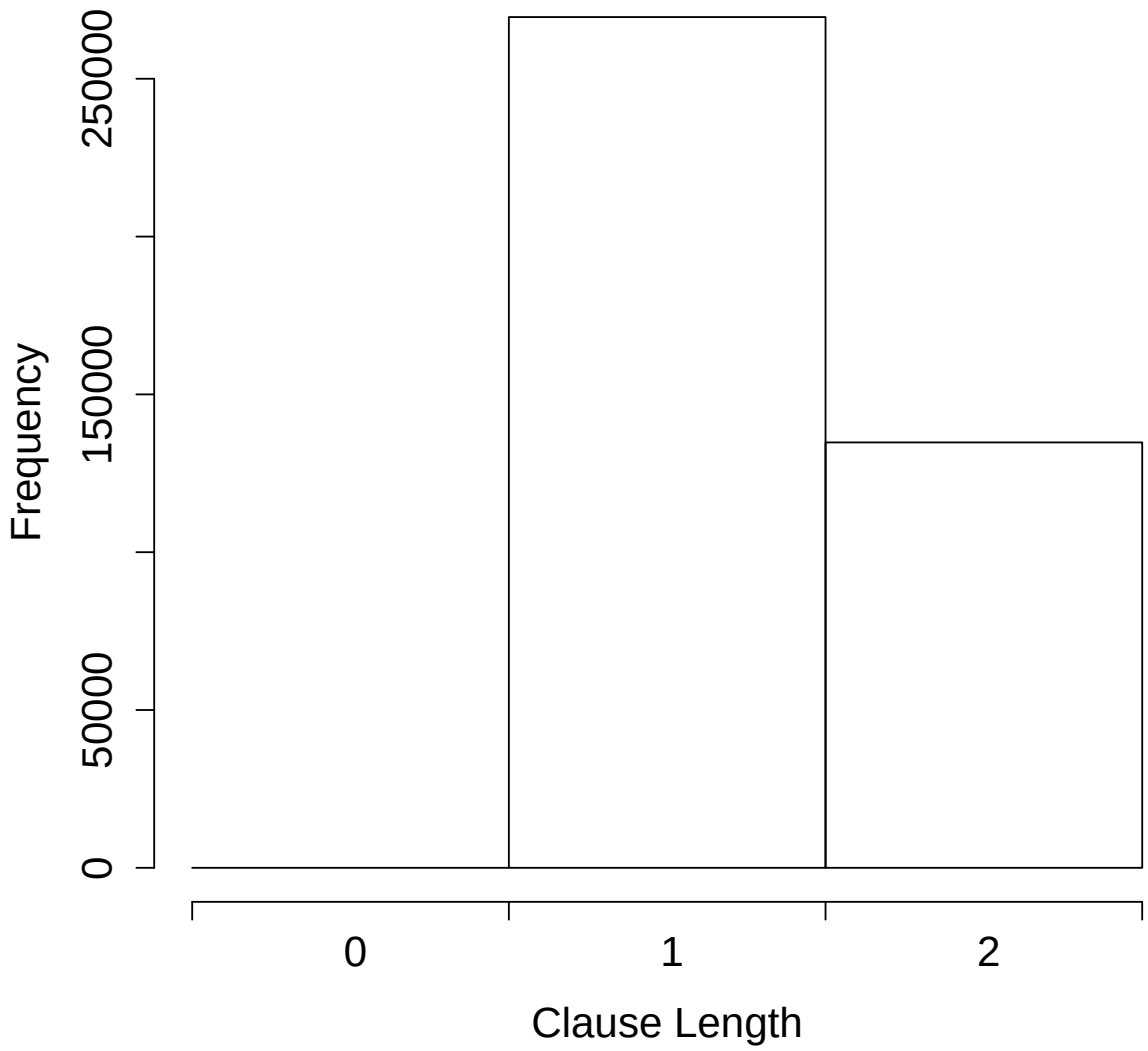


Figure 3.9: A histogram depicting the clause length frequencies of the post-c32s-gcdm16-23 instance from the 2011 SAT Competition. This instance is one of the 6 instances that did not have a power-law distribution over clause length. The other 6 instances had a similar distribution.

Table 3.13: Summary of the three characteristics of industrial instances and their effect on the search space in comparison to uniform instances with fixed clause length.

Instance Type	$P_{success}$	Number of Plateaus	Size of Plateaus	Backbone Size
uni/k3	0.95	28.74	11.85	8.08
pl/k3	0.99	22.00	16.18	7.58
uni/pl	0.95	27.63	13.47	9.20
pl/pl	0.97	18.13	21.48	8.86
uni/k3 (mod)	1.00	64.31	5.43	7.44
pl/k3 (mod)	1.00	64.24	5.45	7.68
uni/pl (mod)	0.16	128.72	5.1	11.80
pl/pl (mod)	0.23	129.02	6.45	11.88

between the backbone size and plateau characteristics. This suggests that a method of estimating the correct setting to the backbone variables could greatly increase the ability of an SLS algorithm to find a globally optimal solution.

Table 3.13 summarizes our results on the three characteristics of industrial instances and their effect on the search space in comparison to uniform instances with $k = 3$ fixed clause lengths. A green (or light colored cell) indicates a significant difference that was less than the measured property in uniform instances with fixed clause length, while a red (or darker colored cell) indicates a significant difference that was greater.

This table summarizes our results and indicates that our plateau characteristics may not have as much of an effect on problem difficulty as the backbone size. We believe that these results indicate the importance of what we call *critical variables*. We define a critical variable as one that, if set incorrectly, can cause the search to set other variables incorrectly and have a low probability of fixing the mistake. We define an incorrect setting as a truth assignment to a critical variable that leaves some clauses unsatisfied that are satisfied by the critical variable in globally optimal solutions. Backbone variables have a high probability of being a critical variable, but a critical variable is not necessarily a backbone variable.

We believe that setting critical variables incorrectly has the potential to cause a cascade effect in SLS algorithms. If a critical variable is set incorrectly, it will force other variables to be set incorrectly as the search attempts to satisfy clauses that are left unsatisfied as a

result. This in turn will cause other variables to be set incorrectly as the search attempts to satisfy clauses. As the cascade effect propagates, the probability that the search will fix the original error becomes smaller and smaller.

We hypothesize that this effect is more pronounced in those instances with community structure due to the family structure and connector variables. If a critical variable in family 'A' is incorrectly set, it may force the connector variable to take on an incorrect truth assignment. This in turn can cause a cascade effect in the families connected to 'A' by this connector variable and further propagate to even more families. While we leave testing this hypothesis for future work, we will see further evidence of critical variables influencing the behavior of search in the following chapter.

Chapter 4

Improving SLS for MAXSAT

We now wish to use the insights gained from our analysis to motivate improvements to state-of-the-art SLS algorithms on industrial instances. Our analysis revealed that the average backbone size was largest on the industrial-like instances and that this factor seemed to be the storngest indicator of difficult instances. Therefore, we conjecture that a good starting place to improve SLS is to develop a method of estimating the correct setting of backbone variables.

We will first show that estimating the backbone variables does in fact improve the performance of AdaptG2WSAT. We do so by first finding the backbone set for generated instances of MAX-SAT. By setting different percentages of the backbone variables correctly at the beginning of search, we observe that the more backbone variables set correctly, the better AdaptG2SWAT can find a globally optimal solution.

Of course, in order to determine the backbone set of an instance we need to know all the globally optimal solutions to that instance. Doing so would eliminate the need to run search. Therefore, we have developed a method of estimating the backbone information using the average evaluations of solutions contained within hyperplanes, subsets of solutions in the search space. Not only can we compute these averages in a tractable manner directly from an instance of MAX-SAT, but we can use these averages to provide a surprisingly good estimate of the correct setting of the backbone variables. We refer to this method as *hyperplane initialization* and it was first published by Hains et al. [31].

Finally, we will discuss the trade-offs between two types of local search: first improving and best improving, which are used in many SLS algorithms [1]. Best improving search is a strictly greedy search that will always take the best improving solution in the Hamming distance 1 neighborhood of the candidate solution. In contrast, first improving search will select an arbitrary improving move.

There is a trade-off between best improving and first improving search in terms of the quality of local optima found and the effort, as measured in CPU time, to find them. However, SLS algorithms do not stop at the first local optimum they find. By examining the behavior of AdaptG2WSAT after the first local optimum is found, we observe that the quality of local optimum has little to no influence over the quality of solution found at the end of the search. Indeed, in several cases the average evaluation of solutions is significantly better when using first improving search.

Our search space analysis in Chapter 3 revealed that the lower level search space in industrial-like instances contains a large number of small plateaus, indicating a rugged search space that is difficult for SLS algorithms to navigate. If search can reach this difficult part of the space faster, it can spend a higher percentage of its overall run time exploring the rugged areas of the space. By replacing the slow, best improving search in AdaptG2WSAT with fast first improving search, we can improve the results by allowing the search to spend more time in the rugged part of the space with little to no negative consequences.

4.1 Initializing Search with Backbone Information

Before discussing methods of estimating backbones, we will first establish that it is a good idea to do so. We generated 50 problems from each of our eight generators with $n = 50$ variables and $m = 214$ clauses (see Table 3.1 for a list of the characteristics of problems constructed from each generator). Our choice of 50 variables is due to that fact that we need to find the backbone variables for each instance. In the previous chapter, we used instances with $n = 15$ variables as this was small enough to allow us to enumerate the entire search space. In this case we do not need to enumerate the space, but we do need to find all the globally optimal solutions. We chose to use $n = 50$ variables as this is the largest instance size that allows us to tractably find all the optimal solutions.

To find the optimal solutions, we used a complete MAX-SAT solver to find a single globally optimal solution, s^*. Let the evaluation of s^* be $f(s^*)$. We then construct a new clause which is the exact negation of the global optimum returned by the complete solver.

Table 4.1: Means and standard deviations of backbone size on 50 generated instances with $n = 50$ and $m = 214$ per problem type. See Table 3.1 for the description of our generators.

pl/k3	uni/k3	pl/pl	uni/pl
11.90±6.48	12.65±8.51	14.28±4.66	17.35±5.78
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
13.00±5.13	12.35±4.72	19.10±3.46	18.70±3.36

This new clause was added to the original formula. Because we evaluate solutions by the number of unsatisfied clauses and every solution but s^* will satisfy the new clause, the evaluation of all solutions but s^* will remain the same. However, because of the new clause, s^* will have an evaluation of $f(s^*)+1$ on the modified instance. Therefore, it will no longer be globally optimal on the modified instance if there are any other solutions with an evaluation of $f(s^*)$. By repeating this process until no more solutions are found with an evaluation of $f(s^*)$, we are able to find all the globally optimal solutions without enumerating the space.

Once the optimal solutions were found for each instance, we then determined the backbone set of each instance by finding which variables were set consistently across all global optima. Therefore, we have the backbone variables and their corresponding truth assignments for all 400 instances. The means and standard deviations of the number of variables in each backbone set per instance type are reported in Table 4.1. The same trend in Table 3.6 from our analysis on smaller instances can be seen in these data: the most industrial-like instances have the largest backbone size.

Our hypothesis is that if we use backbone information to initialize an SLS algorithm, that algorithm will be able to find a global optimum easier than when initialized with a random solution. To test this hypothesis, we constructed five sets of 50 solutions for each instance, yielding us 250 solutions total per instance. Each set of solutions has a different percentage of backbone variables assigned correctly. By correct assignment, we mean that the variable has the same truth assignment as it does in all of the globally optimal solutions. Set 1 has 0% of the backbone variables assigned correctly, set 2 has 25% assigned correctly, set 3 had 50% assigned correctly (as in uniform random initializaiton), set 4 has 75% assigned correctly and set 5 had 100% of the backbone variables correctly assigned. Non-backbone

variables were set to true or false with .5 probability. We rounded up in the case that the number of variables in a set was an odd number.

For example, if the backbone variables were $\{x_2, x_7, x_{10}, x_{12}, x_{26}, x_{28}, x_{39}, x_{42}\}$ and the truth assignments of these variables were $1, 0, 0, 1, 0, 0, 1, 1$ respectively, then a solution in set 1 would be constructed by assigning the backbone variables as $x_2 = 0, x_7 = 1, x_{10} = 1, x_{12} = 0, x_{26} = 1, x_{28} = 1, x_{39} = 0, x_{42} = 1$ and setting the remaining variables to true or false with equal probability.

In sets 2, 3 and 4, a percentage of backbone variables are set correctly while the remaining are set incorrectly. In these cases, the backbone variables that were assigned correctly were chosen uniformly at random from the backbone set. Using the previous example, if we wanted to generate a solution for set 2 in which 25% of the backbone variables are set correctly, we would first choose two backbone variables at random, e.g., x_2 and x_{26}. We would then construct a solution where $x_2 = 1, x_7 = 1, x_{10} = 1, x_{12} = 0, x_{26} = 0, x_{28} = 1, x_{39} = 0, x_{42} = 1$ and the remaining variables would be set at random. In Set 5, the solution is constructed so that all backbone variables are correctly assigned.

We then initialized AdaptG2WSAT from the constructed solutions and allowed the algorithm to run until it made 10,000 bit flips. Thus we had 50 runs per set of constructed solutions for a total of 250 runs per instance. Note that we did not modify the algorithm in any way, we simply used our constructed solutions as the initial candidate solution instead of a randomly generated solution as is the default procedure. We then computed the run-length distributions (RLD) as described in Chapter 2, Section 1 over each set of runs for each of the generator types. These RLDs are shown in Figure 4.1.

As the percentage of correctly assigned backbone variables increases, so does the growth rate of the estimated probability of finding a global optimum. Although this phenomenon can be seen in all our generator types, it is most pronounced in the modular instances with a variable clause length, i.e., uni/pl (mod) and pl/pl (mod). These are the most difficult problems for SLS algorithms as found by our analysis in Chapter 3 and are also the most

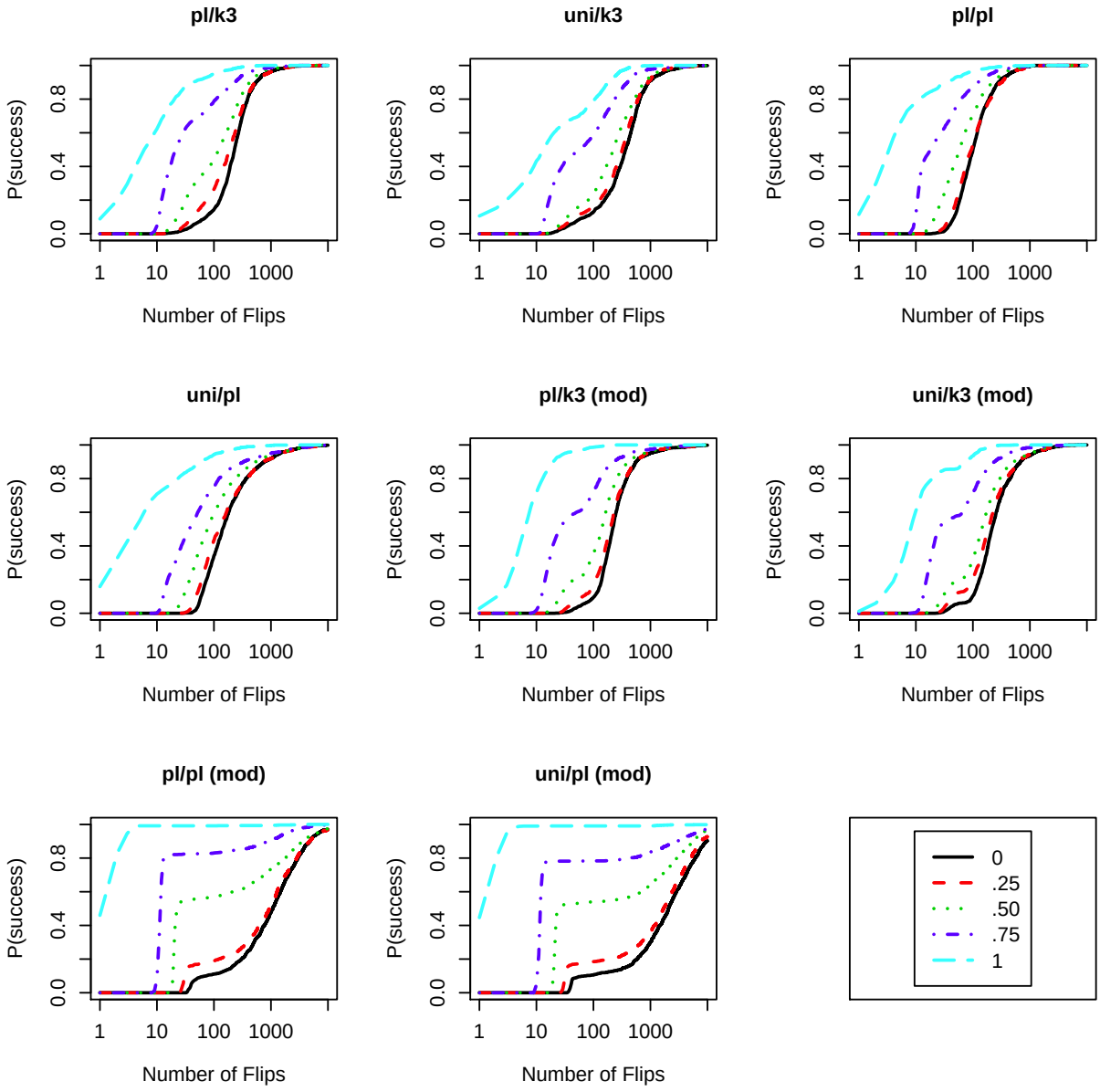


Figure 4.1: The run length distributions of five sets of runs on 50 instances of each generator type (See Table 3.1 for our generator types). Each set consisted of 50 runs initialized by setting the 0, 25%, 50%, 75% and 100% of the backbone variables to their correct settings. The RLD for each set is denoted by a different line style as shown in the bottom right corner.

industrial-like out of our 8 problem types. We believe this is due in part to the higher density of global optima in the hyperplane defined by the backbone as discussed in Chapter 3.

In the third set of solutions, 50% of the backbone variables were set incorrectly and the other 50% were set correctly. This is what we would expect in random solutions. If the initialization of backbone variables can influence the search, we expect to see the number of flips to find a global optimum in sets 1 and 2 to be lower than set 3 because there are more backbone variables set incorrectly. Likewise, in sets 4 and 5 where we set 75% and 100% of the backbone variables correctly we expect to see less flips required to find a global optimum than set 3.

While we see evidence of this in Figure 4.1, we want to determine if there is indeed a significant difference in the mean number of flips required to find a global optimum in our experimental runs. We have 50 problems from each of the eight generators, and five sets of 50 runs each. In each of these runs we counted the number of bit flips required to find a globally optimal solution. If a run did not find a globally optimal solution, we discarded these runs.

We then carried out one-sided t-tests on the data. For each generator type, we used an alternative hypothesis that the mean number of flips to find a global optimum was greater in the 0 and 0.1 run sets than in the 0.5 run set, and that the number of flips in the 0.75 and 1 run sets were less than in the 0.5 set. The p-values of our t-tests are reported in Table 4.2. In all the t-tests, we can reject the null hypothesis at the .001 level for all the modular instances with variable clause lengths. All p-values except the 0.75 set and the 0.5 set of uni/pl instances were significant at the .05 level.

Correctly assigning the majority of backbone variables at the beginning of search can improve the ability of AdaptG2WSAT to find a globally optimal solution. Likewise, incorrectly setting the majority of backbone variables can adversely affect the ability of the search to find a global optimum. This effect is more pronounced on those instances that have a community structure.

Table 4.2: P-values from t-tests comparing the number of flips to a global optimum (f) for each of the 50 runs over 50 instances of each type. We use notation $f(x)$ where x is one of our run sets where 0 is the set of runs with no backbone variables initialized correctly, .25 is 25% are set correctly, etc.

Type	$f(0) > f(.5)$	$f(.25) > f(.5)$	$f(.75) < f(.5)$	$f(1) < f(.5)$
pl/k3	<.001(*)	<.001(*)	<.001(*)	<.001(*)
uni/k3	<.001(*)	0.026	<.001(*)	<.001(*)
pl/pl	<.001(*)	<.001(*)	<.001(*)	<.001(*)
uni/pl	0.010	0.005	0.053	<.001(*)
pl/k3 (mod)	<.001(*)	<.001(*)	<.001(*)	<.001(*)
uni/k3 (mod)	<.001(*)	0.003	<.001(*)	<.001(*)
pl/pl (mod)	<.001(*)	<.001(*)	<.001(*)	<.001(*)
uni/pl (mod)	<.001(*)	<.001(*)	<.001(*)	<.001(*)

4.1.1 Prior Work in Initialization Methods for SLS

We are aware of only two other initialization methods for MAX-SAT that are able to improve over a uniform random solution, due to Zhang et al. [89] and Qasem et al [62], which have been discussed in more detail in Chapter 2. In both of these cases, local search must first be run multiple times with a uniform random initialization to construct a set of local optima. The frequency of assignments for each bit found in the set of local optima are then used to initialize subsequent runs of local search. Zhang hypothesizes that these frequencies can provide an estimation of the backbone [89], however no experimental data has been published to support this hypothesis.

4.1.2 Hyperplane Initialization

The early theoretical analysis of genetic algorithms emphasized the potential for populations to implicitly estimate hyperplane averages and to use this information to guide search [37, 28]. While this line of research has been criticized [64], a similar idea is at the foundation of estimation of distribution algorithms: information about the interaction between variables can be used to guide search [60, 29].

For all k-bounded pseudo-Boolean optimization (PBO) problems, we can convert the evaluation functions into a polynomial form in $O(n)$ time [83]. This allows us to quickly and

exactly compute low order hyperplane averages. We can then explicitly determine which combination of variable assignments will lead to the highest overall combined hyperplane average. By using this information to initialize search we achieve two results: 1) search starts at a solution that *must* be better than average, reducing the number of steps needed to reach a local optimum, and 2) the initial solution is in a region of the search space that is also better than average relative to other regions. Thus, not only is the initial solution better than average, solutions that are nearby in Hamming space must also be better than average.

We note that although we focus on MAX-SAT, hyperplane initialization can be generalized to other PBO problems. A wide range of important optimization problems are naturally expressed as k-bounded PBO problems. In computing, this includes hardware verification, combinatorial auctions, design debugging, software testing and graph coloring[50] as well as classic NP-hard problems such as MAX-SAT, vertex cover, maximum cut, and maximum independent set [10]. In biology, NK-landscapes have been developed as a general model for interacting sets of components (alleles, proteins, amino-acids) with applications in RNA-folding [65] and the study of viruses [20]. In physics, Ising spin glasses correspond to PBO problems [17].

We first show how the Walsh transform can be used to efficiently calculate the hyperplane averages of MAX-SAT instances and then describe a method of using these averages to construct a solution. This method consistently produces solutions with better evaluations than those constructed with a uniform random distribution, the standard practice for SLS algorithms.

We next show that hyperplane information can provide a remarkably good estimation of the correct assignment of backbone variables. We use this information to initialize runs of AdaptG2WSAT and find results consistent to explicitly assigning the correct truth values to the backbone variables.

4.1.3 Theoretical Foundations

A discrete function $f : \{0,1\}^n \mapsto \mathbb{R}$ can be decomposed into an orthogonal basis

$$f(x) = \sum_{i=0}^{2^n-1} w_i \psi_i(x)$$

where w_i is a real-valued weight known as a *Walsh coefficient* and ψ_i is a *Walsh function*. We will represent the index i and vector x as binary strings, and standard binary operations can be applied. The Walsh function

$$\psi_i(x) = -1^{i^T x}(-1)^{bitcount(i \wedge x)}$$

generates a sign: if $i^T x$ is odd $\psi_i(x) = -1$ and if $i^T x$ is even $\psi_i(x) = 1$.

The MAX-SAT objective function is given by

$$f(x) = \sum_{j=1}^{m} f_j(x, mask_j)$$

where each subfunction f_j corresponds to a clause and $mask_j$ has a 1 bit for each bit used by f_j. The MAX-SAT objective function is a linear combination of subfunctions and we can apply the Walsh transform to each clause individually:

$$w = \sum_{j=1}^{m} \mathbf{W} f_j$$

where w is a vector of polynomial coefficients and $\mathbf{W}$ is the Walsh transform. This generates the Walsh coefficients associated with each clause. Rana et al. [63] show that we can dispense with matrix $\mathbf{W}$ and directly compute the Walsh coefficient associated with each clause. Walsh coefficients can overlap if the same bits co-occur in the same clause. In this case, the coefficients are added together if they share the same index. Given a clause of size k, each subfunction f_j contributes at most 2^k nonzero Walsh coefficients to vector w.

4.1.4 Computing Hyperplane Averages

The search space of a MAX-SAT instance with n variables and m clauses corresponds to a n-dimensional hypercube. If we 'fix' the truth values of j variables to 1 or 0, the search space is reduced to a $(n-j)$-dimensional hyperplane.

The Walsh coefficients can be used to efficiently compute the average evaluation of solutions contained in any $(n-j)$-dimensional hyperplane [63] [27]. Let h denote a $(n-j)$ dimensional hyperplane where j variables have preassigned bit values. Let $\alpha(h)$ be a mask with 1 bits marking the locations where the j variables appear in the problem encoding, and 0 bits elsewhere. Let solution x assign values to the j variables. Let $\beta(h) = \alpha(h) \wedge x$. This means $\beta(h)$ has value 0 in all of the positions where the j bits do not appear, and has the assigned values of the relevant j bits in the appropriate bit locations. Then the average fitness of hyperplane h is

$$Avg(h) = f_{\text{avg}} + \sum_{\forall b, b \subseteq \alpha(h)} w_b \psi_b(\beta(h))$$

where $f_{\text{avg}} = w_o$ is the average over the entire MAX-SAT search space, i.e. $f_{\text{avg}} = (2^k - 1)/(2^k) * m$.

Although we can find the averages of any number of hyperplanes using this method, for the current study we compute the averages of the $2^k m$ hyperplanes that exactly correspond to the m clauses. For example, in a MAX-3SAT problem, there are seven assignments that can make a clause true. For each of these assignments, we can compute the hyperplane averages: this tells us how, on average, this assignment will impact the evaluation function over the remainder of the search space. Note that we only need to compute the hyperplane averages once. Thus, we not only have local information (whether the assignment makes a clause true or not), we also have global information about how the assignment affects the rest of the search space. The computational complexity to do this is $O(n)$ assuming $m = cn$ and c is a constant.

4.1.5 Hyperplane Initialization

We now describe a method of exploiting hyperplane averages to construct solutions to MAX-SAT that we call *hyperplane initialization*. While we use MAX-3SAT to describe the method, hyperplane voting can be applied to any MAX-SAT problem.

In MAX-3SAT, given a clause v_i containing the variables p, q, and r, there are eight possible assignments of these variables: 000, 001, 010, 011, 100, 101, 110, and 111. We

compute the averages of the eight hyperplanes formed by fixing p, q and r to each of these partial assignments and leaving the remaining variables free. This process is repeated for each clause in the instance. Thus we compute eight hyperplane averages for each clause for a total of $8m$ hyperplane averages. We then use the hyperplane with the best average from each clause to calculate a probability distribution over all n variables as follows.

Let $v_i = \{p, q, r\}$ be the three variables in clause i. Let $A_i : v_i \mapsto \{0, 1\}$ be the partial assignment of the variables in clause i that correspond to the hyperplane with the highest average for clause v_i. Let true_j count the number of times that variable j is set to 1 across all partial assignments A and let total_j count the total number of times that variable j appears across all clauses (assignments):

$$\text{true}_j = \sum_{\forall i : j \in v_i} A_i(j)$$

$$\text{total}_\text{j} = true_j + \sum_{\forall i : j \in v_i} (1 \oplus A_i(j))$$

where $(1 \oplus A_i(j)) = 1$ when $A_i(j) = 0$. We define the following probability distribution over truth assignments based on the hyperplane voting:

$$P(j = 1) = \frac{\text{true}_j}{\text{total}_\text{j}}$$

Solutions are then constructed by generating a random value in the range of $(0, 1)$ for each variable. If the random value generated for variable j is greater than $P(j = 1)$, j is set to 1, otherwise j is set to 0.

We require the Walsh coefficients, and on problems with large clause lengths, calculating the coefficients can become intractable. Therefore, in practice we limit the max clause length to eight. Any clauses larger than this are ignored. Of course, if an instance consists only of clauses with length nine or greater, this means we could not get any hyperplane information about this instance. In these cases, it would be possible to compute the hyperplane averages over arbitrary subsets of variables, e.g. all possible 3-way combinations of the top 20 most frequent variables. However, in our studies we have not encountered any such instances and, as our results in following sections show, our method of ignoring large clauses provides excellent results.

Table 4.3: Means and standard deviations of the percentage of backbone variables correctly set by hyperplane initialization over 50 generated solutions for each of the 50 instances by generator type (See Table 3.1 for the description of the generator types).

pl/k3	uni/k3	pl/pl	uni/pl
0.76±0.13	0.80±0.14	0.83±0.08	0.83±0.08
pl/k3 (mod)	uni/k3 (mod)	pl/pl (mod)	uni/pl (mod)
0.77±0.10	0.78±0.10	0.94±0.04	0.97±0.03

4.1.6 Evaluating Hyperplane Initialization

We first examine how well hyperplane initialization can estimate the correct assignment of backbone variables and next look at how starting SLS from a hyperplane initialized solution affects the run-length distribution compared to starting the search from a random solution. We will use the same 50 instances from each generator type listed in Table 3.1 as we used in our experiments in Section 4.1 of this chapter to determine the effect of initializing search with the correct backbone assignments.

For each instance, we generated 50 solutions using the hyperplane initialization as described in Section 2.1.5. For each solution, we computed how many of the backbone variables were correctly assigned and then normalized these values by dividing by the total number of backbone variables. This gave us a percentage of backbone variables that were correctly set in each solution. We then found the mean and standard deviation of these percentages over all solutions for each instance type. These values are shown in Table 4.3.

Hyperplane initialization can provide a remarkably good estimation of backbone variables, ranging from 76% of the total backbone variables correctly assigned to 97%. Hyperplane initialization provides the best estimation on the most industrial-like instances, those generated with the pl/pl(mod) and uni/pl(mod) generators (94 and 97 respectively). In expectation a randomly generated solution would correctly assign 50% of the backbone variables. We performed a one-sided t-test to each set of solutions (grouped by generator type as in Table 3.2) with the alternative hypothesis that more backbone variables were set correctly in the hyperplane initialized solutions than in random solutions. In all tests a p-value $< .001$ was found.

Based on these results, we conjecture that hyperplane initialization will have a similar impact to SLS algorithms as our backbone initialization experiments did in Figure 4.1, but recall hyperplane initialization biases all variables, not just the backbone variables. When generating the results for Figure 4.1, non-backbone variables were set to true or false with equal probability. This is not true in hyperplane initialization. We therefore wish to determine if hyperplane initialization will increase the performance of AdaptG2WSAT in a similar manner.

We ran AdaptG2WSAT for 100 runs on each of the 50 instances from our eight generator types. 50 runs per instance were initialized with hyperplane initialization, and 50 runs were initialized with a uniform random solution in which each bit was set to true or false with equal probability. Each run was allowed 10,000 bit flips. To determine the effect of hyperplane initialization on AdaptG2WSAT's ability to find a global optimum, we then computed the run-length distributions for each initialization method on each problem type in the same manner as described in Chapter 3, Section 2.1. Figure 4.2 shows the RLDs for the two initialization methods broken down by problem type.

The RLDs in Figure 4.2 indicate that hyperplane initialization improves AdaptG2WSAT's ability to find the global optima on all instances types. Interestingly, it is more pronounced on the most industrial-like instances: those with a community structure and a variable clause length. We believe that this is due to the fact these structures allow us to extract more information about the space from the hyperplane averages. The smaller clauses that are the most frequent in a power-law distribution of clause lengths will generate Walsh coefficients with a higher magnitude. Of course, these instances also have the largest size backbones. Not only can we get stronger hyperplane information that results in a higher accuracy of the correct backbone settings, but there are also more variables that are set correctly in these problems.

To determine if improvement in the RLDs shown in Figure 4.2 is statistically significant, we counted the number of flips required to find a global optimum for the hyperplane initialized runs and for the runs initialized with a random solution. Table 4.4 reports the means

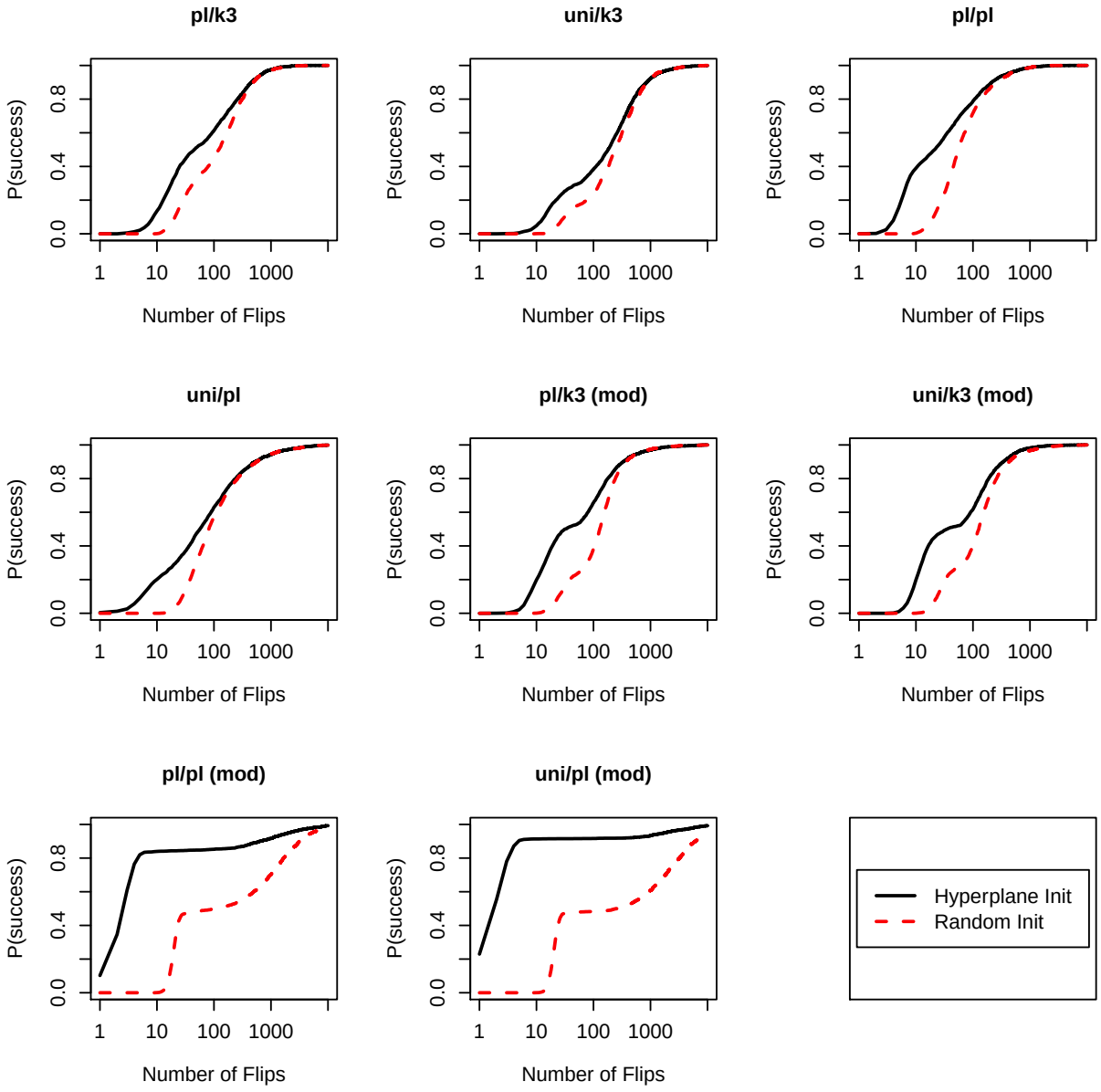


Figure 4.2: The run length distributions of two initialization methods over 50 runs on 50 instances with $n = 50$ and $m = 214$ of each generator type (See Table 3.1 for our generator types). Each set of runs consisted of 50 runs initialized by hyperplane initialization (hyperplane init) and random solutions (random init)

Table 4.4: Mean and standard deviations of the number of flips required for AdaptG2WSAT to find a global optimum on runs initialized with hyperplane initialization and random solutions. There were 50 runs for each instance and 50 instances for each problem type. The p-value column reports the p-value of a one-sided t-test comparing these values with the alternative hypothesis that the hyperplane initialized runs require less flips. Results significant at the .001 level are indicated with a (*).

Problem Type	Random	Hyperplane	p-value
pl/k3	220.30±351.17	172.69±319.33	<.001(*)
uni/k3	425.29±692.45	385.28±796.84	0.029
pl/pl	120.18±208.70	92.73±235.16	<.001(*)
uni/pl	298.39±877.58	275.21±864.81	0.173
pl/k3 (mod)	221.08±524.41	176.36±570.32	0.002
uni/k3 (mod)	230.45±493.38	147.38±367.38	<.001(*)
pl/pl (mod)	1267.91±3152.08	411.03±1954.88	<.001(*)
uni/pl (mod)	2166.21±4444.16	360.28±1990.83	<.001(*)

and standard deviations of these data for each instance type and the p-values of a one-sided t-test with an alternative hypothesis that the hyperplane initialized runs require less flips than the randomly initialized runs.

The mean flips to find a global optimum are always lower for the hyperplane initialized runs than those runs initialized with a random solution. This decrease was significant at the .001 level in five of the eight problem types, and significant at the .05 level in seven of the eight problem types. The difference in mean flips to a global optimum were not statistically significant in the uni/pl problem types, although it was lower for the hyperplane initialized runs. The largest decrease in mean flips to a global optimum are in the modular instances with a variable clause length. Not only are these the most industrial-like instances, but they were also the instances with the highest mean percentage of backbone variables set correctly by hyperplane initialization (See Table 4.3).

4.1.7 Hyperplane Initialization and Industrial Instances

We have shown that hyperplane initialization can increase the performance of Adapt-G2WSAT on generated instances with industrial-like characteristics. We now wish to extend our analysis on the impact of hyperplane initialization from our generated instances to industrial instances. We have selected 30 problems from our set of industrial instances listed

in Appendix A. For the sake of brevity, we will refer to these instances using an identification code as shown in Table 4.5. These instances were chosen from the 380 industrial instances listed in Appendix A to represent a variety in both the underlying application and in number of variables.

Due to the fact that the optimal solutions are not known for the majority of these instances [67, 52], we cannot perform the same run-length distribution analysis as in previous experiments to measure the performance of SLS algorithms. However, in many applications of MAX-SAT a globally optimal solution is not needed; merely finding a *good* solution is often enough to provide good results [50, 44]. Even so, the quality of the returned solution can have an impact on the underlying application [13, 66]. Therefore, improving SLS algorithms can have a valuable impact on real-world applications.

To determine if hyperplane initialization can increase the quality of solutions found by AdaptG2WSAT, we generated 30 random solutions and 30 solutions initialized by hyperplane initialization for each of the industrial instances in Table 4.5. We ran AdaptG2WSAT for $20n$ bit flips starting from each of the generated solutions. Our procedure differs here from Section 2.1.6 because in Section 2.1.6 all our instances had the same number of variables, $n = 50$. Our industrial instances in Table 4.5 range from $n = 300$ to $n = 1,974,822$ variables. Therefore, we wanted to consistently scale the number of bit flips relative to the size of the instance and $20n$ bit flips is tractable even on the largest of our industrial instances. We recorded the evaluation of the initial solution and the best-so-far solution after 10% of the total run, 50% of the total run and at the end of the run.

We first examine the initial solutions found by our generation methods. Our hypothesis is that the average evaluation of initial solutions generated by hyperplane initialization on industrial instances will be lower than those found by random solutions. Table 4.6 reports the mean and standard deviation of the initial solutions found by random and hyperplane initialization on the 30 instances. The right-most column reports the p-value from a one-sided t-test with an alternative hypothesis that the mean evaluations of the hyperplane initialized solutions are lower than the mean evaluation of random solutions.

Table 4.5: Mapping of industrial instances chosen for empirical experiments to identification numbers.

ID	Instance Name	n	m
1	aes-32-1-keyfind-1.cnf	300	1016
2	myciel6-tr.used-as.sat04-320.cnf	570	4625
3	vmpc-25.renamed-as.sat05-1913.cnf	625	76775
4	dp04s04.shuffled.cnf	1075	3152
5	rbcl-xits-18-SAT.cnf	2888	218530
6	E07N15.cnf	4740	41363
7	comb1.shuffled.cnf	5910	16804
8	k2fix-gr-rcs-w8.shuffled.cnf	10056	271393
9	slp-synthesis-aes-bottom12.cnf	17298	57292
10	traffic-b-unsat.cnf	26061	742909
11	gss-21-s100.cnf	31613	95104
12	AProVE11-12.cnf	44805	149118
13	maxxor064.cnf	51064	152039
14	hwmcc10-timeframe-expansion-k45-bc57sensorsp1.cnf	63327	174791
15	countbitssrl064.cnf	75103	225116
16	sokoban-sequential-p145-microban-sequential.040.cnf	87884	1413816
17	11pipe-k.cnf	89315	5584003
18	aaai10-planning-ipc5-TPP-21-step11.cnf	99736	783991
19	UR-10-10p1.cnf	131228	635871
20	post-c32s-gcdm16-23.cnf	135543	404326
21	openstacks-p30-1.045.cnf	171676	858766
22	ibm-2004-23-k100.cnf	207606	861175
23	divider-problem.dimacs-8.filtered.cnf	246943	810105
24	fpu-multivec1-problem.dimacs-14.filtered.cnf	257168	928310
25	wb-problem.dimacs-46.filtered.cnf	300846	789283
26	c2-DD-s3-f1-e2-v1-bug-fourvec-gate-0.dimacs.seq.filtered.cnf	400085	1121810
27	b15-bug-fourvec-gate-0.dimacs.seq.filtered.cnf	581064	1712690
28	mrisc-mem2wire-problem.dimacs-29.filtered.cnf	844900	2905976
29	rsdecoder-problem.dimacs-41.filtered.cnf	1186710	3829036
30	mem-ctrl2-blackbox-mc-dp-problem.dimacs-28.filtered.cnf	1974822	6795573

Table 4.6: Means and standard deviations of the evaluations rounded to the nearest integer of 30 solutions produced by hyperplane initialization (Hyperplane) and random solutions (Random) for the 30 industrial instances from Table 4.5. The p-value column lists the p-value from one sided t-tests comparing the means with an alternative hypothesis that the hyperplane initialized solutions have lower evaluations than those of random solutions.

ID	Hyperplane	Random	p-value
1	111±6	139±9	<.001(*)
2	94±0	1,106±99	<.001(*)
3	48±0	13,226±736	<.001(*)
4	346±8	639±20	<.001(*)
5	860±22	27,190±1,481	<.001(*)
6	1,070±41	4,440±164	<.001(*)
7	2,081±24	3,656±40	<.001(*)
8	1,244±1	67,557±1,875	<.001(*)
9	5,542±52	10,810±121	<.001(*)
10	27,358±462	60,543±685	<.001(*)
11	8,360±68	19,524±128	<.001(*)
12	18,313±123	26,372±190	<.001(*)
13	19,852±161	31,650±171	<.001(*)
14	19,566±97	36,346±181	<.001(*)
15	29,585±193	46,792±195	<.001(*)
16	660±21	343,007±2,832	<.001(*)
17	31,133±25	1,361,003±67,209	<.001(*)
18	19,266±53	190,654±1,303	<.001(*)
19	41,822±132	141,053±595	<.001(*)
20	36,560±124	84,211±263	<.001(*)
21	38,480±129	212,247±798	<.001(*)
22	81,968±193	183,359±670	<.001(*)
23	90,088±163	146,661±415	<.001(*)
24	87,622±203	159,068±407	<.001(*)
25	98,042±225	164,982±490	<.001(*)
26	130,489±273	231,173±584	<.001(*)
27	280,231±440	368,539±577	<.001(*)
28	233,413±662	509,449±1,112	<.001(*)
29	292,240±384	685,530±813	<.001(*)
30	611,118±2,017	1,182,185±1,309	<.001(*)

In all cases the hyperplane initialized solutions had evaluations significantly lower than random solutions. In some cases, this difference was an order of magnitude or more. In two cases, instances 2 and 3, hyperplane initialization constructed 30 solutions with the same evaluation. Further analysis of these solutions revealed that the truth settings for each variable were the exact same in all solutions: all the variables were set to false. Analysis of these two instances revealed that the literals in over 90% of the clauses in both instances were all negated. As we compute hyperplane averages on a clause by clause basis this naturally resulted in an extremely heavy bias towards setting variables false.

Having established that hyperplane initialized solutions have better evaluations than random solutions, we now wish to determine how these solutions influence the subsequent search. We recorded the best found solution at intervals of 10% of the overall run length of $20n$ bit flips. To keep runs tractable on large instances, we limited the maximum number of bit flips for a single run to 5,000,000.

Table 4.7: The means and standard deviations of the evaluations of the best-so-far solutions after 10%, 50% and 100% of the overall run length over 30 runs per instance (see Table 4.5 for a description of the instances). Runs were either initialized with hyperplane initialization or random initialization. The p-value column reports the p-value from a t-test testing a difference in the mean evaluations.

ID	Percent of Run	Hyperplane	Random	p-value
1	10%	9±2	10±2	0.300
	50%	6±1	6±1	0.578
	100%	5±1	5±1	1.000
2	10%	3±0	4±1	<.001(*)
	50%	1±0	1±0	1.000
	100%	1±0	1±0	1.000
3	10%	12±2	12±2	0.054
	50%	9±1	9±2	0.596
	100%	8±1	8±2	0.795
4	10%	15±4	22±4	<.001(*)
	50%	4±2	5±2	0.135
	100%	2±1	2±1	0.196
5	10%	34±57	67±81	0.076
	50%	7±2	7±2	0.608
	100%	6±1	6±2	0.461

Continued on next page

Table 4.7 – *Continued from previous page*

ID	Percent of Run	Hyperplane	Random	p-value
6	10%	67±5	54±8	<.001(*)
	50%	24±6	21±5	0.033
	100%	13±3	12±3	0.366
7	10%	204±11	196±12	0.011
	50%	100±11	109±13	0.005
	100%	69±8	74±10	0.020
8	10%	116±6	115±7	0.550
	50%	104±6	102±6	0.300
	100%	94±7	91±8	0.098
9	10%	34±7	39±6	0.010
	50%	14±4	13±4	0.130
	100%	10±3	10±3	0.361
10	10%	157±31	210±100	0.009
	50%	107±21	108±28	0.861
	100%	96±20	95±23	0.878
11	10%	669±26	420±21	<.001(*)
	50%	349±43	234±20	<.001(*)
	100%	192±16	165±10	<.001(*)
12	10%	1,401±33	1,570±51	<.001(*)
	50%	1,109±51	1,238±68	<.001(*)
	100%	904±38	976±87	<.001(*)
13	10%	899±26	928±25	<.001(*)
	50%	619±48	568±41	<.001(*)
	100%	415±46	352±39	<.001(*)
14	10%	447±36	626±45	<.001(*)
	50%	235±35	311±32	<.001(*)
	100%	100±15	167±24	<.001(*)
15	10%	1,349±27	1,962±69	<.001(*)
	50%	1,110±34	1,338±79	<.001(*)
	100%	961±59	979±74	0.313
16	10%	12±0	28±5	<.001(*)
	50%	12±0	19±3	<.001(*)
	100%	12±0	16±2	<.001(*)
17	10%	338±35	554±25	<.001(*)
	50%	333±34	543±26	<.001(*)
	100%	279±63	516±61	<.001(*)
18	10%	141±1	147±3	<.001(*)
	50%	110±6	117±4	<.001(*)
	100%	87±6	94±6	<.001(*)

Continued on next page

Table 4.7 – *Continued from previous page*

ID	Percent of Run	Hyperplane	Random	p-value
19	10%	111±8	550±38	<.001(*)
	50%	53±8	351±52	<.001(*)
	100%	42±7	168±58	<.001(*)
20	10%	1,260±35	3,063±46	<.001(*)
	50%	1,123±68	2,432±199	<.001(*)
	100%	1,087±54	1,852±86	<.001(*)
21	10%	76±8	67±4	<.001(*)
	50%	40±4	35±3	<.001(*)
	100%	27±2	25±3	<.001(*)
22	10%	3,114±57	3,887±50	<.001(*)
	50%	2,461±169	3,126±222	<.001(*)
	100%	1,811±159	2,301±105	<.001(*)
23	10%	3,088±220	9,449±232	<.001(*)
	50%	2,606±276	7,620±419	<.001(*)
	100%	2,255±259	5,962±357	<.001(*)
24	10%	8,258±84	9,049±116	<.001(*)
	50%	6,659±435	7,162±512	<.001(*)
	100%	5,186±342	5,170±283	0.848
25	10%	3,978±93	7,923±159	<.001(*)
	50%	3,165±191	7,020±260	<.001(*)
	100%	2,442±164	6,019±304	<.001(*)
26	10%	6,190±80	13,857±163	<.001(*)
	50%	5,216±59	10,853±90	<.001(*)
	100%	5,014±74	10,165±273	<.001(*)
27	10%	18,093±261	29,030±208	<.001(*)
	50%	13,371±189	19,820±197	<.001(*)
	100%	12,578±432	18,665±992	<.001(*)
28	10%	3,109±98	59,388±1,501	<.001(*)
	50%	2,854±358	30,602±954	<.001(*)
	100%	2,085±272	29,628±1,607	<.001(*)
29	10%	22,316±122	130,539±598	<.001(*)
	50%	10,283±248	36,567±471	<.001(*)
	100%	9,927±280	34,284±467	<.001(*)
30	10%	72,819±1,084	370,027±683	<.001(*)
	50%	11,171±168	35,972±581	<.001(*)
	100%	10,472±919	34,657±534	<.001(*)

Table 4.7 lists the means and standard deviations of the best-so-far solution at the 10% mark, the 50% mark and the 100% mark of the 30 runs for each instance. We compared the

means using a two-sided t-test with the alternative hypothesis that the mean evaluations of the best-so-far solutions in the hyperplane initialized runs are different than those of the runs starting from the random solutions. A two-sided test was chosen because in some cases the mean of the random runs was higher. The p-values of these tests are also listed in Table 4.7.

At the 10% mark, there were 20 significant differences. In two cases, instances 11 and 6, the mean best-so-far solution from the random runs was higher than the hyperplane means. In all other significant cases, the hyperplane means were better. At the 50% mark, there were 21 significant differences. Again, in these cases the hyperplane means were better except for instances 6, 9, and 11. At the 100% mark, there were 20 significant differences. The mean evaluations of the best-so-far solutions was better in the hyperplane initialized runs in all but two cases: instances 11 and 13. Of the cases in which the random runs were significantly better, only in the case of instance 11 were the random runs significantly better than the hyperplane initialized runs at all marks.

On large instances, instance 14 and up with the exception of instance 22 in which no significant difference was found, the hyperplane runs consistently have a significantly better mean evaluation than those runs starting from random solutions. This occurs at all points along the runs. In some cases the difference between means is an order of magnitude and more.

4.1.8 Summary of Hyperplane Initialization

In our evaluation of hyperplane initialization, we found that hyperplane initialization can provide a significantly better estimation of the correct backbone assignments than random solutions. The best estimations, correctly assigning over 90% of the total backbone variables, occurred on the most industrial-like instances, those generated with the uni/pl(mod) and pl/pl(mod) generators (see Table 3.1 for a full description of our generators).

When we initialized AdaptG2WSAT with the hyperplane initialized solutions, the search was able to find a global optimum faster than runs initialized from random solutions. We conjecture that this is because the search starts closer to the hyperplane defined by the

backbone set that contains all of the globally optimal solutions as discussed in Chapter 3. The search was influenced the greatest on the most industrial-like instances, the same instances that hyperplane initialization provided the best estimation of the correct backbone assignments.

We extended our analysis to industrial instances and found that hyperplane initialization constructed significantly better solutions than random initialization on all instances. In some cases, the evaluations were improved by an order of magnitude or more. We examined the best-so-far solutions found by AdaptG2WSAT when initialized by hyperplane initialization at various points in the search. On all instances with 60,000 variables those runs initialized by hyperplane initialization were able to find significantly better solutions than those runs initialized with random solutions with one exception in which there was no significant difference. This provides strong evidence that hyperplane initialization can significantly improve the quality of solutions found by SLS algorithms.

4.2 Fast Initial Descent

SLS algorithms can be split into two phases. The first phase is the part of the search prior to the first local optimum being found and the second is after the first local optimum is found [78, 1]. We will refer to this first phase as the *initial descent* and look at two strategies for this phase: a first improving search and a best improving search. *Best improving* search for MAX-SAT first identifies the improving moves in the neighborhood of the candidate solution and takes the best improving move. *First improving* search arbitrarily selects an improving move without necessarily considering all improving moves.

AdaptG2WSAT and other SLS algorithms, e.g. GSAT [71], employ a best improving search for the initial descent [48, 49]. Best improving search can be costly; exact implementations employed by most SLS solvers have a computational complexity of $O(n)$ while first improving search has a complexity of $O(1)$ [1]. Prior studies on uniform random instances of MAX-SAT by Gent and Walsh (1992) and by Schuurmans and Southey (2001) [23, 69] indicate that best improving local search methods are no better than first improving local

search, yet many modern algorithms continue to use best improving local search despite the extra computational cost [48, 49, 75].

We therefore wish to revisit this issue by analyzing industrial problems to determine if there is any advantage to using best improving search over next improving search for the initial descent. We will carry out this analysis in two stages.

The first stage is an analysis of the local optima found found by the two types of local search and the time required to find a local optimum. The purpose of this experiment is to establish any differences in the quality of local optima found by the two searches and the effort required to find them in terms of CPU time.

The initial descent to a local optimum is only the first of two phases of SLS. We are therefore also concerned with how the search behaves after a local optimum is found. In the second stage of our analysis we will look at the effect of the two initial descents on the final solutions found by the second phase of SLS.

4.2.1 Prior Analyses of Local Search

Gent and Walsh [25, 24] analyzed hill climbing in GSAT [71] to determine what contributed to its performance. Gent and Walsh [23] examined the influence of greediness (best improving moves) and randomness (first improving moves) on six problems: three SAT encodings of the n-queens problem ($n = 6, 8, 16$) and three random 3SAT problems with clause to variable ratio of 4.3 (for 50, 70 and 100 variables). They found that neither greediness nor randomness were important for GSAT's performance. Gent and Walsh [24] divided search into two phases: a short duration of hill climbing followed by a long duration of plateau search. They tested on random 3SAT problems with up to 1000 variables and showed that hill climbing progresses through increasingly lengthening phases in which the incremental improvement declines until a plateau is reached.

Schuurmans and Southey (2001) identified characteristics of local search that can be used to predict behavior [69]. Seven different algorithms were run on MAX-3SAT problems in

the phase transition region from SATLIB. They found that effective SAT solvers first find a good local optimum and then explore quickly and broadly to find a solution.

These studies assessed performance on small random 3SAT instances. Do these conclusions extend to large problems and industrial applications where the optimal solution is not known? To address this question, we will perform our own analysis of the trade-off between first and best improving search on the industrial problems listed in Table 4.5.

4.2.2 First vs Best Improving Local Search

We first examine the quality of local optima found by first improving and best improving local search on industrial instances. We use a common implementation for the update in both searches in which the neighborhood is updated in amortized constant time [82]. In other words, the list of improving moves in the Hamming distance 1 neighborhood from the candidate solution is updated in constant time for both best improving and first improving search.

In best improving search, we scan this list to find the move with the best improvement to the evaluation of the candidate solution, breaking ties at random. This has a worst case complexity of $O(n)$ in the event that the number of improving solutions is some factor of n. We have empirically shown that indeed best improving search implemented in this way, even with a constant time neighborhood update, has an $O(n)$ worst case computational complexity [1].

First improving search needs only to select an improving move. Therefore, we need only to select an element at random from the list of improving moves. A scan is not required. This implementation of first improving search, along with a constant time update, has $O(1)$ worst case computational complexity.

We constructed 30 random solutions for each of the industrial instances in Table 4.5 by setting each variable to true or false with equal probability. From each solution we started both best improving search and first improving search. Once the search reached a solution with no improving moves in the Hamming distance 1 neighborhood, the search

was terminated. We recorded both the CPU time required to find these solutions and their evaluations. Table 4.8 reports the mean and standard deviations of these evaluations for each instance. We also ran a two-sided t-test comparing the evaluations from each search for each instance. The p-values are reported in the right-most column of Table 4.8.

We see that in the majority of instances, with the exception of instances 3 and 18, there is a significant difference between the evaluations of local optima found by best improving search and first improving search. Instance 18 is a planning problem that has been reduced to an instance of MAX-SAT [44]. In 9 out of the 28 significant cases, specifically instances 2, 6, 8, 10, 16, 17, 19, 21 and 22, first improving search found better solutions on average than best improving search. In the remaining 19 out of 28 significant cases, best improving search found local optima with better evaluations.

We next look at the cost of finding a local optimum in terms of CPU time. To do so, we measure the CPU time required by each run to find a local optimum, time to local optimum (TLO), that was required for each run. Given that first improving search has a constant time complexity and that best improving search has a linear time complexity, we expect to see shorter times from the runs using first improving search. The means and standard deviations of TLO over the 30 runs on each instance are reported in Table 4.9 along with p-values from a one-sided t-test with the alternative hypothesis that the TLO from the first improving runs is lower than those from best improving runs.

Although there is little difference in the smaller instances, as the instance size grows we begin to see significant differences in the TLO. In all instances with more than 3,000 variables, the runs using first improving search have a significantly lower mean TLO than those runs using best improving search. Clearly on the largest instances, the initial descent could have a major impact on the CPU time required by the search. On the largest instances, the difference between times is over a factor of 1,000.

We can make a major savings in the effort required to reach a local optimum in terms of CPU time on large instances by using first improving as opposed to the best improving used in state-of-the-art SLS algorithms. However, we see mixed results in the evaluation of local

Table 4.8: Means and standard deviations of the evaluations of 30 local optima found by first improving search (First) and best improving search (Best) for 30 industrial instances from Table 4.5. The p-value column lists the p-value from a two sided t-test comparing the means.

ID	First Improving	Best Improving	p-value
1	55±5	50±4	<.001(*)
2	18±3	24±3	<.001(*)
3	28±2	26±3	0.023
4	139±9	129±9	<.001(*)
5	349±19	227±10	<.001(*)
6	285±23	317±22	<.001(*)
7	737±22	710±20	<.001(*)
8	417±17	451±12	<.001(*)
9	2,148±40	1,983±30	<.001(*)
10	2,667±218	3,050±168	<.001(*)
11	4,362±49	4,313±53	<.001(*)
12	6,132±74	5,487±45	<.001(*)
13	6,782±73	6,510±71	<.001(*)
14	7,154±84	6,666±62	<.001(*)
15	9,808±85	9,402±101	<.001(*)
16	563±31	1,152±38	<.001(*)
17	2,120±83	4,363±78	<.001(*)
18	2,493±123	2,490±93	0.926
19	10,635±130	12,671±125	<.001(*)
20	18,416±99	18,171±116	<.001(*)
21	6,836±56	7,205±76	<.001(*)
22	24,075±123	24,437±125	<.001(*)
23	34,021±283	31,713±191	<.001(*)
24	36,178±210	32,600±186	<.001(*)
25	44,571±151	40,161±169	<.001(*)
26	56,267±269	55,013±132	<.001(*)
27	74,250±260	70,816±234	<.001(*)
28	118,522±1,125	103,727±440	<.001(*)
29	191,198±457	173,903±378	<.001(*)
30	255,366±817	237,565±537	<.001(*)

Table 4.9: Means and standard deviations of the TLOs over 30 runs of first improving search (First) and best improving search (Best) for 30 industrial instances from Table 4.5. The p-value column lists the p-value from a two sided t-test comparing the means.

ID	First Improving	Best Improving	p-value
1	0.00±0.00	0.00±0.00	1.000
2	0.00±0.00	0.00±0.00	1.000
3	0.00±0.00	0.00±0.00	1.000
4	0.00±0.00	0.00±0.00	1.000
5	0.02±0.01	0.01±0.00	0.985
6	0.01±0.01	0.01±0.00	<.001(*)
7	0.00±0.01	0.01±0.00	<.001(*)
8	0.04±0.01	0.07±0.00	<.001(*)
9	0.01±0.00	0.07±0.00	<.001(*)
10	0.26±0.02	0.31±0.01	<.001(*)
11	0.01±0.00	0.21±0.01	<.001(*)
12	0.02±0.00	0.43±0.01	<.001(*)
13	0.02±0.01	0.66±0.02	<.001(*)
14	0.02±0.00	0.92±0.02	<.001(*)
15	0.04±0.01	1.37±0.02	<.001(*)
16	0.37±0.04	4.99±0.13	<.001(*)
17	1.38±0.21	5.57±0.55	<.001(*)
18	0.22±0.01	4.17±0.06	<.001(*)
19	0.26±0.04	7.83±0.38	<.001(*)
20	0.08±0.01	4.25±0.06	<.001(*)
21	0.33±0.06	10.29±0.62	<.001(*)
22	0.40±0.05	10.93±0.43	<.001(*)
23	0.19±0.02	13.21±0.57	<.001(*)
24	0.22±0.03	14.53±1.36	<.001(*)
25	0.20±0.02	17.25±0.84	<.001(*)
26	0.43±0.03	37.77±3.52	<.001(*)
27	0.79±0.05	189.63±4.94	<.001(*)
28	0.81±0.06	589.20±33.36	<.001(*)
29	1.01±0.05	1,164.21±45.88	<.001(*)
30	2.01±0.10	4,558.74±197.82	<.001(*)

optima found by the two searches: the majority of local optima found by best improving search are better than first improving. Our next experiment will determine if this difference affects the final solution found at the end of the second stage.

We ran AdaptG2WSAT on the industrial instances in Table 4.5 with two types of initial descent: first improving and best improving. We ran 30 runs of each type for $20n$ bit flips with a maximum of 5,000,000 bit flips[1]. Note that AdaptG2WSAT uses a best improving initial descent as default, so our best improving runs are no different from its default behavior. Table 4.10 reports the means and standard deviations of the evaluations from each of the runs along with the p-values from a two-sided t-test testing for a difference in the means.

In the majority of cases, there is no significant difference. Interestingly, in all cases that are significant at the .001 level those runs using first improving search found better solutions on average than the rest. At $\alpha = .05$, only instances 14, 18 and 22 have a significant difference in favor of those runs using best improving search. On the vast majority of instances, there is either no significant difference or a significant difference indicating that first improvement finds better solutions.

We conjecture that first descent is better in some cases as it is not biased towards flipping the critical variables. To illustrate our point, we choose instance 26, as this is one of the larger instances on which first improving search performed better. This instance has 400,085 variables and 1,121,810 clauses. We first counted the variable frequencies of all the variables in this instance. 71 of the 400,085 variables in this instance each appeared in over 1,500 clauses. The next most frequent variable appeared in less than 300 clauses. The vast majority of variables, 345,600 to be exact, appeared in less than 10 clauses.

We then looked at the first 1,000 bit flips of the runs from best improving and first improving search. We plotted this against the step at which each bit was flipped in Figure 4.3. There is a heavy bias towards flipping the bits with $> 1,500$ frequency in the first few

[1]Six instances were terminated before $20n$ flips.

Table 4.10: Means and standard deviations of the evaluations of solutions found by Adapt-G2WSAT starting from 30 local optima found by first improving search (First) and 30 local optima found by best improving search (Best) on 30 industrial instances from Table 4.5. The p-value column lists the p-value from a two sided t-test comparing the means.

ID	First Improving	Best Improving	p-value
1	5.07±1.28	4.97±1.22	0.758
2	1.00±0.00	1.00±0.00	1.000
3	8.43±1.70	8.53±1.36	0.802
4	1.83±0.99	1.83±1.12	1.000
5	5.90±1.97	7.10±2.76	0.058
6	12.30±2.72	12.83±2.69	0.448
7	74.17±9.86	75.27±8.72	0.649
8	90.97±7.50	92.13±7.15	0.540
9	10.33±3.17	11.10±3.85	0.403
10	95.07±23.08	94.67±20.07	0.943
11	165.20±10.50	167.80±16.17	0.464
12	975.57±86.61	969.00±58.27	0.732
13	351.87±38.69	331.90±43.46	0.065
14	166.83±23.63	149.67±19.98	0.004
15	978.67±73.52	1,031.97±78.76	0.009
16	15.93±2.03	16.30±1.66	0.448
17	515.70±60.96	736.67±52.34	<.001(*)
18	93.73±6.00	90.40±5.12	0.024
19	168.37±57.70	265.97±60.02	<.001(*)
20	1,852.17±86.18	1,832.83±102.46	0.432
21	24.63±2.59	25.83±2.51	0.074
22	14,374.20±46,751.12	2,201.50±136.67	0.330
23	5,962.33±357.42	6,022.60±331.71	0.501
24	5,170.03±282.82	5,369.30±400.56	0.030
25	6,019.20±304.18	6,074.67±355.18	0.519
26	10,164.87±272.79	10,350.77±212.12	0.005
27	18,665.27±992.01	21,407.70±1,198.17	<.001(*)
28	29,627.50±1,607.30	30,519.43±2,293.06	0.087
29	34,284.23±467.12	35,515.43±396.03	<.001(*)
30	34,657.31±534.48	35,439.06±707.13	<.001

iterations of the search in best improving search. This makes sense, as it will greedily flip the variables that satisfy the most clauses. Those variables that appear most frequently will have a greater chance to satisfy more clauses.

First improving has no such bias. Although 2 of the variables with $1,500$ frequency are flipped, there is no bias towards the more frequent variables. We conjecture that first improving search performs better as it does not prematurely fix these critical variables early in the search.

By setting the critical variables early, we believe that AdaptG2WSAT with best improving search will have a harder time undoing any mistakes made in setting these variables. To better understand how the search is influenced by the two local searches, we ran both AdaptG2WSAT using first improving search and AdaptG2WSAT using best improving search for 5,000,000 bit flips on the same instance used to generate Figure 4.3. Figure 4.4 shows the improvement over time at three points along the searches.

Figure 4.4(a) shows the first 100,000 bit flips. During the first 100,000 flips, first improvement maintains a relatively constant rate of improvement. Due to the greedy bias, best improving search improves at a faster rate. However, at about 320,000 bit flips first improving finds a better best-so-far solution as shown in Figure 4.4(b). At this point, AdaptG2WSAT with first improving search improves at a faster rate than AdaptG2WSAT with best improving search. We believe this is a direct result of not setting the critical variables early in the search.

In Figure 4.4(c), both searches have stagnated and cannot find improving solutions. At this point, there is little chance that either search will find an improved solution. We conjecture that alternative strategies must be employed in the second phase to alleviate this stagnation by using additional information about the space beyond the immediate gradient, e.g., Sutton's directed plateau search [77] or hyperplane information. Nevertheless, the search using first improving search during the initial descent has found a better solution than the best improving search.

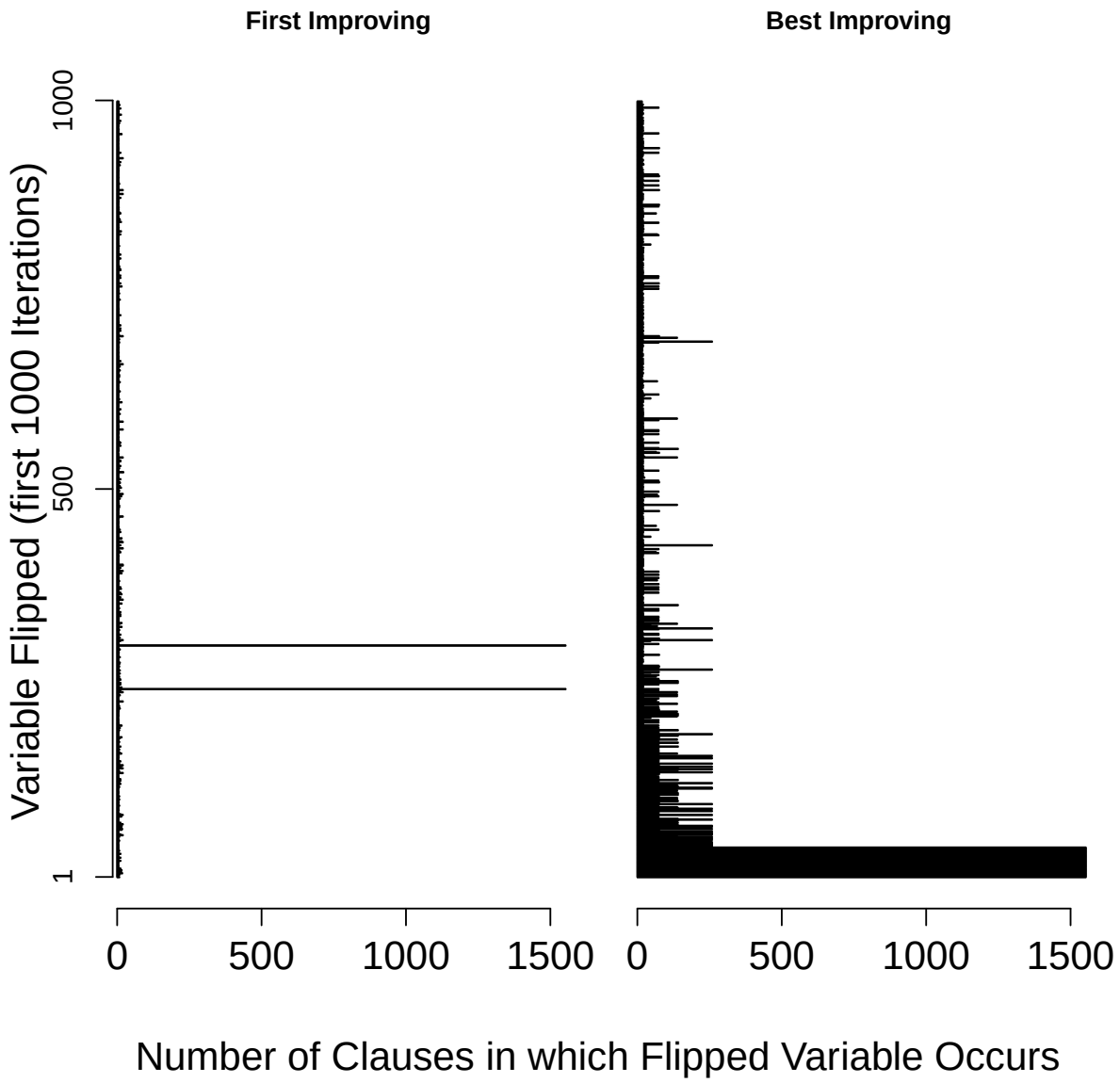
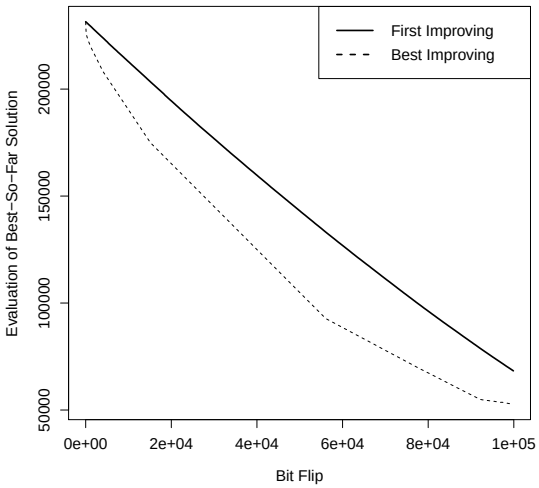
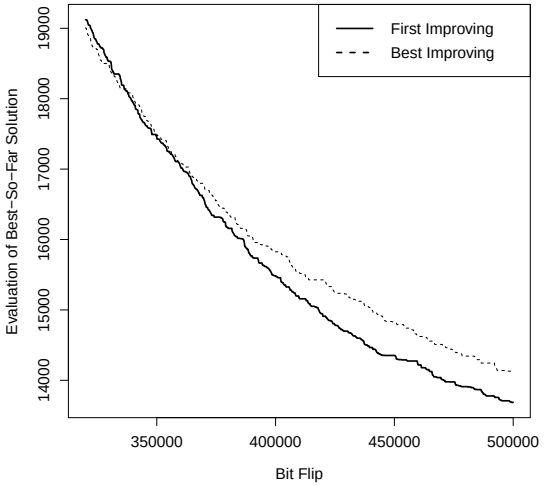


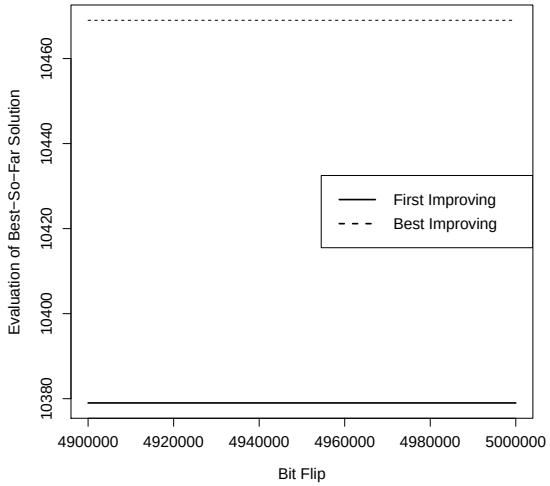
Figure 4.3: The frequency of variables that are flipped by first improving and best improving search during the first 1,000 steps of the two local search algorithms on instance 26 in Table 4.5. The flip sequence is ordered from bottom to top with the first flip at the bottom and the 1,000th flip at the top.



(a) First 100,000 Bit Flips



(b) Crossover Point



(c) Last 100,000 Bit Flips

Figure 4.4: The improvement over time of AdaptG2WSAT with best improving search and with first improving search at three points along runs of 5,000,000 bit flips: (a) the first 100,000 bit flips, (b) the point where AdaptG2WSAT with first improving search finds a better solution than the best improving version, and (c) the last 100,000 bit flips.

The results in Table 4.10 do not reflect another advantage of using first improving search: the savings in computational time required to find a local optimum. Table 4.11 reports the mean and standard deviation of the CPU time required over the 30 runs for each instance.

The p-values are the result of a one-sided t-test with an alternative hypothesis that first improving search will have lower mean times than best improving search.

Although there is little difference in smaller instances, AdaptG2WSAT with first improving search requires a significantly lower time to execute $20n$ bit flips. AdaptG2WSAT with best improving search on the majority of instances. Again we see time savings on the order of several magnitudes on the largest instances. Perhaps more interesting is the fact that the time required for the second phase in AdaptG2WSAT is greatly dwarfed by the cost of the initial descent on large instances.

To better illustrate this phenomenon, we took the time to local optima from each run in Table 4.9 and divided by the overall run times in Table 4.11. This gave us a percentage of time that the search spent in the initial descent. Tale 4.12 lists these percentages. We only report the values for instances 6 and above as the smaller instances had no significant difference in the time to local optima.

As we can see, the majority of the overall running time is spent in the initial descent phase when using best improving search. By allowing the search to reach a local optimum faster, more of the search can be spent in the exploration phase. As we have seen in the previous chapter, the lower levels of the space are more rugged and thus present more difficulty to SLS algorithms. A fast initial descent will allow the search to spend more time in this difficult area of the space in time constrained applications.

It is important to note that the computational time is a factor in SLS algorithms that have a constant time update. If there are additional components to the algorithm that have a computational complexity of $O(n)$ or higher per move, then this advantage may be lost. One example of this is Iterated Robust Tabu Search (IRoTS) [75]. This SLS algorithm incorporates a best improving local search along with a Tabu list of flipped bits. When a bit is flipped, it is put on the Tabu list where it will remain for a fixed number of steps, known as the Tabu expirations. If a bit is on the list, it will not be flipped again.

We replaced the best improving local search in IRoTS with a first improving local search. We ran 30 runs of IRoTS with both best improving and first improving search on our 30

Table 4.11: Means and standard deviations of the time in seconds required to execute 30 runs of AdaptG2WSAT with each run terminated after $20n$ bit flips per run. Two versions of AdaptG2WSAT were used: our modified version using first improving search (First) and the unmodified version using best improving search (Best). 30 industrial instances from Table 4.5 were used as benchmarks. The p-value column lists the p-value from a one sided t-test comparing the means with the alternative hypothesis that the average time of the First runs will be lower.

ID	First Improving	Best Improving	p-value
1	0.00±0.00	0.00±0.00	1.000
2	0.00±0.00	0.01±0.00	0.125
3	0.06±0.01	0.06±0.00	0.315
4	0.01±0.00	0.01±0.00	0.043
5	0.23±0.03	0.24±0.03	0.052
6	0.06±0.01	0.07±0.01	<.001(*)
7	0.04±0.00	0.05±0.00	<.001(*)
8	0.81±0.17	0.89±0.15	0.071
9	0.30±0.03	0.37±0.05	<.001(*)
10	5.92±1.09	6.20±0.97	0.293
11	0.22±0.02	0.43±0.01	<.001(*)
12	0.47±0.08	0.85±0.03	<.001(*)
13	0.32±0.01	0.96±0.01	<.001(*)
14	0.41±0.02	1.30±0.02	<.001(*)
15	0.70±0.03	2.03±0.03	<.001(*)
16	17.20±1.53	22.64±1.27	<.001(*)
17	32.30±34.56	8.16±3.08	<.001(*)
18	0.79±0.03	4.90±0.05	<.001(*)
19	3.28±0.36	11.20±0.38	<.001(*)
20	2.17±0.06	6.35±0.07	<.001(*)
21	4.30±0.42	13.86±0.37	<.001(*)
22	19.05±5.37	28.91±1.73	<.001(*)
23	7.78±0.41	20.99±0.64	<.001(*)
24	7.78±0.13	21.45±0.20	<.001(*)
25	7.16±0.15	23.80±0.39	<.001(*)
26	15.10±2.99	52.87±9.42	<.001(*)
27	16.06±2.74	233.14±85.99	<.001(*)
28	12.04±2.70	743.93±103.84	<.001(*)
29	12.65±2.45	1,465.57±188.25	<.001(*)
30	11.62±2.26	5,117.51±903.11	<.001(*)

Table 4.12: Mean percentage spent in the initial descent by AdaptG2WSAT using either first improving (First) or best improving (Best) for the initial descent. Times are in seconds and are averaged over 30 runs. The 30 industrial instances from Table 4.5 were used.

ID	First Improving	Best Improving
6	9.66%	15.42%
7	12.30%	20.69%
8	4.60%	8.09%
9	2.08%	18.51%
10	4.39%	5.03%
11	4.51%	48.05%
12	3.43%	50.90%
13	5.68%	69.04%
14	5.92%	70.39%
15	5.10%	67.53%
16	2.17%	22.03%
17	4.26%	68.19%
18	27.81%	85.10%
19	8.03%	69.86%
20	3.88%	66.93%
21	7.65%	74.19%
22	2.08%	37.80%
23	2.50%	62.93%
24	2.87%	67.75%
25	2.81%	72.47%
26	2.83%	71.45%
27	4.93%	81.34%
28	6.72%	79.20%
29	7.95%	79.44%
30	18.53%	94.13%

industrial instances for $20n$ bit flips. Due to the computational complexity of updating and enforcing the Tabu list there was no significant difference in the running time between the two versions of IRoTS using first and best improving search. Because the running times were not significantly different between IRoTS first improving and best improving search and IRoTS is very slow, we set a maximum time limit of 20 minutes per run rather than terminate the runs after a fixed number of bit flips. There were, however, significant differences in the average evaluation of the solutions found at the end of the runs. These values are reported in Table 4.13.

We ran a two-sided t-test comparing the mean evaluations of the solutions found at the end of the 30 runs from each version of IRoTS on each instance. The p-values from these tests are also reported in Table 4.13. We can see that in the case of IRoTS, the difference between mean evaluation is significant in 27 of the 30 instances. Of these significant cases, only in instances 3, 8, 16, 17, and 18 are the evaluations found by the first improving run lower than those found by the best improving runs. We conjecture that this is because the Tabu mechanism allows the greedy best improving search to undo mistakes in setting the critical values. Further testing of this hypothesis will be discussed more in the future work section.

4.2.3 Fast Descent Search with Hyperplane Initialization

Finally, we combine our hyperplane initialization with a fast initial descent. We used hyperplane initialization to construct 30 solutions for each of the benchmark instances in Table 4.5. We used these solutions to initialize our version of AdaptG2WSAT that uses first improving search for the initial descent. Again, each run was terminated after $20n$ bit flips with a maximum of 5,000,000 flips per run.

Table 4.14 reports the means and standard deviations of the evaluations of the best solution found by these runs. For comparison, we also report the means and standard deviations of evaluations of solutions found by identical runs using AdaptG2WSAT with best improving for the initial descent initialized by hyperplane initialization, AdaptG2WSAT

Table 4.13: Means and standard deviations of the evaluations of the best found solutions from 30 runs of Iterated Robust Tabu Search (IRoTS) using first improving (first) and best improving (best) local search. The p-values are from a two-sided t-test with an alternative hypothesis that the mean evaluations are different between the two local search types.

Instance	First Improving	Best Improving	p-value
1	8±2	7±1	0.059
2	4±2	4±1	0.183
3	17±2	19±3	<.001(*)
4	29±4	15±3	<.001(*)
5	169±13	6±1	<.001(*)
6	112±14	67±23	<.001(*)
7	176±10	129±14	<.001(*)
8	115±9	126±13	<.001(*)
9	252±25	172±30	<.001(*)
10	284±39	203±36	<.001(*)
11	892±49	264±28	<.001(*)
12	1,985±40	1,386±82	<.001(*)
13	1,014±26	821±61	<.001(*)
14	653±20	660±60	0.546
15	1,960±61	1,069±144	<.001(*)
16	36±5	336±22	<.001(*)
17	799±49	2,691±123	<.001(*)
18	198±11	210±16	<.001(*)
19	2,525±49	1,460±448	<.001(*)
20	2,870±64	2,505±159	<.001(*)
21	94±6	81±4	<.001(*)
22	4,760±69	2,863±360	<.001(*)
23	11,294±205	8,070±993	<.001(*)
24	11,371±120	8,082±1,428	<.001(*)
25	10,086±115	7,497±864	<.001(*)
26	11,992±173	10,996±284	<.001(*)
27	84,893±15,454	30,568±162	<.001(*)
28	312,803±28,461	95,233±26,855	<.001(*)
29	542,494±22,858	251,129±51,078	<.001(*)
30	1,092,200±16,364	768,341±44,946	<.001(*)

with best improving initial descent initialized by random solutions, and AdaptG2WSAT with first improving initial descent initialized by random solutions.

We first note that the RN+Best column of Table 4.14 is the default implementation of AdaptG2WSAT in the UBCSAT package. This implementation was only able to find the best evaluations, on average, on four instances: Instances 1, 2, 4, and 10. In each case, at least one other configuration of AdaptG2WSAT was able to find an equal mean evaluations. We can therefore state that on our benchmark industrial instances, at least one combination of our improvements is able to find equal or better solutions.

We see mixed results on the combination of improvements. No one combination is consistently better than the other. To determine the significance of our improvements we ran an analysis of variance test on the data in Table 4.14. We used two factors with two values each in this test: the initialization method (im), which was either hyperplane initialization or random, and the initial descent type (dt), which was either first improving or best improving. We ran the analysis of variance on an instance by instance basis, thus we had 30 samples from each combination of the two factors. Table 4.15 reports the p-values for the significance level of each factor (im and dt) and the p-value of the significance of an interaction effect between the factors (im:dt).

We will first examine the impact of the initialization method (im) in Table 4.14 and Table 4.15. There were 17 instances in which the initialization method was a significant factor. In all but two of these (instances 11 and 22), the runs initialized by hyperplane initialization were able to find significantly better solutions than those runs initialized by random solutions.

The impact of the descent type (dt) are not as significant. There were six instances in which the descent type was a significant factor. In three of these, first improving was the best (instances 13,19, and 29) and best improving found better solutions in the other three.

The impact of the interaction effect was similarly mixed. There were nine instances in which the interaction between the initialization method and descent type (im:dt in table 4.15 were significant). In four of these instances (instances 13,14,19, and 29) the combination

Table 4.14: Means and standard deviations of the evaluations of solutions found by four versions of AdaptG2WSAT: AdaptG2WSAT with first improving initial descent initialized by hyperplane initialization, AdaptG2WSAT with first improving initial descent initialized with random solutions, AdaptG2WSAT with best improving initial descent initialized with hyperplane solutions and AdaptG2WSAT with best improving initial descent initialized by random solutions. The best average evaluations are in bold.

Id	Hyperplane First Imp.	Random First Imp.	Hyperplane Best Imp.	Random Best Imp.
1	**5**±1	**5**±1	**5**±1	**5**±1
2	**1**±0	**1**±0	**1**±0	**1**±0
3	9±1	**8**±2	**8**±1	9±1
4	**2**±1	**2**±1	**2**±1	**2**±1
5	**6**±1	**6**±2	**6**±1	7±3
6	**12**±3	**12**±3	13±3	13±3
7	**66**±10	74±10	69±8	75±9
8	**91**±7	**91**±8	94±7	92±7
9	**10**±4	**10**±3	**10**±3	11±4
10	97±18	**95**±23	96±20	**95**±20
11	177±12	**165**±10	192±16	168±16
12	**862**±39	976±87	904±38	969±58
13	**283**±30	352±39	415±46	332±43
14	**85**±16	167±24	100±15	150±20
15	968±72	979±74	**961**±59	1,032±79
16	**12**±0	16±2	**12**±0	16±2
17	580±76	516±61	**279**±63	737±52
18	95±6	94±6	**87**±6	90±5
19	**39**±6	168±58	42±7	266±60
20	1,093±73	1,852±86	**1,087**±54	1,833±102
21	27±3	**25**±3	27±2	26±3
22	1,849±173	2,301±105	**1,811**±159	2,202±137
23	**2,096**±284	5,962±357	2,255±259	6,023±332
24	5,238±314	**5,170**±283	5,186±342	5,369±401
25	**2,265**±141	6,019±304	2,442±164	6,075±355
26	5,388±62	10,165±273	**5,014**±74	10,351±212
27	17,547±1,163	18,665±992	**12,578**±432	21,408±1,198
28	3,270±454	29,628±1,607	**2,085**±272	30,519±2,293
29	**9,846**±470	34,284±467	9,927±280	35,515±396
30	14,022±623	34,657±534	**10,472**±919	35,439±707

Table 4.15: P-values from an analysis of variance test on the mean evaluations of solutions found by runs of AdaptG2WSAT with two factors: initialization method (im) and descent type (dt). There are two values for each factor, initialization method is either hyperplane initialization or random solutions and descent type is either first improving or best improving. There are 30 runs per configuration for a total of 90 runs for each instance (see Table 4.5 for a list of the instances

Id	Initialization Method	Descent Type	Initialization Method : Descent Type
1	0.819	0.819	0.819
2	0.319	0.319	0.319
3	0.948	0.648	0.397
4	0.096	0.922	0.922
5	0.219	0.036	0.154
6	0.799	0.309	1.000
7	<.001(*)	0.350	0.775
8	0.558	0.081	0.387
9	0.205	0.883	0.329
10	0.659	0.834	0.918
11	<.001(*)	0.001	0.021
12	<.001(*)	0.106	0.027
13	0.310	<.001(*)	<.001(*)
14	<.001(*)	0.807	<.001(*)
15	0.002	0.074	0.023
16	<.001(*)	0.498	0.416
17	<.001(*)	0.006	<.001(*)
18	0.301	<.001(*)	0.012
19	<.001(*)	<.001(*)	<.001(*)
20	<.001(*)	0.385	0.663
21	<.001(*)	0.239	0.213
22	<.001(*)	0.076	0.380
23	<.001(*)	0.055	0.384
24	0.349	0.235	0.044
25	<.001(*)	0.015	0.199
26	<.001(*)	0.005	<.001(*)
27	<.001(*)	<.001(*)	<.001(*)
28	<.001(*)	0.575	<.001(*)
29	<.001(*)	<.001(*)	<.001(*)
30	<.001(*)	<.001(*)	<.001(*)

of hyperplane initialization and first improving search found better solutions on average. In the other five instances (instances 17,26,27,28, and 30) the combination of hyperplane initialization and best improving search found better solutions.

We know that first improving search can drastically reduce the time to execute $20n$ bit flips from Table 4.11. However, hyperplane initialization requires more time than constructing a random solution due to the calculation of Walsh coefficients and hyperplane averages. Table 4.16 reports the means and standard deviations of the overall run times, including initialization, of the 30 runs per instance by the various configurations of AdaptG2WSAT used in Tables 4.14 and 4.15.

Hyperplane initialization does increase the amount of time required to execute a fixed number of bit flips. This is especially apparent in instances with a large number of clauses, e.g., instances 10 and 17. This is because we must process each clause to both generate Walsh coefficients and compute the hyperplane averages. Although this time is significant, we note that it is possible to improve upon the time required by our initialization method [31].

However, in many cases, especially those larger instances, hyperplane initialization can actually improve the mean run time of those runs using best improving search. We conjecture that this is because the initialization method will start the search closer to a local optimum. Table 4.12 shows the percentage of time spent in the initial descent on large instances dominates the search. By starting closer to a local optimum, the search can find a local optimum using less bit flips, thus transitioning to the second phase more quickly and using less overall time.

4.3 Summary

We have examined two improvements to SLS algorithms: hyperplane initialization and fast initial descent using first improving search. We have found that these improvements can increase both the quality of solutions found and the time taken to find them when compared to the performance of the standard implementation of AdaptG2WSAT. However, the best performing configuration of these improvements varies from instance to instance. These

Table 4.16: Means and standard deviations of the time to execute $20n$ bit flips by four versions of AdaptG2WSAT: AdaptG2WSAT with first improving initial descent initialized by hyperplane initialization, AdaptG2WSAT with first improving initial descent initialized with random solutions, AdaptG2WSAT with best improving initial descent initialized with hyperplane solutions and AdaptG2WSAT with best improving initial descent initialized by random solutions. Means are over 30 runs from each configuration on each of the 30 industrial instances from Table 4.5.

Id	Hyperplane First Imp.	Random First Imp.	Hyperplane Best Imp.	Random Best Imp.
1	0±0	0±0	0±0	0±0
2	0±0	0±0	0±0	0±0
3	1±0	0±0	1±0	0±0
4	0±0	0±0	0±0	0±0
5	5±0	0±0	5±0	0±0
6	0±0	0±0	0±0	0±0
7	0±0	0±0	0±0	0±0
8	3±0	1±0	3±0	1±0
9	1±0	0±0	1±0	0±0
10	68±1	6±1	67±1	6±1
11	1±0	0±0	1±0	0±0
12	2±0	0±0	2±0	1±0
13	1±0	0±0	1±0	1±0
14	1±0	0±0	2±0	1±0
15	2±0	1±0	2±0	2±0
16	24±1	17±2	22±1	23±1
17	189±10	32±35	225±38	8±3
18	3±0	1±0	3±0	5±0
19	5±0	3±0	5±0	11±0
20	3±0	2±0	4±0	6±0
21	6±0	4±0	7±0	14±0
22	18±1	20±1	18±0	29±2
23	13±0	8±0	18±0	21±1
24	18±1	8±0	21±0	21±0
25	11±0	7±0	16±1	24±0
26	26±3	15±3	37±3	53±9
27	26±2	16±3	62±2	233±86
28	50±0	12±3	83±1	744±104
29	43±0	13±2	305±3	1,466±188
30	167±1	12±2	1,477±12	5,118±903

differences may indicate that there are more structural qualities to industrial instances than those captured by our generated instances.

Hyperplane initialization can provide an estimation of the correct assignment of the backbone variables with a high degree of accuracy on our generated instances. On those instances with the most industrial-like properties, hyperplane initialization can assign over 90% of the backbone variables correctly. We find that this not only increases the evaluation of the initial solution but also improves the ability of AdaptG2WSAT to quickly find a global optimum when it is initialized with hyperplane initialization.

We revisit the issue of best improving versus first improving search and conduct a new analysis on industrial instances. SLS algorithms can be split into two phases: the initial descent before a local optimum is found, and the exploration phase after a local optimum is found. We conducted our analysis by using both first improving and best improving search during the initial descent of AdaptG2WSAT. We found that although there can be significant differences in the evaluation of local optima found at the end of the initial descent, this makes little difference in the quality of solution found at the end of the exploration phase. Furthermore, first improving search is much faster on large industrial instances, giving up to a 4000× speedup in the overall run time on fixed length runs.

This gave us four configurations of AdaptG2WSAT: Hyperplane initialized AdaptG2WSAT with first improving initial descent, random initialized AdaptG2WSAT with first improving initial descent, hyperplane initialized AdaptG2WSAT with best improving initial descent, and random initialized AdaptG2WAT with best improving initial descent (the standard implementation in UBCSAT). The first three represent various configurations of our improvements. We found that in all 30 of the industrial instances we used as benchmarks, at least one of these configurations outperformed the standard implementation.

Given our results on generated instances, we would have expected the hyperplane initialized runs to outperform the randomly initialized runs. Although this is indeed what we saw in most cases, there were two instances, instances 11 and 21 from Table 4.5, where the randomly initialized runs found significantly better solutions on average than the hyperplane

initialized runs. Given our previous results on generated instances, we conjecture that these instances must contain some additional structure beyond the community structure, clause distribution and variable frequency found in our generated instances.

Nevertheless, we were able to improve quality of solutions found on almost all of our benchmark industrial instances by using hyperplane initialization and first descent search. In the previous chapter, we have identified why the search space is more difficult on instances having industrial-like properties. In this chapter, we have shown two methods by which we can improve the performance of SLS algorithms on instances with these properties. In the next chapter, we will discuss a method based on applying our hyperplane averages to complete solvers and show that this method can improve the ability of complete solvers to find globally optimal solutions on industrial instances.

Chapter 5

Improving Complete Solvers for SAT

In the previous chapter we discussed several improvements to SLS algorithms. These algorithms are suitable for applications where simply finding a solution with low evaluation is good enough [13, 66]. However, there are some cases when we need to know if the problem is satisfiable or not [50]. Although SLS solvers can determine if a problem is satisfiable by finding a satisfying solution, they typically cannot say anything in the unsatisfiable case. Complete SAT solvers, which are guaranteed to return a satisfying solution if one exists, are ideal for applications in which we need to determine the satisfiability of an instance.

The majority of complete SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [16], which is a backtracking branch-and-bound search based on resolution and other inference rules [55, 19]. There has been much research into the improvement of the DPLL algorithm, such as improved efficiency [55, 19], better conflict detection [6], and improved heuristics for ordering the variables assignments, which dictate the search paths explored [8]. However, rather than focus on improving DPLL or an existing solver directly, we take a different approach.

We use our method of efficiently computing hyperplane averages (See Section 3 in Chapter 4) to reduce the search space of the original problem as a preprocessing step prior to running a complete solver. In Chapter 3, we discussed how the backbone variables define a hyperplane in the search space. In Chapter 4, we have shown how hyperplane averages can provide a good estimation of the correct assignment of backbone variables. Our hyperplane reduction method will use these estimations to produce a new problem from the original by fixing a parameterized number of variables to the best guess truth assignment based on hyperplane averages. Although the use of preprocessors is common practice in solving SAT instances, our method is unique in that we directly use the hyperplane averages to heuristically chose a hyperplane and confine the search to that subspace.

SatELite is perhaps the most widely used preprocessor and incorporates a number of techniques to reduce both the number of variables and the number of clauses in the original formula [18]. The methods used in SatELite and other preprocessors ensure that the reduced formula will be satisfiable if and only if the original formula is satisfiable [18].

Our method differs from SatELite in this regard. We will use hyperplane averages to heuristically chose a subspace that we hope contains an optimal solution. Because our hyperplane averages can only estimate the location of the optimal solution, we cannot be sure a globally optimal solution is contained within this subspace. If a complete solver finds a satisfying solution within the reduced subspace, then the original problem is satisfiable. However, if a solver finds that our reduced problem is unsatisfiable, the original problem may still be satisfiable as it is possible that a global optimum exists outside of the subspace. In this case, we can either perform another reduction to a different hyperplane or run the complete solver on the original problem.

This strategy is effective and has proven to be one of the best strategies in the 2013 SAT Competition in the Application SAT track [68][2]. In the application track, consisting of real-world applications of SAT, our hyperplane reduction method paired with the MiniSAT complete solver [19] was able to solve 113 out of the 150 industrial instances in the track. This was good enough to rank our solver in 3rd place, out of a total of 31 entrants, many of which are considered state-of-the-art in complete solvers for SAT [68, 8, 6].

We first will describe our hyperplane reduction method with some empirical results to demonstrate its effectiveness at both reducing the original problem and finding hyperplanes containing optimal solutions. We will then present our results from the 2013 SAT Competition.

---

[2]Due to a technicality our entry was disqualified because the word 'unsatisfiable' was present in our final answer for three satisfiable problems, despite the fact that our solver produced verifiable satisfying solutions for these problems.

5.1 Reducing Problem Size with Hyperplane Averages

The goal of hyperplane reduction is to select a single hyperplane that not only contains the global optimum, but also significantly reduces the search space. The hypothesis is that by reducing the search space, we make the job of finding a satisfying solution (or determining the space does not contain one) easier for a complete solver. This is the same hypothesis that motivated preprocessors such as SatELite. This strategy has been proven effective for the state-of-the-art generation of DPLL-based solvers [18, 67].

As we have shown in Chapter 4, we were able to generate solutions to our industrial-like generated instances with over 90% of the backbone variables correctly assigned using hyperplane averages. Initializing search with these solutions led to a higher probability of finding a global optimum. When using hyperplane averages to initialize search on large, industrial instances, the search was able to find better solutions on average. We will use a similar technique as a preprocessing step to reduce the size of industrial instances before passing them off to a complete solver.

In Chapter 4, Section 3, we show how the Walsh coefficients can be used to efficiently compute the average evaluation of solutions contained within a hyperplane [82, 31, 83]. The hyperplanes we chose for our initialization method were based on the variables in each clause. For example, on an instance with $n = 6$ variables, if the first clause was $C_1 = (x_1 \vee \neg x_3 \vee x_4)$ then we would compute the hyperplanes defined by all possible combinations of truth assignments over x_1, x_3, and x_4, which we refer to as the fixed variables. These would be $0*00**$, $0*01**$, $0*10**$, $0*11**$, $1*00**$,$1*01**$,$1*10**$,$1*11**$, where '$*$' denotes a free variable. Thus, for each clause of size k, we have the average evaluation for 2^k hyperplanes.

Our strategy is to select a single hyperplane, for example $11*00*$, and reduce the original problem by eliminating the fixed variables and any clauses that are satisfied by them. For example using hyperplane $11*00*$, if our original problem was

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_2 \vee \neg x_4 \vee x_6)$$

it would be reduced to

$$(\neg x_3 \vee x_5$$

A satisfying solution to the reduced problem would be setting $x_3 = 0$ or $x_5 = 1$. To find the satisfying solution to the original problem, we can replace the free variables with the assignments found to satisfy the reduced problem, giving us 110001.

Computing the averages over the hyperplanes corresponding to the clauses works well for our hyperplane initialization. Although using the clause-based hyperplanes to vote on a promising initial assignment works well for hyperplane initialization, in hyperplane reduction we are not constructing a solution. Furthermore, we want to fix the variables that have the greatest potential to reduce the size of the original formula.

Therefore, instead of using the hyperplanes defined by the clauses, we compute the averages of the hyperplanes corresponding to fixing the most frequent variables. The rationale is that we can eliminate the most clauses by fixing the variables that appear in the most clauses. In practice, we first compute the frequency of occurrence of each variable in the original instance. We then select the four most frequently occurring variables. We choose four variables as empirically it gave us the best results. If we choose less variables, we do not gain as big of a reduction as the resulting hyperplanes will contain more solutions. If we choose more variables, hyperplanes become more restrictive. Although this results in better reductions, we also lessen the probability that the hyperplane contains a global optimum.

Once the four variables are chosen, we compute the average evaluations of the 16 hyperplanes associated with the 16 possible truth assignments to the four variables and then select the hyperplane with the best average evaluation. The original problem is then reduced as described above by removing the fixed variables and any clauses that are satisfied under their truth assignments.

To evaluate the ability of hyperplane reduction to reduce the problem size, both in terms of n and m, we selected 100 problems from the application track of the 2011 SAT competition. These problems were selected to represent a range of both size and application. We select

different problems from those in Table 4.5 because many of the industrial problems used in Chapter 2 are from the MAX-SAT 2012 competition [52].

These problems were contributed to the 2012 MAX-SAT competition by Safarpour and represent an application of MAX-SAT to circuit debugging and analysis [13, 66]. Through private correspondence with the contributor, we have learned that these problems are unsatisfiable. Although they are extremely difficult for MAX-SAT solvers, indeed the global optimum is unknown for many of these problems, they are quite easily determined to be unsatisfiable by a SAT complete solver due to obvious contradictions in the formulas. Therefore, these problems would not be suitable for testing a SAT solver and we chose a different set to represent applications of SAT.

We ran SatELite on the 100 problems that we selected from the SAT competition. Of these, 3 were solved by simply reducing the problems so we eliminated these from our set leaving us with 97 reduced problems. We then ran our hyperplane reduction algorithm on the 97 reduced problems. To further reduce our hyperplane reduced problems, we then ran SatELite again on the hyperplane reduced formulas. Therefore, our preprocessing pipeline is as follows.

```
Original Problem -> SatELite -> Hyperplane Reduction -> SatELite
```

Table 5.1 reports the number of variables n and the number of clauses m on the original problem, the initial SatELite reduction and the hyperplane reduction. We also report the percentage of the reduction from the original problem for both n and m on the two reductions. In all cases, hyperplane reduction+SatELite is able to reduce the problem further than just running SatELite.

Table 5.1: The number and percentage of variables (n) and clauses (m) reduced by SatELite and SatELite+hyperplane reduction for 97 industrial instances.

Instance	Reduction	n (% reduced)	m (% reduced)
driverlog3	Original	170	1559
	SatELite	78(54.12%)	834(46.5%)
	Hyperplane	10(48.24%)	136(37.78%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
bart17	Original	231	1166
	SatELite	11(95.24%)	11(99.06%)
	Hyperplane	26(83.98%)	175(84.05%)
aes-32-1	Original	300	1016
	SatELite	154(48.67%)	356(64.96%)
	Hyperplane	8(46%)	52(59.84%)
rovers1	Original	439	5423
	SatELite	292(33.49%)	3,732(31.18%)
	Hyperplane	4(32.57%)	90(29.52%)
aes-32-2	Original	504	1840
	SatELite	206(59.13%)	408(77.83%)
	Hyperplane	8(57.54%)	64(74.35%)
aes-64-1	Original	596	2780
	SatELite	196(67.11%)	532(80.86%)
	Hyperplane	20(63.76%)	160(75.11%)
vmpc-25	Original	625	76775
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.36%)	972(98.73%)
aes-32-3	Original	708	2664
	SatELite	258(63.56%)	460(82.73%)
	Hyperplane	8(62.43%)	64(80.33%)
vmpc-29	Original	841	120147
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.52%)	1,132(99.06%)
ferry5	Original	984	8702
	SatELite	350(64.43%)	405(95.35%)
	Hyperplane	4(64.02%)	138(93.76%)
vmpc-32	Original	1024	161664
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.61%)	1,252(99.23%)
dp04s04	Original	1075	3152
	SatELite	844(21.49%)	2,046(35.09%)
	Hyperplane	163(6.33%)	808(9.45%)
vmpc-34	Original	1156	194072
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.65%)	1,332(99.31%)
vmpc-35	Original	1225	211785
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.67%)	1,372(99.35%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
vmpc-36	Original	1296	230544
	SatELite	0(100%)	0(100%)
	Hyperplane	4(99.69%)	1,412(99.39%)
rbcl-xits-15	Original	2384	164746
	SatELite	827(65.31%)	894(99.46%)
	Hyperplane	5(65.1%)	3,189(97.52%)
rbcl-xits-18	Original	2888	218530
	SatELite	961(66.72%)	1,043(99.52%)
	Hyperplane	5(66.55%)	3,488(97.93%)
ndhf-xits	Original	4020	466486
	SatELite	1,396(65.27%)	1,895(99.59%)
	Hyperplane	5(65.15%)	5,943(98.32%)
abb9-c	Original	6228	484871
	SatELite	16(99.74%)	5,775(98.81%)
	Hyperplane	4(99.68%)	1,213(98.56%)
vda-gr-rcs-w9	Original	6498	130997
	SatELite	763(88.26%)	771(99.41%)
	Hyperplane	4(88.2%)	744(98.84%)
dp10s10	Original	7759	23004
	SatELite	5,919(23.71%)	14,062(38.87%)
	Hyperplane	213(20.97%)	1,183(33.73%)
E05F18	Original	7794	126826
	SatELite	1,181(84.85%)	3,178(97.49%)
	Hyperplane	20(84.59%)	3,612(94.65%)
E04F19	Original	9044	295685
	SatELite	1,018(88.74%)	1,968(99.33%)
	Hyperplane	20(88.52%)	10,487(95.79%)
E02F20	Original	10420	395383
	SatELite	1,432(86.26%)	3,976(98.99%)
	Hyperplane	20(86.07%)	12,566(95.82%)
E04F20	Original	10420	484053
	SatELite	1,733(83.37%)	19,397(95.99%)
	Hyperplane	56(82.83%)	25,652(90.69%)
E05F20	Original	10420	481346
	SatELite	1,512(85.49%)	12,372(97.43%)
	Hyperplane	20(85.3%)	15,701(94.17%)
E02F22	Original	13574	1301188
	SatELite	2,126(84.34%)	39,135(96.99%)
	Hyperplane	20(84.19%)	39,612(93.95%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
abb9-tr	Original	14013	481761
	SatELite	2,243(83.99%)	40,368(91.62%)
	Hyperplane	4(83.96%)	604(91.5%)
gss-14-s100	Original	31229	93855
	SatELite	18,213(41.68%)	39,865(57.52%)
	Hyperplane	442(40.26%)	1,779(55.63%)
gss-16-s100	Original	31248	93904
	SatELite	17,912(42.68%)	38,462(59.04%)
	Hyperplane	406(41.38%)	1,705(57.23%)
gss-19-s100	Original	31435	94548
	SatELite	17,753(43.52%)	37,719(60.11%)
	Hyperplane	448(42.1%)	1,900(58.1%)
gss-21-s100	Original	31613	95104
	SatELite	17,774(43.78%)	37,625(60.44%)
	Hyperplane	447(42.36%)	1,823(58.52%)
gss-22-s100	Original	31616	95110
	SatELite	17,752(43.85%)	37,482(60.59%)
	Hyperplane	445(42.44%)	1,796(58.7%)
gss-27-s100	Original	31951	96161
	SatELite	17,676(44.68%)	36,831(61.7%)
	Hyperplane	496(43.13%)	2,021(59.6%)
mizh-sha0-36-4	Original	50073	210235
	SatELite	29,500(41.09%)	89,892(57.24%)
	Hyperplane	24(41.04%)	128(57.18%)
sha0-36-5	Original	50073	210223
	SatELite	29,496(41.09%)	89,877(57.25%)
	Hyperplane	24(41.05%)	128(57.19%)
pathways-17	Original	53919	308235
	SatELite	12,954(75.98%)	35,636(88.44%)
	Hyperplane	Hyperplane Unsatisfiable	
ibm-2004-01-k90	Original	64699	276210
	SatELite	29,736(54.04%)	63,664(76.95%)
	Hyperplane	1,027(52.45%)	6,636(74.55%)
slpaes-top24	Original	65983	209942
	SatELite	41,792(36.66%)	86,618(58.74%)
	Hyperplane	1,428(34.5%)	7,416(55.21%)
AProVE11-15	Original	66715	228274
	SatELite	31,707(52.47%)	55,377(75.74%)
	Hyperplane	853(51.2%)	5,437(73.36%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
md5-48-3	Original	66892	279258
	SatELite	40,203(39.9%)	121,909(56.35%)
	Hyperplane	4(39.89%)	144(56.29%)
AProVE11-09	Original	68779	234693
	SatELite	34,721(49.52%)	62,554(73.35%)
	Hyperplane	9(49.5%)	4,071(71.61%)
pipesworld-12	Original	68952	1029036
	SatELite	7,433(89.22%)	11,486(98.88%)
	Hyperplane	Hyperplane Unsatisfiable	
slpaes-top25	Original	71356	227126
	SatELite	45,155(36.72%)	93,590(58.79%)
	Hyperplane	1,477(34.65%)	7,674(55.42%)
slpaes-top26	Original	76943	245006
	SatELite	48,647(36.78%)	100,860(58.83%)
	Hyperplane	1,526(34.79%)	7,938(55.59%)
AProVE11-16	Original	84025	282766
	SatELite	41,359(50.78%)	81,027(71.34%)
	Hyperplane	668(49.98%)	4,993(69.58%)
slpaes-top28	Original	88763	282870
	SatELite	56,013(36.9%)	116,257(58.9%)
	Hyperplane	1,624(35.07%)	8,459(55.91%)
slpaes-top29	Original	94998	302862
	SatELite	59,882(36.96%)	124,349(58.94%)
	Hyperplane	1,673(35.2%)	8,721(56.06%)
AProVE11-11	Original	96526	325263
	SatELite	46,397(51.93%)	93,226(71.34%)
	Hyperplane	70(51.86%)	1,801(70.78%)
traffic-r	Original	98651	3362666
	SatELite	1,171(98.81%)	6,648(99.8%)
	Hyperplane	541(98.26%)	157,490(95.12%)
slpaes-top30	Original	101451	323566
	SatELite	63,881(37.03%)	132,723(58.98%)
	Hyperplane	1,722(35.34%)	8,976(56.21%)
SAT-dat.k80-04	Original	104450	457628
	SatELite	62,075(40.57%)	195,213(57.34%)
	Hyperplane	Hyperplane Unsatisfiable	
UTI-10-5t1	Original	108508	527595
	SatELite	72,472(33.21%)	297,745(43.57%)
	Hyperplane	46(33.17%)	382(43.49%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
sokoban.050	Original	109684	1767096
	SatELite	37,888(65.46%)	771,199(56.36%)
	Hyperplane	420(65.07%)	4,382(56.11%)
velev-pipe	Original	118038	8780591
	SatELite	12,110(89.74%)	46,375(99.47%)
	Hyperplane	4(89.74%)	152,920(97.73%)
UR-10-10p1	Original	131228	635871
	SatELite	65,884(49.79%)	238,265(62.53%)
	Hyperplane	52(49.75%)	406(62.47%)
sokoban.060	Original	131484	2120376
	SatELite	41,728(68.26%)	870,079(58.97%)
	Hyperplane	420(67.94%)	4,382(58.76%)
12pipe-bug4	Original	138563	4675040
	SatELite	57,369(58.6%)	120,410(97.42%)
	Hyperplane	Hyperplane Unsatisfiable	
12pipe-bug6	Original	138795	4671352
	SatELite	57,611(58.49%)	123,494(97.36%)
	Hyperplane	1,272(57.58%)	20,051(96.93%)
pipesworld-18	Original	141856	2742601
	SatELite	6,029(95.75%)	15,401(99.44%)
	Hyperplane	Hyperplane Unsatisfiable	
traffic-3	Original	142205	1312352
	SatELite	795(99.44%)	1,100(99.92%)
	Hyperplane	143(99.34%)	39,715(96.89%)
sokoban.070	Original	153284	2473656
	SatELite	45,568(70.27%)	968,959(60.83%)
	Hyperplane	420(70%)	4,382(60.65%)
smtlib-share	Original	164758	456350
	SatELite	136,895(16.91%)	316,415(30.66%)
	Hyperplane	500(16.61%)	3,879(29.81%)
sokoban.080	Original	175084	2826936
	SatELite	49,408(71.78%)	1,067,839(62.23%)
	Hyperplane	420(71.54%)	4,382(62.07%)
smtlib-src-wget	Original	180045	538304
	SatELite	119,072(33.87%)	260,573(51.59%)
	Hyperplane	625(33.52%)	2,949(51.05%)
ibm-2002-30r-k85	Original	181484	890298
	SatELite	93,897(48.26%)	214,240(75.94%)
	Hyperplane	1,482(47.44%)	10,250(74.78%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
SAT-dat.k85	Original	181484	890298
	SatELite	93,897(48.26%)	214,240(75.94%)
	Hyperplane	1,482(47.44%)	10,250(74.78%)
ibm-2002-21r-k95	Original	191522	788339
	SatELite	133,054(30.53%)	355,982(54.84%)
	Hyperplane	86(30.48%)	1,368(54.67%)
UCG-15-10p1	Original	200003	1019221
	SatELite	97,502(51.25%)	352,067(65.46%)
	Hyperplane	58(51.22%)	438(65.41%)
ibm-2004-23-k100	Original	207606	861175
	SatELite	136,410(34.29%)	339,304(60.6%)
	Hyperplane	95(34.25%)	1,395(60.44%)
AProVE11-02	Original	214734	743081
	SatELite	107,919(49.74%)	190,262(74.4%)
	Hyperplane	70(49.71%)	6,570(73.51%)
q-query-3	Original	218792	1020908
	SatELite	190,030(13.15%)	718,824(29.59%)
	Hyperplane	96(13.1%)	11,817(28.43%)
ACG-15	Original	218990	943377
	SatELite	82,534(62.31%)	272,071(71.16%)
	Hyperplane	85(62.27%)	503(71.11%)
UCG-20-5p1	Original	224986	1204430
	SatELite	103,738(53.89%)	370,059(69.28%)
	Hyperplane	60(53.86%)	474(69.24%)
AProVE11-07	Original	233177	782677
	SatELite	117,732(49.51%)	232,867(70.25%)
	Hyperplane	34(49.5%)	6,236(69.45%)
UTI-20-10t1	Original	238008	1278025
	SatELite	168,324(29.28%)	707,581(44.63%)
	Hyperplane	46(29.26%)	409(44.6%)
pipesworld-27	Original	249618	5949456
	SatELite	11,561(95.37%)	25,916(99.56%)
	Hyperplane	Hyperplane Unsatisfiable	
UTI-20-10p1	Original	260342	1391257
	SatELite	121,668(53.27%)	432,772(68.89%)
	Hyperplane	56(53.24%)	464(68.86%)
smtlib-libsmbclient	Original	266663	768347
	SatELite	217,518(18.43%)	521,921(32.07%)
	Hyperplane	345(18.3%)	3,183(31.66%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
smtlib-rfunit	Original	266805	720122
	SatELite	266,649(0.06%)	718,511(0.22%)
	Hyperplane	7(0.06%)	166(0.2%)
AProVE11-10	Original	289828	1008603
	SatELite	146,944(49.3%)	253,446(74.87%)
	Hyperplane	807(49.02%)	20,188(72.87%)
AProVE11-06	Original	302076	1039056
	SatELite	142,963(52.67%)	259,987(74.98%)
	Hyperplane	Hyperplane Unsatisfiable	
ipc5	Original	314455	2920820
	SatELite	5,984(98.1%)	10,079(99.65%)
	Hyperplane	Hyperplane Unsatisfiable	
openstacks	Original	324116	1643601
	SatELite	123,627(61.86%)	314,422(80.87%)
	Hyperplane	64,984(41.81%)	441,590(54%)
smtlib-servers	Original	360364	1076507
	SatELite	190,314(47.19%)	368,673(65.75%)
	Hyperplane	1,997(46.63%)	7,916(65.02%)
traffic-r-uc	Original	467551	4467733
	SatELite	13,985(97.01%)	14,701(99.67%)
	Hyperplane	36(97%)	4,982(99.56%)
transport.2city040	Original	580875	3140965
	SatELite	226,183(61.06%)	705,632(77.53%)
	Hyperplane	21,290(57.4%)	145,167(72.91%)
blocks37-1.150	Original	645431	11795567
	SatELite	155,651(75.88%)	3,094,125(73.77%)
	Hyperplane	11,400(74.12%)	101,906(72.9%)
blocks36-0.160	Original	652445	11706080
	SatELite	145,898(77.64%)	2,858,145(75.58%)
	Hyperplane	10,804(75.98%)	96,524(74.76%)
blocks36-0.170	Original	693135	12437620
	SatELite	145,898(78.95%)	2,858,145(77.02%)
	Hyperplane	10,804(77.39%)	96,524(76.24%)
transport.city060	Original	723130	3869060
	SatELite	222,059(69.29%)	452,886(88.29%)
	Hyperplane	Hyperplane Unsatisfiable	
blocks36-0.180	Original	733825	13169160
	SatELite	145,898(80.12%)	2,858,145(78.3%)
	Hyperplane	10,804(78.65%)	96,524(77.56%)

Continued on next page

Table 5.1 – *Continued from previous page*

Instance	Reduction	n (% reduced)	m (% reduced)
transport.3city020	Original	756832	4116966
	SatELite	390,900(48.35%)	1,546,822(62.43%)
	Hyperplane	14,122(46.48%)	106,216(59.85%)
transport.city030	Original	945406	5079060
	SatELite	371,079(60.75%)	1,109,256(78.16%)
	Hyperplane	57,732(54.64%)	391,289(70.46%)
transport.city040	Original	1260306	6771840
	SatELite	442,679(64.88%)	1,120,116(83.46%)
	Hyperplane	57,732(60.29%)	391,289(77.68%)
clauses-8	Original	1461771	5687554
	SatELite	1,014,482(30.6%)	3,515,162(38.2%)
	Hyperplane	59,444(26.53%)	143,867(35.67%)
transport.city050	Original	1575206	8464620
	SatELite	514,279(67.35%)	1,130,976(86.64%)
	Hyperplane	57,732(63.69%)	391,289(82.02%)

Table 5.2 summarizes this result by reporting the mean and standard deviation percentage reduction over the original problem from both SatELite and SatELite+Hyperplane Reduction. We ran a paired t-test between the number of variables in the SatELite reductions and the number of variables in the SatELite+Hyperplane reductions. The alternative hypothesis in this test was that there were less variables in the SatELite+Hyperplane reduction. The results were significant at the 0.001 level. A similar test comparing the number of clauses also resulted in a p-value of $< .001$.

These results tell us that SatELite+Hyperplane reduction provides a significantly larger decrease to the problem size in both terms of the number of variables and the number of clause than SatELite alone. However, this reduction comes at a cost. By fixing the four most frequent variables to specific truth assignments, we limit the search to the free variables. If there is not a satisfying solution with these same truth assignments, then the search will not find it. We will next examine how often the reduced space contains a satisfying solution.

Table 5.2: The mean percentage of reduction in both the number of variables and the number of clauses on 90 industrial problems (See Table 5.1) by SatELite alone and by SatELite with hyperplane reduction. The p-values are the result of a paired t-test with the alternative hypothesis that there are less variables or clauses in the Hyperplane+SatELite reductions than the SatELite reductions.

Reduction Method	Reduction of Variables	Reduction of Clauses
SatELite Reduction	40.40 ± 24.87%	27.18 ± 22.84%
SatELite+Hyperplane Reduction	48.00% ± 28.62	36.87 ± 29.21%
p-value	< 0.001	< 0.001

5.1.1 Ranked Hyperplanes and Global Optima

To examine how often our hyperplane reduced problems contain a satisfying solution, we need a complete solver that can efficiently search the subspaces. We will use the MiniSAT complete solver to decide the satisfiability of the hyperplane reduced problem [19]. MiniSAT is a lightweight, open source solver designed for use in experimental SAT research.

For each of the 97 industrial instances, we chose the four most frequent variables and computed the average evaluations over the 16 hyperplanes corresponding to the 16 possible truth assignments to these variables. We then sorted the hyperplanes from highest average evaluation to lowest average evaluation and ran MiniSAT for 20 minutes on the reduced problems from the eight hyperplanes with the highest averages. We recorded the result from each run of MiniSAT and report them in Table 5.3.

Table 5.3: The outcome of 20 minute runs of MiniSAT on the eight top ranked hyperplanes for 97 industrial SAT instances. The hyperplanes were chosen by fixing the four most frequent variables in each problem and ranked from highest to lowest by the average evaluation of the solutions within each hyperplane. MiniSAT has three possible outcomes: S, U and I. S (shaded cells) means a satisfying solution was found, U means the hyperplane does not contain a satisfying solution, and I means MiniSAT timed out before deciding on the satisfiability of the hyperplane.

Instance	1	2	3	4	5	6	7	8
12pipe-bug4	S	U	U	U	I	I	I	I
12pipe-bug6	U	U	U	S	I	I	I	I
aaai10-pathways-17	I	I	I	I	I	I	I	I
aaai10-pipesworld-12	I	U	I	I	I	S	U	I
aaai10-pipesworld-18	I	I	I	I	I	I	S	I

Continued on next page

Table 5.3 – *Continued from previous page*

Instance	1	2	3	4	5	6	7	8
aaai10-pipesworld-27	I	I	I	I	I	I	U	I
aaai10-ipc5	I	I	I	S	S	I	S	S
abb9-c	I	I	I	I	I	I	I	I
abb9-tr	S	S	S	S	S	S	S	S
ACG-15	S	I	S	S	I	I	I	I
aes-32-1	U	U	U	S	U	U	U	U
aes-32-2	U	U	U	U	S	U	U	U
aes-32-3	S	U	I	U	I	U	U	I
aes-64-1	I	U	I	U	U	U	U	I
AProVE11-02	S	S	S	S	S	S	S	S
AProVE11-06	I	U	U	U	I	U	I	U
AProVE11-07	S	I	I	I	S	I	I	I
AProVE11-09	S	S	S	S	S	S	S	S
AProVE11-10	U	U	U	U	U	S	U	U
AProVE11-11	U	U	U	U	U	U	U	U
AProVE11-15	U	U	U	U	U	S	U	S
AProVE11-16	U	U	U	S	S	U	S	S
bart17	S	S	S	S	S	S	S	S
blocks36-0.160	S	S	S	S	S	S	S	S
blocks36-0.170	S	S	S	S	S	S	S	S
blocks36-0.180	S	S	S	S	S	S	S	S
blocks37-1.150	S	S	S	S	S	S	S	S
clauses-8	U	U	U	U	U	U	U	U
dp04s04	U	U	I	U	U	I	U	U
dp10s10	S	S	S	S	S	S	S	S
driverlog3	S	I	I	I	I	I	I	I
E02F20	S	S	S	S	S	S	S	S
E02F22	S	S	S	S	S	S	S	S
E04F19	S	S	S	S	S	S	S	S
E04F20	S	S	S	S	S	S	S	S
E05F18	S	S	S	S	U	S	S	S
E05F20	S	S	S	S	S	S	S	S
ferry5	S	U	U	U	U	I	I	I
gss-14-s100	U	U	U	U	U	U	U	U
gss-16-s100	U	U	U	U	U	U	U	U
gss-19-s100	U	S	U	U	U	U	U	U
gss-21-s100	U	U	U	U	U	U	U	S
gss-22-s100	U	U	U	U	S	U	U	U
gss-27-s100	I	I	I	I	I	I	I	I
ibm-2002-21r-k95	S	S	S	S	S	S	S	U
ibm-2002-30r-k85	S	S	S	S	I	S	I	S
ibm-2004-01-k90	S	S	I	S	S	S	S	S

Continued on next page

Table 5.3 – *Continued from previous page*

Instance	1	2	3	4	5	6	7	8
ibm-2004-23-k100	S	S	S	S	S	S	S	S
md5-48-3	S	S	S	S	S	S	S	S
mizh-sha0-36-4	S	S	S	S	S	S	S	S
ndhf-xits	I	I	I	I	I	I	I	I
openstacks	S	S	S	S	S	S	S	S
q-query-3	U	I	U	I	S	I	I	I
rbcl-xits-15	S	S	S	S	S	S	S	S
rbcl-xits-18	S	S	S	S	S	S	S	S
rovers1	S	I	I	I	S	I	I	I
SAT-dat.k80-04	I	I	I	I	I	I	I	I
SAT-dat.k85	S	S	S	S	I	S	I	S
sha0-36-5	S	S	S	I	S	S	S	S
slpaes-top24	I	I	I	I	I	I	I	S
slpaes-top25	S	I	I	S	I	S	I	I
slpaes-top26	S	I	I	I	I	S	S	I
slpaes-top28	S	S	S	S	S	S	I	S
slpaes-top29	S	S	S	S	I	S	S	S
slpaes-top30	I	I	S	S	S	S	S	S
smtlib-libsmbclient	S	S	S	S	S	S	S	S
smtlib-libsmbsharemodes	S	S	S	S	S	S	S	S
smtlib-rfunit	S	S	S	S	S	S	S	S
smtlib-servers	I	I	I	I	I	I	I	I
smtlib-src-wget	S	S	S	S	S	S	S	S
sokoban.050	I	I	I	I	I	I	I	I
sokoban.060	I	I	I	I	I	I	I	I
sokoban.070	I	I	I	I	I	I	I	I
sokoban.080	I	I	I	I	I	I	I	I
traffic-3	I	I	I	S	I	I	S	I
traffic-r	S	S	S	S	S	S	S	I
traffic-r-uc	S	I	S	S	I	S	S	S
transport.city060	I	S	U	U	I	S	U	U
transport.city030	I	I	S	I	I	I	I	I
transport.city040	I	I	I	I	I	I	I	I
transport.city050	I	I	S	I	I	I	I	I
transport.3city020	U	U	I	U	U	I	U	I
transport.2city040	U	I	U	U	U	I	S	U
UCG-15-10p1	S	S	I	S	I	I	S	I
UCG-20-5p1	S	S	S	S	S	I	S	S
UR-10-10p1	S	S	S	S	I	I	I	I
UTI-10-5t1	S	I	I	I	I	I	I	I
UTI-20-10p1	I	S	I	I	I	I	I	I
UTI-20-10t1	S	I	I	I	I	I	I	I

Continued on next page

Table 5.3 – *Continued from previous page*

Instance	1	2	3	4	5	6	7	8
vda-gr-rcs-w9	S	S	S	S	S	S	S	S
velev-pipe	S	U	U	U	U	U	U	U
vmpc-25	S	U	U	U	U	U	U	U
vmpc-29	S	U	U	U	U	U	U	U
vmpc-32	S	U	U	U	U	U	U	U
vmpc-34	I	U	U	U	U	U	U	U
vmpc-35	I	U	U	I	S	U	U	U
vmpc-36	I	I	I	I	I	U	U	U

There are three possible outcomes from a run of MiniSAT on a hyperplane reduction reporting in Table 5.3: Satisfiable, Unsatisfiable, and Indeterminate. Satisfiable means that MiniSAT found a satisfying solution in the hyperplane and that the original problem is satisfiable; these are the shaded cells. Unsatisfiable means that MiniSAT found that the hyperplane reduced problem is unsatisfiable. This, however, does not necessarily mean that the original problem is not satisfiable; only that the hyperplane does not contain a satisfying solution. Indeterminate means that MiniSAT timed out without determining if the reduction was satisfiable or not. Table 5.4 summarizes the results.

Of the 97 industrial instances in Table 5.3, the majority of rank one hyperplanes contain a satisfying solution. The second highest return value from MiniSAT is indeterminate. In the indeterminate cases, it is possible that the reduced problem based on the rank one hyperplane is satisfiable but MiniSAT was unable to find a satisfying solution in the 20 minutes allowed per run. We know for certain that the rank one hyperplane does not contain a satisfying solution in only 17 of the 97 instances.

In the cases where the rank one hyperplane is unsatisfiable, we cannot say for certain anything about the satisfiability of the original problem. Therefore, we have two options. We can either run MiniSAT on the original problem or we can run MiniSAT on a different hyperplane reduction. The second strategy does have some merit. For example, Table 5.3 shows that in some cases where the rank 1 hyperplane is unsatisfiable, e.g., gss-19-s100, the rank 2 or higher ranked hyperplanes are satisfiable.

Table 5.4: Number of Satisfiable, Unsatisfiable and Indeterminate cases after running MiniSAT for 20 minutes on hyperplane reduced problems. 16 hyperplanes averages were computed corresponding to the four most frequently occurring variable in each problem. These hyperplanes were then ranked by averages and we ran MiniSAT for 10 minutes on the reductions corresponding to the top eight hyperplanes for each problem. These results are a summary of the data in Table 5.3.

Hyperplane Ranking	1	2	3	4	5	6	7	8
Satisfiable	53	40	40	45	40	42	41	40
Unsatisfiable	17	26	25	24	20	20	25	21
Indeterminate	27	31	32	28	37	35	31	36

Of course, one of the 16 total hyperplanes *must* contain a satisfying solution if the original solution is satisfiable. The hyperplanes partition the solutions in the original search space into 16 mutually exclusive sets, the union of which equals the original space. Thus, with an unbounded runtime, it is possible to search the rank 1 hyperplane, until either a satisfying solution is found or MiniSAT determines the reduction is unsatisfiable. If a satisfying solution is found, we can stop and conclude that the original problem is satisfiable. Otherwise, we can run MiniSAT on the rank 2 hyperplane, until a satisfying or unsatisfying result is reached. We can then repeat this process on the higher ranked hyperplanes until either a satisfying solution is found or all hyperplanes have been found to be unsatisfiable, in which case we can conclude the original problem is unsatisfiable.

While theoretically this strategy would work, an unbounded run-time is an unreasonable assumption due to the NP-complete nature of SAT. Indeed, some of the problems in our benchmark set are unsolved. Therefore, we need to set an upper bound on the runtime allowed per hyperplane. We can of course, continue with this strategy, however if the time limit is reached on any of the 16 hyperplanes , we can no longer determine the unsatisfiability based on the results of the hyperplane reductions alone. This concept will be the basis of our strategy for combining hyperplane reduction with MiniSAT.

5.2 Hyperplane Reduced MiniSAT

We will now describe our strategy for combining hyperplane reduction and MiniSAT. We will refer to the entire procedure as Hyperplane Reduced MiniSAT (HRMS). HRMS was designed to be submitted to the 2013 SAT Competition [68], therefore some of the design decisions were made to abide by the competition rules. Specifically, the competition imposes a 5,000 second limit per instance.

Given the time limit, we would not have enough time to search all 16 hyperplanes even for 10 minutes a piece. Our strategy was to instead search as many hyperplanes as possible, while leaving 10 minutes for MiniSAT to run on the original problem. We gave each hyperplane an upper bound of 10 minutes. That is, if MiniSAT ran longer than this on any one hyperplane without determining the satisfiability of the reduction, the run was aborted and MiniSAT was run on the next hyperplane. If an unsatisfiable result was found, MiniSAT immediately ran on the next best ranked hyperplane. If a satisfying solution was not found with 10 minutes remaining in the run, MiniSAT aborted its current run and ran on the original unreduced problem for the remainder of the time.

Table 5.5 reports the results of our solver in the 2013 SAT competition. 31 other solvers were submitted to this track. The 'Virtual Best Solver' is a hypothetical solver that shows the results if the best runs on each instance were attributed to a single solver. The rankings were computed by determining how many of the 150 instances were solved by returning a solution that satisfies the original formula. Our solver, which we refer to here as HRMS, came in 3rd place in the ranking order based on the total number of solved problems.

Table 5.6 reports the time in seconds used by each solver on all of the instances (Total Time) and the median time in seconds spent by each solver on the 150 instances (Median Time). If the rankings were done on the total amount of time spent solving the 150 instances or the average time spent per instance, our solver would have ranked second. This tells us two things: 1. our hyperplane reduction strategy is competitive with the state-of-the-solvers

Table 5.5: Results of the application+SAT track of the 2013 SAT Competition. There were 31 solvers that were ranked by the number of instances they solved out of the 150 instances in the competition. Our solver, Hyperplane Reduced MiniSAT (HRMS), ranked 3rd.

Ranking	Solver	Number of Solved Instances
0	Virtual Best Solver (VBS)	149
1	Lingeling aqw	119
2	ZENN 0.1.0	113
3	HRMS	111
4	satUZK 48	110
5	Riss3g 3g	108
6	Lingeling 587f	107
7	CSHCapplLG	106
8	gluebit-lgl 1.0	105
9	MIPSat	105
10	forl nodrup	104
11	MIPSat	104
12	glucose 2.3	103
13	glue-bit 1.0	102
14	Solver43b b	102
15	Riss3g	102
16	strangenight satcomp11-st	102
17	Solver43a a	101
18	BreakIDGlucose 1	100
19	glueminisat 2.2.7j	100
20	CSHCapplLC	100
21	relback v1.1	99
22	gluH 1.0	99
23	ShatterGlucose 1	99
24	gluH-simp 1.0	96
25	Nigma 1.0	96
26	GlucoRed r531	93
27	RSeq V 1.1	93
28	Nigma-NoPB 1.0	92
29	minipure 1.0.1	84
30	sattimeRelbackShr SRShr1.0	73
31	SAT4J SAT COMPETITION 2013	70

at solving industrial instances of SAT. 2. Our strategy is also relatively fast in terms of the running time required to solve industrial instances.

These results are even more intriguing when we also consider that we used MiniSAT as our solver [19]. MiniSAT is a fully featured SAT Solver, but it has not been competitive with more recent SAT solvers for several years [19, 67]. In the last SAT competition in 2011, Glucose was the winner of the application track [6, 67]. Not surprisingly, many of the entrants to the 2013 competition are based on Glucose, e.g., GlucoRed, ShatterGlucose, glueminisat. Not only do we solve more instances than Glucose using an unmodified version of MiniSAT, but we also outperform the Glucose variants.

The two solvers that were able to solve more than HRMS are ZENN and Lingeling [8]. ZENN is a new solver that incorporates a phase shift paradigm [86]. ZENN uses two different phases in the search: one phase that is similar to MiniSAT and a second phase based on a Glucose-like search. ZENN alternates between these two search methods after a threshold number of restarts have been reached [86].

Lingeling is based on the PrecoSAT solver [7] which incorporates preprocessing techniques of reducing the number of clauses and variables into the search itself. In addition, there are multiple improvements to the search procedure used in PrecoSAT and versions of Lingeling that were entered in previous competitions [9].

To examine the difference between the top three solvers, we looked at the result for each of the 150 instances in the SAT track and the time spent on each instance. These values are reported in Table 5.7.

Table 5.7: The results of the top three solvers from the 2013 SAT competition. The results are reported as either S for satisfiable or I for indeterminate in the case the solver timed out after 5,000 seconds. The value in parentheses is the CPU time in seconds that each solver spent on the instance.

Instance	HRMS	ZENN	Lingeling
001a	I(5000)	I(5000)	S(1608)
001b	S(1570)	I(5000)	I(5000)
001c	I(5000)	I(5000)	S(475)
001d	S(936)	S(3484)	I(5000)

Continued on next page

Table 5.7 – *Continued from previous page*

Instance	HRMS	ZENN	Lingeling
002a	S(26)	I(5000)	I(5000)
002b	I(5000)	I(5000)	S(2587)
002c	S(2375)	I(5000)	I(5000)
003a	I(5000)	I(5000)	I(5000)
003b	I(5000)	I(5000)	I(5000)
003c	I(5000)	I(5000)	S(2990)
003d	I(5000)	I(5000)	S(4745)
003e	S(3086)	I(5000)	I(5000)
004a	S(2559)	I(5000)	I(5000)
004b	S(899)	S(1162)	S(622)
005a	I(5000)	S(1103)	I(5000)
005b	S(532)	S(3281)	S(328)
005c	S(1219)	I(5000)	S(1020)
006a	I(5000)	S(1053)	I(5000)
006b	I(5000)	S(1214)	S(114)
007a	I(5000)	S(3351)	I(5000)
007b	I(5000)	S(1714)	I(5000)
007c	I(5000)	I(5000)	S(3216)
007d	S(3355)	S(576)	S(97)
008a	S(3946)	I(5000)	I(5000)
008b	I(5000)	I(5000)	S(4059)
008c	I(5000)	I(5000)	S(1180)
009a	S(463)	I(5000)	I(5000)
009b	S(2250)	I(5000)	S(3160)
009c	S(42)	S(4572)	I(5000)
010	I(5000)	I(5000)	I(5000)
9vliw-bug1	S(73)	S(157)	S(245)
9vliw-bug10	S(70)	S(79)	S(193)
9vliw-bug4	S(71)	S(157)	S(182)
9vliw-bug6	S(67)	S(67)	S(115)
9vliw-bug7	S(101)	S(63)	S(200)
9vliw-bug8	S(115)	S(171)	S(163)
9vliw-bug9	S(108)	S(70)	S(77)
aaai10-pathways-17	I(5000)	S(2898)	S(622)
aaai10-ipc5	S(74)	S(111)	S(225)
ACG-15-10p1	S(1867)	S(1536)	S(358)
ACG-20-5p1	S(1541)	S(1077)	S(330)
aes-16	I(5000)	S(601)	S(2206)
aes-24-2	S(927)	S(4741)	I(5000)
aes-24-4	S(2576)	S(3224)	S(2904)
aes-32-1	S(2125)	S(2378)	S(591)
aes-32-2	S(110)	S(4102)	S(1253)

Continued on next page

Table 5.7 – *Continued from previous page*

Instance	HRMS	ZENN	Lingeling
aes-64	I(5000)	S(516)	S(1340)
AProVE07-11	S(110)	S(63)	S(533)
AProVE09-06	S(992)	S(791)	S(847)
b04-s-2	I(5000)	I(5000)	I(5000)
b04-s-pre	S(539)	S(225)	S(388)
b04-s	I(5000)	S(2942)	S(2235)
blocks-130	S(159)	S(118)	S(72)
blocks-150	S(44)	S(76)	S(53)
E02F20	S(282)	S(86)	S(731)
E02F22	S(730)	S(1133)	S(451)
E04F19	S(172)	S(58)	S(44)
esawn-uw3	S(381)	I(5000)	S(199)
grid-3.045	S(1144)	S(512)	S(450)
grid-3.055	S(59)	S(465)	S(102)
grid-3.065	S(107)	S(223)	S(503)
grieu-vmpc-31	S(277)	S(1250)	S(1765)
gss-17-s100	S(61)	S(190)	S(182)
gss-18-s100	S(358)	S(308)	S(221)
gss-19-s100	S(1038)	S(915)	S(627)
gss-20-s100	S(1197)	S(720)	S(383)
gss-21-s100	S(348)	S(4959)	S(574)
gss-22-s100	S(1313)	I(5000)	I(5000)
gss-23-s100	I(5000)	I(5000)	I(5000)
gss-24-s100	S(2827)	I(5000)	I(5000)
gss-25-s100	I(5000)	I(5000)	I(5000)
hitag2-7	I(5000)	I(5000)	S(70)
hitag2-8	I(5000)	I(5000)	S(1375)
itox-vc1033	S(5)	S(2)	S(31)
itox-vc1130	S(7)	S(2)	S(39)
md5-47-1	S(53)	S(130)	S(59)
md5-47-2	S(44)	S(120)	S(75)
md5-47-3	S(84)	S(38)	S(191)
md5-47-4	S(42)	S(19)	S(89)
md5-48-1	S(454)	S(173)	S(178)
md5-48-2	S(97)	S(160)	S(133)
md5-48-3	S(488)	S(475)	S(35)
md5-48-4	S(67)	S(63)	S(385)
md5-48-5	S(69)	S(23)	S(128)
ndhf-xits-19	I(5000)	S(1309)	S(1)
ndhf-xits-21	S(633)	S(5)	S(2)
openstacks	S(252)	S(41)	S(26)
partial-10-11-s	S(1101)	S(1946)	S(751)

Continued on next page

Table 5.7 – *Continued from previous page*

Instance	HRMS	ZENN	Lingeling
partial-10-17-s	I(5000)	I(5000)	S(4922)
partial-10-19-s	I(5000)	I(5000)	S(1964)
partial-5-13-s	S(127)	S(60)	S(61)
partial-5-15-s	S(178)	S(165)	S(186)
partial-5-17-s	I(5000)	S(1134)	S(614)
partial-5-19-s	S(456)	S(1199)	S(533)
pb-200-03-lb-03	S(90)	S(377)	S(56)
pb-300-01-lb-00	S(48)	S(37)	S(217)
pb-300-02-lb-06	S(38)	S(119)	S(88)
pb-300-02-lb-07	S(10)	S(2)	S(25)
pb-300-03-lb-13	S(15)	S(46)	S(53)
pb-300-05-lb-16	S(2203)	S(1340)	S(497)
pb-300-05-lb-17	S(938)	S(180)	S(820)
pb-300-06-lb-02	S(518)	S(642)	S(1340)
pb-300-09-lb-07	S(89)	S(55)	S(71)
pb-300-10-lb-12	S(3447)	S(415)	S(1715)
pb-300-10-lb-13	S(87)	S(163)	S(793)
pb-400-02-lb-15	S(328)	S(257)	S(339)
pb-400-04-lb-19	S(164)	S(365)	S(507)
pb-400-05-lb-00	S(380)	S(1420)	S(728)
pb-400-09-lb-05	S(52)	S(60)	S(174)
pb-400-10-lb-00	S(17)	S(41)	S(34)
post-cbmc	S(76)	S(76)	S(199)
q-query-3	S(45)	S(50)	S(143)
rbcl-xits-14	S(5)	S(1)	S(2)
rpoc-xits-15	S(606)	S(1)	S(1)
slp-top25	I(5000)	I(5000)	I(5000)
slp-top26	I(5000)	I(5000)	I(5000)
slp-top27	S(1388)	S(430)	I(5000)
smtlib-aigs-	S(14)	S(2)	S(1053)
total-10-15-s	S(23)	S(491)	S(71)
transport-city-25-020	S(58)	S(30)	S(63)
transport-city-25-030	S(445)	S(38)	S(34)
transport-city-25-040	S(203)	S(460)	S(885)
transport-city-25-050	S(49)	S(545)	S(1136)
transport-city-25-060	S(383)	S(1588)	S(261)
transport-city-35-020	S(958)	S(458)	S(641)
transport-city-35-030	I(5000)	S(3392)	S(3355)
transport-city-35-040	S(1417)	I(5000)	I(5000)
transport-three-020	I(5000)	I(5000)	I(5000)
transport-three-030	I(5000)	I(5000)	I(5000)
transport-two-020	S(984)	S(323)	S(1026)

Continued on next page

Table 5.7 – *Continued from previous page*

Instance	HRMS	ZENN	Lingeling
transport-two-030	S(1105)	S(1048)	S(1214)
transport-two-040	S(841)	S(115)	S(2436)
UCG-15-10p1	S(1409)	S(842)	S(369)
UR-15-10p1	S(1608)	S(1362)	S(724)
UR-20-10p1	I(5000)	S(4452)	S(2691)
UR-20-5p1	S(2451)	S(1530)	S(786)
UTI-15-10p1	S(136)	S(310)	S(381)
UTI-20-10p1	S(2652)	S(1451)	S(1953)
UTI-20-5p1	S(1236)	S(990)	S(1108)
velev-npe	S(80)	S(247)	S(490)
velev-pipe-b7	S(1241)	S(97)	S(181)
velev-pipe-b9	S(96)	S(104)	S(49)
vmpc-29	S(51)	S(213)	S(527)
vmpc-30	S(2)	S(169)	S(1517)
vmpc-32	I(5000)	S(1013)	S(4012)
vmpc-33	S(736)	S(4444)	S(1326)
vmpc-34	I(5000)	S(3166)	I(5000)
vmpc-35	I(5000)	I(5000)	I(5000)
vmpc-36	I(5000)	I(5000)	I(5000)
zfcp-2.8-u2-nh	S(81)	S(81)	S(202)

Table 5.7 shows that there were several instances that HRMS solved that the other two solvers did not. Indeed, in instance 003e, HRMS was the only solver among the 31 entrants that solved this instance [68]. Additionally, HRMS was often the fastest solver to reach a solution when other solvers were able to find a solution. This tells us that there are cases where not only is hyperplane reduction an effective strategy at finding solutions efficiently, but hyperplane reduction often provides the best results in terms of both finding solutions and the time required to do so.

5.3 Summary

We have used our hyperplane averages to heuristically select a hyperplane in industrial instances that we hope contains a globally optimal solution. We then perform a reduction

Table 5.6: Total time (in seconds) spent by each solver on the 150 instances and the median time spent per solver on each instance. Our solver, HRMS, was the second fastest in total time and median time per instance.

Solver	Total Time	Median Time
Virtual Best Solver (VBS)	67391	452
Lingeling aqw	250712	1671
ZENN 0.1.0	285593	1904
HRMS	273180	1821
satUZK 48	288204	1921
Riss3g 3g	298632	1991
Lingeling 587f	306968	2046
CSHCapplLG	309761	2065
gluebit-lgl 1.0	291967	1946
MIPSat	307557	2050
forl nodrup	316879	2113
MIPSat	325120	2167
glucose 2.3	306865	2046
glue-bit 1.0	291928	1946
Solver43b b	315139	2101
Riss3g	322501	2150
strangenight satcomp11-st	324324	2162
Solver43a a	312244	2082
BreakIDGlucose 1	313876	2093
glueminisat 2.2.7j	314200	2095
CSHCapplLC	355796	2372
relback v1.1	308343	2056
gluH 1.0	316141	2108
ShatterGlucose 1	324754	2165
gluH-simp 1.0	334811	2232
Nigma 1.0	345460	2303
GlucoRed r531	363050	2420
RSeq V 1.1	391404	2609
Nigma-NoPB 1.0	354518	2363
minipure 1.0.1	420429	2803
sattimeRelbackShr SRShr1.0	480920	3206
SAT4J SAT COMPETITION 2013	464330	3096

using the SatELite preprocessor on the hyperplane subspace. We have shown that this reduction produces instances that are significantly smaller than a reduction based on SatELite alone. When we rank hyperplanes based on the four most frequent variables, the highest ranking hyperplane often contains a satisfying solution. In these cases, a complete solver such as MiniSAT can find a satisfying solution to the original problem by examining only the best hyperplane.

We expanded this strategy to develop an entry to the 2013 SAT competition [68]. If the best hyperplane is unsatisfiable, or if the satisfiability cannot be determined in ten minutes, we move on to the next highest hyperplane. This strategy has proven to be very effective. Our entry was ranked 3rd out of 31 solvers in the application track of the competition by solving 113 out of 150 instances. Our strategy is also efficient: we were the second fastest solver in the competition. Furthermore, there is some evidence that we can solve exceptionally hard problems with hyperplane reduction; our solver using hyperplane reduction was the only one to solve the 003e instance [68].

This provides further evidence that hyperplane information can be used to guide search. While we have shown that this is true for SLS algorithms in the previous chapter, we have now shown that it is also applicable to complete SAT solvers. We conjecture that this again comes down to hyperplane averages being able to provide a good estimation of the backbone and critical variables. By using these estimations to fix the variables prior to search, we reduce the work required by complete solvers to determine the satisfiability of an instance.

Chapter 6

Conclusion and Future Work

Random uniform problems are easy to generate, and their uniformity allows for a fairly straightforward analytical analysis. While these properties are valuable for developing theory and initial testing for new algorithms, there is some risk in focusing too much on these artificial problems.

In MAX-SAT and SAT, many of the SLS algorithms that are considered state-of-the-art are based on prior algorithms that have been developed using uniform random benchmarks [46]. One issue that arises from this practice can clearly be seen by examining the results of recent SAT competitions. In the 2011 SAT competition, the top 4 performing algorithms in the uniform random SAT track were SLS algorithms. Although these same SLS algorithms, along with others, were also submitted to the industrial SAT track, not a single SLS algorithm ranked above 40th place on the applied problems [67].

Instances from real-world applications tend to be structured in a way that is not captured in expectation by uniform randomly generated instances [47, 39, 41]. It has been observed that industrial instances have several common characteristics: a power-law distribution of variables, a power-law distribution over clause length and a community structure [3, 5]. However, prior to our work, these characteristics have not been connected to the performance of SLS algorithms nor have they been correlated to underlying structures in the search space.

To better understand why structured instances are difficult for SLS algorithms, we have extended an existing generator to several instance generators based on the generators that can produce instances with a power-law distribution over variable frequency, clause lengths following a power distribution and a community structure. This allowed us to control for these characteristics and by doing so, we found that SLS algorithms have more difficultly finding the optimal solutions on those instances with a community structure and variable clause length. Further analysis of these industrial-like instances reveals that the search space

is more rugged than in uniform random instances and the backbone size is larger, two search space features which are associated with difficult for SLS.

In light of these results, we next looked at improving the performance of SLS algorithms on industrial instances. We presented a method of tractably computing the average evaluation of solutions in a subspace of the search space that we call a hyperplane [83, 31]. We have shown that hyperplane averages can be used to provide a remarkably good estimation of the correct backbone settings, with a greater than 90% accuracy on the most industrial-like instances. We then used the hyperplane averages to construct solutions that were used to initialize an existing SLS algorithm called AdaptG2WSAT. By using the hyperplane initialized solutions, AdaptG2WSAT was able to find significantly better solutions in the vast majority of cases.

We then examined the difference between two types of local search: first improving and best improving [1]. Current SLS algorithms typically employ a best improving search despite evidence from Gent and Walsh [25, 24] that suggests there is no advantage to using best improving over first improving on uniform random instances and its greater computational cost. Our own analysis of the trade-off between the two searches revealed a similar result on industrial instances: the end solutions found by AdaptG2WSAT using first improving search were the same, or significantly better, than those found by AdaptG2WSAT using best improving search in the majority of cases.

We conjecture that AdaptG2WSAT using first improving search can do better than AdaptG2WSAT with best improving search due to the effect of critical variables. Best improving search will always take the bit flip that maximizes the number of satisfied clauses in the resulting solution. By doing so, we believe this forces the search to set critical variables in the first few iterations of the search. If these variables are set incorrectly, it may trigger a cascade effect causing the search to set other variables incorrectly that occur in the same clauses as the critical variables. As this effect propagates to more and more variables, it becomes increasingly unlikely that the search will repair the mistake it made early on. As first improving search takes an arbitrary improving move, it is not as heavily biased towards

flipping the critical variables early in the search. Because of this, AdaptG2WSAT using first improving search will have more opportunities to find improving moves in the second phase.

Finally, we examined how the hyperplane averages can be used to improve complete SAT solvers on industrial instances. We first show that by fixing the variables corresponding to a promising hyperplane, we can significantly reduce the size of the original problems in terms of both the number of variables and the number of clauses. We then ranked the hyperplanes by their averages and show that the first rank hyperplane contains a satisfying solution in the majority of cases. Based on this result, we formulate a strategy for using the hyperplane averages in a preprocessing step. Our hyperplane reduction strategy coupled with the MiniSAT complete solver ranked 3rd in the 2013 SAT competition. Our strategy not only beat many state-of-the-art complete solvers, but we also solved one instance that was not solved by any other solvers.

6.1 Future Work

We have shown that randomly generated instances with two of the three characteristics common to industrial instances are indeed more difficult for SLS algorithms. Further analysis of these instances has revealed search space properties that are at least partial explanations of the increase in difficulty. We used these results to improve the performance of both incomplete and complete solvers on industrial instances. Our work has advanced both the knowledge of structured instances and the state-of-the-art in solving these instances. We hope that this dissertation will also serve as a foundation for future work on solving real-world applications of combinatorial optimization. To that end, in the following sections we outline several open questions from this work that should serve as excellent starting points for future research.

6.1.1 Understanding Community Structure

In our analysis of structured instances, we generated instances with a community structure by using 'connector variables' to link several families of highly connected variables. This

creates a ring-like structure of families linked to neighboring families. However, there are other community structures that could be considered. For example, a central family with radial connections to multiple families that are connected only through the central family. A second example is an instance in which any one family has a probability of being connected to any other family.

It would be fairly straightforward to repeat our search space analysis on various community structures and observe how they influence the underlying structure of the search space. Experiments on other types of community structure may give us a better understanding of how modularity affects performance critical search space features such as ruggedness and backbone size.

Algorithms for computing the modularity of MAX-SAT and SAT instances provide a partitioning of the variables into families [5]. This partitioning may be leveraged by constructing subproblems that correspond the inter-family variable interactions. These subproblems could then be solved independently of one another, and the final solutions merged together to form a solution for the original formula. Partition and patch strategies have proven successful in other problems of combinatorial optimization, i.e., the cycle patching in Lin-Kernighan-Helsgaun or the Edge Assembly Operator for the TSP [36, 56].

An additional artifact of our construction method is the cardinality of the families created by our modular generator. Our generators will construct modular instances with l families with the same cardinality. Although we have not exmained the cardinality of families found in industrial instances, we suspect that they are not uniform in size. Additional analysis should be done not only on the connections between between families in modular instances, but also on the relative sizes of the families.

6.1.2 Critical Variables

We hypothesize that there are critical variables, perhaps more so in those instances with a community structure, that if set incorrectly can cause cascade effects that make it difficult for the search to repair the mistake. We believe that critical variables are the reason that

first improving search can sometimes perform better than best improving search. The greedy bias of best improving search will cause the critical variables to become fixed early in the search. First improving does not have such a bias and may allow the search more freedom to try different settings of the critical variables.

There is some indication from our results with IRoTS that using a Tabu mechanism can at least offset the bias of best improving search. These results showed that IRoTS with best improving search almost always found better solutions than IRoTS with first improving search. By making other bits Tabu, even if the search fixes a critical variable early on, it may be forced to flip the variable later in the search due to the bias from the Tabu list. More experimentation is required to better understand the interaction effect between the local search type, Tabu lists and critical variables.

6.1.3 Hyperplane Reduction

Although our hyperplane reduction strategy shows great promise, there are still several ways it might be improved. One is our selection of variables that define the hyperplane. We choose to use the most frequent variables as we believe these are likely to be critical variables and also have the most potential to reduce the problem size. However, if a variable is negated half the time it appears in an instance, its Walsh coefficient will be 0. Therefore, it will not contribute as much to the hyperplane average as a variable that is negated 90% of the time. Pair-wise and higher order interactions can have similar effects.

Likewise, the size of the clauses in which each variable appears in can also reduce the variables impact on the hyperplane averages. Variables appearing in larger clauses will have a smaller Walsh coefficient than those in smaller clauses. Thus we may be able to extract more information about the hyperplanes by using alternative heuristics that bias the choice towards those variables that will contribute more to the hyperplane information.

Another aspect that we have left unexplored for the time being is the parallelization of our strategy. Katsirelos et al. have shown that efficient parallelization is extremely difficult for SAT solvers, and in fact may not even be possible due to the resolution proofs used in

modern solvers [43]. Portfolio methods, e.g. SATZilla, are the standard way of utilizing parallel architectures. CPU resources are allocated to a portfolio of solvers based on the predictions of the solver performance that are made from problem characteristics.

While hyperplane reduction could be included in a portfolio of solvers, it also suggests an alternative approach to parallelizing SAT solvers. Hyperplanes are mutually exclusive partitions of the space and can therefore be searched independently in parallel without any overlap. We can also easily adjust the number of hyperplanes that are generated simply by using more variables. Choosing k variables will partition the space into 2^k hyperplanes. Thus if p parallel nodes are available, we can choose $\sqrt{p}$ variables and search each of the resulting p hyperplanes in parallel.

The advantage of this is that on difficult problems that are intractable without hyperplane reduction, the reduced problems associated to the hyperplanes may be tractable. Indeed, in the 2013 SAT competition our hyperplane reduction strategy solved one problem (003.cnf) that was not solved by any other solvers [68]. Another advantage is that we can also decide an instance is unsatisfiable without resorting to the original problem. As soon as one node finds a satisfiable solution, the rest can stop. If all nodes return that the hyperplane reduction is unsatisfiable, we can conclude that the original problem is unsatisfiable. Thus, our hyperplane reduction strategy could easily be used to efficiently parallelize a SAT solver, albeit at a coarse-grain level.

References

[1] Greedy or not? best improving versus first improving stochastic local search for MAXSAT, author=Whitley, D. and Howe, A. and Hains, D., journal=Proceedings of the National Conference on Artificial Intelligence, year=2013.

[2] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected Papers of Hirotugu Akaike*, pages 199–213. Springer, 1998.

[3] C. Ansótegui, M. Bonet, and J. Levy. On the structure of industrial SAT instances. *Principles and Practice of Constraint Programming-CP 2009*, pages 127–141, 2009.

[4] C. Ansótegui, M.L. Bonet, and J. Levy. Towards industrial-like random SAT instances. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 387–392. Morgan Kaufmann Publishers Inc., 2009.

[5] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 410–423. Springer Berlin Heidelberg, 2012.

[6] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, volume 9, pages 399–404, 2009.

[7] Armin Biere. PrecoSAT @ SC'09. *Proceedings of SAT Competition 2019*, 4:41–43, 2009.

[8] Armin Biere. Lingeling and friends at the SAT competition 2011. *FMV Report Series Technical Report*, 11(1), 2011.

[9] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition 2013*, page 51, 2013.

[10] E. Boros and P.L. Hammer. Pseudo-boolean optimization. *Discrete applied mathematics*, 123(1):155–225, 2002.

[11] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On finding graph clusterings with maximum modularity. *Graph-Theoretic Concepts in Computer Science*, pages 121–132, 2007.

[12] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artifical Intelligence*, pages 331–337, 1991.

[13] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris. Automated design debugging with maximum satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(11):1804–1817, 2010.

[14] Aaron Clauset, Mark Newman, and Christopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6), 2004.

[15] Aaron Clauset, Cosma Rohilla Shalizi, and Mark Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

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

[17] C. DeSimone, M. Diehl, M. Juenger, P. Mutzel, G. Reinelt, and G. Rinaldi. *Exact Ground States of Two-Dimensional J Ising Spin Glasses*. Max-Planck-Inst. für Informatik, Bibliothek & Dokumentation, 1996.

[18] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75. Springer, 2005.

[19] Niklas Een and Niklas Sörensson. Minisat. In *6th International Conference on Theory and Applications of Satisfiability Testing*, page 2003, 2007.

[20] S.F. Elena, R.V. Solé, J. Sardanyés, et al. Simple genomes, complex interactions: Epistasis in RNA virus. *Chaos*, 20, 2010.

[21] J. Frank, P. Cheeseman, and J. Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[22] M.R. Garey and D.S. Johnson. *Computers and intractability*, volume 174. 1979.

[23] Ian Gent and Toby Walsh. The enigma of SAT hill-climbing procedures. 1992.

[24] Ian P. Gent and Toby Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1, 1993.

[25] Ian P Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 28–33, 1993.

[26] Jean Charles Gilbert and Claude Lemaréchal. *Numerical optimization: theoretical and practical aspects*. Springer, 2006.

[27] D. Goldberg. Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction. *Complex Systems*, 3:129–152, 1989.

[28] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[29] D. Goldberg, B. Korb, and K. Deb. Messy Genetic Algorithms: Motivation, Analysis, and First Results. *Complex Systems*, 4:415–444, 1989.

[30] D. Hains, D. Whitley, and A. Howe. Improving Lin-Kernighan-Helsgaun with crossover on clustered instances of the TSP. *Proceedings of Parallel Problem Solving from Nature*, pages 388–397, 2012.

[31] Doug Hains, Darrell Whitley, Adele Howe, and Wenxiang Chen. Hyperplane initialized local search for MAXSAT. *Proceedings of the Annual Conference on Genetic and Evolutionary Computation Conference*, 2013.

[32] DR Hains, LD Whitley, and AE Howe. Revisiting the big valley search space structure in the TSP. *Journal of the Operational Research Society*, 62(2):305–312, 2010.

[33] S. Hampson and D. Kibler. Large plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:437–455, 1996.

[34] J.A. Hartigan and M.A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[35] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[36] K. Helsgaun. General k-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation*, 1(2):119–163, 2009.

[37] John Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[38] H. Hoos, K. Smyth, and T. Stützle. Search space features underlying the performance of stochastic local search algorithms for MAX-SAT. In *Proceedings of Parallel Problem Solving from Nature*, pages 51–60, 2004.

[39] H.H. Hoos. A mixture-model for the behaviour of SLS algorithms for SAT. In *Proceedings of the National Conference on Artificial Intelligence*, pages 661–667, 2002.

[40] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann, 2004.

[41] Holger H Hoos and Thomas Stutzle. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence*, 112(1):213–232, 1999.

[42] David S. Johnson and Lyle A. Mcgeoch. The traveling salesman problem: A case study in local optimization. In *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, 1997.

[43] G. Katsirelos, A. Sabharwal, H. Samulowitz, and L. Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence*, 2013.

[44] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[45] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 58–60, 1998.

[46] H.A. Kautz, A. Sabharwal, and B. Selman. Incomplete algorithms. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, 185:185–204, 2009.

[47] L. Kroc, A. Sabharwal, C.P. Gomes, and B. Selman. Integrating systematic and local search paradigms: A new strategy for MaxSAT. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 544–551, 2009.

[48] C. Li and W. Huang. Diversification and determinism in local search for satisfiability. In *Theory and Applications of Satisfiability Testing*, pages 158–172, 2005.

[49] C. Li, W. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 121–133, 2007.

[50] J. Marques-Silva. Practical applications of boolean satisfiability. In *Proceedings of the 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008.

[51] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[52] MAX-SAT 2012 competition. `http://maxsat.ia.udl.cat`, 2012.

[53] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 321–326, 1997.

[54] Peter Merz and Bernd Freisleben. Fitness landscapes and memetic algorithm design. *New ideas in optimization*, pages 245–260, 1999.

[55] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, pages 530–535, 2001.

[56] Y. Nagata. Fast EAX algorithm considering population diversity for traveling salesman problems. *Evolutionary Computation in Combinatorial Optimization*, pages 171–182, 2006.

[57] Mark Newman. Power laws, pareto distributions and zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[58] Mark Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[59] A.J. Parkes. Clustering at the phase transition. In *Proceedings of the National Conference on Artificial Intelligence*, pages 340–346, 1997.

[60] Martin Pelikan, David Goldberg, and Fernando Lobo. A survey of optimization by building and using probabilistic model. Technical Report 99018, IlliGAL, 1999.

[61] A. Prugel-Bennett and M.H. Tayarani-Najaran. Maximum satisfiability: Anatomy of the fitness landscape for a hard combinatorial optimization problem. *IEEE Transactions on Evolutionary Computation*, 16(3):319–338, 2012.

[62] M. Qasem and A. Prugel-Bennett. Learning the large-scale structure of the MAX-SAT landscape using populations. *IEEE Transactions on Evolutionary Computation*, 14(4):518–529, 2010.

[63] S. Rana, R.B. Heckendorn, and D. Whitley. A tractable Walsh analysis of SAT and its implications for genetic algorithms. In *Proceedings of the National Conference on Artificial Intelligence*, pages 392–397, 1998.

[64] Colin R. Reeves and Jonathan E. Rowe. Landscapes. In *Genetic Algorithms – Principles and Perspectives: A guide to GA theory*, pages 231–263. Springer, 2002.

[65] C.M. Reidys and P.F. Stadler. Combinatorial landscapes. *SIAM review*, 44(1):3–54, 2002.

[66] S. Safarpour, H. Mangassarian, A. Veneris, M.H. Liffiton, and K.A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 13–19. IEEE, 2007.

[67] Satisifiability Competition 2011. `http://www.satcompetition.org/2011/`, 2011.

[68] Satisifiability Competition 2013. `http://www.satcompetition.org/2013/`, 2013.

[69] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[70] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 337–337, 1994.

[71] B. Selman, H. Levesque, D Mitchell, et al. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial intelligence*, pages 440–446, 1992.

[72] B. Selman, D.G. Mitchell, and H.J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996.

[73] Carsten Sinz. Visualizing SAT instances and runs of the DPLL algorithm. *Journal of Automated Reasoning*, 39(2):219–243, 2007.

[74] J. Slaney and T. Walsh. Backbones in optimization and approximation. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 254–259, 2001.

[75] Kevin Smyth, Holger H Hoos, and Thomas Stützle. Iterated robust tabu search for MAX-SAT. In *Advances in Artificial Intelligence*, pages 129–144. Springer, 2003.

[76] K.R.G. Smyth. Understanding stochastic local search algorithms: An empirical analysis of the relationship between search space structure and algorithm behaviour. Master's thesis, The University of British Columbia, 2004.

[77] A.M. Sutton, A.E. Howe, and L.D. Whitley. Directed plateau search for MAX-k-SAT. In *Third Annual Symposium on Combinatorial Search*, 2010.

[78] Dave AD Tompkins, Adrian Balint, and Holger H Hoos. Captain Jack: new variable selection heuristics in local search for SAT. In *Theory and Applications of Satisfiability Testing*, pages 302–316. Springer, 2011.

[79] UBCSAT. `http://ubcsat.dtompkins.com/`, 2013.

[80] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *Circuits and Systems Magazine, IEEE*, 3(1):6–20, 2003.

[81] Duncan Watts and Steven Strogatz. Collective dynamics of small-worldnetworks. *Nature*, 393(6684):440–442, 1998.

[82] D. Whitley. Defying gravity: constant time steepest ascent for MAX-kSAT. Technical report, Department of Computer Science, Colorado State University, December 2011.

[83] D. Whitley and W. Chen. Constant Time Steepest Ascent Local Search with Statistical Lookahead for NK-Landscapes. In *GECCO '12: Proceedings of the Annual Conference on Genetic and Evolutionary Computation Conference*, 2012.

[84] D. Whitley, D. Hains, and A. Howe. Tunneling between optima: partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, pages 915–922. ACM, 2009.

[85] D. Whitley, D. Hains, and A. Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. *Parallel Problem Solving from Nature–PPSN XI*, pages 566–575, 2011.

[86] Takeru Yasumoto and Takumi Okugawa. ZENN. *Proceedings of SAT Competition 2013*, page 95, 2013.

[87] W. Zhang. Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *Principles and Practice of Constraint Programming-CP-2001*, pages 153–167, 2001.

[88] W. Zhang and M. Looks. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *International Joint Conference on Artificial Intelligence*, volume 19, page 343, 2005.

[89] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *Proceedings of International Joint Conference on Artificial Intelligence*, volume 18, pages 1179–1186, 2003.

Appendix A

Names and Sizes of Industrial Instances

Table A.1: The number of variables and number of clauses of 320 instances from the 2011 SAT competition [67] and the 2012 MAX-SAT competition [52]. Note some instances names have been shortened due to width restrictions but remain uniquely identifiable.

Instance	Number of Variables (n)	Number of Clause (m)
driverlog3	170	1559
driverlog1	207	588
bart17.shuffled	231	1166
homer16.shuffled	264	1476
homer17.shuffled	286	1742
aes-32-1	300	1016
homer14.shuffled	300	2130
rovers1	439	5423
aes-32-2	504	1840
myciel6	570	4625
aes-64-1	596	2780
vmpc-25	625	76775
aes-32-3	708	2664
dp04u03	825	2411
vmpc-29	841	120147
aes-32-4	912	3488
ferry5	984	8702
aes-64-2	1000	5176
vmpc-32	1024	161664
dp04s04	1075	3152
aes-32-5	1116	4312
vmpc-34	1156	194072
aes-128-1	1192	10656
vmpc-35	1225	211785
rbcl-xits-08	1278	68055
vmpc-36	1296	230544
aes-64-3	1404	7572
rpoc-xits-09	1430	87044
rbcl-xits-09	1430	79453
aes-64-4	1808	9968
aes-128-2	2000	20544
rbcl-xits-15	2384	164746
aes-128-3	2808	30432
rbcl-xits-18	2888	218530

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
rand-net60-25	3000	8881
rand-net60-30	3600	10681
ndhf-xits-19	4020	466486
am-7-7	4264	14751
gripper13u	4268	38965
minandmaxor016	4271	12620
mulhs016	4656	13871
E07N15	4740	41363
korf-15	4740	45569
E05X15	4740	41379
E15N15	4740	44327
rand-net60-40	4800	14281
comb1.shuffled	5910	16804
abb313GPIA-9-c	6228	484871
vda-gr-rcs	6498	130997
E02F17	6664	69700
E03N17	6664	93544
korf-17	6664	89966
dp10s10	7759	23004
E04N18	7794	120068
E05F18	7794	126826
korf-18	7794	186934
aes-128-10	8080	96704
rand-net70-60	8400	25061
countbitsrotate032	8527	25484
countbitsarray04	8750	25865
x1mul.miter	8756	55571
E04F19	9044	295685
c6288mul.miter	9540	61421
k2fix	10056	271393
E02F20	10420	395383
E04F20	10420	484053
E05F20	10420	481346
k2fix	11313	305160
k2mul.miter	11680	74581
maxxor032	13240	39143
E02F22	13574	1301188
abb313GPIA-9-tr	14013	481761
E00N23	15364	2210893
E00X23	15364	2133873
6pipe	17064	545612
slp-bottom12	17298	57292

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
velev-pipe-1.1-6	17710	304026
dekker	19473	58308
slp-bottom13	19995	66333
all-986	21317	63664
slp-bottom14	22886	76038
velev-pipe-1.0-7	24415	711050
pathways-13-step17	25631	142227
slp-bottom15	25972	86411
traffic-b	26061	742909
smtlib-qfbv	27224	68879
li-exam-61	28147	108436
slp-bottom16	29254	97456
aaai10i-rovers-18-step11	29317	277090
gss-14	31229	93855
gss-16	31248	93904
gss-19	31435	94548
gss-21	31613	95104
gss-22	31616	95110
gss-27	31951	96161
rovers-18-step12	32523	308440
slp-bottom17	32733	109177
q-query-3-l46	33090	165828
traffic-f	33320	763101
traffic-pcb	33320	766757
traffic-kkb	34510	701694
traffic-fb	35819	773506
slp-bottom18	36410	121578
li-test4-100.shuffled-370	36809	142491
traffic-3b-unknown	39151	533919
slp-bottom19	40286	134663
slp-bottom20	44362	148436
hwmcc10	44692	129620
AProVE11-12	44805	149118
dividers10	45552	162874
smulo064	47129	141002
slp-bottom21	48639	162901
wb2	49490	140056
wb1	49525	140091
mizh-sha0-36-4	50073	210235
sha0-36-5	50073	210223
pathways-17-step20	50277	283903
maxxor064	51064	152039

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
slp-top21	51138	162526
slp-bottom22	53118	178062
pathways-17-step21	53919	308235
slp-top22	55875	177646
k50	56536	158531
slp-bottom23	57800	193923
slp-top23	60823	193450
slp-bottom24	62686	210488
k45	63327	174791
pipesworld-12-step15	63349	943492
ibm-2004-01-k90	64699	276210
slp-top24	65983	209942
sokoban.030	66084	1060536
AProVE11-15	66715	228274
md5-48-3	66892	279258
slp-bottom25	67777	227761
AProVE11-09	68779	234693
pipesworld-12-step16	68952	1029036
gus-md5-12	69553	226752
gus-md5-11	69561	226787
bjrb07amba2andenv	70167	209513
bc57sensorsp3	70297	194006
bc57sensorsp2	71285	196970
slp-top25	71356	227126
slp-bottom26	73074	245746
countbitssrl064	75103	225116
slp-top26	76943	245006
bobsm5378d2	82007	241364
AProVE11-16	84025	282766
sokoban.040	87884	1413816
11pipe-11-ooo	88289	4187694
k50-pdtpmsns2	88352	262658
slp-top28	88763	282870
pdtpmspalu	89190	265718
11pipe-k	89315	5584003
maxxororand064	92464	276623
slp-top29	94998	302862
openstacks-1.025	95456	477186
openstacks-3.025	95456	483561
smtlib-qfbv	95810	287045
manol-c6bidw-i	96089	283993
pdtswvtms14x8p1	96095	285404

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
AProVE11-11	96526	325263
countbitswegner128	98048	293759
traffic-r-sat	98651	3362666
pdtpmsgoodbakery	98935	296405
TPP-21-step11	99736	783991
slp-top30	101451	323566
pdtviseisenberg1	102621	307469
SAT-dat.k80-04	104450	457628
dividers-multivec1	106128	397650
UTI-10-5t1	108508	527595
sokoban.050	109684	1767096
pdtviseisenberg2	115266	345359
pdtvisns2p3	116405	346850
velev-pipe-sat-1.0-b9	118038	8780591
pipesworld-18-step15	130021	2509375
UR-10-10p1	131228	635871
sokoban.060	131484	2120376
openstacks-p30-1.035	133566	667976
openstacks-p30-3.035	133566	676901
post-c32s-gcdm16-23	135543	404326
12pipe-bug4	138563	4675040
12pipe-bug6	138795	4671352
pipesworld-18-step16	141856	2742601
traffic-3	142205	1312352
SM-AS-TOP	145900	694438
sokoban.070	153284	2473656
minxorminand128	153834	459965
pdtswvsam6x8p3	154038	460862
velev-vliw-4.0-9-i1	154309	3230738
ACG-10-10p0	155107	675624
pdtvissoap1	156058	466691
UCG-15-5p0	162696	821000
pdtvisns3p02	163622	488120
smtlib-libsmbsharemodes	164758	456350
openstacks-p30-1.045	171676	858766
openstacks-p30-3.045	171676	870241
sokoban.080	175084	2826936
smtlib-src-wge	180045	538304
ibm-2002-30r-k85	181484	890298
SAT-dat.k85	181484	890298
pdtvisns3p00	183325	546914
ibm-2002-21r-k95	191522	788339

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
eijkbs6669	192503	564953
c10idw-i	196306	582994
UCG-15-10p0	199304	1005834
UCG-15-10p1	200003	1019221
maxor128	200308	598619
maxxor128	200440	599015
c5-DD-s3v1	200944	540984
c5-DD-s3v2	200944	540984
2dlx-ca-bp	202253	4313014
smtlib-libmsrpc	206768	599363
ibm-2004-23-k100	207606	861175
9dlx-vliw	209724	3634677
AProVE11-02	214734	743081
q-query-3-L70	218792	1020908
ACG-15-5p1	218990	943377
pipesworld-27-step13	224228	5314673
UCG-20-5p1	224986	1204430
divider-problem-2	228874	750705
divider-problem-5	228874	750705
AProVE11-07	233177	782677
SM-RX-TOP	235456	934091
UTI-20-10t0	237336	1262609
rsdecoder4	237783	933978
UTI-20-10t1	238008	1278025
rsdecoder-fsm2	238290	936006
rsdecoder5	238290	936006
rsdecoder6	238290	936006
divider-problem-8	246943	810105
minandmaxor128	249327	746444
pipesworld-27-step14	249618	5949456
smtlib-VS3-benchmark-S2	257030	769313
fpu-multivec1-problem-14	257168	928310
grid-strips	257380	2139615
UR-20-10p0	258781	1372095
UR-20-10p1	259234	1387934
UTI-20-10p0	259616	1374599
smtlib-lfsr-008-063-080	259762	656573
UTI-20-10p1	260342	1391257
dme-03-1-k247	261352	773077
smtlib-libsmbclient	266663	768347
rsdecoder1-blackbox-32	277950	806460
wb-conmax1	277950	1221020

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
wb-conmax3	277950	1221020
grid-strips-grid-y	281936	2464339
AProVE11-10	289828	1008603
c6-DD-s3-f1-e1-v1	298058	795900
wb-problem-46	300846	789283
AProVE11-06	302076	1039056
wb-problem-45	309491	806440
manol-pipe-f7nidw	310434	923497
TPP-30-step11	314455	2920820
openstacks-p30-3.085-SAT	324116	1643601
smtlib-lfsr-004-127-112	350506	878969
smtlib-servers-slapd	360364	1076507
smtlib-nlzbe256	361856	1084031
hard-6-U-7061	370048	1118066
9dlx-vliw-at-b-iq8	371419	7170909
c1-DD-s3-f1-e2-v1	391897	989885
rsdecoder-multivec1	394446	1626312
i2c-problem-26	397668	1205454
c2-DD-s3-f1-e2-v1	400085	1121810
rsdecoder2	415480	1632526
manol-pipe-c10nidw	433601	1291714
bc57-sensors-1-k303	435701	1379987
smtlib–lfsr-008-079-112	448370	1130525
c4-DD-s3-f1-e2-v1	448465	1130672
wb-4m8s1	463080	1759150
wb-4m8s3	463080	1759150
wb-4m8s4	463080	1759150
traffic-r-uc-sat	467551	4467733
9dlx-vliw-at-b-iq9	482093	9676386
q-query-3-L150	486992	2456708
blocks-blocks-36-0.120	489685	8779920
AProVE11-13	504189	1742927
blocks-blocks-37-1.120	516641	9436757
i2c-problem-25	521672	1581471
hard-25-U-7061	541150	1645836
blocks-blocks-37-1.130	559571	10223027
1dlx-c-iq57-a	569795	8562505
transport-cities-15nodes	580875	3140965
b15-bug-fourvec-gate-0.seq	581064	1712690
rsdecoder-multivec1-33	627993	2125620
blocks-blocks-37-1.150	645431	11795567
1dlx-c-iq60-a	651875	9927770

Continued on next page

Table A.1 – *Continued from previous page*

Instance	Number of Variables (n)	Number of Clause (m)
blocks-blocks-36-0.160	652445	11706080
blocks-blocks-36-0.170	693135	12437620
rsdecoder1-blackbox-30	707330	1106376
transport-city-25nodes	723130	3869060
blocks-blocks-36-0.180	733825	13169160
transport-three-14nodes	756832	4116966
c4-DD-s3-f1-e1-v1	797728	2011216
mrisc-mem2wire-29	844900	2905976
SM-MAIN-MEM-buggy1	870975	3812147
velev-npe-1.0-9dlx-b71	889302	14582074
transport-city-35nodes	945406	5079060
valves-gates-1-k617	985042	3113540
mem-ctrl1	1128648	4422185
transport-three-14nodes	1134832	6175026
rsdecoder-problem-41	1186710	3829036
rsdecoder-problem-31	1197376	3863287
rsdecoder-problem-38	1198012	3865513
rsdecoder-problem-39	1199602	3868693
rsdecoder-problem-36	1220616	3938467
rsdecoder-problem-40	1220616	3938467
transport-city-35nodes	1260306	6771840
clauses-8	1461771	5687554
transport-three-14nodes	1512832	8233086
rsdecoder-problem-37	1513544	4909231
transport-city-35nodes	1575206	8464620
mem-ctrl2	1974822	6795573
hard-18-U-10652	1987351	5963534
wb-4m8s-problem-47	2691648	8517027
wb-4m8s-problem-48	2766036	8774655
wb-4m8s-problem-49	2785108	8812799
mem-ctrl-problem-27	4426323	15983633
post-cbmc-zfcp	10950109	32697150