SAND2016-4057C

The Automated Design of Variable Selection Heuristics for CDCL Solvers To Target Problem Classes

Alex R Bertels[1], Caleb A Roberts[1], Daniel R Tauritz[1], Samuel A Mulder[2], and Denis Bueno[2]

[1] Missouri University of Science and Technology, Rolla, MO, U.S.A.
[2] Sandia National Laboratories, Albuquerque, NM, U.S.A.
arb9z4@mst.edu dtauritz@acm.org

Abstract

Current CDCL solvers employ an all-purpose variable scoring heuristic to perform variable selection when solving arbitrary SAT problems. Previous work has shown that instances derived from a particular problem class exhibit a unique underlying structure – as shown by the variable interactions – 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 proof-of-concept system for automating the design of variable scoring heuristics for CDCL solvers to make it feasible to target arbitrary, but particular, problem classes. Additionally, this paper gives experimental results demonstrating that this system, which evolves variable scoring heuristics using an asynchronous parallel hyper-heuristics approach employing genetic programming, has the potential to create efficient solver portfolios targeting particular problem classes.

1 Introduction

Classes of structured Boolean Satisfiability (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 [23, 24, 20]. To efficiently solve instances in a specific class, we must find the solver and parameter configuration that perform best for that class.

Recent work has automatically optimized SAT solver parameter configuration to target specific classes of structured instances [13, 12, 7], and other work automatically evolved variable selection techniques for stochastic local search (SLS) solvers [1, 15, 8, 9, 10]. However, conflict-driven clause learning (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 [3, 2], we know of no work that automatically evolves CDCL operations. We believe that appropriate CDCL operations will increase the effectiveness of a CDCL SAT solver in targeting classes of instances with unique structure.

To test this belief, we developed a proof-of-concept experimental system for automating the design of variable scoring heuristics for the variable selection schemes in CDCL solvers. Our system, **ADSSEC** (**A**utomated **D**esign of **S**AT **S**olvers employing **E**volutionary **C**omputation), makes it feasible to target arbitrary, but particular, classes of structured instances. The diagram in Fig. 1 illustrates how ADSSEC-generated SAT solvers relate to the entire SAT solver space.

Our research makes the following contributions:

- Introduces a generative hyper-heuristic approach employing genetic programming to automatically construct variable scoring heuristics for a CDCL solver.

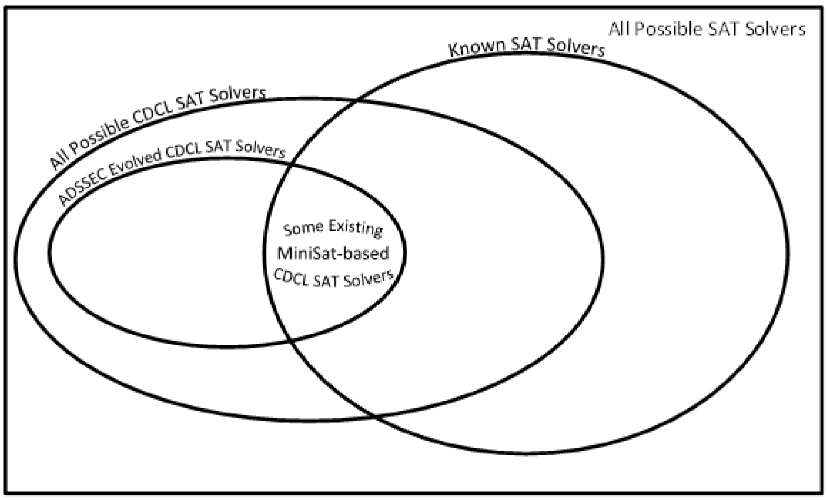

All Possible SAT Solvers
Known SAT Solvers
All Possible CDCL SAT Solvers
ADSSEC Evolved CDCL SAT Solvers
Some Existing MiniSat-based CDCL SAT Solvers

Figure 1: Search space of ADSSEC (not to scale)

- Significantly decreases variable scoring 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 our hyper-heuristic approach to create efficient solver portfolios targeting particular problem classes.

In this paper, we describe the ADSSEC system, evaluate it on two small datasets, and present results demonstrating its potential.

2 Related Work

Ideally, a single solver would be able to adapt to an application at runtime. Tools such as ParamILS [13], SMAC [12], and – most recently – SpySMAC [7] 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 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 [21, 22]. 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 [14]. 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.

We take the next step by applying generative hyper-heuristics [4], 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 [4]. 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 the variable scoring heuristic is being modified and the generative approach could explore more of the search space. As is typical, our hyper-heuristic employs genetic programming (GP) to automatically reorganize and manipulate the algorithmic primitives constituting the variable scoring heuristic [17, 18]. 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 variable scoring heuristics. 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.

3 Evolutionary Approach

Influenced by the success of Fukunaga in improving SLS runtimes by evolving specific heuristics [8], we use genetic programming (GP) to evolve variable scoring heuristics to automatically target CDCL solvers to specific classes of structured instances. In particular, we use Koza-style GP [19] as it is well suited to representing the parse trees representing variable scoring heuristics. Biere and Fröhlich demonstrated that the current best variable scoring heuristics are roughly equal in runtime performance when evaluated across many instance classes [3]. Biere and Fröhlich's work motivated our choice to adapt CDCL solver variable scoring heuristics.

Martin et al. showed the benefits of using an asynchronous parallel evolutionary algorithm (APEA) when evolving populations of individuals with heterogeneous evaluation times [16]. As SAT solving is a canonical example of a problem with heterogeneous evaluation times, we implemented ADSSEC using an APEA.

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 [5], a commonly used efficient and deterministic CDCL solver with dense source code. While ADSSEC employs a standard parent selection before producing offspring, we constructed a survival selection specifically for the APEA. ADSSEC returns the heuristics from the final population after reaching the termination criteria.

3.1 Heuristic Representation

Mapping variable scoring heuristic functions to objects easily manipulated in a GP is fairly straight-forward. We represent each scoring heuristic as a parse tree where non-terminal nodes are operators and terminal nodes are state-related values (see below).
Derived from currently implemented variable scoring heuristics [3], 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 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):

$$h_i^m = \begin{cases} 0.5 \cdot s & \text{if } m \text{ divides } i \text{ evenly} \\ s & \text{otherwise} \end{cases} \tag{1}$$

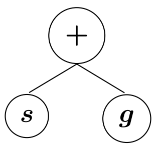
where m is a power of 2: $\{2, 4, 8, \ldots, 1024\}$.

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 allow us to evolve 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, we convert the parse tree into a C++ statement. We replace the original variable scoring heuristic in a pre-built MiniSat 2.2 by compiling and linking in the new variable scoring heuristic (C++ statement). We execute the resulting solver on test instances to evaluate the effectiveness of the heuristic. This method allows us to take advantage of MiniSat and the performance derived from being implemented in C++ while reducing development time.

3.2 Objective

The evolutionary algorithm (EA) 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's 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 we aim 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. Instead of runtime, we use the number of variable decisions as a consistent metric. An improved variable scoring heuristic should reduce this value. Our 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. We define limiting functions 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. 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, our 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. 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. 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 set number of strikes, all remaining sub-scores are assigned the current decision-limit multiplier.

3.3 Evolutionary Algorithm

3.3.1 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 (Sect. 3.1). 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. We introduced this bias because most current schemes appear to rely on the previous variable score. We arbitrarily set the maximum depth of a tree generated for the initial population to eight. Smaller depths contained much less genetic diversity while larger trees produced complex heuristics that rarely solved instances in the decision limits.

3.3.2 Variation Operators.

ADSSEC uses one of two methods to develop a single offspring (variant): mutation or recombination. For mutation, ADSSEC simply randomly selects a node in a random individual's parse tree and replaces it with a new branch generated using the rules established in Population Initialization (Sect. 3.3.1) – without a depth limitation. Typically, APEA's mitigate tree growth by promoting an implicit parsimony pressure. This is under the assumption that smaller trees have short evaluation times and return to the population sooner. However, the strike-based sampling terminated bad heuristics quickly, which 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.

3.3.3 Selection.

The survival selection chooses which individuals in the population continue into the next generation. In ADSSEC, we want to encourage genetically diverse selection 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 [6]. 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. We developed an asynchronous crowding function that allows offspring to compete with either their parents or any *'siblings'* – or potentially descendants of siblings – that replaced the parents. We use 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. The population will always hold at a fixed size given that as an existing individual in the population is removed an offspring will be accepted.

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.

3.3.4 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 upon in a user-defined number of evaluations, ADSSEC introduces new material to the gene pool. Currently, we replace all variants whose performance is worse than that of the original MiniSat. This mechanism is useful in restarting the exploration of the variable scoring heuristics search space.

4 Experiments

Ideally, we 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. Our experiments with the prototype ADSSEC system require datasets that have:

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

Unfortunately, these requirements make many of the usual SAT datasets inappropriate for our initial prototype experiments. Many publicly available datasets contain too few instances to fully 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, we 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. We chose to generate datasets for ADSSEC as generators provide us with enough control to meet these criteria while keeping us from presenting bias by hand-selecting specific instances.

We used a k-colorable graph generation tool [1] – developed by Joseph Culberson, Adam Beacham, and Denis Papp – to create 66 satisfiable instances for the first dataset. These graphs each had 5000 vertices with an average degree of 4.31. We used a SAT conversion tool from the same source to transform each graph into a CNF. Each CNF contained 52324 variables in 15000 clauses. We solved all instances with default MiniSat, and we used the ten instances with the highest number of decisions 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.

We used a modularity-based generator developed by Jesús Giráldez-Cru [2] 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 CNF contained 5000 variables in 19000 clauses. The generator allows the user to specify the structure of each instance; we generated 90 communities with a modularity of 0.8 and 3 literals per clause. We used seeds 1 through 40. Again, we trained ADSSEC on MiniSat's worst ten from this dataset.

We used an identical configuration for ADSSEC on both datasets (Table 1); we used manual tuning 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 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

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

Table 1: ADSSEC EA parameter settings

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

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.

We executed ADSSEC 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, we used Amazon EC2 m3.medium instances to collect the runtimes of the original MiniSat and the evolved solvers on both datasets. We executed MiniSat serially on all instances – including those trained on – from both the k-colorable graph and modularity datasets, but we only executed the evolved solvers on the datasets on which they were trained.

5 Results

Figure 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.

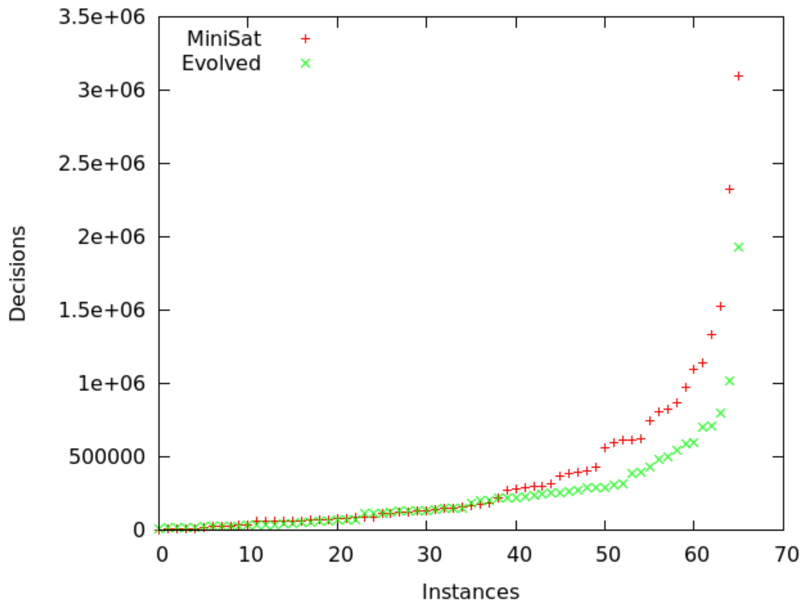


Figure 2: k-colorable graph cactus plot comparing number of decisions to solve

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 Fig. 3).

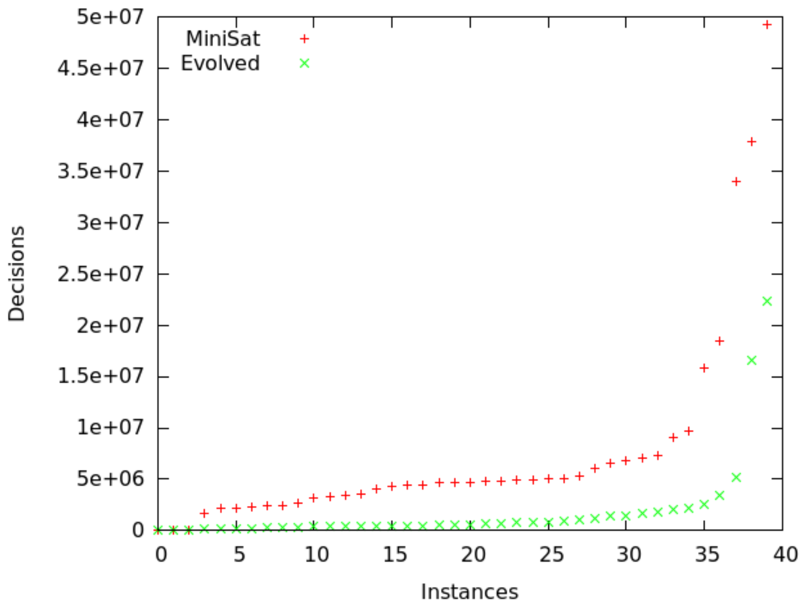


Figure 3: Modularity cactus plot comparing number of decisions to solve

Table 2 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 we executed both the original MiniSat and the evolved solver on the same instances, we were able to compare runtimes to determine whether the evolved solver is actually more efficient. With a p-value of 0.3562, we cannot conclusively state that these values definitely reflect an improvement in terms of runtime.

Table 2: 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	

Table 2 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.

As expected, the number of decisions seemed to be proportionate to the CPU time for the k-colorable graph dataset (Fig. 4); however, this was not true for the modularity dataset (Fig. 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.

We explore whether the two solvers for each dataset perform well (or poorly) on the same instances by keeping the lower runtime for each instance; these times are labeled "*Minimum*" in Fig. 4 and Fig. 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.

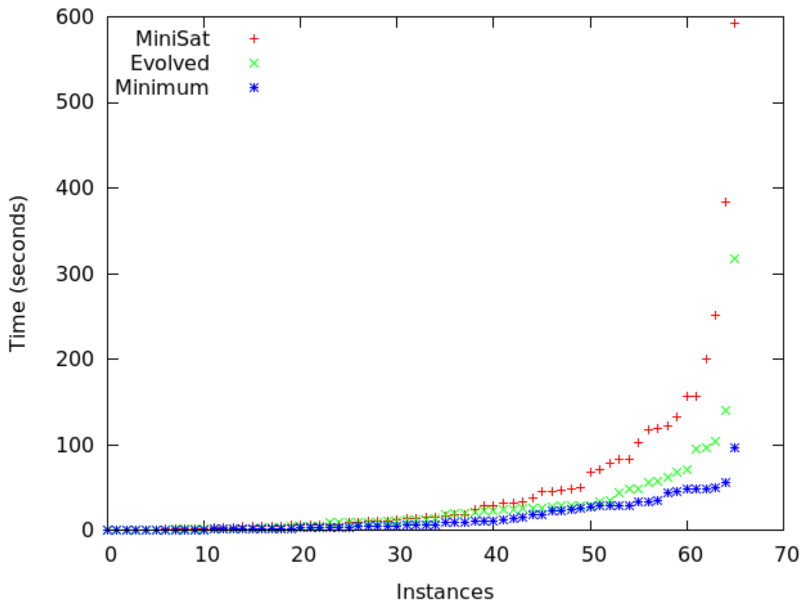


Figure 4: k-colorable graph cactus plot comparing CPU time needed to solve

As seen here, ADSSEC cannot depend solely on the number of decisions in evaluating an instance. To discover what other metric(s) we should use, we compare the final state metrics in the worst modularity instances for the original MiniSat and the evolved solver (Table 3). Evolved has fewer restarts, conflicts, decisions, and propagations, but as we are comparing different instances, we cannot draw meaningful conclusions 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 we see that the number of conflict literals affects runtime, we believe 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.

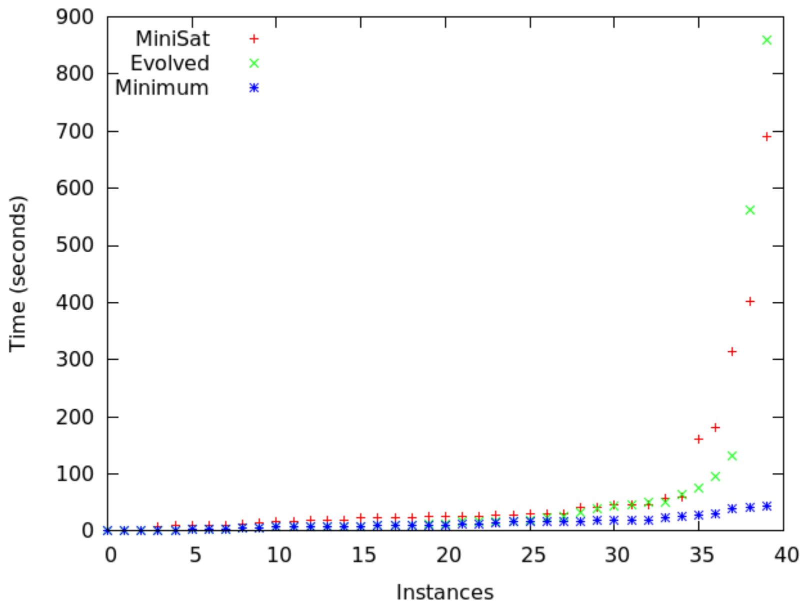


Figure 5: Modularity cactus plot comparing CPU time needed to solve

Table 3: 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

6 Discussion

Fig. 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 we were to add a pruning function in ADSSEC, we would only prune the final heuristic after termination.

Fig. 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 work, we could explore looking back through the ancestors of the individual to discover 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

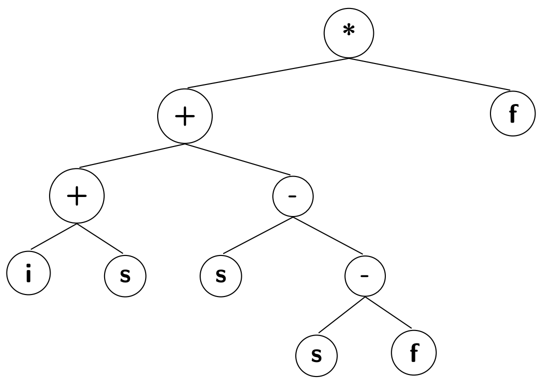


Figure 6: Evolved heuristic for k-colorable graph instances

that represent other solver state-based information.

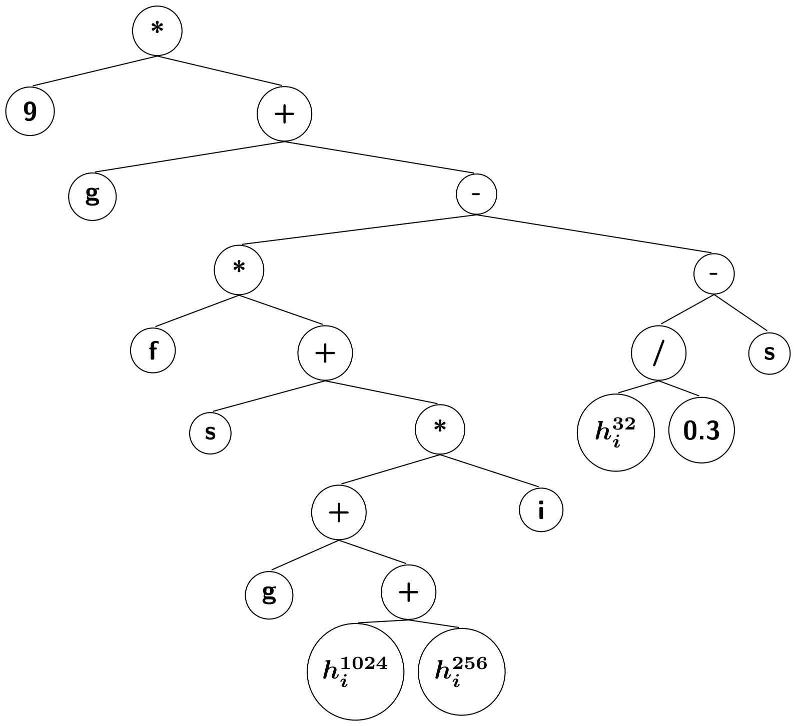


Figure 7: Evolved heuristic for modularity instances

7 Conclusions

Our results show that ADSSEC 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.

Interestingly, we found that a lower number of decisions does not always guarantee faster performance. Other metrics also should be considered when automatically modifying components of an existing solver to target specific problem classes; for example, number of conflict literals is a promising metric.

8 Future Work

ADSSEC offers many possible areas of exploration:

- Developing a more accurate objective function using other metrics such as number of conflict literals should provide better evaluations and ultimately better variants.
- Allowing different data structures to store the variable scores may speed up the execution time of ADSSEC [3].
- Adjusting bounds on maximum variable activity scores or preventing exponential growth of scores would reduce the number of times variable scores need to be scaled down.
- Evolving restart and clause learning schemes in conjunction with variable scoring heuristics may result in more flexible solvers.
- 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 [11]; 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 we targeted a solver to the worst instances in a dataset, it would be interesting to explore potential selection schemes to target evolving a single solver for an entire dataset.
- 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.
- Evolving the variable scoring heuristic alongside the 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 a multi-objective objective score.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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).

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.