Effective Auxiliary Variables
via Structured Reencoding

Andrew Haberlandt[1] ✉ 📧
Carnegie Mellon University, Pittsburgh, PA, USA

Harrison Green[1] ✉ 📧
Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule ✉ 🏠 📧
Carnegie Mellon University, Pittsburgh, PA, USA

---- **Abstract** ----

Extended resolution shows that auxiliary variables are very powerful in theory. However, attempts to exploit this potential in practice have had limited success. One reasonably effective method in this regard is bounded variable addition (BVA), which automatically reencodes formulas by introducing new variables and eliminating clauses, often significantly reducing formula size. We find motivating examples suggesting that the performance improvement caused by BVA stems not only from this size reduction but also from the introduction of *effective auxiliary variables*. Analyzing specific packing-coloring instances, we discover that BVA is *fragile* with respect to formula randomization, relying on variable order to break ties. With this understanding, we augment BVA with a heuristic for breaking ties in a *structured* way. We evaluate our new preprocessing technique, Structured BVA (SBVA), on more than 29 000 formulas from previous SAT competitions and show that it is robust to randomization. In a simulated competition setting, our implementation outperforms BVA on both randomized and original formulas, and appears to be well-suited for certain families of formulas.

1 Introduction

Pre-processing techniques that introduce and eliminate auxiliary variables have been shown to be helpful both in theory and practice. Theoretically, auxiliary variables lift the power of solvers from the resolution proof system to Extended Resolution (ER) [23, 8, 11]. In practice, efforts to exploit this full power of ER have had limited success; however, auxiliary variables have been used to reencode formulas in a way that drastically reduces their size [4, 2, 15], often leading to a decreased solve time. In this work we show that this speedup may not be caused entirely by the reduction in formula size, but by the introduction of certain *effective auxiliary variables*.

---

[1] Authors contributed equally.

A very powerful pre-processing technique is *Bounded Variable Elimination* (BVE) [9]. As its name suggests, it eliminates a variable x by resolving each clause containing a literal x with every clause containing literal $\overline{x}$. Importantly, BVE only performs such an elimination if it helps reduce the formula size (measured as the number of clauses plus the number of variables). A pre-processing technique that introduces new auxiliary variables is *Bounded Variable Addition* (BVA) [15], which is the focus of this article. It is well known that the introduction of auxiliary variables is crucial for many succinct encodings (e.g., the Tseitin transformation [23], or cardinality constraints [20, 14]). Following the intuition of BVE, BVA will only introduce a new variable if it can eliminate a larger number of clauses than it adds.

Auxiliary variables may not only be useful to reduce the size of a formula, but they can also capture some semantic meaning about the underlying problem to encode, as we detail in Section 3. As a case study, we consider a recent encoding used by Subercaseaux and Heule for computing the packing chromatic number of the infinite square grid via SAT solving [22]. In their work, BVA was found to generate auxiliary variables that represented clusters of neighboring vertices of the grid. The encoding resulting from running BVA on a direct encoding of the problem inspired a more efficient encoding, by suggesting the usefulness of having auxiliary variables capturing clusters of vertices. In this paper, we offer new insight into this *"meaningful variables"* phenomenon, which we believe can generalize to other problems as well. Furthermore, even though the reencoding resulting from BVA suggested meaningful new variables for the packing coloring problem, it was not as effective as manually designing a more structured encoding based on some of those variables. We take this as motivation to identify shortcomings of BVA and improve upon its design.

In general, on problems where BVA is effective, the effect tends to be extreme. BVA is able to reduce the number of clauses by a $\times 10$ factor or more, improving solve time by orders of magnitude. However, in this paper, we find that this reduction in solve time is highly sensitive to randomly scrambling the formula (even when controlling for how CDCL solvers are generally sensitive to this form of randomization [3]). In particular, randomizing the order of variables and clauses prior to BVA substantially reduces the positive effect of BVA on solve time, despite maintaining the same overall reduction in formula size. Using the packing k-color problem, we show that the effectiveness of BVA relies on the introduction of a few specific variables that account for only a small fraction of the reduction in formula size. Moreover, we identify that the lack of effective tie-breaking in BVA is the cause of this high sensitivity to randomization. Inspired by these new insights into the behavior of BVA, we present SBVA (Section 4), a version augmented with a tie-breaking heuristic that enables it to introduce better auxiliary variables at each step, even when the original formula is randomized. Our heuristic is based on a connectivity measure between variables in the *incidence graph* of CNF formulas, which is preserved under randomization of the formula. As a result, SBVA is able to identify effective auxiliary variables even when the original formula is scrambled. We evaluate our implementation by running it on more than 29 000 formulas from the Global Benchmark Database [12]. Experimental results, presented in Section 5, demonstrate that our approach outperforms the original implementation of BVA.

In summary, the main contributions of this article are:

1. We offer new insight into the behavior of BVA, by exhibiting its ability to introduce effective auxiliary variables and showing its sensitivity to formula randomization.

2. We design SBVA, a heuristic-guided form of BVA, that introduces new variables in a way that is robust to randomization.

3. We perform a large-scale evaluation of both BVA and SBVA on benchmark problems from the SAT Competition and study their behavior on different families of instances.

4. We release an open-source implementation of SBVA that supports DRAT proof logging.

2 Preliminaries

A *literal* is either a variable x, or its negation ($\overline{x}$). A *propositional formula* in *conjunctive normal form* (CNF) is a conjunction of *clauses*, which are themselves disjunctions of literals. An *assignment* is a mapping from variables to truth values. A positive (negative) literal is true if the corresponding variable is assigned to true (false, respectively). An assignment satisfies a clause if at least one of its literals is true, and we say an *formula* is satisfied if all of its clauses are. A formula is *satisfiable* (SAT) if there exists an assignment that satisfies it, or *unsatisfiable* (UNSAT) otherwise. For example, the formula $(x \vee \overline{y}) \wedge (\overline{x} \vee z)$ is made up of two clauses, $(x \vee \overline{y})$ and $(\overline{x} \vee z)$, each with two literals. This formula is satisfiable, since the assignment of x and z to true and y to false satisfies it.

Auxiliary Variables. There can be many equivalent ways of encoding a problem into CNF, differing in the meaning assigned to individual variables. Problems often have a *direct encoding*, in which variables are assigned for each individual decision element present in a problem. For example, in a direct encoding of graph coloring, there are $k|V|$ variables, where each $v_{i,c}$ represents whether node i has color c and k is the number of colors.

Although direct encodings are often the most intuitive, more efficient encodings are known for a wide variety of problems. These encodings often add *auxiliary variables* to the formula, which capture properties about a group of variables. One of the simplest examples is an $\text{AtMostOne}(x_1, \ldots, x_n)$ constraint, which requires that at most one of the variables $x_1, \ldots, x_n$ is true. Without adding auxiliary variables, this constraint requires $\Theta(n^2)$ clauses, which are typically binary clauses between every pair of variables [14]. However, with the introduction of auxiliary variables, this constraint can be encoded in a linear number of clauses and variables as follows [13]:

$$\text{AtMostOne}(x_1, \ldots, x_n) = \text{AtMostOne}(x_1, x_2, x_3, y) \wedge \text{AtMostOne}(x_4, \ldots, x_n, \overline{y}) \qquad (1)$$

where the pairwise encoding is used for $\text{AtMostOne}(x_1, \ldots, x_n)$ where $n < 4$. The split AtMostOne constraints require that at most one of $\{x_1, x_2, x_3\}$ is true, and at most one of $\{x_4, \ldots, x_n\}$ is true, respectively. The added auxiliary variable y prevents a variable in both of the groups from being true. The auxiliary variable y is forced false if any of x_1, x_2, or x_3 are true, and forced true if any of $x_4, \ldots, x_n$ are false. If a literal from both groups is true, the auxiliary variable y prevents the formula from being satisfiable.

Extended Resolution. Starting from the original formula, the Extended Resolution proof allows only two simple rules:
1. *Resolution*: Given clauses $C \vee p$ and $D \vee \overline{p}$, add the clause $C \vee D$ to the proof.
2. *Extension*: Define a new variable x as $x \leftrightarrow \overline{a} \vee \overline{b}$, where a and b are literals in the current proof. Add the clauses $x \vee a$, $x \vee b$, and $\overline{x} \vee \overline{a} \vee \overline{b}$ to the proof.

In *resolution*, the clause $C \vee D$ is implied by the first two clauses, resulting in a logically equivalent formula. In *extension*, however, the introduction of a new variable x is not implied by the original clauses, and results in a formula that preserves satisfiability and is only logically equivalent over the original variables.

Using the extension rule, new variables can be defined in terms of existing variables. The original rule defined by Tseitin [23] only allows for definitions of the form $x \leftrightarrow \overline{a} \vee \overline{b}$, the construction for which is given in the definition above. However, the extension rule can be applied repeatedly to construct variables corresponding to arbitrary propositional formulas over the original variables. This flexibility is key to the success of Extended Resolution, but it provides no guidance on how these extensions should be chosen.

Bounded Variable Addition. Bounded Variable Addition (BVA) [15] is a pre-processing technique that reduces the number of clauses in a formula by adding new variables. Each application of BVA first identifies a "grid" of clauses, as shown in Figure 1. Then, BVA adds a new variable and clauses which resolve together to generate all clauses in the grid.
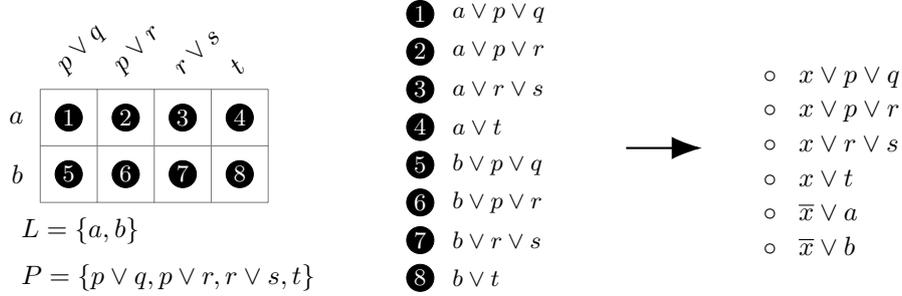
Figure 1 Bounded variable addition transforms groups of clauses (those that form a grid) by adding a new variable and eliminating a number of clauses.

Collectively, for a formula F, the grid constitutes a set of literals L and a set of partial clauses P, such that $\forall l \in L, \forall C \in P : (l \vee C) \in F$. While bounded-variable elimination eliminates variables by replacing all the clauses containing a variable with their resolvents, BVA tries to identify grids of *resolvents* which can be generated by the introduction of a new variable and a smaller number of clauses. These grids of clauses capture the fact that either the entirety of L must be satisfied, or the entirety of P must be satisfied. More precisely, $F \implies (\bigwedge_{l \in L} l) \vee (\bigwedge_{C \in P} C)$. By identifying these grids, BVA replaces $|L| \cdot |P|$ clauses with a single, new variable x and $|L| + |P|$ clauses (which can generate the original set by resolution on x in $\{x \vee C \mid C \in P\} \times \{\overline{x} \vee l \mid l \in L\}$). Therefore, if $|L| \cdot |P| > |L| + |P| + 1$, then this replacement results in a reduction in formula size.

Note that a BVA replacement step can be simulated by extended resolution: First, add the definition $x \leftrightarrow \text{AND}(L)$. In the example above, this means adding the clauses $x \vee \overline{a} \vee \overline{b}$, $\overline{x} \vee a$, and $\overline{x} \vee b$. Afterwards, the clauses $x \vee p \vee q$, $x \vee p \vee r$, $x \vee r \vee s$, and $x \vee t$ can each be derived using $|L|$ resolution steps. For example, to derive $x \vee p \vee q$, resolve $x \vee \overline{a} \vee \overline{b}$ with $a \vee p \vee q$ and the result with $b \vee p \vee q$. Afterward the clauses used in these resolution steps can be deleted.

The SIMPLEBOUNDEDVARIABLEADDITION **algorithm.** Manthey et al. [15] propose a greedy algorithm to identify these grids of resolvents that prioritizes literals which appear in many clauses called SIMPLEBOUNDEDVARIABLEADDITION. An abbreviated version of a single variable addition in this algorithm is shown in Algorithm 1.

Each identified grid starts from the most frequently occurring literal l in the current formula. The grid starts with dimension $1 \times |F_l|$, where F_l is the set of clauses containing l. From there, the algorithm searches for a literal $l_{\max}$ to add to the grid, which maximizes the number of remaining resolvents.

To identify the literal $l_{\max}$, the BVA algorithm looks for the literal for which $(l_{\max} \vee C)$ appears in F for the greatest number of clauses $C \in P$ (line 4). At each step, a literal is added to L (line 6), and clauses may be removed from P (line 7). The grid will continue to shrink until the addition of a literal to the grid would not increase the size of the formula reduction (line 5), as shown in Figure 2.

In BVA, variable additions are performed as long as there is a reduction in formula size.
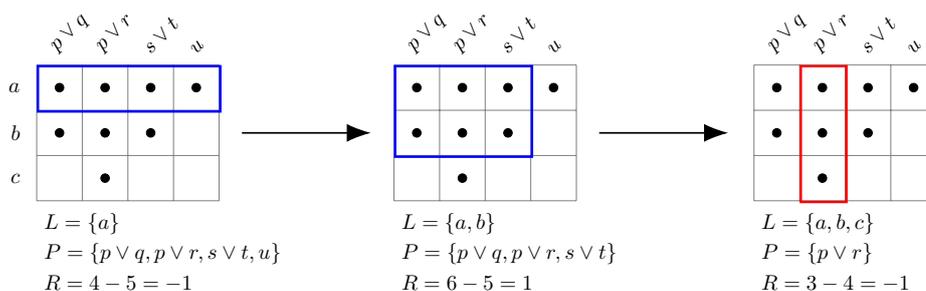
Algorithm 1 A single variable addition in SIMPLEBOUNDEDVARIABLEADDITION [15].

> PARTIALCLAUSES$(F, l) := \{C \setminus \{l\} \mid (C \in F) \wedge (l \in C)\}$
> $F :=$ the clauses in the current formula
> $l :=$ a literal in F

1: $L \leftarrow \{l\}$
2: $P \leftarrow$ PARTIALCLAUSES(F, l)
3: **while** True **do**
4:     $l_{\max} = \text{argmax}_{l_m \in \text{Lits}(F)} |P \cap \text{PARTIALCLAUSES}(F, l_m)|$ ▷ *Sensitive to tiebreaking*
5:     **if** adding $l_{\max}$ results in a greater reduction **then**
6:        $L \leftarrow L \cup \{l_{\max}\}$
7:        $P \leftarrow P \cap \text{PARTIALCLAUSES}(F, l_{\max})$
8:     **else**
9:        **break**
10: **if** $|L| \cdot |P| > |L| + |P| + 1$ **then** ▷ *If adding this variable would reduce the formula size*
11:     $S_{\text{add}} \leftarrow \{x \vee C \mid C \in P\} \cup \{\overline{x} \vee l_m \mid l_m \in L\}$ ▷ *Introduce a new variable x*
12:     $S_{\text{remove}} \leftarrow \{l_i \vee C \mid (l_i, C) \in L \times P\}$
13:     $F \leftarrow (F \setminus S_{\text{remove}}) \cup S_{\text{add}}$

$L = \{a\}$
$P = \{p \vee q, p \vee r, s \vee t, u\}$
$R = 4 - 5 = -1$

$L = \{a, b\}$
$P = \{p \vee q, p \vee r, s \vee t\}$
$R = 6 - 5 = 1$

$L = \{a, b, c\}$
$P = \{p \vee r\}$
$R = 3 - 4 = -1$

Figure 2 BVA adds variables to form a grid, until the reduction stops increasing. Here, the largest reduction was 1, and the variable corresponding to the middle grid will be added.

The entirety of Algorithm 1 is repeated using different literals for l to construct multiple new auxiliary variables. Specifically, the original implementation defines a priority queue of literals ordered by the number of clauses each literal appears in. Our adaptation of BVA (Section 4) reuses this implementation detail. These repeated applications of BVA enable the algorithm to achieve large reductions in formula size, and auxiliary variables added in previous steps can even be re-used in future variable introductions.

3 Motivating Example

To motivate the need for a heuristic-guided version of BVA, we will first demonstrate the effect of randomization on existing implementations of BVA, and the disproportionate impact of a few critical variable additions.

3.1 Packing Colorings

BVA has been shown to be effective on the grid packing k-coloring problems, whose constraints are based on coloring a circular grid of tiles, shown in Figure 3a. Unlike a standard graph coloring problem, each color in the packing k-coloring problem is associated with a integer

distance from 1 to k. When coloring the grid, two tiles can only have the same color if the taxicab distance between them is greater than the color number. For example, two tiles of color 3 must have at least 3 tiles between them, while color 1 tiles cannot be adjacent. The $D_{r,k}$ problem asks whether the grid of radius r can be colored with k colors.

The direct encoding for this problem consists of variables $v_{i,c}$, denoting that grid location i has color c. There are three types of clauses [22]:

1. At-Least-One-Color: $\forall i, (v_{i,1} \vee v_{i,2} \vee \cdots \vee v_{i,k})$. Each tile must be colored with a color between 1 and k.

2. At-Most-One-Distance: $\forall i, j, c : d(i, j) \leq c, (\overline{v_{i,c}} \vee \overline{v_{j,c}})$. If the distance between two tiles is less than or equal to the color, they cannot both have that color.

3. Center-Clause: $v_{(0,0),c}$ for a fixed color c. This is a symmetry-breaking optimization [21], which has no effect on BVA since it ignores unit clauses.

Previous work [22] showed that BVA can reduce the size of such formulas by a factor of 4, and induces more than a $\times 4$ speedup on the larger instance ($D_{6,11}$). They found that auxiliary variables capture *regions* of grid tiles within a particular color, i.e. the grid replacement happens entirely within the binary at-most-one-distance constraints.

We visualize the variables introduced by BVA on $D_{5,10}$, the packing k-coloring problem with radius 5 and 10 colors. In the first row of Figure 3b, each of the four plots introduces a new auxiliary variable x for one of the colors $c \in \{1, \ldots, 10\}$ (denoted above each plot). All the binary clauses for color c (At-Most-One-Distance clauses) of the form $(\overline{v_{i,c}} \vee \overline{v_{j,c}})$ with grid location i corresponding to a green square and grid location j corresponding to a yellow square will be replaced with a smaller number of clauses: $(x \vee \overline{v_{i,c}})$ for each green location i and $(\overline{x} \vee \overline{v_{j,c}})$ for each yellow location j.

3.2   Negative Impact of Randomization

We discovered that randomizing packing k-coloring formulas prior to running BVA significantly increases the resulting solve time. Furthermore, the variables added by BVA after randomization fail to capture the clustered *regions* within the problem's 2D space that are
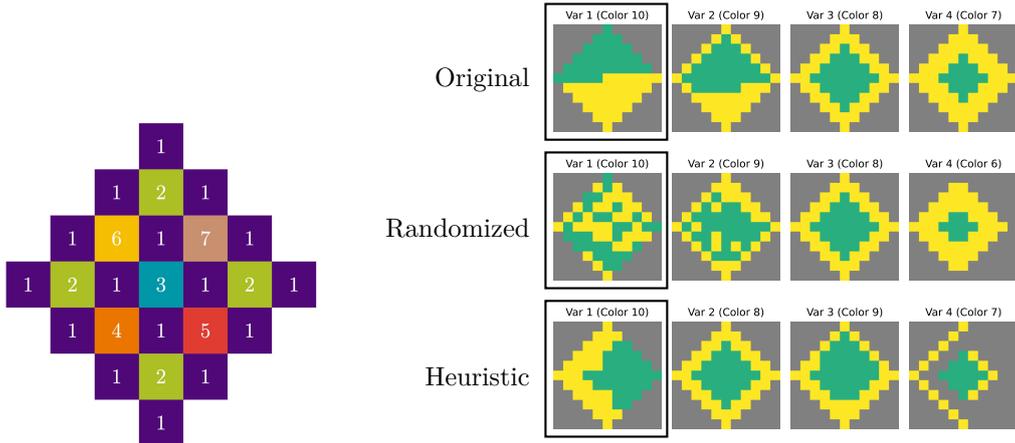
(a) Figure from [22] showing the $D_{3,5}$ grid packing k-coloring problem

(b) The effect of variable randomization on the first four BVA substitutions in $D_{5,10}$. The black boxes indicate the first variable addition, the effect of which is isolated in Table 1.

Figure 3 The auxiliary variables introduced by BVA on the packing k-coloring problem are sensitive to randomization.

■ **Table 1** CaDiCaL solve time for the $D_{5,10}$ packing problem, breaking BVA ties using the original variable order (sorted), randomized variable order (randomized), or the heuristic proposed in Section 4 (heuristic). Breaking ties differently has a *significant* effect on solve time even when the resulting formula is the same size (Single BVA Step).

	# Vars	# Clauses	Solve (s)
Original formula	610	10688	590.545
Single BVA Step (sorted)	611	9819	105.635
Single BVA Step (randomized)	611	9819	429.396
Single BVA Step (heuristic, **this paper**)	611	9819	213.018
Full BVA (sorted)	973	2313	38.749
Full BVA (randomized)	971	2305	107.675
Full BVA (heuristic, **this paper**)	972	2290	55.482

identified without randomization. Figure 3b shows the first four variable additions performed by BVA on $D_{5,10}$. The effect is especially noticeable in the first few variable additions. The structure of these variables is more than a visual artifact. Running BVA to completion produces a formula that requires more than $\times 2$ the solve time in CaDiCaL compared with running BVA on the original formula, despite a similar reduction in formula size (see Table 1).

We found that the *first* variable added by BVA in $D_{5,10}$ had a disproportionate impact on the solve time of the formula. We isolated the effect of a single replacement by allowing BVA to only produce one new auxiliary variable and then evaluating the solve time of the resulting formula. Table 1 shows that a single variable addition (outlined in black in Figure 3b) can achieve a $\times 6$ speedup over the original formula and that the impact of this single addition is also substantially affected by randomization. Although randomization before BVA did not affect the size reduction of the first variable addition, the randomized formula with a single BVA step is 2 times slower compared to the original formula.

The importance of individual variable additions and their sensitivity to randomization suggests that BVA's impact is derived not only from the size reduction but from the *structure* of the variable additions.

3.3 Ties in Bounded Variable Addition

The reason for the BVA's sensitivity to randomization is due to a detail in the way implementations treat ties between literals. As described in Section 2, the algorithm chooses the literal that maximizes the number of remaining resolvents to be eliminated (Algorithm 1, line 4). If there is a tie between two literals, the original algorithm does not specify which literal should be used. The original implementation provided by [15] breaks ties using the variable number in the original formula. Figure 4 shows how breaking ties differently leads to different variable additions. Note that since BVA eliminates the clauses in the grid when adding a variable, it is *not* possible for multiple applications of BVA to eventually add both variables resulting from a tie.

In the $D_{5,10}$ packing problem, colors 9 and 10 are almost fully connected; coloring a tile with color 10 means that no tile within 10 spaces of it can also be colored 10. When BVA creates a variable for these pairwise constraints, all of the clauses are tied for the number of preserved resolvents (since every pair of color-10 variables appears in a at-most-one-distance clause). Since the original implementation used variable number to break ties
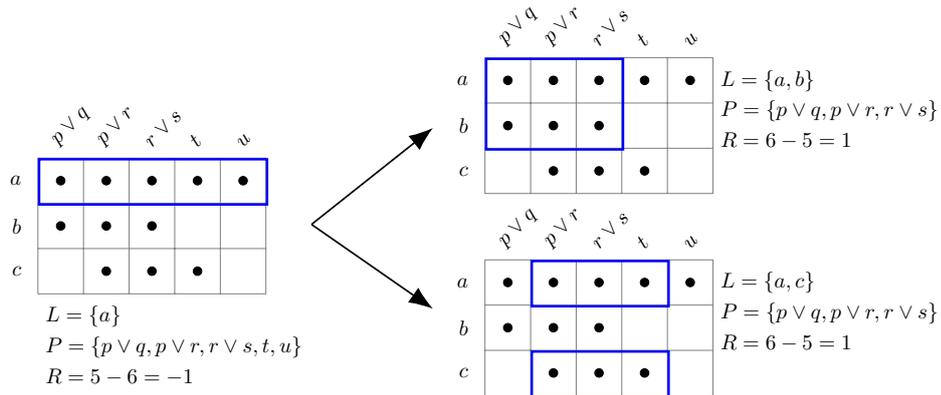
Figure 4 The addition of b or c both lead to a 2×3 grid of resolvents. Breaking this tie in different ways leads to different variable additions.

and ordered variables from top-left to bottom-right, the variable additions it produces follow that structured pattern. However, when the variable order is randomized, the resulting region lacks structure and the formula takes longer to solve.

3.4 Recovering Structure

After randomization, BVA struggles to introduce variables that represent coherent clusters of tiles. However, we note that the original structure is still captured by the original formula as a whole. For example, in the $D_{5,10}$ packing problem, two variables representing color 1, $v_{i,1}$ and $v_{j,1}$, only share a pairwise constraint if they are adjacent (i.e. if i and j represent adjacent tiles). If we could recover a generic metric for how *close* variables are to each other (e.g. in the 2D space of $D_{5,10}$), this metric could be used to help BVA recover structure in problems where the original variable order does not result in structured variable additions.

The intuition for our heuristic, which is detailed in Section 4, is based on the structure observed in the packing problem. We notice that while variables in color 10 are indistinguishable after randomization (i.e. all fully connected with At-Most-One-Distance clauses), the variables in color 1 preserve the structure of the original problem: these variables only share At-Most-One-Distance clauses with their immediate neighbors. Additionally, variables for the same *tile location* but different *colors* are all linked by an At-Least-One-Color constraint, even after randomization. One could deduce which variables in color 10 are neighbors by looking at the connectivity of the equivalent tile positions in color 1. Specifically this requires 3 "hops" through clauses: starting at a variable $v_{i,10}$ in color 10, we find $v_{i,1}$ in color 1 (via an At-Least-One-Color clause), then find $v_{j,1}$ in color 1 (via an At-Most-One-Distance clause), and finally find $v_{j,10}$ in color 10 (via an At-Least-One-Color clause); the full path is $v_{i,10} \rightarrow v_{i,1} \rightarrow v_{j,1} \rightarrow v_{j,10}$.

While it is possible to construct an algorithm to recover this structure specifically for the k-coloring packing problem, we generalize this concept by *counting* paths. Specifically, between two variables $v_{i,10}$ and $v_{j,10}$ in color 10 there are *many* paths of length 3: for example $v_{i,10} \rightarrow v_{a,10} \rightarrow v_{b,10} \rightarrow v_{j,10}$ (using only At-Most-One-Distance clauses). However, only *adjacent* variables in color 10 will have the additional path that goes through color 1: $v_{i,10} \rightarrow v_{i,1} \rightarrow v_{j,1} \rightarrow v_{j,10}$. For a given variable in color 10, it will have the most 3-hop paths to variables of the immediately adjacent grid tiles. We formalize this intuition in Section 4.

4   Structured Reencoding

In this section we define our implementation of a heuristic for breaking ties during variable
selection in BVA. While our heuristic was initially designed to mitigate the detrimental
effects of randomization on the packing coloring problems, we found that it is also effective
for other problems, even ones which have not been randomized. In Section 5 we show that
our heuristic-guided BVA is effective on a wide variety of problems and offers a significant
improvement to solve time for certain families of formulas.

The 3-Hop Heuristic.   Our heuristic is based on the intuition that BVA should prefer to
break ties by adding variables that are *close* to one another. In Subsection 3.4, we noticed
that in the k-coloring problem, there are some paths between variables that are only present
when variables are *close* in the problem's 2-D space. The *variable incidence graph* compactly
captures this notion of variable adjacency. Here we formally define a heuristic for "variable
distance" based on the number of *paths* between pairs of variables in the *variable incidence
graph*.

▶ **Definition 1.** *The Variable Incidence Graph (VIG) of a formula F is an undirected graph
$G = (V, E)$ where V is the set of variables in F, and E contains an edge between variables if
they appear in a clause together. The weight on an edge (v_1, v_2) is the number of clauses in
which v_1 and v_2 appear together: $w(v_1, v_2) = |\{C \in F : \{v_1, v_2\} \subset \text{Vars}(C)\}|$*

We measure variable distance by counting the number of distinct paths between two
variables (i.e. using different intermediate variables or clauses). Edges in the VIG indicate
the number of clauses shared by pairs of variables. For a given sequence of variables
$(v_1, v_2, ..., v_n)$ the number of distinct paths through different combinations of clauses is given
by $w(v_1, v_2) \cdot w(v_2, v_3) \cdot ... \cdot w(v_{n-1}, v_n)$. Since edge weights are multiplicative along a path,
the number of different paths of length n through the VIG is given by A^n, where A is the
adjacency matrix of the VIG. Since we identified that adjacent tiles in the packing problem
have more length-3 paths between them, we define a simple heuristic that counts the number
of paths of length 3 in the VIG, which we call the *3-hop heuristic*.

▶ **Definition 2.** *The 3-hop heuristic $H(x, y)$ is defined as the number of distinct paths of
length 3 between two variables x and y in the VIG. Two paths are distinct if they travel
through a different sequence of variables or clauses. Given the VIG adjacency matrix A, the
3-hop heuristic can be computed as $H(x, y) = (A^3)_{x,y}$.*

We modify Algorithm 1 to use our heuristic as a tie-breaker, specifically augmenting
the computation of argmax in line 4: when multiple values of l_m have the same number
of remaining resolvents, we choose the literal l_m with the highest value of $H(l, l_m)$. Our
implementation of BVA, called SBVA, is written in C++ and uses the `Eigen` library for
sparse matrix operations. It is capable of generating DRAT proofs describing the sequence
of variable additions and clause deletions and thus could be used with a solver to generate
certificates of unsatisfiability.

In Figure 5, we show the value of $H(x, y)$ in $D_{5,10}$ for variables representing color 10
between a variable of interest (outlined in black) and all other variables of color 10. Grid tiles
that are closer in the 2-D space of the packing k-coloring problem have more 3-hop paths
between them and thus have a higher heuristic value. Using our heuristic on a randomized
formula for the packing problem, we recover variables that capture the spatial structure of
the problem. In the third row of Figure 3b, we show the first 4 variables added by SBVA,
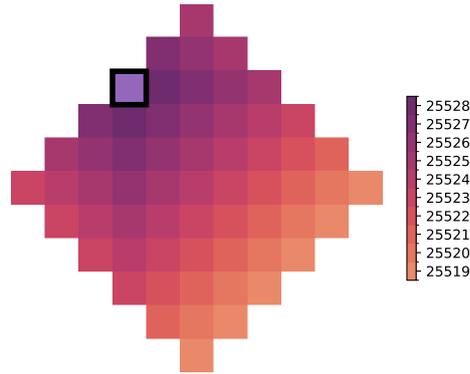
Figure 5 The value of the 3-hop heuristic in $D_{5,10}$ between the color-10 variable for the location outlined in black and all other color-10 locations.

which cluster variables together using the notion of distance that is inherent in the original problem. Furthermore, we find that applying this heuristic to the packing problem results in formulas that solve much faster than BVA on a randomized formula (Table 1).

5 Experimental Details

We evaluated BVA on more than 29,000 formulas from the Global Benchmark Database [12] in order to study the effects of randomization and our heuristic on BVA. In this section, we discuss the experimental setup and provide a brief overview of the results. In Section 6, we analyze the results in more detail and discuss families of formulas that were significantly impacted by BVA and/or SBVA.

Configurations.   We constructed three solver configurations that use BVA in different ways. All three variants take a formula, (optionally) randomize it with `scranfilize`, run BVA (with or without heuristic), and pass it to CaDiCaL to solve. For comparison, we include a baseline variant that does not run BVA. Since the particular ordering of clauses and variables in a formula can impact solver performance [3], we also use the `scranfilize` tool immediately prior to running CaDiCaL in all configurations. To mitigate this variance, we run the entire sequence three times for each configuration, averaging across the three runs. The list of configurations is shown in Table 2. Note that all four configurations have randomization applied prior to solving with CaDiCaL but only BVA-RAND-ORIG and BVA-RAND-3HOP have randomization applied prior to BVA/SBVA.

Table 2 Experimental configurations. *Pre* and *Post* refer to arguments passed to `scranfilize` before and after running the preprocessor respectively. An empty space indicates the step was skipped for this variant.

Variant	Pre	Preprocessor	Post	Solver
BASELINE			`-p -P -f 0.5`	CaDiCaL
BVA-ORIG		BVA	`-p -P -f 0.5`	CaDiCaL
BVA-RAND-ORIG	`-p -P -f 0.5`	BVA	`-p -P -f 0.5`	CaDiCaL
BVA-RAND-3HOP	`-p -P -f 0.5`	SBVA	`-p -P -f 0.5`	CaDiCaL

Table 3 PAR-2 scores and number of formulas solved for each variant split by problem type (ALL/UNSAT/SAT) and dataset (FULL/ANNI-2022). Bold cells indicate the lowest PAR-2 score or highest number solved for that group.

Dataset	Variant	ALL PAR-2	#	UNSAT PAR-2	#	SAT PAR-2	#
FULL	BASELINE	1077.91	21602	756.14	6495	1196.99	15107
	BVA-ORIG	867.04	22140	**635.71**	6562	948.85	15578
	BVA-RAND-ORIG	870.20	22077	673.58	6533	953.25	15544
	BVA-RAND-3HOP	**862.29**	**22173**	650.41	**6568**	**935.38**	**15605**
ANNI-2022	BASELINE	1262.18	3953	1164.61	2048	**1309.41**	1905
	BVA-ORIG	**1174.80**	3987	**967.85**	2085	1338.31	1902
	BVA-RAND-ORIG	1193.27	3958	1053.75	2060	1350.09	1898
	BVA-RAND-3HOP	1188.63	**3995**	982.84	**2088**	1350.98	**1907**

Benchmarks.   We evaluated our variants on 29 402 benchmark instances (downloaded on February 20, 2023) from the Global Benchmark Database (GBD) [12]. We also report results against the Anniversary Track from the SAT Competition 2022 [1] (labeled as "ANNI-2022" within this paper) which is included as a subset in the GBD (5355 benchmarks).

Hardware.   All experiments were performed on the Bridges-2 system at the Pittsburgh Supercomputing Center [7] on nodes with 128 cores and 256 GB RAM.

Experimental Setup.   We compare the four configurations in a simulated competition setting with a fixed time limit of 5 000 seconds per benchmark. The total time is computed as the sum of BVA and CaDiCaL runtimes (scranfilize time is not counted towards this limit). As noted by Manthey et al. [15], BVA can be quite expensive, even on formulas that do not reduce significantly. We allow all versions of BVA to run for 200 seconds and if it has not terminated by then, we instead run the original formula with CaDiCaL. On our full benchmark, BVA terminates within 200 seconds on approximately 95% of problems. We ran 128 instances in parallel per node, leaving approximately 2GB of memory (for reference, in the SAT Competition 2022 [1], solvers were allotted 128GB) for each BVA/CaDiCaL process. This limit is enough for most formulas, but in cases where BVA runs out of memory, we instead run the original formula in CaDiCaL. In both cases (timeout and out-of-memory), the *already-used* time is added to the subsequent solve time of the original formula. This setup provides a fair comparison as BVA could be realistically configured this way in a competition.

We report the PAR-2 scores and number of formulas solved for each variant in Table 3. The PAR-2 score is computed as the total time it took to solve an instance (BVA runtime + CaDiCaL runtime) or twice the time limit if the formula was not solved within 5 000 seconds. We compute the PAR-2 score individually for each run and average across the three runs of a given formula. A formula is marked as solved in Table 3 if any of the three runs solved it within the time limit. Additionally, the set of formulas over which PAR-2 is computed consists of instances where at least one of the four configurations was able to solve it. Instances that were not solved by any configuration were not included in the PAR-2 score. Adding these entirely unsolved instances would not change the number solved and would simply scale the PAR-2 scores equally for all configurations.
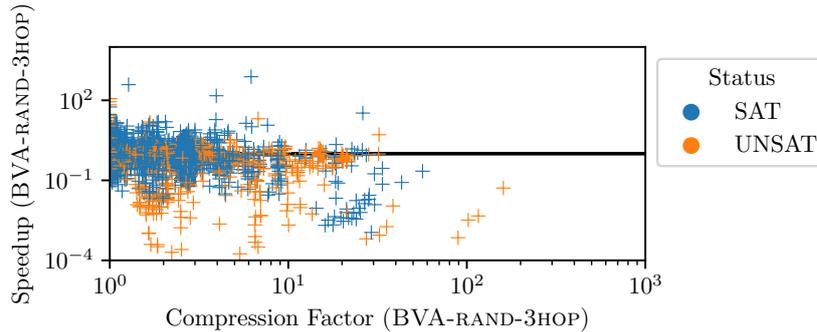
Figure 6 Formula speedup compared to compression factor for BVA-RAND-3HOP.

6 Results and Analysis

This section takes a closer look at the performance of BVA-ORIG, BVA-RAND-ORIG, and BVA-RAND-3HOP in comparison to the BASELINE configuration. We explore both the effects of randomization and the effects of the heuristic, in general and on specific families of formulas. Specifically, we explore the following questions:

Q1: Does compression factor correlate with solve time in the context of BVA?
Q2: What is the effect of randomization on the performance of BVA?
Q3: Can our heuristic outperform randomized BVA?
Q4: How does the performance of our heuristic vary across different families of formulas?

We address these questions directly in the following paragraphs:

A1: Formulas with larger compression factors tend to be solved faster, but this is not always the case. As demonstrated in Table 1, even small reductions from BVA can have a large impact on solve time. For example, on the packing k-coloring problem, a *single added variable* can reduce solve time by over a factor of 5 if picked correctly (Table 1).

We compute the *compression factor* of a formula as the ratio of the formula size *before* to the size *after* running BVA. For example, a factor of 1 indicates no reduction, a factor of 2 indicates the formula was reduced to 50% the original size, and a factor of 10 indicates the formula was reduced to 10% of the original size. Similarly, we compute the *speedup* as the ratio of solve time to BASELINE solve time (values below 1 indicate the formula was solved faster). In Figure 6 we plot the speedup of BVA-RAND-3HOP against the compression factor for every problem in the benchmark. Equivalent figures for BVA-ORIG and BVA-RAND-ORIG look similar and are available in the appendix (Figure A1 and Figure A2).

For formulas that could be greatly reduced, there is an observable trend towards a greater speedup. However, for small reductions, the speedup is much more variable. In some cases, even formulas that are reduced to less than 10% of the original size may be *slowed down* by BVA. With BVA-RAND-3HOP, 60% of formulas had a compression factor greater than 1, 40% had a factor greater than 2, and 4% had a factor greater than 10.

A2: Randomization is Detrimental to BVA. Randomization has a negative effect on the performance of BVA; in all benchmark groups, BVA-RAND-ORIG solved fewer formulas and has a higher PAR-2 score than BVA-ORIG. Interestingly, this effect appears to be entirely due to the *structure* of the resulting formula and not the resulting *size* of the formula.
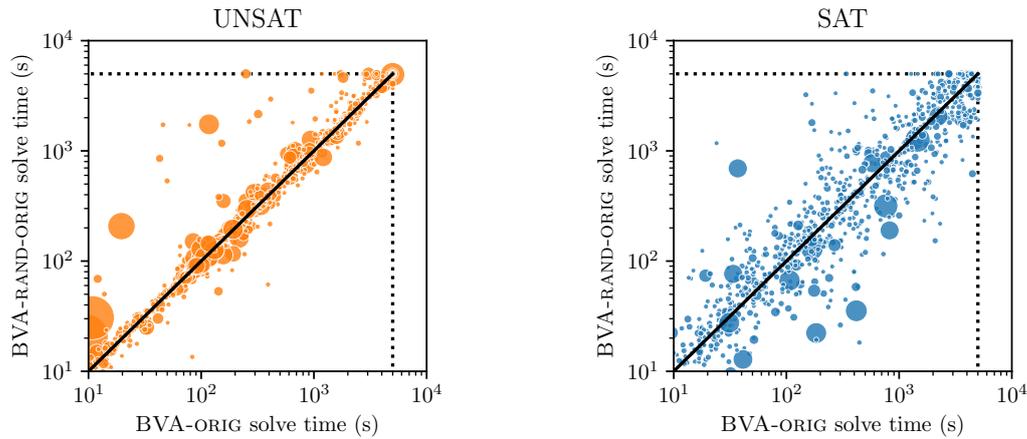
(a) Relative solve time for UNSAT formulas.
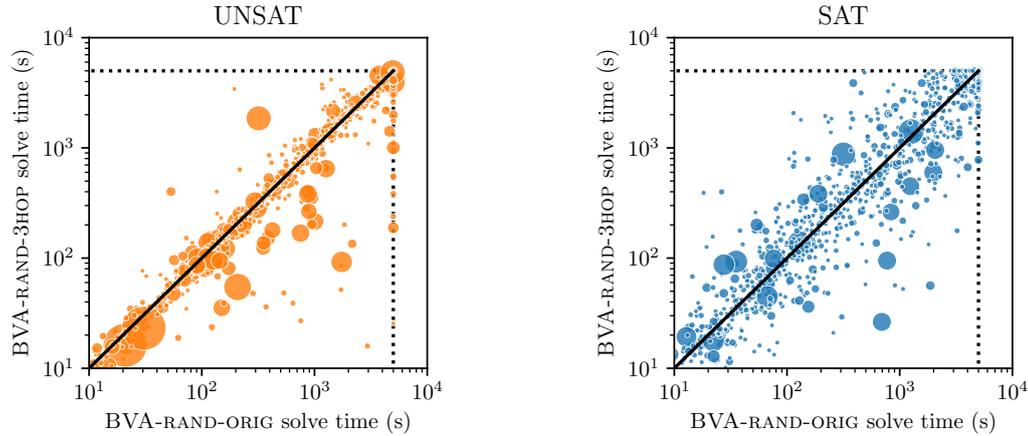
(b) Relative solve time for SAT formulas.

Figure 7 Difference in reduction size and solve time between BVA-RAND-ORIG and BVA-ORIG on formulas from ANNI-2022. Larger points indicate a more-reduced formula.

In Figure 7, we plot the relative solve times of formulas from the ANNI-2022 benchmark for BVA-RAND-ORIG and BVA-ORIG. While there is almost no difference in the reduction sizes of the formulas produced by BVA-RAND-ORIG and BVA-ORIG (formula sizes differ by less than 1.5%), a number of formulas were substantially slowed down (Figure 7). Note that in these plots, randomization prior to BVA is more detrimental for UNSAT formulas and introduces a lot of variance to SAT formulas.

A3: 3-Hop Heuristic is Robust to Randomization.   While randomization has a negative effect on the original implementation of BVA, we observe that our heuristic-guided BVA is robust to this effect. Despite being provided with randomized formulas, it is able to generate high quality variable additions and recover all of the performance loss of BVA-RAND-ORIG, even surpassing BVA-ORIG in many cases on number of problems solved and PAR-2 score. We believe the slight performance improvement over BVA-ORIG in several cases is due to the presence of "pre-randomized" formulas in the benchmark; in these cases BVA-ORIG already suffers the effects of randomization while BVA-RAND-3HOP is able to recover the original structure of the problem.

In Figure 8, we compare the relative solve times of formulas from the ANNI-2022 benchmark for BVA-RAND-3HOP and BVA-RAND-ORIG. As in the previous section, the formula sizes between the two variants differs by less than 1.5% on average. However, BVA-RAND-3HOP is able to speed up many formulas, especially UNSAT instances.

A4: SBVA performs similar to BVA in most cases and performs extremely well for a few families.   We found that both the original implementation of BVA and our heuristic-guided version have strong effects for specific families of formulas. In Figure 9, we plot the relative performance of the four configurations on 10 formula families for which BVA was effective. For these plots we allow BVA/SBVA to run for the full 5 000 seconds and consider only the CaDiCaL solve time in the plots in order to understand the effectiveness of the formula rather than the speed of BVA. In this section, we briefly describe some of the families where BVA was most effective.

(a) Relative solve time for UNSAT formulas.

(b) Relative solve time for SAT formulas.

Figure 8 Difference in reduction size and solve time between BVA-ʀᴀɴᴅ-3ʜᴏᴘ and BVA-ʀᴀɴᴅ-ᴏʀɪɢ on formulas from ANNI-2022. Larger points indicate a more-reduced formula.

Figure 9 Performance of BVA/SBVA on 10 families of formulas where it was effective.

Pigeonhole / PHNF / FPGA-Routing. Pigeonhole formulas try to uniquely assign n pigeons to m holes. Like the packing k-coloring problem, these formulas consist primarily of AtLeastOne constraints (a pigeon must be in at least one hole) and pairwise AtMostOne constraints (two pigeons cannot share a hole). Our benchmark also contains variants of this problem, e.g. allowing multiple pigeons in a hole. These formulas are difficult for SAT solvers due to the number of possible permutations.

We found that SBVA was quite effective for UNSAT instances of pigeonhole problems (note that SAT instances of pigeonhole problems are trivial), able to solve new instances that the other three configurations could not solve. Interestingly, we found that these newly solved problems consist mainly of *pre-shuffled* pigeonhole problems. A full list of solved UNSAT pigeonhole problems is provided in Table A1. Other pigeonhole-like families in the dataset include PHNF (Pigeonhole Normal Form) [19] and FPGA-Routing [17], which consists of problems generated by combining two pigeonhole problems. All forms of BVA were very effective on these problems compared to Bᴀsᴇʟɪɴᴇ.

Petri Net Concurrency. Petri nets are a model of concurrent computation that consists of places and transitions [6]. They are used to model a variety of systems, including chemical reactions, manufacturing processes, and computer programs. The Petri Net Concurrency family consists of formulas that encode the satisfiability of Petri nets. All three configurations of BVA are able to generate very compact encodings for these formulas, with an average compression factor of more than 20.

Bioinformatics. The bioinformatics family consists of problems that encode genetic evolutionary tree computations into SAT [5]. As noted by the authors of the original BVA paper, these problems are also reduced significantly with BVA. For the problems in this family, we found that the average compression factor was more than 7 for all three BVA configurations, i.e. the formulas were reduced to less than 15% total size on average.

Puzzle / Rooks / Battleship. We found BVA to be useful in several families of formulas derived from 2-D games. The puzzle family consists of formulas that encode the satisfiability of a sliding-block puzzle and were contributed by van der Grinten to SAT Comp 2017. The rooks family asks if it is possible to place $N + 1$ rooks on a $N \times N$ chessboard such that no two rooks can attack each other [16]. The battleship family consists of problems that are derived from the battleship guessing game and were contributed by Skvortsov to SAT Comp 2011. BVA was effective in all three families and SBVA was especially effective for the puzzle and battleship families.

Antibandwidth / Spectrum Repacking. The antibandwidth [10] and spectrum repacking [18] formulas are both related to assigning radio stations to channels. Specifically, the antibandwidth family asks if it possible to assign a given set of stations to a given set of channels such that the difference in channel between any two stations is at least k. Similarly, the spectrum repacking family asks if it is possible to reassign a given set of stations into a smaller set of channels, taking into account physical distances between stations and the bandwidth of each channel. All configurations of BVA were effective on these problems.

7 Conclusion

Bounded Variable Addition is surprisingly effective at reducing the size of formulas and improving solve time by introducing auxiliary variables. We discovered that this speedup is caused not only by the reduction in formula size but also the introduction of certain *effective* auxiliary variables. We found that the original implementation was sensitive to randomization and proposed a new heuristic-guided implementation, SBVA, that is robust to this effect. In a competition-style benchmark, we show that using SBVA resulted in the most formulas solved in every category, outperforming both BVA and the baseline (no preprocessor). Additionally, SBVA was extremely effective on certain families of formulas, demonstrating that auxiliary variables can be useful in practice if they are chosen carefully.

──── **References** ────

1   Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions.* Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022.

2   A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, NLD, 2009.

3   Armin Biere and Marijn Heule. The effect of scrambling CNFs. In *Proceedings of Pragmatics of SAT*, volume 59, pages 111–126, 2019.

4   Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. *Handbook of Satisfiability*, 336:391–435, 2021.

5   Maria Luisa Bonet and Katherine St John. Efficiently calculating evolutionary tree measures using SAT. In *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings 12*, pages 4–17. Springer, 2009.

6   Pierre Bouvier and Hubert Garavel. Efficient algorithms for three reachability problems in safe petri nets. In Didier Buchs and Josep Carmona, editors, *Application and Theory of Petri Nets and Concurrency*, pages 339–359, Cham, 2021. Springer International Publishing.

7   Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing*, PEARC '21, New York, NY, USA, 2021. Association for Computing Machinery.

8   Stephen A Cook. A short proof of the pigeon hole principle using extended resolution. *Acm Sigact News*, 8(4):28–32, 1976.

9   Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. *SAT*, 3569:61–75, 2005.

10   Katalin Fazekas, Markus Sinnl, Armin Biere, and Sophie Parragh. Duplex encoding of staircase at-most-one constraints for the antibandwidth problem. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 186–204. Springer, 2020.

11   Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

12   Markus Iser and Carsten Sinz. A problem meta-data library for research in SAT. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 144–152. EasyChair, 2019.

13   W. Klieber and G. Kwon. Efficient CNF encoding for selecting 1 from n objects. In *Fourth Workshop on Constraints in Formal Verification (CFV)*, 2007.

14   Petr Kučera, Petr Savickỳ, and Vojtěch Vorel. A lower bound on CNF encodings of the at-most-one constraint. *Theoretical Computer Science*, 762:51–73, 2019.

15   Norbert Manthey, Marijn JH Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Haifa Verification Conference*, pages 102–117. Springer, 2012.

16   Norbert Manthey and Peter Steinke. Too many rooks. *Proceedings of SAT competition*, pages 97–98, 2014.

17   Gi-Joon Nam, Fadi Aloul, Karem Sakallah, and Rob Rutenbar. A comparative study of two boolean formulations of FPGA detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design*, pages 222–227, 2001.

18   Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2017.

19   O. Roussel. Another SAT to CSP conversion. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pages 558–565, 2004.

20   Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

21   Bernardo Subercaseaux and Marijn JH Heule. The packing chromatic number of the infinite square grid is at least 14. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

22   Bernardo Subercaseaux and Marijn JH Heule. The packing chromatic number of the infinite square grid is 15. *arXiv preprint*, 2023. `arXiv:2301.09757`.

23   Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Mathematics and Mathematical Logic 2*, pages 115–125, 1968.

A   Appendix

A.1   Reduction Size vs. Solve Time
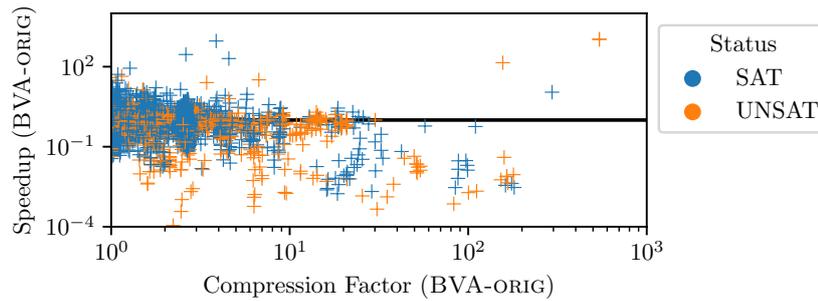
Figure A1 Formula speedup compared to compression factor for BVA-ORIG.
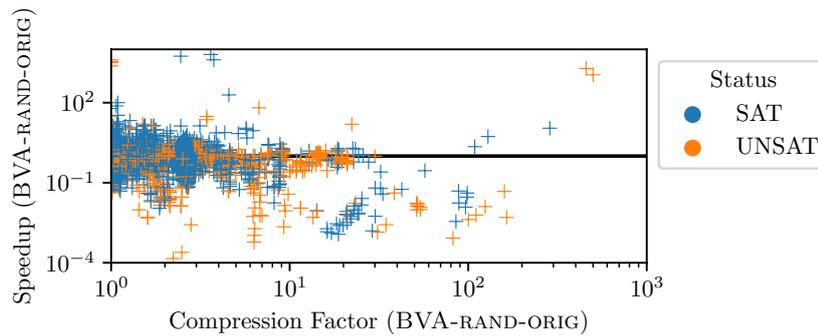
Figure A2 Formula speedup compared to compression factor for BVA-RAND-ORIG.

A.2   Performance on Pigeonhole Problems

■ **Table A1** Performance on unsatisfiable instances of problems in the pigeon-hole family.

Instance (unsatisfiable)	Solve Time (s)			
	BASELINE	BVA-ORIG	BVA-RAND-ORIG	BVA-RAND-3HOP
a_rphp035_05	499.77	478.28	**327.31**	328.82
a_rphp045_05	2165.96	2069.67	**1748.25**	1952.39
a_rphp055_04	79.75	81.91	79.09	**75.39**
a_rphp065_04	168.98	164.66	**137.85**	146.82
a_rphp085_04	760.00	752.18	707.73	**661.66**
a_rphp098_04	**1945.51**	2136.84	2726.63	2318.10
ae_rphp035_05	410.54	501.24	426.94	**407.24**
ae_rphp045_05	2402.73	2549.29	**2206.80**	2307.20
ae_rphp055_04	83.73	86.48	81.64	**74.17**
ae_rphp075_04	513.63	515.97	558.53	**504.86**
ae_rphp095_04-sc2018	1937.08	1686.12	2074.11	**1678.99**
ae_rphp095_04	1706.39	1611.31	**1560.96**	1572.97
clqcolor-08-06-07.shuffled-as.sat05-1257	3.28	**0.62**	0.91	0.81
counting-clqcolor-unsat-set-b-clqcolor-08-06-07.sat05-1257.reshuffled-07	2.69	0.83	**0.76**	1.02
counting-easier-fphp-012-010.sat05-1214.reshuffled-07	111.76	33.26	30.03	**0.11**
counting-easier-fphp-014-012.sat05-1215.reshuffled-07	**T.O.**	**T.O.**	**T.O.**	**1.62**
counting-easier-php-012-010.sat05-1172.reshuffled-07	139.59	27.07	29.35	**3.45**
counting-easier-php-018-014.sat05-1175.reshuffled-07	**T.O.**	**T.O.**	**T.O.**	**4224.30**
counting-harder-php-014-013.sat05-1187.reshuffled-07	**T.O.**	**T.O.**	**T.O.**	**1760.18**
e_rphp035_05-sc2018	424.51	460.54	461.25	**387.80**
e_rphp035_05	482.43	480.30	**419.09**	501.65
e_rphp055_04	**73.62**	81.31	77.03	78.85
e_rphp065_04	139.17	**125.17**	143.14	144.34
e_rphp096_04	1626.72	1830.11	**1492.40**	1503.88
easier-fphp-020-015.sat05-1218.reshuffled-07	**T.O.**	**T.O.**	**T.O.**	**4205.14**
fphp-010-008.shuffled-as.sat05-1213	0.49	0.22	0.22	**0.02**
fphp-010-009.shuffled-as.sat05-1227	5.43	4.30	4.22	**0.06**
fphp-012-010.shuffled-as.sat05-1214	113.38	45.04	36.16	**0.12**
fphp-012-011.shuffled-as.sat05-1228	1722.86	1525.19	1844.10	**0.68**
fphp-014-012.shuffled-as.sat05-1215	**T.O.**	**T.O.**	**T.O.**	**1.95**
fphp-014-013.shuffled-as.sat05-1229	**T.O.**	**T.O.**	**T.O.**	**564.50**
fphp-016-013.shuffled-as.sat05-1216	**T.O.**	**T.O.**	**T.O.**	**383.94**
fphp-016-015.shuffled-as.sat05-1230	**T.O.**	**T.O.**	**T.O.**	**1027.46**
fphp-018-014.shuffled-as.sat05-1217	**T.O.**	**T.O.**	**T.O.**	**549.07**
fphp-020-015.shuffled-as.sat05-1218	**T.O.**	**T.O.**	**T.O.**	**944.65**
harder-fphp-016-015.sat05-1230.reshuffled-07	**T.O.**	**T.O.**	**T.O.**	**3496.64**
hole10.cnf.mis-98.debugged	2.20	**0.95**	1.51	1.05
ph9	5.68	1.55	4.02	**0.08**
ph10	120.63	13.83	50.17	**8.64**
ph11	3061.40	35.76	790.89	**26.45**
php-010-008.shuffled-as.sat05-1171	0.64	0.15	0.25	**0.03**
php-010-009.shuffled-as.sat05-1185	6.59	3.03	3.58	**0.07**
php-012-010.shuffled-as.sat05-1172	138.30	31.52	23.96	**3.02**
php-012-011.shuffled-as.sat05-1186	2695.99	1006.29	755.79	**26.97**
php-014-012.shuffled-as.sat05-1173	**T.O.**	**T.O.**	**T.O.**	**237.47**
php-016-013.shuffled-as.sat05-1174	**T.O.**	**T.O.**	**T.O.**	**3024.16**
php11e11	3599.98	**500.38**	780.09	796.55
rphp4_065_shuffled	146.66	148.59	**129.96**	132.01
rphp4_070_shuffled	**213.59**	249.18	280.29	227.72
rphp4_075_shuffled	448.36	459.23	**415.65**	419.72
rphp4_080_shuffled	532.00	519.81	516.74	**503.20**
rphp4_085_shuffled	666.98	723.74	654.52	**648.68**
rphp4_090_shuffled	853.68	**850.60**	919.62	890.40
rphp4_095_shuffled	1571.45	1362.84	1611.21	**1342.79**
rphp4_100_shuffled	2314.76	2545.95	2361.29	**2154.53**
rphp4_105_shuffled	3168.06	2948.21	2520.33	**2249.50**
rphp4_110_shuffled	3726.95	3389.56	**3208.10**	3665.63
rphp4_115_shuffled	4155.81	4406.08	4169.75	**4095.25**
rphp4_120_shuffled	**T.O.**	**4840.06**	4883.44	4948.49
rphp4_125_shuffled	**T.O.**	4613.90	4676.45	**3989.60**
rphp_p6_r28	**T.O.**	**T.O.**	**4895.39**	**T.O.**
tph6	226.78	12.06	68.97	**0.52**
tph7	**T.O.**	249.13	**T.O.**	**0.88**
tph8	**T.O.**	**T.O.**	**T.O.**	**188.30**

A.3   Performance on Bioinformatics Problems

■ **Table A2** Performance on unsatisfiable instances of problems in the bioinformatics family.

Instance (unsatisfiable)	Solve Time (s)			
	Baseline	BVA-orig	BVA-rand-orig	BVA-rand-3hop
ndhf_xits_09_UNSAT	**T.O.**	**9.69**	11.10	10.52
ndhf_xits_10_UNSAT	**T.O.**	70.30	70.43	**63.04**
ndhf_xits_11_UNSAT	**T.O.**	612.65	869.25	**401.50**
ndhf_xits_12_UNSAT	**T.O.**	**T.O.**	**T.O.**	**1003.12**
rbcl_xits_06_UNSAT	5.27	0.20	0.22	**0.20**
rbcl_xits_07_UNSAT	110.14	0.60	0.60	**0.47**
rbcl_xits_08_UNSAT	3603.02	2.09	2.15	**1.72**
rbcl_xits_09_UNKNOWN	**T.O.**	**8.11**	9.55	10.69
rbcl_xits_10_UNKNOWN	**T.O.**	69.24	**56.82**	95.77
rbcl_xits_11_UNKNOWN-sc2009	**T.O.**	311.98	**309.87**	505.25
rbcl_xits_11_UNKNOWN	**T.O.**	**331.32**	392.19	528.49
rbcl_xits_12_UNKNOWN	**T.O.**	**3607.56**	4857.58	**T.O.**
rpoc_xits_07_UNSAT	54.86	**0.95**	1.03	0.97
rpoc_xits_09_UNSAT	**T.O.**	41.37	30.13	**24.29**
rpoc_xits_10_UNKNOWN	**T.O.**	237.25	**172.23**	182.65
rpoc_xits_11_UNKNOWN-sc2009	**T.O.**	3863.50	2369.91	**1672.29**
rpoc_xits_11_UNKNOWN	**T.O.**	1813.61	4624.68	**1413.96**

■ **Table A3** Performance on satisfiable instances of problems in the bioinformatics family.

Instance (unsatisfiable)	Solve Time (s)			
	Baseline	BVA-orig	BVA-rand-orig	BVA-rand-3hop
ndhf_xits_19_UNKNOWN-sc2011	143.44	31.96	**9.59**	13.37
ndhf_xits_20_SAT	29.44	**2.20**	3.32	3.38
ndhf_xits_21_SAT	6.27	2.16	**1.13**	2.27
ndhf_xits_22_SAT	3.09	1.11	**0.50**	0.86
rbcl_xits_14_SAT	1.31	0.47	**0.46**	1.91
rbcl_xits_18_SAT	0.22	0.04	0.05	**0.03**
rpoc_xits_17_SAT	1.22	0.14	0.11	**0.11**