SAND2017-7363C

Improving Performance of CDCL SAT Solvers by Automated Design of Variable Selection Heuristics

Marketa Illetskova*, Alex R. Bertels†, Joshua M. Tuggle*, Adam Harter*, Samuel Richter*,
Daniel R. Tauritz*, Samuel Mulder†, Denis Bueno†, Michelle Leger†, and William M. Siever‡
*Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, U.S.A.
Email: miwdf@mst.edu, dtauritz@acm.org
†Sandia National Laboratories, Albuquerque, NM, U.S.A., Email: abertel@sandia.gov
‡Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, U.S.A.

***Abstract*—Many real-world engineering and science problems can be mapped to Boolean satisfiability problems (SAT). It has been shown that improvement in a SAT solver's performance correlates with improvement in its energy efficiency. CDCL SAT solvers belong among the most efficient solvers. Previous work showed that instances derived from a particular problem class exhibit a unique underlying structure which impacts the effectiveness of a solver's variable selection scheme. Thus, customizing the variable scoring heuristic of a solver to a particular problem class can significantly enhance the solver's performance; however, manually performing such customization is very labor intensive. This paper presents a system for automating the design of variable scoring heuristics for CDCL solvers, making it feasible to tailor solvers to arbitrary problem classes. Experimental results are provided demonstrating that this system, which evolves variable scoring heuristics using an asynchronous parallel hyper-heuristics approach employing genetic programming, has the potential to create more efficient solvers for particular problem classes.**

I. Introduction

The boolean satisfiability problem (SAT) is one of many problems in computer science that belongs to the NP complete complexity class, and therefore it has no efficient solution unless P = NP. Various real-world engineering and science problems can be modeled as instances of SAT, and searching for solutions to such real-world problems requires extensive utilization of computational resources.

Over the past few decades, SAT solver efficiency has improved dramatically, leading to even more utilization of SAT in engineering and science solutions [28]. For example, SAT has been used to minimize power consumption in an energy management system while satisfying varying requirements of its users [12]. There are many other applications of SAT in various fields such as computational biology [11], [34], software testing [25], hardware and software model checking and verification [33], [37], Internet of Things [27], and robotics [22], [23].

Encoding specific problem classes in SAT creates classes of structured logical expressions called SAT instances. Empirical evidence shows that each solver has an ideal underlying instance structure and that each class of instances has an optimal solver [40], [41], [36]. Recent work has automatically optimized SAT solver parameter configuration to target specific classes of instances [20], [19], [15]. KhudaBukhsh et al. automatically composed stochastic local search (SLS) solvers from parts of existing SLS solvers [24], and other work automatically evolved variable selection techniques for SLS solvers [2], [26], [16], [17], [18]. However, conflict-driven clause learning (CDCL) solvers are still the most efficient SAT solvers for industrial instances. Based on Biere and Fröhlich's demonstration of the drastic impact that restart and variable selection schemes have on CDCL solver efficiency for specific problem classes [8], [7], Bertels introduced the idea of automatic evolution of CDCL operations as a case study in [5] and as a proof-of-concept system in [4]. His work showed promising results in increasing the effectiveness of a CDCL SAT solver by targeting classes of instances with unique structure. We know of no other work that automatically evolves CDCL heuristics.

In this paper we describe the system, **ADSSEC** (**A**utomated **D**esign of **S**AT **S**olvers employing **E**volutionary **C**omputation), evaluate its performance on datasets and benchmarks from random and industrial tracks, and present results demonstrating its potential for producing more effective solvers targeted to arbitrary, but particular, problem classes (datasets).

II. Related Work

Ideally, a single solver would be able to adapt to an application at runtime. Tools such as ParamILS [20], SMAC [19], and SpySMAC [15] automatically tailor the parameter configurations of reasonably versatile solvers to particular datasets. While the solver parameter configurations are adjustable, most of the internal methods of the solvers remain the same. The effectiveness of adapting a solver to a problem class solely through external parameter optimization is limited by the appropriateness of the solver's architecture, such as its variable scoring heuristic, 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 [38], [39]. 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 [21]. 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.

ADSSEC takes the next step by applying generative hyper-heuristics [10] to generate CDCL SAT solvers tailored for an arbitrary, but particular, problem class. *Hyper-heuristics* approaches automatically develop heuristics or algorithms; generative hyper-heuristics combine primitive heuristics, exploring a broader search space, while selective hyper-heuristics choose the best option(s) from a set of pre-defined heuristics [10]. 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 paper, only the variable scoring heuristic is being modified, so ADSSEC uses the generative approach. ADSSEC employs genetic programming (GP) to automatically reorganize and manipulate the algorithmic primitives constituting the variable scoring heuristic [29], [30]. 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 employs CDCL state-based information and binary operations to automatically develop new variable scoring heuristics tailored to problem classes. ADSSEC's primitives are more granular than statements in the source code and are therefore much more versatile in developing new solver components.

We describe the ADSSEC hyper-heuristics framework in detail in the next section.

III. ADSSEC

Genetic programing (GP) utilizes a biologically inspired approach in searching for individuals adapted to a given environment by preserving genes important for success. An individual in GP may be optimized for a given input to produce desirable output in a given environment. Finding an optimal SAT solver for a given problem class is a complex and expensive task, suitable for automation using a GP-based hyper-heuristic system. Influenced by the success of Fukunaga in improving SLS runtimes by evolving specific heuristics [16], ADSSEC uses GP to evolve variable scoring heuristics to automatically tailor CDCL solvers to specific classes of structured instances.

In particular, ADSSEC utilizes Koza-style GP trees [32], which are well suited to the parse trees representing variable scoring heuristics used by CDCL solvers to select decision variables. Biere and Fröhlich's work demonstrating that the current best variable scoring heuristics are roughly equal in runtime performance when evaluated across many instance classes [8] motivated the choice to adapt a CDCL solver's variable scoring heuristics [4].

TABLE I: Terminal nodes in ADSSEC

Score ($s[v]$)	The previous score of the variable.
Conflict Index (i)	The current number of conflicts encountered.
MiniSat Variable Decay Amount (f)	f is also used to derive MiniSat's Variable Increment Value (*MiniSat Default: 0.95*).
MiniSat's Variable Increment Amount (g)	($g = (1/f)^i$) The amount MiniSat increases a score when a variable is bumped.
Constant (C)	A constant value in $\{1, 2, 3, \ldots, 10\}$ or $\{0.0, 0.1, 0.2, \ldots, 0.9\}$.
Special Component (H)	Derived from the Chaff CDCL SAT solver for scaling variable scores [8]. $h_i^m = \begin{cases} 0.5 \cdot s & \text{if } m \text{ divides } i \text{ evenly} \\ s & \text{otherwise} \end{cases}$ where m is a power of 2: $\{2, 4, 8, \ldots, 1024\}$.
Variables in the First UIP ($U[v]$)	The number of times a variable occurred in a first Unique Implication Point (UIP) conflict.
Eliminated Variables in Simplification Step ($E[v]$)	The number of times a variable is eliminated in the simplification of conflict clauses.

Evolutionary algorithms and SAT solving are independently computationally heavy tasks, and combining the two magnifies the effect. Bertels et al. showed using ADSSEC that an asynchronous parallel evolutionary algorithm (APEA) approach leads to a significant speed up in hyper-heuristic systems [5], [4]. This motivated us to use an asynchronous parallel implementation of ADSSEC.

ADSSEC creates an initial population of variable scoring heuristics and evolves the population through mutation and recombination, both explained in detail later. ADSSEC evaluates these heuristics by replacing the variable selection heuristic in MiniSat 2.2 [13], a commonly used efficient and deterministic CDCL solver with dense source code, or in Glucose 4.0 [1], one of the current state-of-the-art solvers based on MiniSat. ADSSEC employs a standard parent selection before producing offspring and a crowding survival selection specifically modified for the APEA [4]. ADSSEC returns the best heuristics from the final population after reaching the termination criteria.

Derived from currently implemented variable scoring heuristics [8], [13], [35], ADSSEC defines the terminal nodes described in Table I and basic non-terminal (binary) operator nodes: Addition ($+$), Subtraction ($-$), Multiplication ($*$), and Division ($/$). These arithmetic operators may be applied because all terminal nodes are either integer or floating-point values. These nodes allow evolution of novel schemes while still being able to represent current schemes.

A. Heuristic Representation

Mapping variable scoring heuristic functions to objects that can be easily manipulated in a GP tree is fairly straightforward. Each scoring heuristic can be represented as a parse tree where non-terminal nodes are operators and terminal

nodes are state-related values. See Figure 9 for an example of a tree representation of an evolved heuristic.

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 in a pre-built solver is replaced by compiling and linking with the new variable scoring heuristic. The resulting solver, termed the *evolved variant*, is executed on test instances to evaluate the effectiveness of the heuristic.

B. Objectives

The evolutionary algorithm (EA) objective score represents how well an evolved variant performs on a provided training set of instances. Traditionally, the aim is to reduce the runtime needed to either find a solution or prove unsatisfiability. In Sect. IV-A, we compare performance measures, i.e., objective functions, to use in ADSSEC; our ultimate goal is to reduce the average runtime across instances in a problem class.

ADSSEC's per-instance sub-score for an evolved variant is the ratio of the performance of the evolved solver to that of the original solver. The objective score is then simply the average of all the instance sub-scores. Thus, any evolved individual that performs identically to the original solver's variable scoring heuristic will end up with an objective score of 1.0, and lower scores indicate better schemes.

Occasionally, the EA will construct inadequate heuristics that cause the solver to require inordinate resources to terminate on some instances. Limiting functions prevent wasting evaluation time on such heuristics. ADSSEC relies on the original solver's performance to approximate reasonable limits for any given SAT instance. Initially, ADSSEC limits an evolved variant to three times the number of variable decisions the original solver needed to solve that instance. These generous limits allow ADSSEC to collect diverse genetic material in a population; they do not time out on all tested instances, but they are generally worse than the original solver. However, as ADSSEC progresses through the evolutionary process, interest shifts from collecting diverse heuristics to exploiting heuristics that are strictly better than the original. As such, the decision limit linearly decreases down to the exact number of decisions the original solver needed for a specific instance. 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 training dataset of interest. This is generally too costly, therefore ADSSEC utilizes strike-based sampling to gauge the effectiveness of an evolved variant. ADSSEC randomly selects a number of instances in a user-defined range from the given training set to evaluate a variant. For each instance in this selection, ADSSEC executes the evolved variant and assigns a sub-score ratio as described before. If the 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 set number of strikes, all remaining sub-scores are assigned the current decision-limit multiplier without evaluation.

C. Evolutionary Algorithm

1) Population Initialization: To avoid local optima, ADSSEC creates each individual in the initial population of variable scoring heuristics by randomly generating a parse tree from the primitives (nodes). First, 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. 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. Otherwise, ADSSEC randomly assigns one of the other terminal node options. This bias was introduced because most current schemes appear to rely heavily on the previous variable score. The maximum depth of a tree generated for the initial population is arbitrarily set to eight. Smaller depths seemed to contain much less genetic diversity while larger depths produced complex heuristics that rarely solved instances within the decision limits.

2) Variation Operators: As in traditional tree GP, ADSSEC uses one of two methods to develop a single offspring (variant): mutation or recombination. For mutation, ADSSEC randomly selects a node in a random individual's parse tree and replaces it with a new subtree generated using the rules established in Population Initialization (Sect. III-C1) – without a depth limitation. 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. Both procedures produce a single child.

3) Selection: The survival selection function determines which individuals in the population continue into the next generation. In ADSSEC, genetically diverse selection is desirable so that smaller parse trees (which are generated more easily) do not flood the population. Certain small heuristics have adequate performance and, if one is discovered early on in evolution, it can overwhelm the population if diversity is not maintained.

Crowding functions are selection functions that excel at promoting genetic diversity in the population [14]. 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. Bertels developed an asynchronous crowding function that allows offspring to compete with either their parents or any *'siblings'* – or descendants of siblings – that replaced the parents [4], [5]. The function uses a computationally cheap distance function comparing histograms of node types (e.g., addition, constant, conflicts, etc.) to determine the closest remaining relative in the current population. As in a standard

crowding function, only the offspring or its closest remaining relative remains in the population, and the population remains at a fixed size.

We use k-Tournament parent selection, which is easily applicable to parallel systems and which provides selection pressure modification through the k parameter. We selected a low k to increase the chance that genetic material in the pool propagates to many individuals in the population while maintaining some pressure to eliminate less fit individuals.

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 converges. 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 solver are replaced. This mechanism is useful in restarting the exploration of the variable scoring heuristic search space.

IV. Experiments

A. Comparison of Objectives

The main motivation for this experiment was to determine a good objective for the EA employed by ADSSEC. ADSSEC evaluates several instances in parallel on the same hardware, and so runtimes for individual instances are inconsistent. For that reason, ADSSEC was originally implemented both as a single objective EA (SOEA) with the number of variable decisions as an objective and as a multi-objective EA (MOEA) with the number of variable decisions and the number of learnt clause literals (conflict literals) as objectives. An improved variable scoring heuristic should reduce the values for the number of decisions and conflict literals, potentially reducing the runtime, but lower values in CDCL solvers may not correlate well with the runtime in specific cases.

Thus, we decided to explore the performance of solvers produced by ADSSEC when using CPU time as an objective. We compare an SOEA with decisions as an objective to an SOEA with CPU time as an objective, and we compare an MOEA with decisions and conflict literals as objectives to an MOEA with decisions, conflict literals, and CPU time as objectives. We explore both SOEA and MOEA because experiments in [4] suggested that ADSSEC produced more performant evolved variants using an MOEA.

In order to achieve a statistically significant results, we ran the evolutionary process with the same settings 30 times for each of the four different classes that we compare. Since evolutionary algorithms are computationally expensive, we needed a dataset whose instances can be quickly solved. For this experiment, we chose the ibm-26 dataset, one of many subsets of the IBM - Hardware Verification set from the IBM Formal Verification Benchmark Library, from the Configurable SAT Solver Challenge (CSSC) 2014 benchmarks [31]. MiniSat can solve all instances in less than 4 seconds.

We evolved the variable scoring heuristic from the MiniSat solver, using an identical parameter configuration (Table II; more on parameter configuration in Sect. IV-B) for all SOEA and MOEA experiments. ADSSEC evaluated each evolved variant on a randomly ordered subset of all of the instances from the ibm-26 dataset, employing the evaluation described previously.

We executed ADSSEC on the same machine for each objective function. Solvers were then compiled with the best heuristic produced by each run. To obtain more accurate CPU solve times, we averaged the CPU time collected over 30 serial runs of each evolved variant and the original MiniSat on each instance in the ibm-26 dataset. All evaluations were performed on the same machine. Results are presented in Sect. V-A.

B. Evolving Variable Scoring Heuristics

Based on preliminary results from experiments like that in Sect. IV-A, and in order to achieve results comparable to results in [4], we ran our main experiments using an MOEA with decisions and conflict literals as objectives for the training phase. The purpose of these experiments was to determine whether ADSSEC could produce more efficient solvers tailored to specific real-world problem classes.

Due to the nature of evolutionary computation and SAT solving, ADSSEC requires datasets with enough instances to represent a distinct instance class for training and testing. These datasets must contain instances that ADSSEC can train on in a short period of time, and the dataset must be difficult enough to benefit from a fitted heuristic. Many available datasets do not fulfill these requirements. For these experiments, we used the unif-k5 dataset, a benchmark from the random track of CSSC 2014 consisting of 600 instances generated by Balint et al. according to the uniform generation model [3], and the ibm-18 dataset, another subset of the IBM - Hardware Verification dataset from the CSSC 2014 industrial track benchmarks [31].

We used an identical configuration for ADSSEC on both datasets (Table II), the same as used in [4]. For each experiment, ADSSEC created an initial population of 30 random individuals. The master process used several slave processes (the exact number varied based on the number of nodes available on the machine used for each experiment) to evaluate offspring populations, asynchronously creating new offspring as each node became available.

ADSSEC selected parents in a k-Tournament selection with k = 2 for either recombination or mutation – with a mutation probability of 0.10 and a recombination rate of 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 individuals in 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.

We used the same experimental setup to evolve variable scoring heuristics in both MiniSat and Glucose. We executed ADSSEC on several locally networked machines of varying

TABLE II: ADSSEC EA parameter settings

Population (μ)	Offspring (λ)	Mutation Rate	Termination Evaluations	Restart Evaluations
30	63	0.10	5000	250

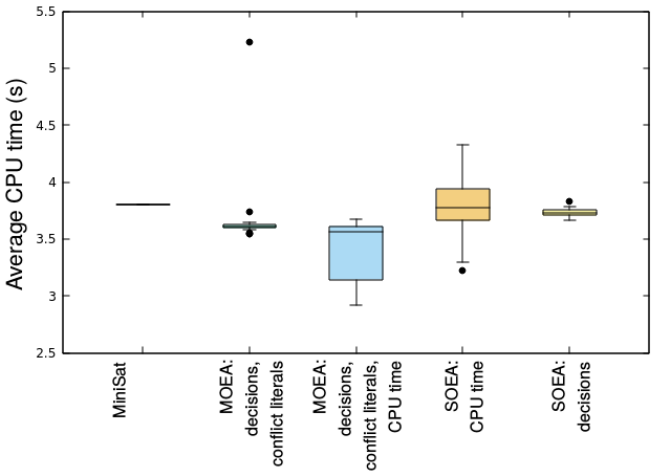


Fig. 1: Comparison of objectives on ibm-26

loads. Solvers were then compiled with the best heuristics produced by each run. For each evolved variant and each original solver, we averaged the CPU time collected over 30 serial runs on each instance. All evaluations were performed on the same machine. Results are presented in Sect. V-B.

V. Results

A. Comparison of Objectives

Figure 1 compares the CPU time that the evolved solvers produced by ADSSEC took to solve all instances in ibm-26 dataset. The SOEA with decisions as the objective produced statistically more efficient solvers than SOEA with CPU time as the objective. The MOEA with decisions, conflict literals, and CPU time produced more efficient solvers than any other combination of objectives that we compare, but these results had higher variance than those produced when not using CPU time as an objective. Our results serve as a guide for further experimentation: different EA parameter configuration or different datasets may yield different results.

B. Evolving Variable Scoring Heuristics

Figure 2 compares the CPU time required to solve instances in the ibm-18 dataset using the original MiniSat solver against that required using an evolved variant with a new variable scoring heuristic. While the minimum CPU times required are fairly close, there is a marked improvement with the new heuristic in the maximum CPU time needed. Because the dataset contains only 15 instances, ADSSEC's training and the test instances are the same. However, in Figure 3 we show that the solver evolved on instances from the ibm-18 showed improvement in CPU time on other IBM Hardware Verification

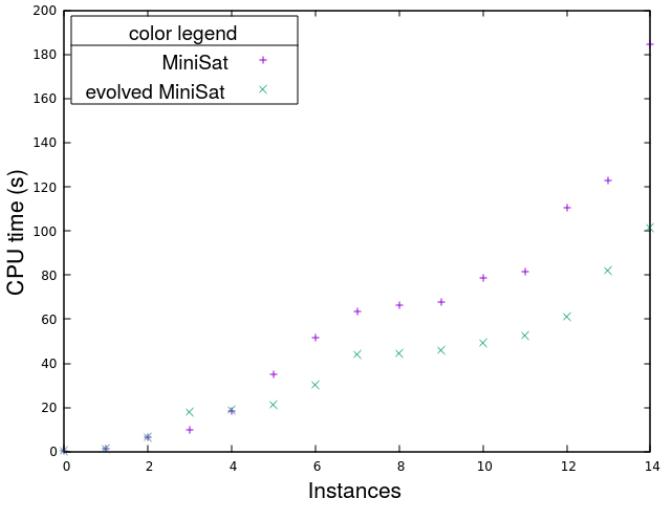


Fig. 2: ibm-18 cactus plot comparing CPU time to solve

dataset subsets on which it was not trained (ibm-15 and ibm-03), especially in maximum CPU time.

Figure 4 compares the CPU time to solve instances from the unif-k5 dataset using the original MiniSat solver against that using MiniSat with an evolved heuristic. Significant improvement can be seen across the board using the evolved variable selection heuristic. The training set for ADSSEC was a randomly selected subset of ten instances from the dataset; both training and test instances are covered in the plot.

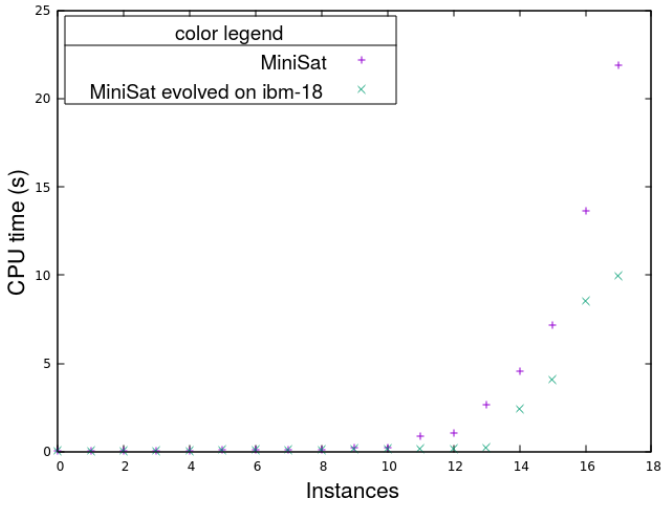
We show in Figure 5 that the solver evolved for the ibm-18 dataset did not show improvement in CPU time on instances from the unif-k5 dataset, which is from a different track. This supports the hypothesis that improved performance is obtained by targeting the specific structure of a problem class.

Figure 6 compares the CPU time to solve instances from the ibm-18 dataset using the original Glucose solver against that using Glucose with an evolved heuristic. Evolving a variable scoring heuristic in Glucose to target problems from the ibm-18 dataset did not show as much improvement as evolving the variable scoring heuristic in MiniSat. Still, some improvement was seen: the maximum CPU time to solve improved, and the average CPU time to solve all instances also improved slightly.

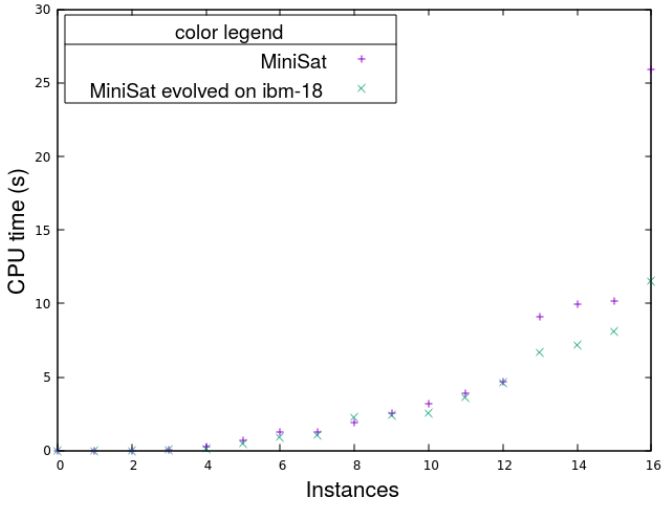
Figure 7 compares the CPU time to solve instances from the unif-k5 dataset using the original Glucose solver against that of Glucose with an evolved heuristic. As with MiniSat, the new heuristic for the unif-k5 dataset in Glucose significantly improved the solver for this dataset. The training set for ADSSEC was a randomly selected subset of ten instances from the dataset; both training and test instances are covered in the plot.

VI. Discussion

In Figure 8 we compare solvers with evolved heuristics to some of the top state-of-the-art CDCL solvers, specifically Glucose 4.1 [1] and the SAT 16 competition version of Lingeling [6], on instances from the ibm-18 and unif-k5



(a) ibm-15



(b) ibm-03

Fig. 3: Cactus plot comparing CPU time to solve of evolved solver trained on ibm-18 dataset and tested on other datasets from IBM Hardware Verification dataset

datasets. We also show MiniSat for comparison. On the unif-k5 dataset, the evolved solvers significantly outperformed both Glucose and Lingeling. On ibm-18, the solvers did better than Lingeling. The evolved Glucose solver also did better than Glucose in overall runtime. Again, we averaged the CPU time collected over 30 runs of each solver on same machine in order to obtain more accurate CPU times.

In Figure 9, we present the tree representation of the best heuristic evolved by ADSSEC for the unif-k5 dataset using MiniSat. Interestingly, it is heavily based on the current number of conflicts encountered (i).

VII. Conclusions

Our results show that ADSSEC is capable of evolving variable scoring heuristics that are able to outperform default MiniSat or Glucose on a specific problem class. Even better,

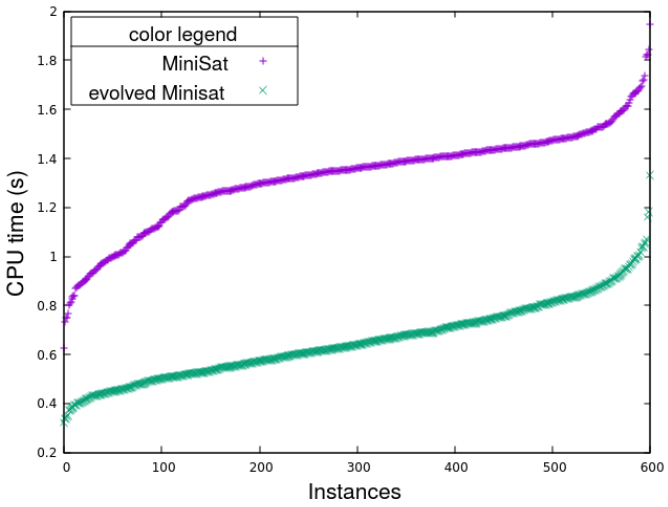


Fig. 4: unif-k5 cactus plot comparing CPU time to solve

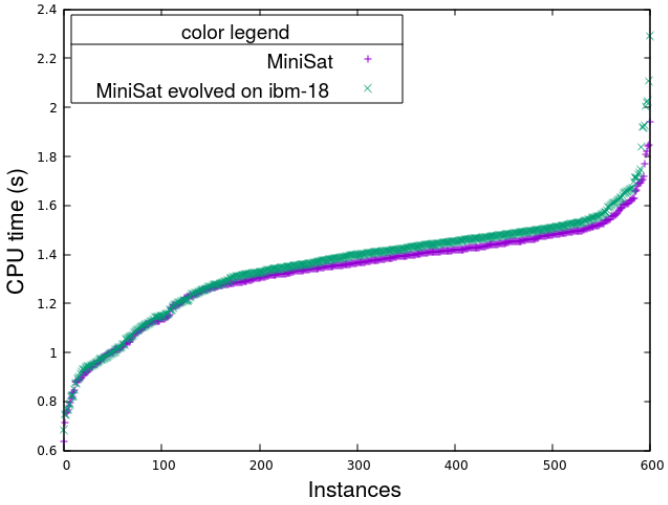


Fig. 5: unif-k5 cactus plot comparing CPU time to solve, evolved solver trained on ibm-18 dataset

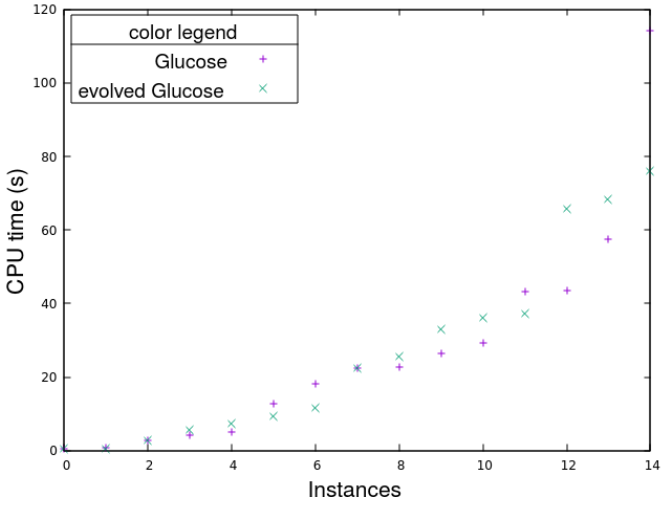


Fig. 6: ibm-18 cactus plot comparing CPU time to solve

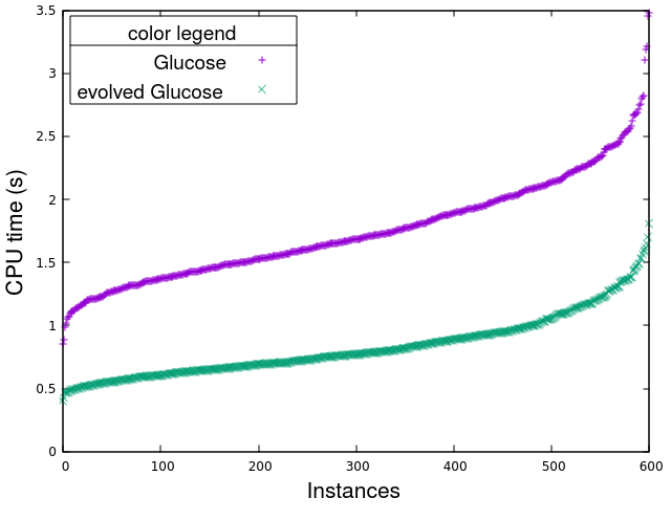
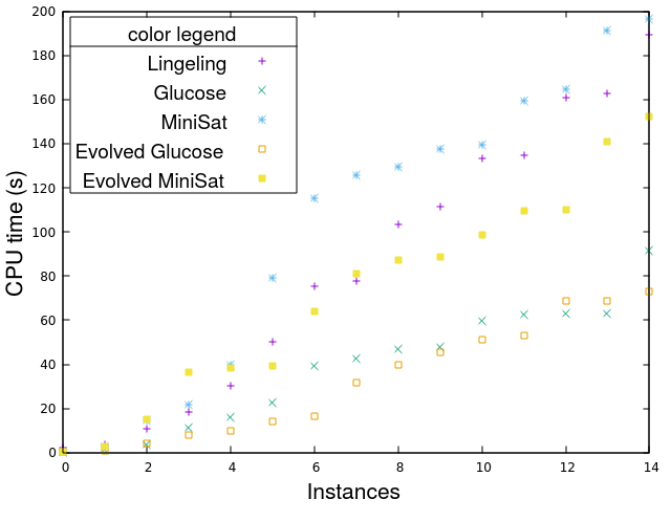
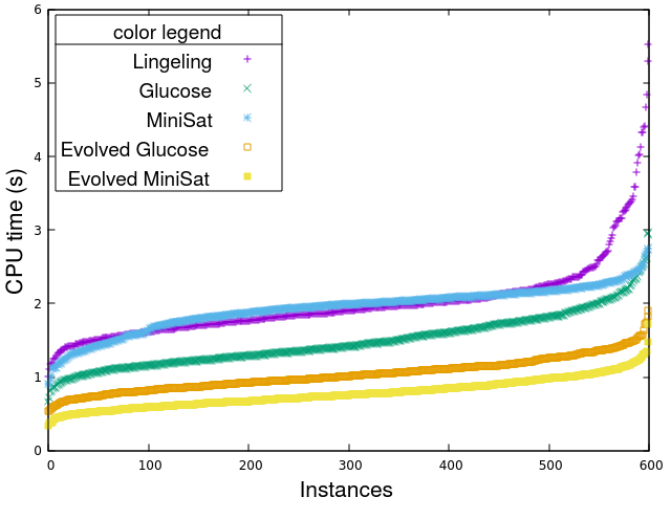


Fig. 7: unif-k5 cactus plot comparing CPU time to solve



(a) ibm-18



(b) unif-k5

Fig. 8: Cactus plot comparing CPU time to solve of evolved solvers and top solvers

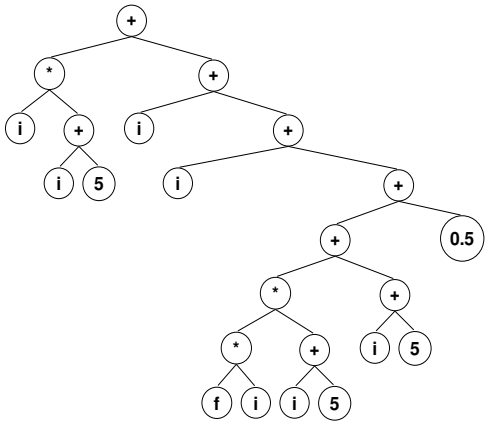


Fig. 9: Evolved heuristic for MiniSat for the unif-k5 dataset

the evolved heuristics seem to outperform some of the state-of-the-art solvers.

We found that even when not using CPU time as an objective during the training phase, we were still able to show major improvement in CPU times of evolved solvers. As our results showed, a multi-objective evolutionary algorithm using CPU time as one of the objectives might lead to even better results than the ones we achieved.

Previous research showed that an automatically optimized solver that needs less CPU time to solve a set of instances is more energy efficient than an original solver [9]. We showed that ADSSEC found solvers that required less CPU time to solve instances from some benchmarks than some of the most energy efficient state-of-the-art solvers; thus, we believe that our system can produce more energy efficient solutions to real-world problems.

References

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

[2] 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.

[3] Adrian Balint, Anton Belov, Matti Jrvisalo, and Carsten Sinz. SAT Challenge 2012 Random SAT Track: Description of Benchmark Generation. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, pages 72–73, Department of Computer Science Series of Publications B, University of Helsinki, Finland, 2012.

[4] Alex R. Bertels. Automated Design of Boolean Satisfiability Solvers Employing Evolutionary Computation. Master's thesis, Missouri University of Science and Technology, Rolla, Missouri, USA, 2016.

[5] Alex R. Bertels and Daniel R. Tauritz. Why Asynchronous Parallel Evolution is the Future of Hyper-heuristics: A CDCL SAT Solver Case Study. In *Proceedings of the 18th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '16)*, pages 1359–1365, Denver, Colorado, USA, July 2016.

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

[7] 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.

[8] 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.

[9] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1327–1334, New York, NY, USA, 2015. ACM.

[10] 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.

[11] George Chin, Daniel G. Chavarria, Grant C. Nakamura, and Heidi J. Sofia. Biographe: high-performance bionetwork analysis using the biological graph environment. *BMC Bioinformatics*, 9(6):S6, 2008.

[12] F. Corno and F. Razzak. Intelligent energy optimization for user intelligible goals in smart home environments. *IEEE Transactions on Smart Grid*, 3(4):2128–2135, Dec 2012.

[13] 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.

[14] Agoston E Eiben and James E Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] F. Imeson and S. L. Smith. A language for robot path planning in discrete environments: The tsp with boolean satisfiability constraints. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5772–5777, May 2014.

[23] F. Imeson and S. L. Smith. Multi-robot task planning and sequencing using the sat-tsp language. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5397–5402, May 2015.

[24] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Satenstein: Automatically building local search sat solvers from components. *Artificial Intelligence*, 232:20 – 42, 2016.

[25] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434, 2004.

[26] 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.

[27] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. Sift: Building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 298–309, New York, NY, USA, 2015. ACM.

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

[29] 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.

[30] 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.

[31] Research Group on Learning, Optimization, and Automated Algorithm Design at Freiburg University. Configurable SAT Solver Challenge (CSSC) 2014 - Benchmarks. http://aclib.net/cssc2014/benchmarks.html. [Online; accessed 02-April-2017].

[32] Riccardo Poli, William B Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, March 2008. (With contributions by J. R. Koza).

[33] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying uml/ocl models using boolean satisfiability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1341–1344, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[34] Ricardo Soto, Hakan Kjellerstrand, Orlando Durn, Broderick Crawford, Eric Monfroy, and Fernando Paredes. Cell formation in group technology using constraint programming and boolean satisfiability. *Expert Systems with Applications*, 39(13):11423 – 11427, 2012.

[35] Tichy, Richard and Glase, Thomas. Clause Learning in SAT. University of Potsdam., April 2006.

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

[37] Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, Nov 2015.

[38] 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.

[39] 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.

[40] 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.

[41] 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.