Summer 2016

Automated design of boolean satisfiability solvers employing evolutionary computation

Alex Raymond Bertels

AUTOMATED DESIGN OF BOOLEAN SATISFIABILITY SOLVERS EMPLOYING EVOLUTIONARY COMPUTATION

by

ALEX RAYMOND BERTELS

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2016

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Bruce McMillin
Dr. Samuel Mulder

ABSTRACT

Modern society gives rise to complex problems which sometimes lend themselves to being transformed into Boolean satisfiability (SAT) decision problems; this thesis presents an example from the program understanding domain. Current conflict-driven clause learning (CDCL) SAT solvers employ all-purpose heuristics for making decisions when finding truth assignments for arbitrary logical expressions called SAT instances. The instances derived from a particular problem class exhibit a unique underlying structure which impacts a solver's effectiveness. Thus, tailoring the solver heuristics to a particular problem class can significantly enhance the solver's performance; however, manual specialization is very labor intensive. Automated development may apply hyper-heuristics to search program space by utilizing problem-derived building blocks. This thesis demonstrates the potential for genetic programming (GP) powered hyper-heuristic driven automated design of algorithms to create tailored CDCL solvers, in this case through custom variable scoring and learnt clause scoring heuristics, with significantly better performance on targeted classes of SAT problem instances. As the run-time of GP is often dominated by fitness evaluation, evaluating multiple offspring in parallel typically reduces the time incurred by fitness evaluation proportional to the number of parallel processing units. The naive synchronous approach requires an entire generation to be evaluated before progressing to the next generation; as such, heterogeneity in the evaluation times will degrade the performance gain, as parallel processing units will have to idle until the longest evaluation has completed. This thesis shows empirical evidence justifying the employment of an asynchronous parallel model for GP powered hyper-heuristics applied to SAT solver space, rather than the generational synchronous alternative, for gaining speed-ups in evolution time. Additionally, this thesis explores the use of a multi-objective GP to reveal the trade-off surface between multiple CDCL attributes.

ACKNOWLEDGMENTS

Before applying to graduate school, I was asked why I wanted to get a master's degree. Was it for a higher salary or because I was not ready to abandon the warm embrace of academia? While both can be perks of the position, I was primarily eager to make my mark on the field I had spent my undergraduate years immersed in. As every academic can attest to, each attempt to make a contribution is not without sacrifice, and we must acknowledge those people in our lives that give us strength in the times that we struggle. Without a doubt, I am forever indebted to my advisor, Dr. Daniel Tauritz. He gave me encouragement in the face of overwhelming obstacles. From my freshman year, Dr. Tauritz has never passed the opportunity to push me to my limits; I am certainly better for all of his efforts.

I must also thank Dr. Samuel Mulder and Dr. Bruce McMillin. Dr. Mulder has always steered me in the right direction during my semesters in school and summers at internships. Dr. McMillin was more than willing to provide guidance in my course work and ensured that I was worthy of this degree.

I am grateful for all my family and friends that encouraged me through the sleepless nights and holidays away from home. I would especially like to thank my brother, Jacob, my sister, Meredith, and, most importantly, my parents, Matthew and Michele, for their love and support throughout my entire life. To them and all those that I could not have completed this degree without, I dedicate my thesis.

I appreciate the funding providing by Sandia National Laboratories through their Critical Skills Master's Program that made my graduate studies possible. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1 INTRODUCTION

In some cases, real-world problems can be modelled as assertions on interactions between components in a defined domain; when these assertions are found to be correct, the assertions have been satisfied. Boolean satisfiability (SAT) instances – the subset of assertion problems that can be represented with Boolean components and associations – are typically divided further into three distinct sets determined by the method used to generate each instance. The less restrictive being random generation, with hand crafted and hard combinatorial instances meeting a few more criteria, and the industrial (or application) class being a direct mapping of real-world problems. Industrial SAT instances inherit associations from their parent problems. Whether the instance is derived from mathematical applications, such as graph coloring or solving Polarium puzzles, or computing-related fields (e.g., cryptography and scheduling) [1], will determine the "structure" of the Boolean expressions and their respective variables. Figure 1.1 illustrates the variable dependency graphs constructed by the DPvis tool* on several application instances. The top-left, top-right, and bottom-left are program understanding propositional formulas from the bounded model checker, BMC†. These particular instances are barrel2, longmult2, and queueinvar2 respectively. The fourth formula, called flat30-1, was taken from the graph coloring dataset flat30-60‡. There are very few pairs of nodes connected by multiple edges in the graph coloring figure, while every edge in the other structures is accompanied by at least one other edge. The higher collocations in the program understanding instances suggest that variable assignments are much more dependent on the assignment of other variables than in other applications. The relationship

*http://www-sr.informatik.uni-tuebingen.de/~sinz/DPvis/
†http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html
‡http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

becomes more apparent when considering the domain from which program understanding instances are derived.

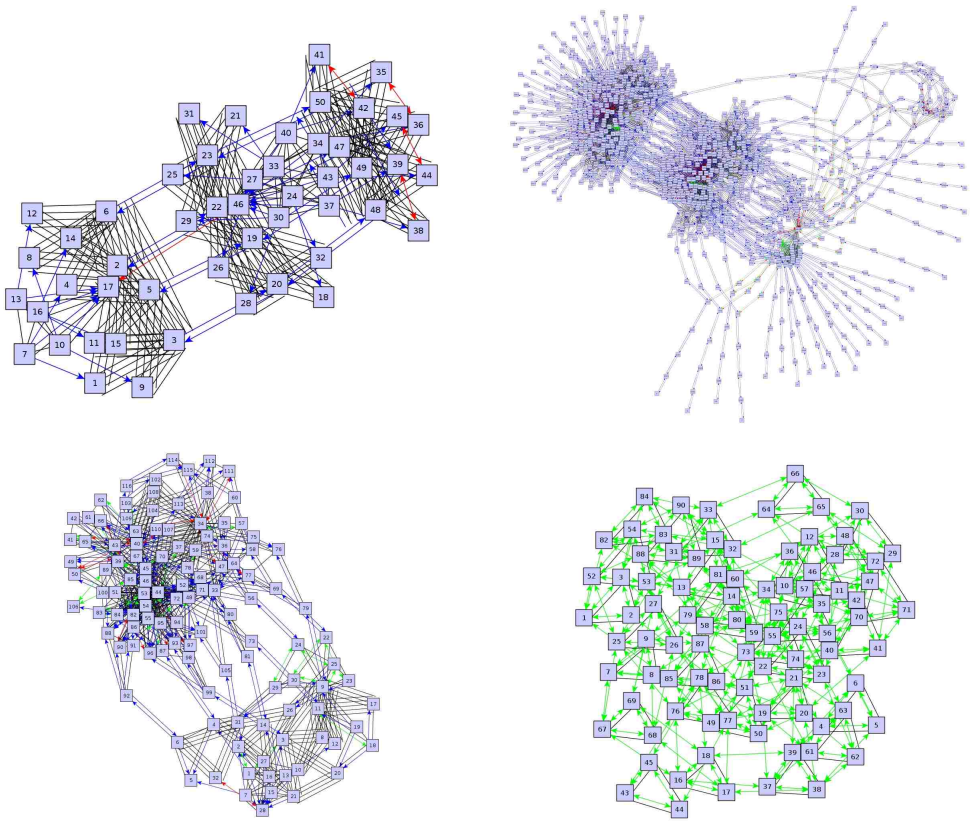


Figure 1.1 DPvis Structure of Application SAT Instances

Digital systems store integers as binary values and interpret arithmetic and comparison operations on these values with logic gates. Occasionally, the values are presented as variables, rather than constants, that depend on the outcome of previously performed operations. The logic gates establish a bit-wise relationship between the given integer values and/or variables and the resultant value. While the result of an arithmetic operation is another n-bit integer, comparison operators

produce a 1-bit status (i.e., 1 for true and 0 for false). All assignments, arithmetic operations, and comparisons with a status that is true can be represented as boolean expressions that are satisfied by at least one set of bits. A path through a program is simply the conjunction of multiple expressions. Given that any boolean expression can be transformed into `AND`, `OR`, and `NOT` gates, a program path can be translated into a SAT instance. Many SAT-solvers have been developed to scale to large instances with low complexity. SAT solvers can be employed to verify paths with predetermined values. Also, if solutions can be found to paths that contain variables with unknown values, then the solution will contain a possible value for that variable.

If no solution is found for a given SAT instance, then the instance can be either undetermined or unsatisfiable. A solution can be proven unsatisfiable if a contradiction is found or if all possible variable assignments have been attempted. In this case, the path through the program will never be executed. However, if the SAT solver has not performed an exhaustive search and cannot find a contradiction, then nothing can be stated about the path through the program.

As previously mentioned, classes of structured SAT instances are created by encoding a specific problem class in SAT. The variable interactions or associations in each instance in a class define a distinct structure. Empirical evidence shows that each SAT solver has an ideal underlying instance structure and that each class of structured instances has an optimal solver [2, 3, 4]. Efficiently solving instances in a specific class requires finding the solver and parameter configuration that perform best for that class.

SAT instances are often structured in the conjunctive normal form (CNF), where clauses are the disjunctive components [5]. Each clause is composed of literals, and each literal can be either a variable or the negation of a variable (e.g., x or $\neg x$). As SAT solvers attempt to find a solution to an instance or prove that the instance is unsatisfiable, the solver must make decisions for the assignment of Boolean variables – often determined by a variable scoring heuristic – in the logical expression. A simple

example of variable assignments is illustrated by Figure 1.2. When many of the decisions cause conflicts in the Boolean statement, a conflict-driven clause learning (CDCL) solver determines that a restart is necessary. Restart heuristics dictate how frequently the solver backtracks to either the beginning of the search or to some stored decision. Static heuristics will restart a search after a set number of conflicts have occurred; smaller limits have been shown to be effective for certain satisfiable problem classes [6]. However, an empirical analysis comparing static to dynamic heuristics indicates that restarts dependent on detailed state-related information are much more effective [6]. Evolving heuristics that borrow primitives from both types may improve the performance for a targeted class.

$$\underbrace{(x \vee \neg a)}_{clause} \wedge (\underbrace{\neg x}_{literal} \vee b) \;=\; \mathbf{T}$$

$a = \mathbf{T}$ and $b = \mathbf{F}$	$(x \vee \mathbf{F}) \wedge (\neg x \vee \mathbf{F})$	CONFLICT $(x \neq \mathbf{T}/\mathbf{F})$
$a = \mathbf{F}$ and $b = \mathbf{F}$	$(x \vee \mathbf{T}) \wedge (\neg x \vee \mathbf{F})$	NO CONFLICT $(x = \mathbf{F})$

Figure 1.2 The conflict shows that under those assignments of a and b the expression cannot be true.

When a CDCL solver encounters a conflict, a "reason" for the conflict is constructed and stored. These "reasons" are called learnt clauses [7]; and, learnt clauses are employed to help make decisions. CDCL solvers can reach millions of conflicts in a run and cannot afford to keep all the learnt clauses. As such, some learnt clauses must be discarded. The clauses that are the most useful are kept and the rest are forgotten. The usefulness of a learnt clause is computed through a scoring heuristic. Generating novel learnt clause scoring heuristics will ensure that the most important clauses are not removed. Developing primitives from CDCL state-related values can make evolving these heuristics possible.

While evolving CDCL heuristics in a population, each solver containing a generated heuristic must be evaluated against several instances from the target problem classes to measure the overall effectiveness of the new heuristic. When dealing with sampled fitness, some instances may take significantly more computational time to evaluate than others. This often leads to large variations in the computational time needed to evaluate the fitness of a trial solution. Additionally, this occurs when the complexity of the representation in a population can significantly vary, such as is often the case in Genetic Programming (GP), where typically limits are placed on genotype size (tree depth in Koza style GP) and larger genotypes are penalized to create parsimony pressure to combat bloat [8, 9]. Hyper-heuristics are a type of meta-heuristic which search program space for the purpose of automating the design of algorithms. They typically employ GP and their fitness evaluation relies on a sample consisting of multiple test cases [10, 11]. With hyper-heuristics, especially in the case of evolving CDCL heuristics, the evaluation time varies drastically with the difficulty of the datasets, the sample size, and the quality of the heuristic. Additionally, these factors can also result in lengthy evolution times.

Given a distributed computing resource, such as a multi-core machine, or parallel cluster, hyper-heuristics implemented as Evolutionary Algorithms (EAs) are often able to reduce overall runtime by distributing individuals in the population to be evaluated concurrently. Synchronous Parallel EAs (SPEAs) maintain the generational step that is typical to most EAs; however, this approach suffers from idle CPU cycles when the fitness evaluation times vary. Asynchronous Parallel EAs (APEAs) eliminate this wasted time by producing offspring as slave nodes in the distributed computing resource become idle.

This thesis demonstrates how structure is introduced into a particular problem class, namely program understanding. In the interest of targeting solvers to particular problem classes, the design and modifications of the ADSSEC (**A**utomated **D**esign of **S**AT **S**olvers employing **E**volutionary **C**omputation) system is discussed in detail.

Empirical studies are included supporting the performance gain of APEAs when compared to SPEAs on global populations of CDCL SAT solver variable scoring heuristics, as opposed to distributed populations such as in island-model EAs and diffusion EAs [12, 13, 14, 15]. Additionally, this thesis investigates the quality of the heuristics produced by the ADSSEC system as well as the effectiveness of its approach. Possible directions for this research are presented to address the limitations and expansion of automatically designing CDCL SAT solvers.

This thesis makes the following contributions:

- Provides empirical evidence of the substantial performance gains of the asynchronous hyper-heuristics approach over the synchronous alternative.
- Introduces a generative hyper-heuristic approach employing genetic programming to automatically construct variable scoring heuristics and learnt clause scoring heuristics for a CDCL solver.
- Significantly decreases CDCL SAT solver heuristic evolution time using an asynchronous parallel evolutionary algorithm.
- Evolves novel variable scoring heuristics that target classes of structured SAT instances.
- Demonstrates the potential of the hyper-heuristic approach to create efficient solver portfolios targeting particular problem classes.
- Discusses the automated design of CDCL SAT solver restart schemes as well as combining multiple components during evolution.

2 RELATED WORK

Populations within a parallel evolutionary algorithm (PEA) can either be structured as a single, centralized population or as multiple decentralized subpopulations [15]. Distributing subpopulations over available machines achieves near-linear scalability, with the sole overhead due to inter-population communication through interchange of select individuals at typically fixed time intervals called epochs. This allows for each subpopulation to evolve semi-independently, while slowly diffusing genetic material throughout all populations. Alba et al. have reported on the effectiveness of various behaviors, particularly distributed and cellular reproduction, in distributed PEAs [12, 13, 14].

Durillo et al. have shown empirical evidence supporting the significant improvement in terms of various quality metrics when employing APEAs rather than SPEAs for NSGA-II [16]. The APEA master process creates and sends individuals to be evaluated as the slave processors become idle. In the generational version, the population is replaced when enough offspring have been generated. With the steady-state alternative, the offspring are considered as each is received. The researchers employed homogeneous populations as the test cases during experimentation. While these results still apply to heterogeneous populations, an in-depth runtime analysis should be completed to measure performance.

Those that have specifically addressed heterogeneous populations note that APEAs are biased toward individuals with shorter evaluation times [17, 18, 19]. This is a result of the master process receiving those individuals sooner and more often, flooding the population. This potentially reduces the search space that can be reached within the runtime. Yagoubi and Schoenauer attempt to circumvent this with a duration-based selection on the received offspring [18]. This supposed defect can be taken advantage of in various situations, one of which is evolving genetic programs,

which must use a mechanism such as parsimony pressure or must minimize a size-related objective value to prevent any individual from becoming too large. If the size of the individual is proportionate to the evaluation time, then the bias provided by heterogeneous evaluation times can be used to produce an implicit parsimony pressure [20, 21]. This characteristic may not necessarily be present in some hyper-heuristics if the size of the genotype has little effect on the evaluation time, as is noted in the present application.

This research applies generative hyper-heuristics [22], approaches to automatically develop heuristics or algorithms, to generate CDCL SAT solvers tailored for an arbitrary, but particular, problem class, to populate solver portfolios. Alternatively to a generative approach, selective hyper-heuristics are provided with pre-existing heuristics from which to choose the best option [22]. While not quite as flexible as generating new heuristics, selecting heuristics can be beneficial if multiple components need to be matched together and effective heuristics are known. In this case, only a single heuristic is being modified during evolution and the generative approach can explore more of the search space. As is typical, the hyper-heuristic employs genetic programming (GP) to automatically reorganize and manipulate the algorithmic primitives constituting the heuristic [11, 23]. These primitives can be as general as state- related variables and binary operations or as specific as carefully constructed functions with tunable inputs. ADSSEC is a hyper-heuristics framework that uses CDCL state-based information and binary operations to automate the development of new restart or learnt clause scoring heuristics. ADSSEC's primitives are more granular than statements in source code and are therefore much more versatile in developing new solver components than the line substitution approach proposed by Petke et al. designed to optimize entire SAT solvers [24, 25].

Ideally, a single solver would be able to adapt to an application at runtime. Tools such as ParamILS [26], SMAC [27], and − most recently − SpySMAC [28] automatically tailor the parameter configurations of reasonably versatile solvers to

particular datasets. While the parameter configurations are adjustable, most of the internal methods of solvers remain the same. The effectiveness of adapting a solver to a problem class solely through parameter optimization is limited by the appropriateness of the solver's architecture for that problem class.

Running all computable solvers with all configurations simultaneously would guarantee the shortest possible time to solve a given instance. However, obtainable resources restrict this parallel procedure to a subset of existing solvers with carefully selected parameters. This method is referred to as a portfolio approach. Xu et al. were able to predict which solvers in a portfolio were able to perform well in particular domains [29, 30]. They did this by calculating values for a set of instances, testing the portfolio on the instances, and using machine learning to relate solvers to a given instance. This pairing of solvers with instance classes allowed Xu et al. to reduce the portfolio to the best suited solvers. Hutter et al. expanded on this work by employing these calculated values to predict the runtime of SAT solvers [31]. Portfolios of algorithms provide high flexibility in discovering the right *existing* solver for the job assuming that the right solver is in the portfolio to begin with.

Previous work has automatically evolved variable selection techniques for stochastic local search (SLS) solvers [32, 33, 34, 35, 36]. However, CDCL solvers are still the most efficient SAT solvers, and although Biere and Fröhlich demonstrated that restart and variable selection schemes drastically impact CDCL solver efficiency for specific problem classes [6, 37], no known work focuses on automatically evolving CDCL heuristic functions. It is believed that appropriate CDCL operations will increase the effectiveness of a CDCL SAT solver in targeting classes of instances with unique structure. The diagram in Figure 2.1 illustrates how ADSSEC-generated SAT solvers relate to the entire SAT solver space.

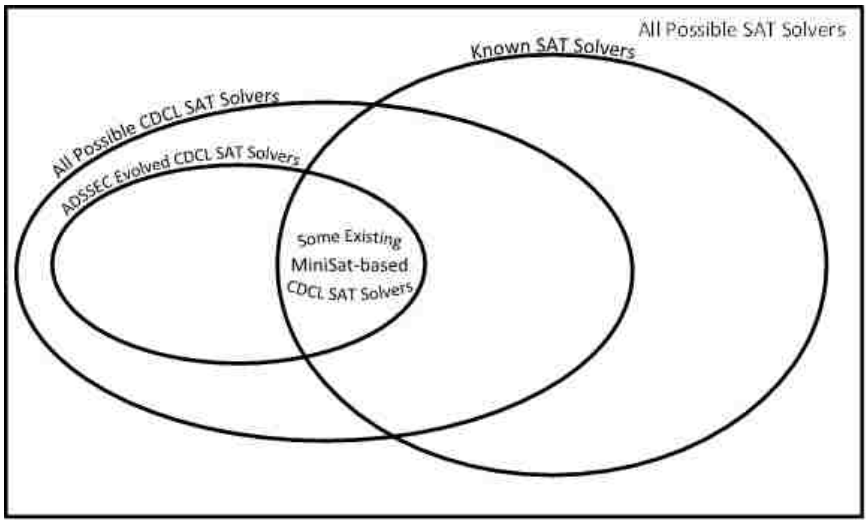


Figure 2.1 Search space of ADSSEC (not to scale)

3 MOTIVATIONAL STUDY OF STRUCTURE IN SAT INSTANCES

As a motivational study of the structure in SAT instances, the following sections describe a brief working example that investigates an encoding of the program understanding program class. This transformation is merely a demonstration of interactions of clauses and variables in application-based instances. The encoding is not a novel contribution of this thesis. Clarke et al. provide a much more expansive and detailed explanation of this problem class [38, 39].

3.1 TRANSFORMING PROGRAM UNDERSTANDING PROBLEMS

In order to illuminate the encapsulation of structure in SAT instances, a tool for converting rudimentary pseudocode into the CNF format poses an introduction to the program understanding SAT problem class. At the current state of the framework, src2sat (read as source-to-SAT) is attempting to answer what possible inputs to a program will follow the path defined by the pseudocode.

3.1.1 Specifications. The steps that src2sat needs to take to discover the possible pseudocode inputs are as follows:

1. Parse the pseudocode representing a path through a program. This includes storing variables to be represented as n-bit integers.
2. Translate each arithmetic, comparison, and assignment operation into equivalent logic gate operations. These resulting boolean expressions will all be combined as a logical conjunction.
3. The conjunction will be converted to the DIMACS CNF to be used later as a SAT instance.
4. Solving this SAT instance will provide an input that will follow this path in a program.

3.1.2 Design. The first step in designing src2sat was to develop a simple pseudocode language that could represent the same operations that might be seen in very basic C programs. When following a path through a program, certain conditions must be met in order for the program to enter specific branches. These conditions along the path can be asserted as being true in this particular application. In this section, the following pseudocode will be employed as a working example during src2sat's conversion process. To keep the example simple, the integers will be represented with 3-bits.

```
input x
z = 2
s = x + 1
assert(s < 2)
y = s - z
assert(y > -2)
```

Both positive and negative values are supported by src2sat. There are a few differences when including negative values. Negative values require the use of two's complement; in that situation, arithmetic operators must allow the most significant carry bit − or borrow bit − to be either a `0` or a `1`. If only positive values are used, that bit can be set to `0`. Also, when comparing a positive to negative value, the most significant bit must be checked. If that bit is not checked, then negative values will register as being greater. Allowing for both will add additional clauses to the SAT instance.

Floating-point values, or floats, have not been incorporated into src2sat. In addition to needing support for multiple types, these numbers would require different logical representations for each operation. Floats often consist of three components and are structured as a *significand* multiplied by some *base* to a given *exponent*. Operations would need to consider that operands may not necessarily have the same type, and, more importantly, the same structure.

3.1.3 Pseudocode Parsing. Each program will contain at least one line that starts with the keyword `input`. These are the only commands that will not produce a boolean expression. The instruction simply indicates that the program needs to store that variable in the symbol table as an n-bit integer. In the example, the input variable will be stored as

$$\mathtt{x} = \mathtt{x}_2\mathtt{x}_1\mathtt{x}_0$$

All other instructions will produce boolean expressions that were designed from logic gates that at some level can be represented with just `AND`s, `OR`s, and `NOT`s. However, to make things easier and less verbose, other more complex boolean operations were used, and the translation takes place in the next step.

The symbol table stores each variable name with their respective number of assignments. Given that the result of each assignment to a specific variable is independent, additional instances of that variable are needed to store each assignment's result. For example, the first time `x` is assigned in the pseudocode, the boolean relation will use `x0`. The second assignment to `x` will use `x1` in the expression, the third `x2`, and so on.

3.1.3.1 Assignment operation. In src2sat's current state, the assignment operation always has a single variable on the left-hand side. The right-hand side can be another variable, an integer, or an expression. To ensure that the value of bit `a` on the left-hand side is equivalent to bit `b` on the right-hand side, the following expression will be generated

$$\mathtt{a}\ \texttt{XNOR}\ \mathtt{b}\ \textit{or}\ \overline{\mathtt{a} \oplus \mathtt{b}}$$

In the example, the line `z = 2` would be interpreted as

$$(\overline{\mathtt{z}_2 \oplus 0})(\overline{\mathtt{z}_1 \oplus 1})(\overline{\mathtt{z}_0 \oplus 0})$$

3.1.3.2 Comparison operations. All comparison operators will appear as an assert statement (i.e., `assert(`*`condition`*`)`). The comparison operations implemented so far are *equivalence*, *not equal*, *greater than*, *greater than or equal to*, *less*

than, and *less than or equal to*. Each allow for both sides of the operator to be either an integer or a variable. The logical disjunction of the *equal* operator and the *greater than* and *less than* operators allows the formations of the *greater than or equal to* and *less than or equal to* operators respectively.

3.1.3.2.1 Equal. Asserting that two values are equal is the same as assigning one value to the other. This is handled in exactly the same manner as the assignment operation previously described. To define the *not equal* operator, the resulting expression for *equal* is simply negated.

3.1.3.2.2 Greater than. In deciding whether one value is greater than another, the boolean expression must rely on the relation between several bits in the two values. The logical statement that defines `a > b` for three bit integers is

$$((\overline{a_2 \oplus 0}) \cdot (\overline{b_2 \oplus 1})) +$$

$$((\overline{a_2 \oplus b_2}) \cdot (a_2\overline{b_2} + (x_2a_1\overline{b_1}) + (x_2x_1a_0\overline{b_0}))); \text{ where } x_i = (\overline{a_i \oplus b_i})$$

The statement `assert(y > -2)` would have the following translation

$$((\overline{y_2 \oplus 0}) \cdot (\overline{1 \oplus 1})) +$$

$$((\overline{y_2 \oplus 1}) \cdot (y_2\overline{1} + (\overline{y_2 \oplus 1})y_1\overline{1} + (\overline{y_2 \oplus 1})(\overline{y_1 \oplus 1})y_0\overline{0})))$$

3.1.3.2.3 Less than. The representation for `a < b` is as follows (assuming three bit integers)

$$((\overline{a_2 \oplus 1}) \cdot (\overline{b_2 \oplus 0})) +$$

$$((\overline{a_2 \oplus b_2}) \cdot (\overline{a_2}b_2 + (x_2\overline{a_1}b_1) + (x_2x_1\overline{a_0}b_0))); \text{ where } x_i = (\overline{a_i \oplus b_i})$$

The statement `assert(s < 2)` would be translated to the following expression

$$((\overline{s_2 \oplus 1}) \cdot (\overline{0 \oplus 0})) +$$

$$((\overline{s_2 \oplus 0}) \cdot (\overline{s_2}0 + (\overline{s_2 \oplus 0})\overline{s_1}1 + (\overline{s_2 \oplus 0})(\overline{s_1 \oplus 1})\overline{s_0}0)))$$

3.1.3.3 Arithmetic operations. Similar to the comparison operators, both sides of an arithmetic operator must be either an integer or a variable. Both addition and subtraction introduce new variables to handle the carry and borrow bits. Currently src2sat only allows for integer values.

3.1.3.3.1 Addition. The boolean expression for addition is based on the logic gates that construct a full adder. Two expressions are needed with the addition of the carry bits (c). The solution is represented by the variable s.

$$(\overline{\mathtt{s} \oplus (\mathtt{a} \oplus \mathtt{b} \oplus \mathtt{c}_{in})}) \cdot (\overline{\mathtt{c}_{out} \oplus (\mathtt{ab} + \mathtt{ac}_{in} + \mathtt{bc}_{in})})$$

The expression associated with the solution, s, in the example `s = x + 1` is defined as

$$(\overline{\mathtt{s}_2 \oplus (\mathtt{x}_2 \oplus 0 \oplus \mathtt{c}_2)}) \cdot (\overline{\mathtt{s}_1 \oplus (\mathtt{x}_1 \oplus 0 \oplus \mathtt{c}_1)}) \cdot (\overline{\mathtt{s}_0 \oplus (\mathtt{x}_0 \oplus 1 \oplus 0)})$$

and the carry bits are handled by

$$(\overline{\mathtt{c}_3 \oplus (\mathtt{x}_2 0 + \mathtt{x}_2\mathtt{c}_2 + 0\mathtt{c}_2)}) \cdot (\overline{\mathtt{c}_2 \oplus (\mathtt{x}_1 0 + \mathtt{x}_1\mathtt{c}_1 + 0\mathtt{c}_1)}) \cdot$$

$$(\overline{\mathtt{c}_1 \oplus (\mathtt{x}_0 1 + \mathtt{x}_0 0 + 1 \cdot 0)})$$

3.1.3.3.2 Subtraction. The subtraction operator is similar to the addition operator; however, the operation relies on the outcome of the borrow bits (d).

$$(\overline{\mathtt{s} \oplus (\mathtt{a} \oplus \mathtt{b} \oplus \mathtt{d}_{in})}) \cdot (\overline{\mathtt{d}_{out} \oplus (\neg\mathtt{ab} + \neg\mathtt{ad}_{in} + \mathtt{bd}_{in})})$$

The statement `y = s - z` is interpreted with the following two expressions

$$(\overline{\mathtt{y}_2 \oplus (\mathtt{s}_2 \oplus \mathtt{z}_2 \oplus \mathtt{d}_2)}) \cdot (\overline{\mathtt{y}_1 \oplus (\mathtt{s}_1 \oplus \mathtt{z}_1 \oplus \mathtt{d}_1)}) \cdot (\overline{\mathtt{y}_0 \oplus (\mathtt{s}_0 \oplus \mathtt{z}_0 \oplus 0)})$$

$$(\overline{\mathtt{d}_3 \oplus (\neg\mathtt{s}_2\mathtt{z}_2 + \neg\mathtt{s}_2\mathtt{d}_2 + \mathtt{z}_2\mathtt{d}_2)}) \cdot$$

$$(\overline{\mathtt{d}_2 \oplus (\neg\mathtt{s}_1\mathtt{z}_1 + \neg\mathtt{s}_1\mathtt{d}_1 + \mathtt{z}_1\mathtt{d}_1)}) \cdot$$

$$(\overline{\mathtt{d}_1 \oplus (\neg\mathtt{s}_0\mathtt{z}_0 + \neg\mathtt{s}_0 0 + \mathtt{z}_0 0)})$$

3.1.4 Resultant DIMACS SAT Instance. The src2sat program is dependent upon the PyEDA library (version 0.25)*. This library can take the boolean

*https://pyeda.readthedocs.org/en/latest/

expressions as input and produce a DIMACS SAT instance. This library can also solve the generated instance. With the example pseudocode above, the SAT instance produced contains 18 variables in 59 clauses when assuming 3-bit integers. The number of clauses and variables is dependent on the number of bits needed to represent the values (Table 3.1). Also, three possible inputs would follow this path through a program. When solving the SAT instance, `x` is found to be `0`, `7`, and `-8`. The last two results are the boundary conditions when arithmetic operations can cause the values to wrap around from positive to negative and negative to positive.

Table 3.1 Clauses and Variables needed to represent example using n-bit values

***n*-bits**	**Clauses**	**Variables**
3	18	59
4	24	83
5	30	107
6	36	131
7	42	155
8	48	179
9	54	203
10	60	227
⋮	⋮	⋮
32	192	755

The clause-to-variable ratio relies on varying aspects of the program path being analyzed. More variables, rather than values, used throughout the path requires that more bits be solved for in the SAT instance. In the example, if the variable, `z`, had been replaced by the value `2` along the path, then there would have only been 15 clauses and 40 variables. The bloat introduced by using `z` could potentially be mitigated by employing some preprocessing techniques on the path before the conversion.

Table 3.2 shows extended versions of the earlier example. These cases indicate that each operation has a different cost with regard to the number of clauses in the SAT instance. Additionally, the use of variables and values in the source code have an influence over the number of variables in the SAT instance. In the left extended version, the 3-bit conversion results in a SAT instance with 30 clauses and 79 variables. On the right side, the result contains 30 variables and 113 variables. A positive linear relationship exists between the number of lines and variable references in the source code and the number of clauses and variables in the SAT instance, respectively.

Table 3.2 Extended Examples

`input x`	`input x`
`z = 2`	`z = 2`
`s = x + 1`	`s = x + 1`
`assert(s < 2)`	`assert(s < 2)`
`y = s - z`	`y = s - z`
`assert(y > -2)`	`assert(y > -2)`
`m = 0 + 0`	`m = x + x`
`n = 0 - 0`	`n = m - x`
`assert(1 == 1)`	`assert(x == n)`

3.2 CONDITIONAL CODE EXECUTION

While not implemented in src2sat, branching and looping are instrumental in representing most common languages. The following sections provide a brief overview how these affect the encoding of program understanding to SAT instances.

3.2.1 Branching. Branching in programs requires that some subset of statements are associated with, or implied by, one or more conditions. Each branch may make certain assertions or produce assignments to variables that determine future

code execution. The outcome of a set of branches can be represented with a disjunction of implications. Below is a simple example of a ternary operation:

```
a = b ?  0 :  1
```

In this case, if `b` is true (`1`), then `a` becomes `0`; and, if `b` is false (`0`), then `a` becomes `1`. This can be converted to the following disjunction:

$$(\neg b \cdot a) + (b \cdot \neg a)$$

A more general scenario would be to see the ternary structured as an *if-else* branch. See the following pseudocode:

```
...
if (c)
  x = 1
else
  x = 3
assert(x > 2)
...
```

In order to determine if there is a solution to the SAT instance – or a path through the program – where `x > 2`, the statements in both branches must first be associated with their respective conditions. Given that the logic operations behind the assignment operator and greater than operator are discussed above, these operators will be used in the boolean representation. Also as previously mentioned, each assignment to the same variable must be tied to a unique identifier. The pseudocode will be reinterpreted such that `x0 = 1`, `x1 = 3`, and `x2 > z`. With these new symbols, the relationship can be expressed as

$$((c \cdot (x2 = x0)) + (\neg c \cdot (x2 = x1))) \cdot (x2 > 2)$$

With the inclusion of `elseif` conditions and the increase in scope complexity, these expressions become more difficult to represent. The dependencies of variables and conditions are more manageable when stored as edges in a tree or graph.

3.2.2 Looping. Loops provide additional complexity in that the number of iterations is linearly related to the number of clauses needed to represent the loop in the SAT instance. If the number of iterations is known, then the loop can be "unrolled" before generating clauses. Other cases cannot be addressed with such simplicity.

One approach to loops with an unknown number of iterations is to generate multiple SAT instances. Initially, develop a boolean expression when the loop is not executed. If a solution cannot be found for the instance, then iterate through the loop once in the expression. Continue to increase the number of iterations until the instance can be satisfied. If some prior knowledge about the loop is known (e.g., `do-while` loops), then this method may be modified.

The greatest common denominator (gcd) problem will be used as an example to show how this process would take place. The pseudocode is structured as follows:

```
function gcd(a, b)
while b != 0
  t := b
  b := a mod b
  a := t
return a
```

In this example, assume that `a` and `b` will have known values. This initial boolean expression would attempt to solve for:

0 iterations

```
b0 == 0
```

If `b` does not equal zero, then no solution can be found for the generated SAT instance; and, another iteration of the loop will need to be included.

1 iteration

```
t0 = b0
b1 = a0 mod b0
```

```
a1 = t0
b1 == 0
```

This process will continue until the expression is satisfied.

2 iterations

```
t0 = b0
b1 = a0 mod b0
a1 = t0
t1 = b1
b2 = a1 mod b1
a2 = t1
b2 == 0
⋮
```

This may require a maximum number of iterations or maximum solver wall time to determine when a SAT instance is no longer feasible to generate and solve for the given problem.

3.2.3 Remarks on Program Understanding Structure. The pseudocode language serves as a proof-of-concept. The src2sat tool is still far from encapsulating more commonly used languages. The pseudocode would better represent the program understanding problem class by incorporating conditional code execution and memory addressing. A possible starting point would be to use an assembly language (e.g., x86, ARM) as a reference. Also, specific paths will need to be extracted from the program to answer the question of possible inputs. While verification was the focus in this example, other questions may be answered with this conversion. More general than program understanding, the above working example demonstrates how structure is introduced in industrial SAT instances. In attempting to take advantage of that structure, there is certainly potential to discover or develop SAT solvers specifically tailored for this class of SAT instances. The remainder of this thesis details an approach to automate the design of problem-specific SAT solvers.

4 METHODOLOGY

The ADSSEC (**A**utomated **D**esign of **S**AT **S**olvers employing **E**volutionary **C**omputation) system is a hyper-heuristics framework designed to automate the development of SAT solvers. The SAT solvers generated by ADSSEC are trained against specific sets of instances that represent a particular encoding or problem class. ADSSEC aims at discovering solver heuristics that perform well on datasets of interest, not necessarily produce a solver that is better in general. The following two sections feature the initial and updated implementations of the system, as well as further discussion on the integration of potential heuristic representations and existing CDCL SAT solver mechanisms.

4.1 ADSSEC VERSION 1.0

Influenced by the success of Fukunaga in improving SLS runtimes by evolving specific heuristics [34], ADSSEC uses GP to evolve variable scoring heuristics to automatically target CDCL solvers to specific classes of structured instances. In particular, the system employs Koza-style GP [40] as it is well suited to representing the parse trees of variable scoring heuristics. These heuristics assign high scores to more important variables leading CDCL solvers to select influential variables when making a decision. Biere and Fröhlich demonstrated that the current best variable scoring heuristics are roughly equal in runtime performance when evaluated across many instance classes [37]. While no one heuristic function is best in all cases, each instance class has at least one heuristic that provides the best average runtime. Biere and Fröhlich's work motivated the choice to adapt CDCL solver variable scoring heuristics. ADSSEC creates an initial population of variable scoring heuristics and evolves the population through mutation and recombination. ADSSEC evaluates

these heuristics by replacing the variable selection heuristic in MiniSat [7], a commonly used efficient and deterministic CDCL solver with dense source code. While ADSSEC employs a standard parent selection before producing offspring, its unique survival selection was specifically constructed for use by APEAs. ADSSEC returns the heuristics from the final population after reaching the termination criteria.

4.1.1 Heuristic Representation. Mapping variable scoring heuristic functions to objects easily manipulated in a GP is fairly straight-forward. Each scoring heuristic is represented as a parse tree where non-terminal nodes are operators and terminal nodes are state-related values.

Derived from currently implemented variable scoring heuristics [37], ADSSEC defines the following terminal nodes:

- Score (s): The previous score of the variable.
- Conflict Index (i): The current number of conflicts encountered.
- MiniSat Variable Decay Amount (f): Initially used as the rate of variable score decay in MiniSat. Now f is just used to derive MiniSat's Variable Increment Value (*MiniSat Default: 0.95*).
- MiniSat's Variable Increment Amount ($g = (1/f)^i$): The amount MiniSat increases the score of a variable.
- Constant (C): A constant value in $\{1, 2, 3, \ldots, 10\} \cup \{0.0, 0.1, 0.2, \ldots, 0.9\}$.
- Special Component (H):

$$h_i^m = \begin{cases} 0.5 \cdot s & \text{if } m \text{ divides } i \text{ evenly} \\ s & \text{otherwise} \end{cases} \tag{4.1}$$

  where m is a power of 2: $\{2, 4, 8, \ldots, 1024\}$.

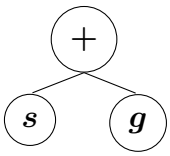
ADSSEC defines the following non-terminal (binary) operator nodes: Addition ($+$), Subtraction ($-$), Multiplication ($*$), and Division ($/$). These arithmetic

operators may be applied because all the terminal nodes relate to either integer or floating-point values.

These nodes permit the evolution of novel schemes while still representing current schemes. For example, consider MiniSat's current variable scoring heuristic:

$$s' = s + g \ .$$

In ADSSEC, the following parse tree represents this heuristic:



Put into words, the updated score of the variable is the sum of the previous score and the variable increment value maintained by MiniSat.

Again, ADSSEC evolves the parse tree genetic encodings. To evaluate each variable scoring heuristic, the parse tree is converted into a C++ statement. The original variable scoring heuristic is replaced in a pre-built MiniSat 2.2 by compiling and linking in the new variable scoring heuristic (C++ statement). The resulting solver is executed on test instances to evaluate the effectiveness of the heuristic. This method takes advantage of MiniSat and the performance derived from being implemented in C++ while reducing development time.

4.1.2 Objective. The objective score represents how well an evolved version of MiniSat performs on a provided training set of instances. The true objective score is a function (e.g., average) over all instances in the problem class being targeted. However, as it is infeasible to compute over a potentially infinite set of instances, instead a small sampling of instances provides an approximation as is discussed later. Determining the best performance measure to use in ADSSEC is difficult. Traditionally the intent is to reduce the runtime needed to either find a solution or prove unsatisfiability. However, because ADSSEC evaluates several instances in parallel on the same hardware, runtimes for individual instances are inconsistent even with a deterministic solver. Therefore, runtime is substituted with the number of variable

decisions, which is a more consistent metric. An improved variable scoring heuristic should reduce this value. The per-instance sub-score for an evolved variant is the ratio of the number of decisions needed by the variant to that needed by the original selection scheme.

The objective score is then simply the average of all the instance sub-scores. Given that all the sub-scores are expressed relative to the original MiniSat, any evolved individual that performs identically to MiniSat's variable scoring heuristic will end up with an objective score of 1.0. Lower scores indicate better schemes.

Occasionally, the EA will construct inadequate heuristics that cause the solver to require an inordinate number of decisions to reach a conclusion. The limiting functions are defined to prevent wasting evaluation time on such heuristics. ADSSEC relies on default MiniSat's performance to approximate reasonable limits for any given SAT instance. Initially, ADSSEC limits an evolved MiniSat to three times the number of decisions the original MiniSat needed to solve that instance. While the multiplier of three is user-configurable, manual tuning indicated that this limit was fairly generous without wasting an excessive amount of evaluation time. These generous limits are required to collect decent heuristics in the population – decent heuristics provide complex genetic material for later optimization; they do not time out on all tested instances, but are generally worse than the original MiniSat. However, as ADSSEC progresses through the evolutionary process, the interest shifts to exploiting the heuristics that are strictly better than the original. As such, the decision limit linearly decreases down to the exact number of decisions MiniSat needed for a specific instance or the average number of decisions for the sample set. For example, if ADSSEC is to complete 5000 evaluations throughout the run and the decision limit multiplier decreases from 3.0 to 1.0, then the multiplier is decremented by $((3.0 - 1.0)/5000 = 0.0004$ after each evaluation.

Ideally, an accurate objective score would be determined by executing the evolved variable scoring heuristic against the entire dataset of interest. Because this is

generally too costly, ADSSEC utilizes strike-based sampling to gauge the effectiveness of a MiniSat variant. ADSSEC randomly selects a user-defined number of instances from the given training set to evaluate a variant. At the start of evolution, there is a bias toward selecting easier instances. The initial population contains mostly low-quality heuristics and evaluating these heuristics against difficult instances wastes evaluation time. As such, a bias is placed in favor of selecting easier instances in early evolution. This bias linearly transforms to a uniform selection at the end of the evolutionary process. For each instance in this selection, ADSSEC executes the evolved variant and assigns a sub-score ratio as described before. If that variant reaches the decision limit for that instance, then the variant receives a strike and a sub-score of the current decision-limit multiplier. After a variant reaches a user-defined number of strikes, all remaining sub-scores are assigned the current decision-limit multiplier.

4.1.3 Evolutionary Algorithm. The following sections define the evolutionary operations implemented within ADSSEC.

4.1.3.1 Population initialization. To create each individual in the initial population of variable scoring heuristics, ADSSEC randomly generates a parse tree from the primitives, or nodes, described previously (Section 4.1.1). First, as experimentation shows that no single terminal node produces an effective scoring scheme, ADSSEC assigns a random operator node to the root of the tree. ADSSEC then assigns two random nodes to the left and right branches of the operator node. There is a 50% chance that each node will be terminal (versus non-terminal). If the node is non-terminal, then ADSSEC repeats the process and assigns a random operator node. If the node is terminal, then there is a 50% chance that the node will be the previously assigned score, s; if not s, ADSSEC randomly assigns one of the other terminal node options. This bias was introduced because most current schemes appear to rely on the previous variable score. The maximum depth of a tree generated for the initial population was manually tuned to eight. Smaller depths contained much

less genetic diversity while larger trees produced complex heuristics that rarely solved instances in the decision limits.

4.1.3.2 Parent selection and variation operators. ADSSEC uses one of two methods to develop a single offspring (variant): mutation or recombination. For mutation, ADSSEC simply randomly selects a subtree in a random individual's parse tree and replaces it with a new branch generated using the rules established in population initialization (Section 4.1.3.1) – without a depth limitation. Typically, APEAs mitigate tree growth by promoting an implicit parsimony pressure. This is under the assumption that smaller trees have shorter evaluation times and return to the population sooner. However, the strike-based sampling terminated bad heuristics quickly, which partially eliminated the implicit pressure. Fortunately, most overly-complicated heuristics receive poor objective scores and are removed during survival selection. For recombination, ADSSEC implements a sub-tree crossover: the system randomly selects two individuals in the population and replaces a random branch from the first parent with a random branch from the second parent. Again, this procedure only produces a single child.

4.1.3.3 Survival selection. The survival selection chooses which individuals in the population continue into the next generation. In ADSSEC, genetically diverse selection is encouraged so that smaller parse trees (which are generated more easily) do not flood the population. Certain small heuristics have adequate performance and, had one been discovered early on in evolution, could be spread throughout the population if diversity was not maintained.

Crowding functions are selection functions that excel at promoting genetic diversity in the population [41]. In a standard crowding function, an offspring competes with its closest parent, either replacing the parent or being dropped from the population in favor of the parent. In an APEA, however, generations are not clearly delineated and a parent can have multiple offspring being evaluated simultaneously. An asynchronous crowding function was developed that allows offspring to compete

with either their parents or any *'siblings'* – or potentially descendants of siblings – that replaced the parents. A computationally cheap distance function compares histograms of node types (e.g., addition, constant, conflicts, etc.) to determine the closest remaining relative in the current population. To provide a fair comparison between the models, the SPEA version of ADSSEC employs the same crowding selection method.

Parents have to be uniformly selected at random to ensure that each individual has an equal chance of providing genetic material to the pool. Additionally, uniform selection allows an equal chance of producing offspring, which can eliminate less fit parents from the population with the employed survival selection.

4.1.3.4 Termination. ADSSEC terminates the evolutionary cycle after completing a user-defined number of evaluations. However, throughout the run individuals may be replaced by randomly generated parse trees if the population has become stale or has converged. If the best individual has not been improved in a user-defined number of evaluations, ADSSEC introduces new material to the gene pool. Currently, all variants whose performance is worse than that of the original MiniSat are replaced. This mechanism is useful in restarting the exploration of the variable scoring heuristics search space.

4.2 ADSSEC VERSION 2.0

The second version of ADSSEC attempts to address the limitations of the previous version. Also, ADSSEC Version 2.0 introduces a representation for learnt clause scoring heuristics and discusses the potential for incorporating restart schemes.

4.2.1 Heuristic Representations. Similar to the representation in evolving CDCL variable scoring heuristics, the primitives for the restart and learnt clause scoring heuristics need to be derived from modern CDCL solvers. Recently, these two heuristics of several solvers have adopted an attribute of learnt clauses as a metric

of interest. This measurement, labeled as the Literal Block Distance (LBD), essentially indicates how many decisions were made that contributed to the creation of a learnt clause. Audemard and Simon describe the significance of this value as well as how LBDs can be applied to restart schemes [42, 43]. Their solver, Glucose, stores the previous W LBDs and, if the average of the LBDs in that window multiplied by a constant (K) exceeds a calculated threshold, then a restart is triggered. Biere and Fröhlich attempt to replace this window with an exponential moving average where the more recent LBDs are more influential [6]. This function still contains the constant (K), but substitutes the window size (W) with an exponential smoothing parameter (α). Currently, the MiniSat restart simply backtracks to the start when a set number of conflicts (C) has been reached. Evolving this heuristic will require selecting both the approach and the values of the associated parameters. Recombination of individuals in the population will require that the selected approaches share parameters. Audemard and Simon conclude that solvers with successful restart schemes will still perform poorly without useful learnt clauses [43]. As such, effective learnt clause scoring heuristics will need to be discovered before evolving restart schemes, which are not currently implemented in ADSSEC Version 2.0.

The MiniSat default learnt clause scoring heuristic determines the usefulness of a learnt clause by how active the clause is [7]. If a learnt clause is employed when a decision is made, then the activity score of that clause is bumped. The more active clauses are assumed to be more useful. Less active clauses are removed to allow for more learnt clauses to be stored. The Glucose solver takes a different approach to determining the usefulness of a learnt clause. When a solver makes a decision for the assignment of a variable, this can propagate to assigning values for other variables without needing to make a decision. The decisions that propagate through many variables tend to be important and present themselves as low LBD scores in learnt clauses. Glucose assumes that the useful learnt clauses will have smaller LDB scores [42]. ADSSEC manipulates primitives derived from both of these

assumptions to evolve novel learnt clause scoring heuristics. In particular, the system uses Koza-style GP [40] as it is well suited to represent the parse trees representing the heuristics. ADSSEC defines the following terminal nodes:

- Score (s): The previous score of the learnt clause.
- Conflict Index (i): The current number of conflicts encountered.
- Minisat's Clause Increment (g^i): The amount MiniSat increases the score of a clause. As this value grows exponentially, ADSSEC only allows for g to be selected from the continuous range of $[1.0 - 1.1]$.
- LBD (lbd): The LBD of the learnt clause.
- LBD Inverse ($lbdI$): Computed as $(1/lbd)$. The higher learnt clause scores are kept and, therefore, low LBDs are more useful.
- Number of Literals (n): ADSSEC Version 2.0 promotes solvers that use minimal memory, so smaller learnt clauses are preferred.
- Constant (C): A constant value in $\{1, 2, 3, \ldots, 10\} \cup \{0.0, 0.1, 0.2, \ldots, 0.9\}$.

ADSSEC also defines the following non-terminal (binary) operator nodes for the learnt clause scoring heuristics: Addition ($+$), Subtraction ($-$), Multiplication ($*$), and Division ($/$). Similar to the operators in the variable scoring heuristics, these arithmetic operators may be applied because all the terminal nodes relate to either integer or floating-point values.

Similar to the variable scoring heuristic, the learnt clause scoring heuristic is translated to C++ and then replaces the original heuristic. Also, a few additional lines are added for maintaining the values of nodes with type g^i. Due to the varying complexity of the heuristics, a design decision had to be made when addressing the reduction of learnt clauses. MiniSat considers two conditions when selecting which learnt clauses to remove. After the learnt clauses are sorted, MiniSat removes the clauses with the lowest scores until the count is half of the stored learnt clauses. MiniSat will continue to remove clauses with scores less than a set threshold: g^i divided by the number of learnt clauses. The magnitude of scores when employing

evolved heuristics may no longer be consistent with this threshold and, as such, this particular threshold cannot be used. However, ADSSEC may benefit from having a mechanism for eliminating useless learnt clauses, so the threshold was set at one divided by the number of learnt clauses. Any clauses with small or negative scores will be removed from the set. This allows much of the MiniSat code to remain unchanged and for ADSSEC to utilize existing functionality with the new heuristics.

4.2.2 Objective. While measuring SAT solver performance is typically dependent solely on runtime, concurrently executing multiple solvers heavily skews these results. Using the number of decisions the solver needed to find a solution is a more consistent metric. However, this value alone is not always proportionate to the runtime of a solver and cannot be the only objective score. As supported by the results obtained using Version 1.0, ADSSEC Version 2.0 includes the total number of learnt clause literals (i.e., conflict literals) as a second objective. This promotes solver variants that can solve instances in as few decisions as possible while also minimizing memory allocation.

The previous version of the ADSSEC system suffered from an inadequate "timeout" technique. For a given instance being executed by an evolved MiniSat variant, the solver was set to terminate if a solution had not been discovered in a calculated number of decisions relative to the default MiniSat's performance. Unfortunately, these limitations could be unreasonable for the evolved variant if MiniSat quickly solved the instance. Version 2.0 loosens these bounds to be dependent on the maximum of either the average number decisions for the sample set of instances or the number of decisions MiniSat needed. This modified method should allow each heuristic a fair comparison against the original MiniSat heuristic on the entire sample set rather than a single instance. Additionally, the objective sub-scores were modified to be ratios relative to the average number of decisions and conflict literals for the entire sample set rather than just the same instance. This promotes an overall improvement rather than an improvement on an instance-by-instance basis. Also,

should a heuristic be re-discovered and evaluated multiple times, the objective scores will be averaged over the number of occurrences. Such a check verifies or corrects the previous evaluations and prevents a single heuristic from flooding the population.

4.2.3 Selection. The crowding survivor selection could no longer be used with the addition of the second objective. Instead, a multi-objective approach similar to the NSGA-II algorithm is employed [44]. This allows individuals in the population to be removed without requiring that direct offspring be the replacement. The uniform parent selection was altered to a k-tournament selection.

5 EXPERIMENTATION

In the following three experiments, ADSSEC is employed to collect results, and an analysis of these results offers a discussion on the design of such a system. The first aims to more clearly understand the potential for asynchronous parallel approaches in hyper-heuristics, particularly in the case of ADSSEC Version 1.0. Utilizing the same version of ADSSEC, the second experiment considers the effectiveness of automatically constructing SAT solvers for specific sets of instances. Finally, the third explores and empirically analyzes the performance of an expanded multi-objective version of ADSSEC (Version 2.0).

5.1 ASYNCHRONOUS VERSUS SYNCHRONOUS APPROACHES

This section empirically evaluates the performance of asynchronous parallel evolutionary algorithms (APEAs) when compared to synchronous parallel evolutionary algorithms (SPEAs) on global populations of CDCL SAT solver variable scoring heuristics.

5.1.1 Experimental Setup. Ideally, one would want to construct entire solvers for a given problem, and ADSSEC demonstrates the obstacles and, more importantly, the potential of adapting a single component of a CDCL solver. The experiments with the prototype ADSSEC system require datasets that have:

- instances that ADSSEC can feasibly train on in a short period of time (each instance should require at most a few seconds for the original MiniSat to solve)
- enough instances to sufficiently represent a distinct instance class for both training and testing
- instances that are difficult enough that the instance can benefit from a fitted heuristic

Unfortunately, these requirements make many of the usual SAT datasets inappropriate for the initial prototype experiments. Instances from previous SAT competitions attempt to challenge the capabilities of the solvers, so many require too significant an amount of time to solve. Many publicly available datasets contain too few instances to sufficiently represent the distinct problem class or the instances are so simple that nothing is gained by creating a fitted heuristic. The idea behind generating datasets for ADSSEC is that generators provide enough control to meet these criteria while preventing bias by hand-selecting specific instances.

A modularity-based generator developed by Jesús Giráldez Cru* was used to create 80 instances for the datasets. These instances simulate an underlying structure that may be found in an industrial class of instances. Each CNF contained 5000 variables in 19000 clauses. The generator allows the user to specify the structure of each instance; the tool generated 90 communities with 3 literals per clause. The seeds 1 through 40 were used. Half of the instances were configured with a modularity of 0.85 and the other 40 with 0.9. A majority of the instances were satisfiable while 32 were unsatisfiable. ADSSEC was trained on select subsets of 32 instances where the sample size was 15 allowing up to 5 strikes per evaluation. To obtain the subsets, the 80 instances were first sorted by the number of decisions default MiniSat needed to solve each. Then, the instances were divided into five groups of sixteen instances, where Group 0 needed the least number of decisions and Group 4 needed the most decisions. The final subsets used for training were constructed as follows:

- Dataset A: combined Group 0 and Group 1
- Dataset B: combined Group 0 and Group 2
- Dataset C: combined Group 0 and Group 3
- Dataset D: combined Group 0 and Group 4

*http://www.iiia.csic.es/~jgiraldez/

Dataset A has the least variation in evaluation time and Dataset D has the most. Some instances can be solved in less than a thousandth of a second while the longer ones can take several seconds.

The computationally extensive search made automated tuning of ADSSEC's parameters infeasible; thus, manual tuning was used to discover the configuration in Table 5.1. ADSSEC created an initial population of 20 random individuals. The master process used 20 slaves processes for evaluating offspring, either synchronously producing offspring at each generation or asynchronously creating new offspring as each node became available.

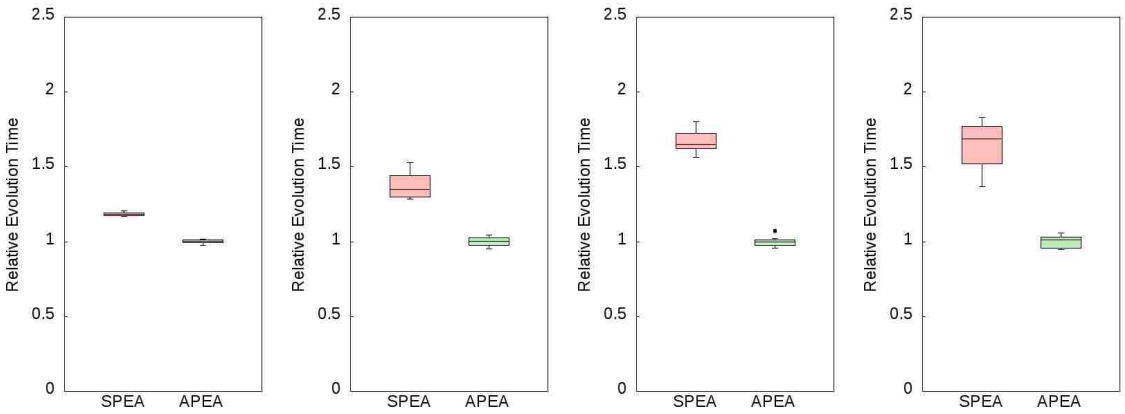
ADSSEC selected parents uniformly for either recombination or mutation – with a mutation probability of 0.10 and, subsequently, a recombination rate of 0.90 – and used a shared crowding method for survival selection. ADSSEC was configured to terminate after 1000 evaluations and restart after 100 evaluations without improvement.

Table 5.1 ADSSEC EA parameter settings

Population (μ)	**Offspring (λ)**	**Mutation Rate**	**Crossover Rate**
20	20	0.10	0.90
Termination Evaluations	**Restart Evaluations**	**Dec. Limit Multiplier**	**Sample Size**
1000	100	$3.0 \rightarrow 1.0$	15 (5 strikes)

ADSSEC was executed on a machine with dual Intel Xeon E5-2630 v3 2.4 GHz octa-core processors and 128 GB 2133 MHz DDR4 RDIMM ECC RAM running Ubuntu 14.04. Both the synchronous and asynchronous models were run 8 times on Datasets A, B, C, and D.

5.1.2 Results and Discussion. The aggregate user time across all processes was collected from each run of ADSSEC for both the synchronous model and asynchronous model on the datasets. The variance in the number of decisions needed to solve the instances in each dataset directly influences the variation in evaluation time for each heuristic. Increased variation in evaluation times allows for greater speed-ups in the asynchronous model over the synchronous approach. As illustrated in Figure 5.1, the growth in variance of MiniSat decisions in each dataset results in more evident speed-ups for the asynchronous evolution times. Additionally, there may be a proportional increase in variation of evolution time for both models.



(a) Dataset A (b) Dataset B (c) Dataset C (d) Dataset D

Figure 5.1 Boxplots of evolution time relative to the mean of the evolution time for the asynchronous runs on each respective dataset. The average relative asynchronous evolution time will always be at 1.0 for each plot. Using default MiniSat, Dataset A had an average of 7,441 decisions and standard deviation of 11,798 decisions. Dataset B had an average of 99,313 decisions and standard deviation of 118,761 decisions. Dataset C had an average of 297,925 decisions and standard deviation of 309,996 decisions. Dataset D had an average of 522,668 decisions and standard deviation of 565,955 decisions. The average and standard deviation of the decisions needed by MiniSat to solve the instances used in each dataset describe the difficulty and variation that can be expected to influence evaluation times.

The results from ANOVA tests confirm the improvement provided by the asynchronous method (see Table 5.2); as the p-values are all approximately zero, there is very high confidence in this conclusion. In Dataset D, the synchronous method needed an average of 5.24 hours to complete the same number of evaluations that asynchronous finished in an average of 3.19 hours. These results were obtained where the longest time to solve an instance is approximately five seconds. In the read-world, industrial instances can require several minutes to hours to complete. Employing the asynchronous evolution when training ADSSEC on datasets containing those instances would measure speed-ups in CPU days or weeks.

Table 5.2 ANOVA results of evolution time in seconds comparing both models of ADSSEC

	Synchronous	**Asynchronous**
	Dataset A	
Mean	515.1038	435.1900
Variance	35.4130	33.4400
P-value	0.0000	
	Dataset B	
Mean	1,647.6300	1,198.9300
Variance	11,554.0380	1,455.4835
P-value	0.0000	
	Dataset C	
Mean	15,453.0900	9,260.6925
Variance	493,279.9961	112,446.0049
P-value	0.0000	
	Dataset D	
Mean	18,865.0775	11,470.6363
Variance	3,539,070.1957	230,474.1322
P-value	0.0000	

5.2 ADSSEC VERSION 1.0 EXPERIMENT

This section investigates the quality of the heuristics produced by the initial single-objective version of the ADSSEC system as well as the effectiveness of its approach.

5.2.1 Experimental Setup. Ideally, one would want to construct entire solvers for a given problem, and ADSSEC demonstrates the obstacles and, more importantly, the potential of adapting a single component of a CDCL solver. The experiments with the prototype ADSSEC system require datasets that have:

- instances that ADSSEC can feasibly train on in a short period of time (each instance should require seconds to minutes for the original MiniSat to solve)
- enough instances to sufficiently represent a distinct instance class for both training and testing
- instances that are difficult enough that the instance can benefit from a fitted heuristic

Unfortunately, these requirements make many of the usual SAT datasets inappropriate for the initial prototype experiments. Many publicly available datasets contain too few instances to sufficiently represent the distinct problem class or the instances are so simple that nothing is gained by creating a fitted heuristic. Also, instances from previous SAT competitions attempt to challenge the capabilities of the solvers, so many require too significant an amount of time to solve. The time needed for a single evaluation is the product of two numbers: (1) the amount of time needed by a solver to find a solution to the average instance in the dataset and (2) the number of instances needed for a representative sample. Additionally, one must consider approximately how many evaluations are needed to discover an improved heuristic and how many physical machines are available to ADSSEC. Applying ADSSEC to hard problems becomes more feasible with smaller samples, fewer evaluations, and more/faster CPUs. The idea behind generating datasets for ADSSEC is that generators provide

enough control to meet these criteria while preventing bias by hand-selecting specific instances.

A k-colorable graph generation tool[†] – developed by Joseph Culberson, Adam Beacham, and Denis Papp – was used to create 66 satisfiable instances for the first dataset. These graphs each had 5000 vertices with an average degree of 4.31. A SAT conversion tool from the same source was employed to transform each graph into an instance. Each instance contained 52324 variables in 15000 clauses. All instances were solved with default MiniSat, and the ten instances with the highest number of decisions served as ADSSEC's training set. These ten instances encapsulated the shared structure of the class that the original heuristic could not exploit, focusing the evolved heuristic on the structure that gave the most room for improvement.

A modularity-based generator developed by Jesús Giráldez-Cru[‡] was used to create 40 satisfiable instances for the second dataset. These instances simulate an underlying structure that may be found in an industrial class of instances. Each instance contained 5000 variables in 19000 clauses. The generator allows the user to specify the structure of each instance. The tool generated 90 communities with a modularity of 0.8 and 3 literals per clause. The seeds 1 through 40 were used. Again, ADSSEC trained on MiniSat's worst ten from this dataset.

An identical configuration was employed for ADSSEC on both datasets (Table 5.3); manual tuning was used to discover this configuration. The computationally extensive search makes automated tuning of ADSSEC's parameters infeasible. Evolution of variable scoring heuristics is very time consuming. ADSSEC created an initial population of 30 random individuals. The master process used 63 slaves processes for evaluating offspring, asynchronously creating new offspring as each node became available. ADSSEC selected parents uniformly for either recombination or mutation – with a mutation probability of 0.10 and, subsequently, a recombination rate of

[†] `https://webdocs.cs.ualberta.ca/~joe/Coloring/Generators/generate.html`
[‡] `http://www.iiia.csic.es/~jgiraldez/`

0.90 – and used an asynchronous crowding method for survival selection. Although ADSSEC terminated after 5000 evaluations, if the best individual objective score had not improved in 250 evaluations, ADSSEC replaced the worst part of the population with randomly generated parse trees. ADSSEC evaluated each individual on the ten SAT instances in the training set (randomly ordered); each individual was limited to four strikes against the decision limit described previously.

Table 5.3 ADSSEC EA parameter settings

Population (μ)	**Offspring** (λ)	**Mutation Rate**	**Crossover Rate**
30	63	0.10	0.90
Termination Evaluations	**Restart Evaluations**	**Dec. Limit Multiplier**	**Sample Size**
5000	250	$3.0 \rightarrow 1.0$	10 (4 strikes)

ADSSEC was executed on several locally networked machines of varying loads all running Ubuntu. Solvers were then compiled with the best heuristics produced by those runs. In hopes of obtaining more accurate runtimes, Amazon EC2 m3.medium instances were used to collect the runtimes of the original MiniSat and the evolved solvers on both datasets. MiniSat was executed serially on all instances – including those trained on – from both the k-colorable graph and modularity datasets, but the evolved solvers were only executed on the datasets on which they were trained.

5.2.2 Results. Cactus plots are employed to compare the effectiveness of the solvers containing the evolved heuristics against the default MiniSat solver. To construct these plots, the metric of interest (e.g., decisions or runtime) is collected on all the instances of a given dataset for both solvers; the values are then sorted independent of the order of values for the other solver. While this representation

does not provide a direct instance-by-instance comparison, cactus plots illustrate the general effectiveness of the solvers over the entire dataset. Given that the solvers attempt to minimize the metrics in this experiment, the instances that each solver performed best on will be on the left of the plot and the worst performance at the right.

Figure 5.2 compares the number of decisions of the original MiniSat solver against that of the evolved solver for the k-colorable graph dataset. While the instances requiring fewer decisions are fairly close, there is a marked improvement with the new heuristic in the maximum number of decisions needed to solve the k-colorable graph dataset.

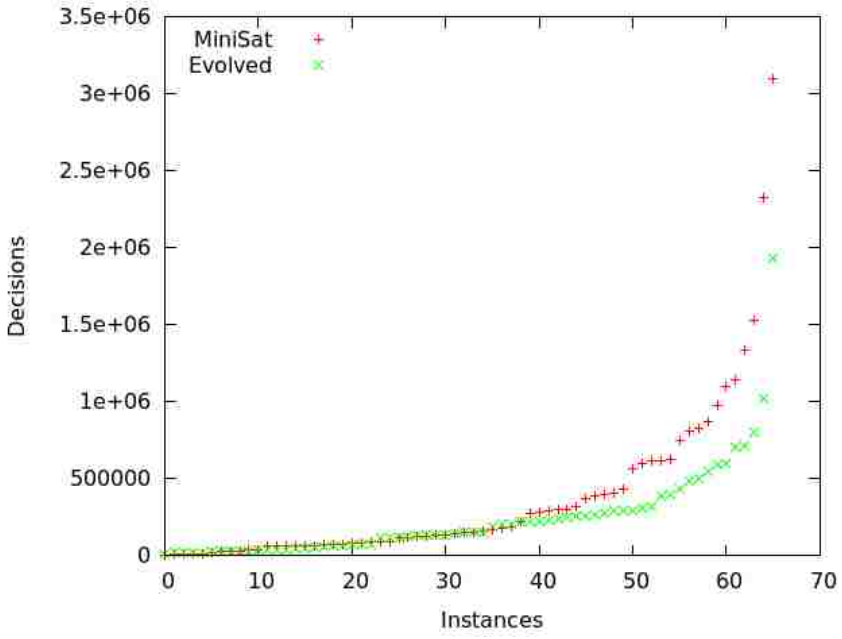


Figure 5.2 k-colorable graph cactus plot comparing number of decisions to solve

The new heuristic for the modularity dataset presented a much more significant difference (in some cases, an order of magnitude fewer decisions). The worst

instance for the evolved solver needed less than half the number of decisions MiniSat accrued at its worst (see Figure 5.3).

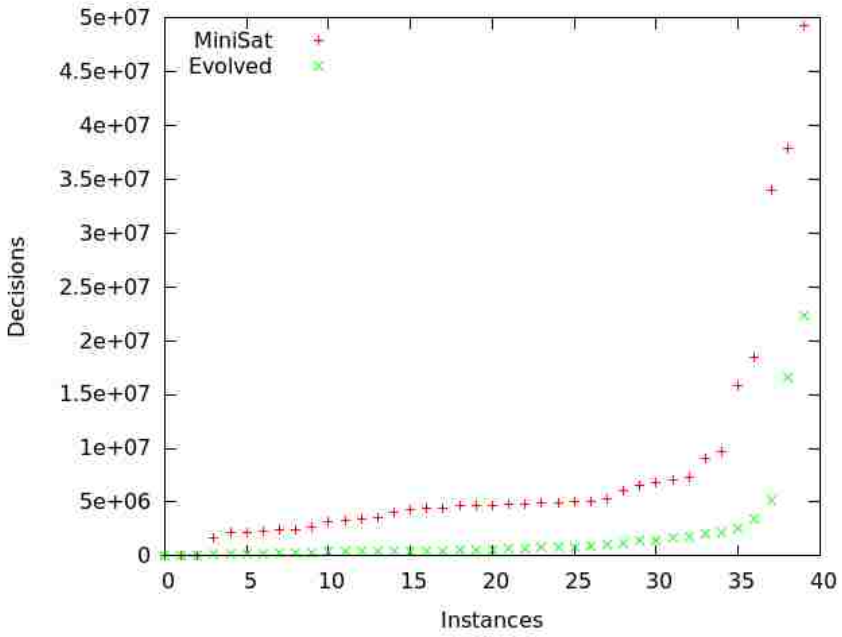


Figure 5.3 Modularity cactus plot comparing number of decisions to solve

Table 5.4 shows that the improvement for the k-colorable graph dataset seems to be reflected in CPU time as well as the number of decisions. Because both the original MiniSat and the evolved solver were executed on the same instances, the runtimes could be compared to determine whether the evolved solver is actually more efficient. With a p-value of 0.3562, the results cannot conclusively state that these values definitely reflect an improvement in terms of runtime.

Table 5.4 provides evidence of an improved performance with the evolved heuristic for the modularity dataset. The average and median CPU times were lower for the new heuristic, and the p-value is significant enough to state that the evolved heuristic is more efficient than the default MiniSat heuristic.

Table 5.4 Statistical Comparison on mean CPU time

	k-colorable Graph		Modularity	
	MiniSat	*Evolved*	*MiniSat*	*Evolved*
Mean	52.21	28.31	64.74	57.83
Variance	9144.35	2120.67	16845.47	25008.72
Median	14.7317	12.9167	24.9044	11.9451
Observations	66		40	
Number Evolved Improved	35		26	
P(X $\geq$ # Improved)	0.3562		0.0403	

As expected, the number of decisions seemed to be proportionate to the CPU time for the *k*-colorable graph dataset (Figure 5.4); however, this was not true for the modularity dataset (Figure 5.5). While the decisions were significantly lower, the evolved modularity heuristic performed similarly to the original heuristic and even had a higher maximum solve time.

To explore whether the two solvers for each dataset perform well (or poorly) on the same instances, the lower runtime for each instance was kept; these times are labeled "*Minimum*" in Figure 5.4 and Figure 5.5. Interestingly, the evolved solvers seem to complement the performance of the original MiniSat. For the modularity dataset, while the worst runtimes for MiniSat and Evolved were 690.62 seconds and 860.702 seconds respectively, the worst minimum runtime was less than 44 seconds. The *k*-colorable graph dataset showed a similar complementary improvement: 592.816 seconds, 317.763 seconds, and a minimum of 96.9906 seconds. As portfolios of two, each pair provides very significant speed-ups for their respective datasets.

As seen here, ADSSEC cannot depend solely on the number of decisions in evaluating an instance. To discover what other metric(s) that should be used, the final state metrics were compared between the worst modularity instances for the original MiniSat and the evolved solver (Table 5.5). Evolved has fewer restarts, conflicts, decisions, and propagations, but as different instances are being compared,

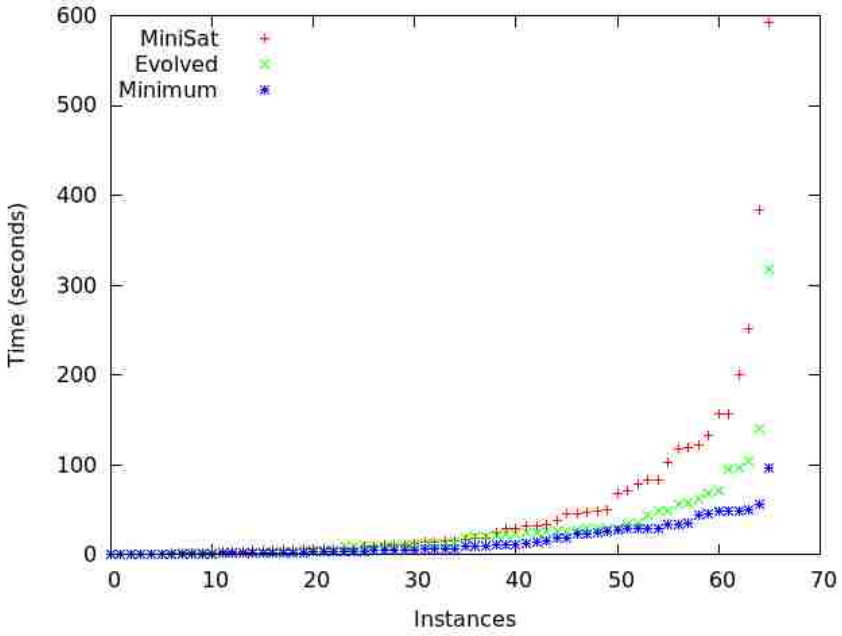


Figure 5.4 k-colorable graph cactus plot comparing CPU time needed to solve

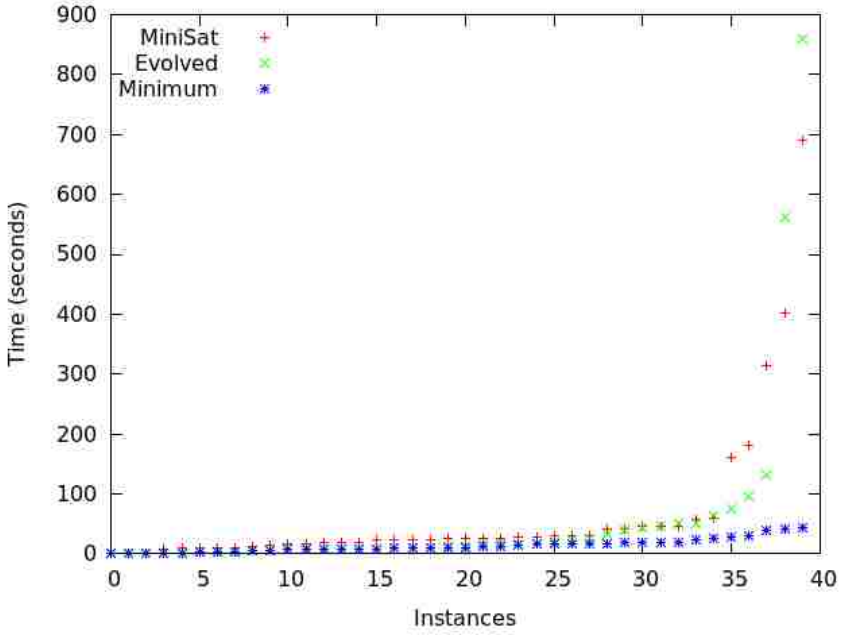


Figure 5.5 Modularity cactus plot comparing CPU time needed to solve

Table 5.5 Metrics of worst modularity instances for MiniSat and the evolved variant

	MiniSat's Worst	Evolved's Worst
restarts	11,771	8,190
conflicts	7,163,905	4,729,239
decisions	49,310,829	22,325,993
propagations	1,835,833,635	1,373,888,853
conflict literals	274,729,206	854,679,218
Memory used (MB)	115.00	183.00
CPU time (sec)	690.62	860.70

meaningful conclusions cannot be drawn from this comparison. However, the evolved solver used more conflict literals, significantly more memory, and more CPU time. In the evolved solver's worst instance, each conflict resulted in more conflict literals (and less of the search space excised) than in MiniSat's worst case. Since the number of conflict literals can be seen to affect runtime, this suggests that in subsequent work the objective function should take the number of conflict literals into consideration as well. For example, the components related to the management of conflict clauses could be adapted alongside the variable selection heuristic.

5.2.3 Discussion. Figure 5.6 shows the heuristic evolved for the k-colorable graph dataset. Arithmetic simplification in a standard optimizing compiler should easily reduce some of the branches in this particular tree. However, a pruning function that simplifies complicated heuristics before termination removes valuable genetic structures from further evolution. If a pruning function were added in ADSSEC, the final heuristic would only be pruned after termination.

Figure 5.7 shows the heuristic evolved for the modularity dataset. This heuristic was much more complex and employed every type of node available. However, the best heuristic discovered may not be the most efficient heuristic discovered in runtime; some components in this heuristic may be less useful or influential. In future

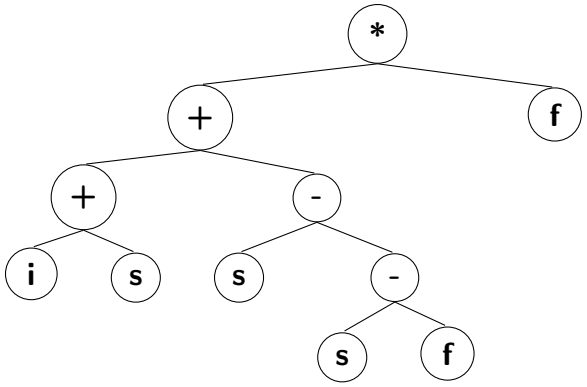


Figure 5.6 Evolved heuristic for k-colorable graph instances

work, looking back through the ancestors of the individual could assist in discovering which components or combinations are most influential and which components hinder performance and should be pruned. Additionally, ADSSEC might find even better heuristics using new node types that preserve pre-existing node structure or new nodes that represent other solver state-based information.

5.3 ADSSEC VERSION 2.0 EXPERIMENT

This section investigates the quality of the heuristics produced by the expanded multi-objective version of the ADSSEC system as well as the effectiveness of its approach.

5.3.1 Experimental Setup. The same two datasets used in the ADSSEC Version 1.0 Experiment were employed in analyzing the performance of the second version. With the more generous decision limits placed on executing the easier instances in a given dataset, the training set was expanded to include all instances rather than the ten instances that MiniSat performed worst on. Additionally, ADSSEC was evolved on the variable scoring heuristic and learnt clause scoring heuristic separately. Evolving both heuristics simultaneously introduces additional complexity. The quality of the heuristics would have to be measured as a pair; as such, if one of the

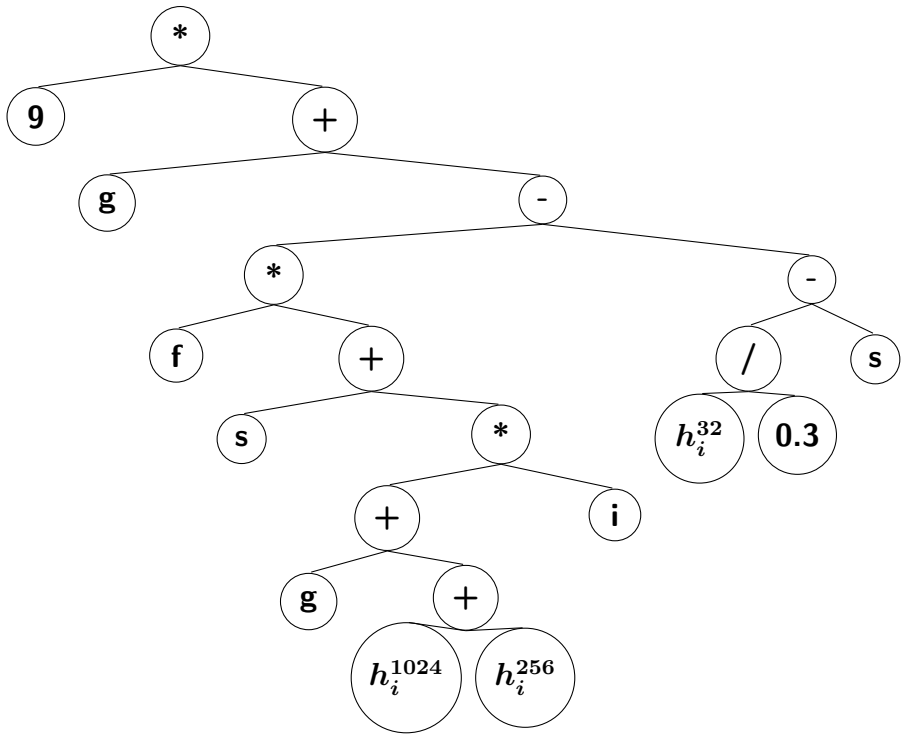


Figure 5.7 Evolved heuristic for modularity instances

heuristics performs poorly, the other will have receive a low fitness. Until ADSSEC can be modified to handle that, the heuristics will be evolved and evaluated independently.

An identical configuration was employed for ADSSEC on both datasets (Table 5.6); manual tuning was used to discover this configuration. ADSSEC created an initial population of 20 random individuals. The master process used 63 slave processes for evaluating offspring, asynchronously creating new offspring as each node became available. ADSSEC selected parents with a tournament size of 2 for either recombination or mutation – with a mutation probability of 0.10 and, subsequently, a recombination rate of 0.90 – and used an asynchronous crowding method for survival selection. Although ADSSEC terminated after 2000 evaluations, if the best individual objective score had not improved in 100 evaluations, ADSSEC replaced the worst

part of the population with randomly generated parse trees. ADSSEC evaluated each individual on a sample of 12 SAT instances in the training set; each individual was limited to four strikes against the decision limit described previously.

Table 5.6 ADSSEC EA parameter settings

Population (μ)	**Offspring** (λ)	**Mutation Rate**	**Crossover Rate**
20	63	0.10	0.90
Termination Evaluations	**Restart Evaluations**	**Dec. Limit Multiplier**	**Sample Size**
2000	100	$3.0 \rightarrow 1.0$	12 (4 strikes)

In multi-objective EAs, the final population typically consists of sets, or fronts, of individuals that do not outperform each other on all – in this case, both – of the objectives. The front that contains individuals that outperform all other individuals except those in the same set is called the non-dominated front. For ADSSEC, the non-dominated front encapsulates the best heuristics found for a given run. In this experiment, the non-dominated fronts only contained a single best heuristic at the end of each run.

ADSSEC was executed on two locally networked machines of varying loads both running Ubuntu. The machines have dual Intel Xeon E5-2630 v3 2.4 GHz octa-core processors and 128 GB 2133 MHz DDR4 RDIMM ECC RAM. Solvers were then compiled with the best heuristics produced by those runs. MiniSat was executed serially on all instances – including those trained on – from both the k-colorable graph and modularity datasets, but the evolved solvers were only executed on the datasets on which they were trained.

5.3.2 Results and Discussion. Figure 5.8 and Figure 5.9 depict the objective scores – numbers of decisions and conflict literals needed to solve each instance – for the modularity dataset when employing MiniSat and the evolved variants; one variant included the evolved variable scoring heuristic and the other contained the evolved learnt clause scoring heuristic. While ADSSEC aimed to reduce these objective values, the plots suggest that very little improvement, if any, was provided by the evolved learnt clause scoring heuristic. This may be the effect of a number of possible factors. For example, unrepresentative sampling with primarily improved instances may bias toward less effective heuristics. Early termination of evolution may also play a part in not finding an effective heuristic. Additionally, the mechanisms and primitives available to ADSSEC shape the search space of the generated learnt clause scoring heuristics. Just as the developers of CDCL SAT solvers take careful consideration in their designs, ADSSEC must automate the process of intertwining the intricate implementations of CDCL heuristics.

Interestingly, the evolved variable scoring heuristic for the modularity dataset provided a substantial improvement over the default MiniSat heuristic on both of the objectives as well as the runtime, as can be seen in Figure 5.10. This evolved solver was also compared against two top solvers, Lingeling_sr15bal [45] and Glucose-Syrup [46], as shown in Figure 5.11. Given that these solvers were executed on the same instances, a Wilcoxon Signed-Rank Test was employed to evaluate the performance gain of the evolved modularity variable scoring heuristics. The small p-values in Table 5.7 confirm that the evolved solver with the novel variable scoring heuristic is more efficient than MiniSat as well as several top CDCL SAT solvers.

The results for the evolved k-colorable graph heuristics provide additional insight into the relation between the objective values and the actual runtime of the CDCL SAT solvers. Consider the objective values of the worst instances for both evolved variants and default MiniSat in Figure 5.12 and Figure 5.13. While the evolved learnt clause scoring heuristics appear to have the lowest objective scores,

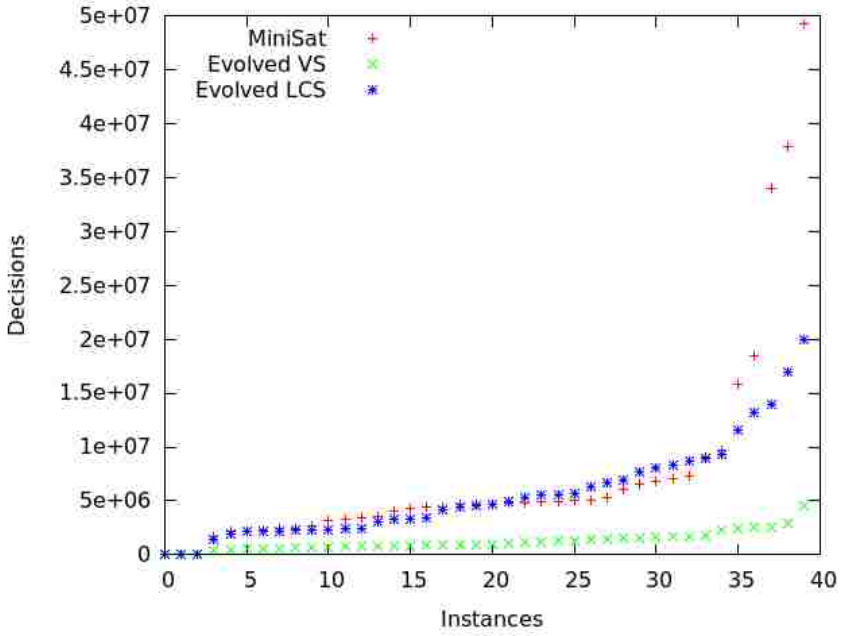


Figure 5.8 Modularity cactus plot comparing number of decisions to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

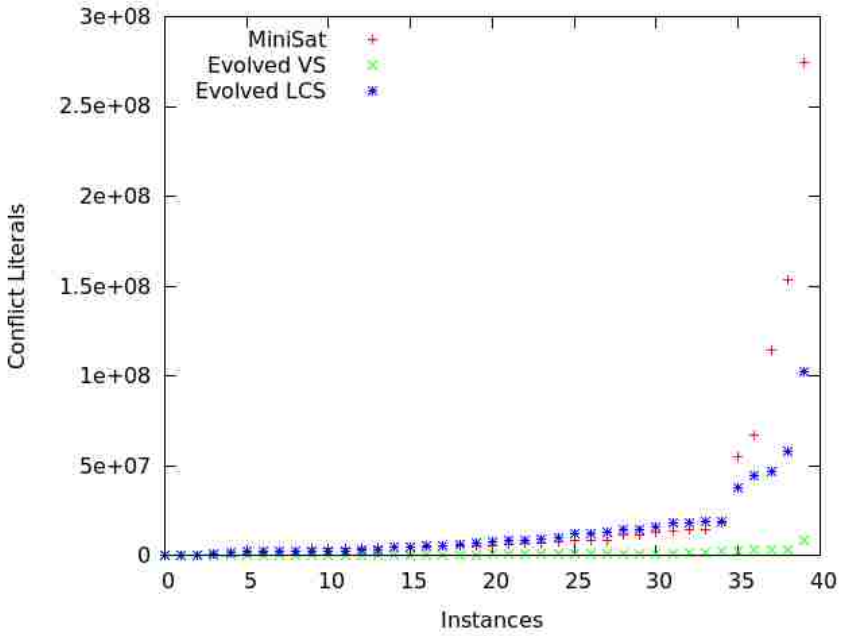


Figure 5.9 Modularity cactus plot comparing number of conflict literals to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

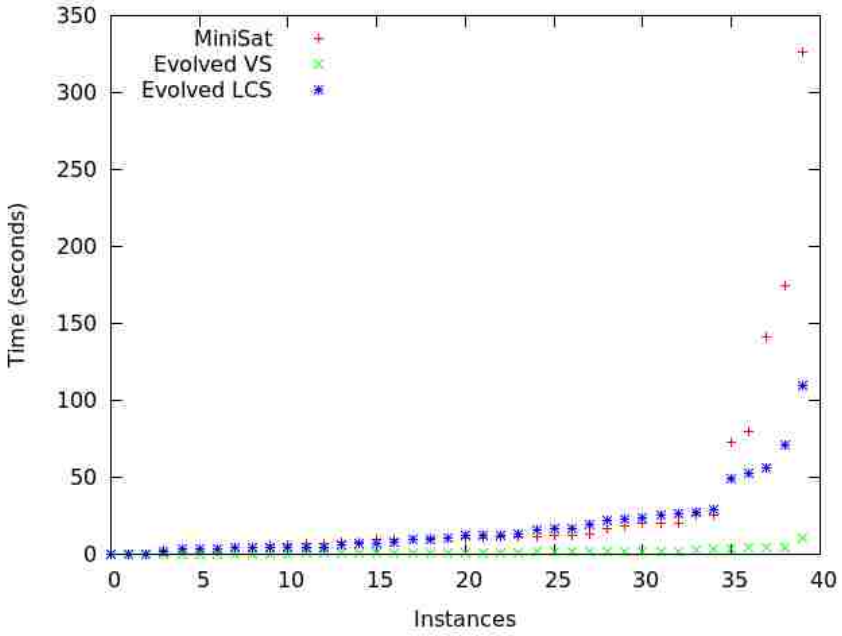


Figure 5.10 Modularity cactus plot comparing runtime to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

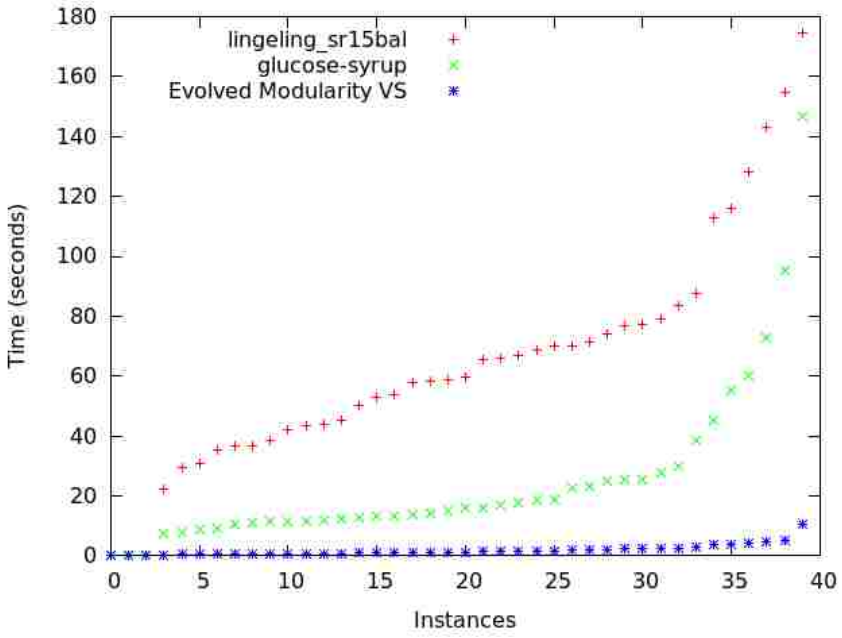


Figure 5.11 Modularity cactus plot comparing runtime to solve with top solvers (VS: Variable Scoring)

Table 5.7 Wilcoxon Signed-Ranks Test for Paired Samples comparing CPU time of evolved modularity variable scoring heuristic against other solvers

	Evolved VS	**MiniSat**	**Lingeling_sr15bal**	**Glucose-Syrup**
Mean	1.8401	29.0480	64.5425	24.8163
Variance	3.6014	3519.8687	1478.0364	755.0245
Median	1.1610	10.8205	59.2000	15.4022
P-Value		0.0000	0.0000	0.0000

the associated runtime in Figure 5.14 is clearly worse. This suggests that the solving time may not necessarily be bound strictly by decisions and conflict literals, and that further investigation is needed to determine the solver attributes that have the most significant influence on runtime.

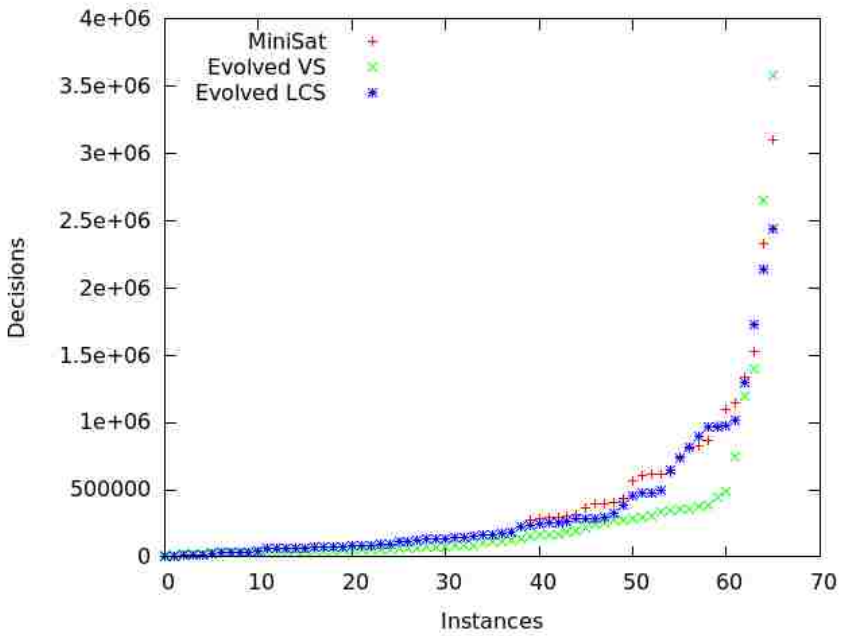


Figure 5.12 k-colorable graph cactus plot comparing number of decisions to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

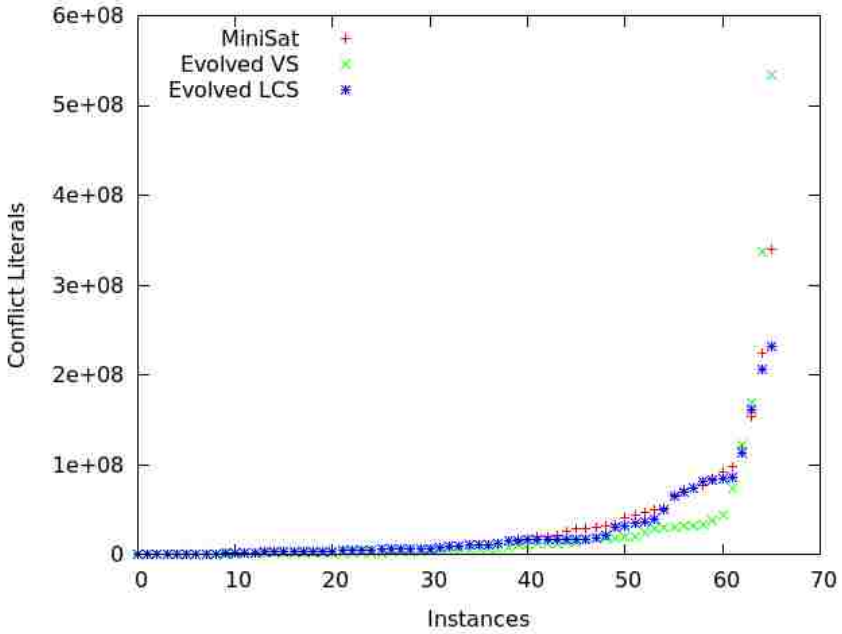


Figure 5.13 k-colorable graph cactus plot comparing number of conflict literals to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

ADSSEC is attempting to specialize the solvers to the given datasets; therefore, an evolved heuristic for one application is not guaranteed to be more efficient for another problem class. While the evolved modularity variable scoring heuristic was more efficient for the modularity instances, this was not found to be the case when executed on all instances from the k-colorable graph dataset. Figure 5.15 shows that the performance of the modularity variable scoring heuristic is comparable to that evolved for the k-colorable dataset and the heuristic in default MiniSat.

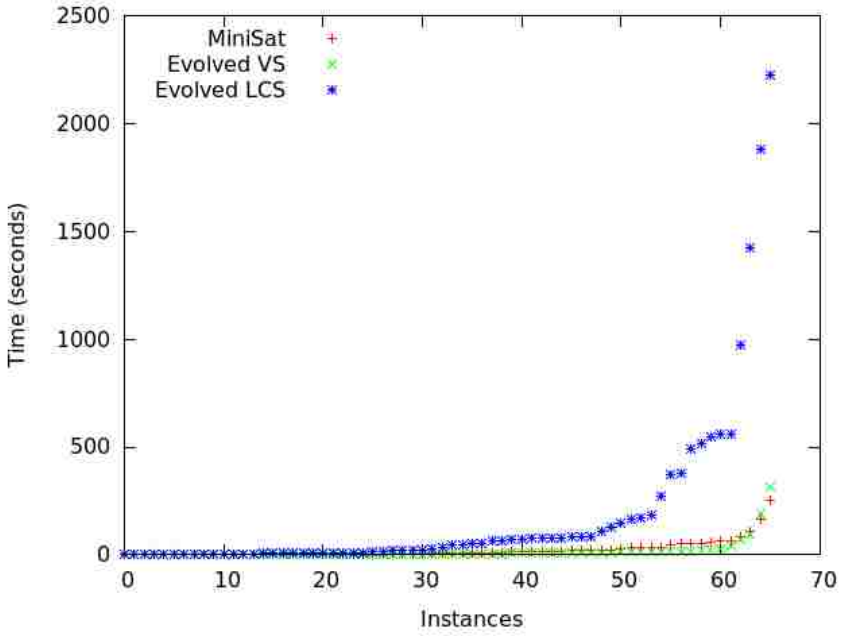


Figure 5.14 k-colorable graph cactus plot comparing runtime to solve (VS: Variable Scoring, LCS: Learnt Clause Scoring)

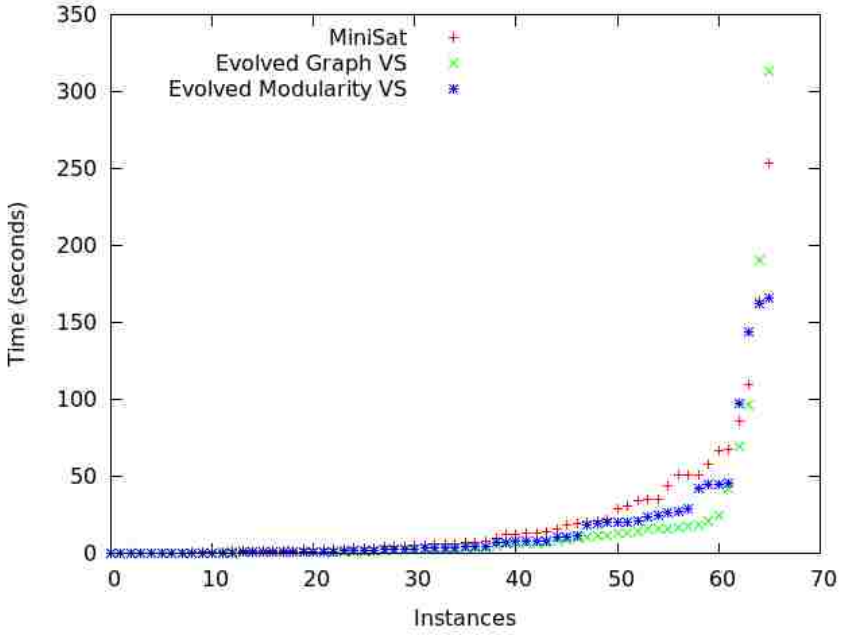


Figure 5.15 k-colorable graph cactus plot comparing runtime to solve with evolved variable scoring (VS) heuristics of both datasets

6 CONCLUSION

For EAs where the fitness evaluation times can vary drastically, especially in the case of hyper-heuristics, just parallelizing the evaluations to minimize evolution time is not always sufficient. The asynchronous approach to modelling hyper-heuristics will certainly provide a significant speed-up in evolution time over the synchronous alternative. As the variation in evaluation time increases, so does the speed-up. Instances from the SAT competitions and within industrial problem classes often have large variations in solving times. The asynchronous approach is necessary for addressing training systems similar to ADSSEC on the industrial instances. This paper has provided empirical evidence of the substantial performance gains of the asynchronous approach demonstrating that APEAs are the future of hyper-heuristics.

Even with the sole objective of reducing the number of decisions, ADSSEC Version 1.0 is capable of evolving variable scoring heuristics that are able to outperform the default MiniSat on a specific problem class. Even better, the evolved heuristics seem to complement the performance of the original MiniSat – even when the evolved solver is not strictly better than the default MiniSat. A portfolio approach using the two solvers provides great speed-ups over using a single solver.

ADSSEC Version 2.0 supported the employment of a multi-objective approach by producing a variable scoring heuristic that significantly outperformed MiniSat as well as top CDCL SAT solvers. Interestingly, lower number of decisions and conflict literals do not always guarantee faster performance. Other metrics may need to be considered when automatically modifying components of an existing solver to target specific problem classes. The learnt clause scoring heuristics and k-colorable graph variable scoring heuristic had much smaller impacts on the potential effectiveness of ADSSEC. This may result from inaccurate objective scores if the samples are not representative of the problem class, or possibly from early termination of evolution.

Also, in the case that these heuristics are constrained by the available genetic material, additional CDCL variables and mechanisms may need to be made available to ADSSEC. Another limiting factor may be that ADSSEC is only evolving one heuristic of the solver during evolution. ADSSEC is still in an early stage of development, but this proof-of-concept framework has shown sufficient evidence that the automated design of SAT solvers can target problem classes of instances.

7 FUTURE WORK

ADSSEC offers many possible areas of exploration:

- Measuring the rate of convergence between the models may conclude whether asynchronous or synchronous approaches produce superior heuristics in less time. While asynchronous evolution completed the same number of evaluations in less time, more evidence is needed to conclude that the quality of heuristics produced will match the synchronous counterpart at any set number of evaluations.
- Use either different datasets or larger sample sizes to determine how the generated heuristic quality varies with the training instances.
- While hyper-heuristics can be computationally expensive, providing some level of automated parameter tuning may further reduce the evolution time. This is dependent on the sensitivity of the parameters or inputs selected for automation.
- Developing a more accurate objective function using other metrics should provide better evaluations and ultimately better variants. This may require running an executable profiler on MiniSat and the evolved solvers to discover the most influential metrics.
- Allowing different data structures to store the variable scores may speed up the execution time of ADSSEC [37].
- Adjusting bounds on maximum variable / learnt clause scores or preventing exponential growth of scores would reduce the number of times the scores need to be scaled down.
- Tuning MiniSat's external parameters, either in the final evolved solver or during evolution, could result in more effective CDCL solver components.

- Harris et al demonstrated the significance of choice of GP type for the performance of the algorithms evolved by a GP powered hyper-heuristic [47]; exploring heuristic representation through alternative GP types may improve ADSSEC's performance.
- Training ADSSEC on an appropriately selected cross-section of a dataset might result in a single solver that performs reasonably on the entire dataset. Although the solvers were targeted to the worst or a random sample of instances in a dataset, it would be interesting to explore potential selection schemes to target evolving a single solver.
- Re-purposing ADSSEC to develop full portfolios of complementing MiniSat variants could result in extremely effective portfolios in which each solver targets a subset of the instance class.
- Exploring the runtime trade-off between the complexity of a heuristic function and the effectiveness of that heuristic could provide an interesting guideline for limiting the heuristics introduced into the population.
- Combining learnt clause reduction mechanisms with learnt clause scoring heuristics may allow for the design of more flexible clause management components.
- Evolving the variable scoring heuristic alongside the restart schemes or conflict clause management components could create an effective targeted solver. For example, a cooperative EA might have one population handle variable scoring heuristics while the other handles learnt clause scoring heuristics, and the populations would periodically share the best heuristics found so far. Interesting work here might include tinkering with the multi-objective objective scores.
- Currently, ADSSEC assumes that the targeted solver will take a serial approach to solving each problem. However, parallel CDCL solvers may also be targeted to problem classes by evolving unique heuristics for each process in the solver and allowing the processes to share learned information.

- Providing a sensitivity analysis on the arrangement and order of variables and clauses in instances from a dataset may better define the bounds of a class of instances.

BIBLIOGRAPHY

[1] Anton Belov, Daniel Diepold, Marijn J H Heule, and Matti Järvisalo. Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions. Technical report, University of Helsinki, Helsinki, 2014.

[2] Emmanuel Zarpas. Benchmarking SAT Solvers for Bounded Model Checking. In *Theory and Applications of Satisfiability Testing–SAT 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 340–354, St Andrews, UK, 2005. Springer Berlin Heidelberg.

[3] Emmanuel Zarpas. Back to the SAT05 Competition: an a Posteriori Analysis of Solver Performance on Industrial Benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:229–236, January 2006.

[4] Allen Van Gelder. Another Look at Graph Coloring via Propositional Satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, January 2008.

[5] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[6] Armin Biere and Andreas Fröhlich. Evaluating CDCL Restart Schemes. In *Proceedings of the International Workshop on Pragmatics of SAT (POS'15)*, Austin, TX, September 2015. 16 pages.

[7] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing–SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, May 2003. Springer Berlin Heidelberg.

[8] Stefan Bleuler, Martin Brack, Lothar Thiele, and Eckart Zitzler. Multiobjective Genetic Programming: Reducing Bloat Using SPEA2. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 536–543. IEEE, 2001.

[9] Julian Miller. What bloat? Cartesian Genetic Programming on Boolean problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.

[10] Matthew A Martin and Daniel R Tauritz. Multi-Sample Evolution of Robust Black-Box Search Algorithms. In *Proceedings of the 16th Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*, pages 195–196. ACM, 2014.

[11] Matthew A Martin and Daniel R Tauritz. A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic. In *Proceedings of the 16th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '14)*, pages 1389–1396, Vancouver, BC, Canada, July 2014. ACM.

[12] Enrique Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[13] Enrique Alba and Marco Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[14] Enrique Alba and José M Troya. Analyzing Synchronous and Asynchronous Parallel Distributed Genetic Algorithms. *Future Generation Computer Systems*, 17(4):451–465, 2001.

[15] Mouloud Oussaidene, Bastien Chopard, Olivier V Pictet, and Marco Tomassini. Parallel Genetic Programming and its application to trading model induction. *Parallel Computing*, 23(8):1183–1198, 1997.

[16] Juan José Durillo, Antonio J Nebro, Francisco Luna, and Enrique Alba. A Study of Master-Slave Approaches to Parallelize NSGA-II. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[17] Alexander W Churchill, Phil Husbands, and Andrew Philippides. Tool Sequence Optimization using Synchronous and Asynchronous Parallel Multi-Objective Evolutionary Algorithms with Heterogeneous Evaluations. In *2013 IEEE Congress on Evolutionary Computation (CEC)*, pages 2924–2931. IEEE, 2013.

[18] Mouadh Yagoubi and Marc Schoenauer. Asynchronous Master/Slave MOEAs and Heterogeneous Evaluation Costs. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, pages 1007–1014. ACM, 2012.

[19] Mouadh Yagoubi, Ludovic Thobois, and Marc Schoenauer. Asynchronous Evolutionary Multi-Objective Algorithms with Heterogeneous Evaluation Costs. In *2011 IEEE Congress on Evolutionary Computation (CEC)*, pages 21–28. IEEE, 2011.

[20] Eric O Scott and Kenneth A De Jong. Evaluation-Time Bias in Asynchronous Evolutionary Algorithms. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1209–1212, Madrid, Spain, July 2015. ACM.

[21] Matthew A Martin, Alex R Bertels, and Daniel R Tauritz. Asynchronous Parallel Evolutionary Algorithms: Leveraging Heterogeneous Fitness Evaluation Times for Scalability and Elitist Parsimony Pressure. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1429–1430, Madrid, Spain, July 2015. ACM.

[22] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, December 2013.

[23] Matthew A Martin and Daniel R Tauritz. Hyper-Heuristics: A Study On Increasing Primitive-Space. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1051–1058, Madrid, Spain, July 2015. ACM.

[24] Justyna Petke, William B Langdon, and Mark Harman. Applying Genetic Improvement to MiniSAT. In *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262. Springer Berlin Heidelberg, St. Petersburg, Russia, August 2013.

[25] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming*, volume 8599 of *Lecture Notes in Computer Science*, pages 137–149. Springer Berlin Heidelberg, Granada, Spain, April 2014.

[26] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, September 2009.

[27] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg, Rome, Italy, January 2011.

[28] Stefan Falkner, Marius Lindauer, and Frank Hutter. SpySMAC: Automated Configuration and Performance Analysis of SAT Solvers. In *Theory and Applications of Satisfiability Testing–SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 215–222. Springer International Publishing, Austin, TX, USA, September 2015.

[29] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-Based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, May 2008.

[30] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla2009: An Automatic Algorithm Portfolio for SAT. In *SAT 2009 Competitive Events Booklet*, pages 53–55, September 2009.

[31] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm Runtime Prediction: Methods & Evaluation. *Artificial Intelligence*, 206:79–111, January 2014.

[32] Mohamed Bader-El-Den and Riccardo Poli. Generating SAT Local-Search Heuristics Using a GP Hyper-Heuristic Framework. In *Artificial Evolution*, volume 4926 of *Lecture Notes in Computer Science*, pages 37–49, Tours, France, October 2008. Springer Berlin Heidelberg.

[33] Raihan H Kibria and You Li. Optimizing the Initialization of Dynamic Decision Heuristics in DPLL SAT Solvers Using Genetic Programming. In *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin Heidelberg, Budapest, Hungary, April 2006.

[34] Alex S Fukunaga. Evolving Local Search Heuristics for SAT Using Genetic Programming. In *Genetic and Evolutionary Computation Conference–GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA, June 2004. Springer Berlin Heidelberg.

[35] Alex S Fukunaga. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evolutionary Computation*, 16(1):31–61, April 2008.

[36] Alex S Fukunaga. Massively Parallel Evolution of SAT Heuristics. In *2009 IEEE Congress on Evolutionary Computation (CEC)*, pages 1478–1485, Trondheim, Norway, May 2009. IEEE.

[37] Armin Biere and Andreas Fröhlich. Evaluating CDCL Variable Scoring Schemes. In *Theory and Applications of Satisfiability Testing–SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer International Publishing, Austin, TX, USA, September 2015.

[38] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2):105–127, September 2004.

[39] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 570–574, Berlin, Heidelberg, April 2005. Springer Berlin Heidelberg.

[40] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, March 2008.

[41] Agoston E Eiben and James E Smith. *Introduction to Evolutionary Computing*. Springer, second edition, 2015. page 93.

[42] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solver. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 399–404, Pasadena, CA, USA, July 2009.

[43] Gilles Audemard and Laurent Simon. Refining Restarts Strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126, Québec City, QC, Canada, October 2012. Springer Berlin Heidelberg.

[44] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[45] Armin Biere. Lingeling and Friends Entering the SAT Race 2015. In *SAT Race 2015*, Austin, TX, USA, September 2015. 2 pages.

[46] Gilles Audemard and Laurent Simon. Glucose and Syrup in the SAT Race 2015. In *SAT Race 2015*, Austin, TX, USA, September 2015. 2 pages.

[47] Sean Harris, Travis Bueter, and Daniel R Tauritz. A Comparison of Genetic Programming Variants for Hyper-Heuristics. In *Proceedings of the 17th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '15)*, pages 1043–1050, Madrid, Spain, July 2015. ACM.

VITA

Alex Raymond Bertels was raised in Dorsey, Illinois. He received his diploma from Edwardsville High School in Edwardsville, Illinois in Spring of 2010 and enrolled in Missouri University of Science and Technology in Fall of 2010, earning a Bachelor of Science degree in Computer Science in Spring of 2014. During the summer semesters from 2012 to 2014, he worked as a technical intern in the Center for Cyber Defenders at Sandia National Laboratories in Albuquerque, New Mexico. He then transitioned into the Critical Skills Master's Program at Sandia National Laboratories from which he received his funding for his graduate studies. He earned his Master of Science degree in Computer Science in July 2016.