Supplemental Material for “Understanding Conflict-Driven Clause Learning SAT Solvers Through the Lens of Proof Complexity”

February 1, 2018

Abstract

This document contains supplemental material for the submission *Understanding Conflict-Driven Clause Learning SAT Solvers Through the Lens of Proof Complexity* with:

1. detailed specifications of the benchmarks used;
2. detailed descriptions of the options implemented in the instrumented solver and which combinations were explored in the experiments;
3. some illustrations of the visual tools used in our analysis.

We describe the theoretical benchmarks used in our experiments in Section 1. In Section 2 we describe the implementation of our instrumented CDCL solver and report for which combinations of settings we ran our experiments. A brief discussion of the tools we have used to analyse the experimental results follows in Section 3.1.

In Appendix A we give a detailed specification of how the benchmarks are encoded into CNF.

1 Description of Theoretical Benchmarks

An overview of our combinatorial benchmark formulas is given in Table 1. In this table, we use the notation $n = X..Y..S$ to mean that the parameter n ranges between X and Y with step size S. Many, though not all, of the formulas have been generated using the tool *CNFgen* [CNF, LENV17]. The parameter ranges were chosen to ensure reasonable sizes of the CNF formulas, and so that in addition most formulas in each family can be solved with at least some combination of settings for the CDCL algorithm.

We want to emphasize that all of these instances have short resolution proofs that can in principle be found by CDCL without any preprocessing, and for most of the formulas even without any restarts given an appropriate (fixed) variable order. More specifically, all formulas in Table 1 except the relativized pigeonhole principle (RPHP) and dominating set formulas have resolution proofs in size linear in the number of clauses (with reasonably small multiplicative constants), and the RPHP and dominating set formula families has proofs of size scaling like n^k, which is still a reasonably small polynomial for the small, constant k that we consider. Thus, one way of viewing our experiments is to benchmark CDCL against resolution proofs: Can CDCL solvers produce proofs of unsatisfiability that are somewhat close to the short proofs that do exist, as the theoretical results in [AFT11, PD11] would suggest, or are there formulas that are very easy in theory but very hard in practice?

In what follows, we give a brief description of each family and provide pointers to references that discuss them in more detail. We remark that since all formulas are from the proof complexity literature, they are all unsatisfiable. In principle, one could try to generate satisfiable instances of these formulas by just removing

Table 1: Benchmark Characteristics

Benchmark Family	Scaling Parameter, n	Other Parameter	Version	Parameter Ranges	Filename	#Inst.	#Vars	Formula Size
Tseitin	#Columns	r: #Rows	Grids	$n = 5..200..5$, $r = 4$	tseitin_reggrid_<r>_<n>.cnf	40	$O(n)$	$O(n)$
				$n = 5..200..5$, $r = 5$		40		
				$n = 3..120..3$, $r = 6$		40		
			Hex. Grids	$n = 8..200..4$, $r = 5$	tseitin_hexgrid_<r>_<n>.cnf	49		
			Diag. Grids	$n = 3..120..3$, $r = 3$	tseitin_diaggrid_<r>_<n>.cnf	40		
Ordering Principle	Set Size		Partial	$n = 2..80..2$	op_partial_<n>.cnf	40	$O(n^2)$	$O(n^3)$
			Total	$n = 2..80..2$	op_total_<n>.cnf	40		
Pebbling	Pyramid Height		Pyr	$n = 4..180..4$	peb_pyr_neq3_<n>.cnf	45	$O(n^2)$	$O(n^2)$
			PyrOfPyr	$n = 1..15..1$	peb_pyrofpyr_neq3_<n>.cnf	15	$O(n^4)$	$O(n^4)$
Stone	#Vertices	m: #Stones	Chain-$m = \frac{n}{2}$	$n = 2..25..1$, $c = m$	stone_width3chain_nhalfmarkers_<n>.cnf	24	$O(nm + m)$	$O(m^3 n)$
		c:	Chain-$m = n$	$n = 2..18..1$, $c = m$	stone_width3chain_nmarkers_<n>.cnf	17		
		#Stones/node	Pyr-$m = n$	$n = 2..40..1$, $c = 15$	stone_pyr_15_sparse_<n>.cnf	39		
Subset Cardinality	Matrix Dimension		geq-$\{1248\}$	$n = 10..60..1$	sc_fixedbw_1248_geq_<n>.cnf	51	$O(n)$	$O(n)$
			eq-$\{1248\}$	$n = 10..60..1$	sc_fixedbw_1248_eq_<n>.cnf	51		
			geq-$\{1247\}$	$n = 10..60..1$	sc_fixedbw_1247_geq_<n>.cnf	51		
			eq-$\{1247\}$	$n = 10..60..1$	sc_fixedbw_1247_eq_<n>.cnf	51		
Even Colouring	#Columns	r: #Rows	Grids	$n = 5..250..5$, $r = 4$	ec_reggrid_<r>_<n>.cnf	50	$O(n)$	$O(n)$
				$n = 5..250..5$, $r = 5$		50		
				$n = 5..250..5$, $r = 6$		50		
			Diag. Grids	$n = 5..250..5$, $r = 3$	ec_diaggrid_<r>_<n>.cnf	50		
RPHP	#Resing Places	p: #Pigeons		$n = 5..131..3$, $p = 4$	rphp_<p>_<n>.cnf	43	$O(n^2)$	$O(n^2)$
				$n = 6..90..2$, $p = 5$		43		
Dominating Set	#Vertices	k: Size of D.S.		$n = 8..220..4$, $k = 3$	domset_<k>_<n>.cnf	54	$O(n)$	$O(n^2)$
				$n = 10..120..2 + 3$, $k = 4$		45		
Clique	#Vertices	k: Size of clique		$n = 12..420..8$, $k = 5$	partiteclique_<k>_<n>.cnf	52	$O(n)$	$O(n^2)$
				$n = 20..300..5$, $k = 6$		57		

one or a couple of randomly chosen clauses from each instance, but it is not clear whether this would yield interesting instances and we have not pursued this direction.

1.1 Tseitin Formulas

Tseitin formulas [Tse68] encode systems of linear equation over $\mathrm{GF}(2)$ (i.e., XOR constraints) generated from connected graphs $G = (V, E)$ with *charge function* $\chi : V \to \{0, 1\}$ chosen so that $\sum_{v \in V} \chi(v) \not\equiv 0 \pmod 2$. Every edge $e \in E$ corresponds to a variable x_e, and for every vertex $v \in V$ there is an equation $\sum_{e \ni v} x_e \equiv \chi(v) \pmod 2$ encoded in CNF. A simple counting argument shows that all equations cannot be satisified simultaneously, since every edge is counted exactly twice but the sum of all charges is odd. When G has bounded degree and is well-connected, the formula is exponentially hard for resolution [Urq87].

We study Tseitin formulas on long, narrow grid graphs, where every vertex has edges horizontally and vertically to its 4 neighbours, and where edges "wrap around" so that the topmost and bottommost rows are connected, as are the rightmost and leftmost columns (more formally, these are rectangular toroidal grids). We also consider slight variations of these grid graphs, which instead have diagonal edges or form hexagonal grids. For the right settings of parameters, Tseitin formulas over families of similar graphs have been proven to exhibit strong size-space trade-offs for resolution [BBI16, BNT13]. These parameter settings are not appropriate for practical experiments, however, since the formulas are infeasible for CDCL solvers in practice. Instead, we fix the number of rows r to a small constant and then vary the number of columns n to scale the size of the instances. Formula sizes and number of variables in this family scale linearly with n, as does the size of minimal resolution proofs. Even tree-like resolution, corresponding to DPLL without clause learning, can refute these formulas efficiently, although at the cost of a small polynomial blow-up in the proof size depending on r. By fixing the parameters in this way we ensure that the formulas do *not* exhibit theoretical time-space trade-offs, but our intuition is that these formulas might be quite sensitive to memory management anyway, since they are "morally close" to the formulas in [BBI16, BNT13].

1.2 Ordering Principle Formulas

Ordering principle formulas claim that there is a finite set $\{e_1, \ldots, e_n\}$ with an ordering $\preceq$ such that no element e_j is minimal with respect to this ordering, where variables $x_{i,j}$, $i \neq j \in [n]$, encode whether $e_i \preceq e_j$ or not. We study two variants of formulas in this family encoding that the ordering is partial and total, respectively.

These formulas have an interesting history in that they were conjectured to be exponentially hard for resolution [Kri85] but were later shown to have proofs of size linear in the formula size [Stå96]. Ordering formulas have clauses of size $n - 1$, but if they are converted to 3-CNF in some appropriate way, they can be shown to require large width [BG01]. Combining this with the size-width lower bounds in [BW01], it follows that ordering principle formulas are exponentially hard for tree-like resolution and DPLL. (One can also avoid this conversion by instead studying the asymmetric width measure in [Kul99], which is meaningful also for CNF formulas of unbounded width, and use that ordering principle formulas can be shown to require large asymmetric width.)

1.3 Pebbling Formulas

Pebbling formulas [BW01] are generated from directed acyclic graphs (DAGs) $G = (V, E)$, with vertices $v \in V$ identified with variables x_v, and contain clauses saying that (a) sources s are true (a unit clause x_s) and (b) truth propagates through the DAG (a clause $\bigvee_{i=1}^{\ell} \overline{x}_{u_i} \vee x_v$ for each non-source v with predecessors $u_1, \ldots, u_\ell$) but that (c) sinks z are false (a unit clause $\overline{x}_z$). Pebbling formulas are trivially refuted by unit propagation, but become more interesting if we replace every variable x by some suitably chosen Boolean function $f(x_1, \ldots, x_d)$ for new variables $x_1, \ldots, x_d$, where f should have the property that no single variable assignment can fix the value of f to true or false (exclusive or $x_1 \oplus x_2$ is perhaps the simplest function with this property). It is not hard to show that all pebbling formulas, even after substitution with a Boolean function of constant arity, have resolution proofs of size linear in the formula size and of constant width. It was proven in [BN08, BN11, Nor12] that the properties of pebbling formulas with respect to space are much more varied, however, and are governed by the complexity of the underlying DAG G with respect to the so-called *black-white pebble game* [CS76]. This means that strong space lower bounds and size-space trade-offs can be obtained by considering suitable families of DAGs. Pebbling formulas over DAGs with high black-white pebbling space complexity also yield exponentially hard formulas for tree-like resolution [BIW04].

Although from a theoretical point of view any function f that satisfies the properties above yields formulas with similar properties, in practice there can be significant differences. A fairly extensive experimental evaluation of pebbling formulas with different substitution functions was performed in [JMNŽ12]. Guided by these results, we use not-all-equal over 3 variables as the substitution function in our experiments.

The families of DAGs that yield the strongest results, as briefly discussed above, are somewhat intricate to construct, and the properties are guaranteed to hold only asymptotically for quite large graphs. We therefore consider pebbling formulas over two simpler, more explicit, families of DAGs, which yield weaker lower bounds but nevertheless seem to produce suitably challenging formulas:

1. Pyramid graphs of height n, consisting of $n+1$ layers $i = 0, 1, \ldots, n$ with $i+1$ vertices in layer i, and with edges from vertices j and $j+1$ in layer i to vertex j in layer $i-1$.

2. Pyramid-of-pyramid graphs, where each vertex in the pyramid of height n is itself blown up to a pyramid of height n. and where in addition to the internal edges inside each blown-up pyramid there are also edges from the sinks of the pyramids at position j and $j+1$ in layer i to all sources in the pyramid at position j in layer $i-1$.

For pyramids and pyramids-of-pyramids the space complexity of the formulas scale like $\sqrt{n}$ and $\sqrt[4]{n}$, respectively, where n is the size of the formula, and they require exponential size $\exp\bigl(\Omega\bigl(\sqrt{n}\bigr)\bigr)$ and $\exp\bigl(\Omega\bigl(\sqrt[4]{n}\bigr)\bigr)$, respectively, for tree-like resolution.

1.4 Stone Formulas

Stone formulas are also generated from DAGs and are similar in flavour to pebbling formulas, but here we think of every vertex of the graph as containing a stone or marker, where every marker has colour red or blue and the formulas specify that (a) stones on sources are red and (b) a non-source with all predecessors red also has a red stone, but (c) the sink has a blue stone. It was shown in [AJPU07] that for appropriately chosen families of DAGs and with $m = 3n$ stones for graphs over n vertices these formulas separate general resolution from so-called *regular resolution*, which is a subsystem of resolution that is strong enough to capture the DP algorithm using variable elimination [DP60]. Stone formulas have also been investigated as candidates for showing that CDCL without restarts cannot simulate the full power of resolution, but the results so far have been inconclusive. It was shown in [BK14] that a model of CDCL without restarts can refute these formulas efficiently, but in contrast to [AFT11, PD11] this model seems too general to yield any compelling practical conclusions (in particular, the solver can choose whether to apply unit propagation or not do so, and it also has very wide latitude how to learn clauses from conflicts, which makes the model too strong to be realistic).

The parameters for which the theoretical results in [AJPU07] hold yield far too large formulas to be manageable in practice. Instead, we run experiments on the following three scaled down versions of stone formulas:

1. For *chain* or *road graphs* (as described in [JMNŽ12, CLNV15]) of length n with $n/2$ stones.

2. For chain/road graphs of length n with n stones.

3. For pyramid graphs of height n with n stones, where for every vertex v only one out of a limited number c of randomly chosen stones can be placed on v. We call this last version of the formulas *sparse stone formulas*.

All of these formulas have resolution proofs in size linear in the formula size. As mentioned above, for the right graphs and number of stones the non-sparse stone formulas are provably hard for regular resolution, and hence also for tree-like resolution, but the concrete parameter values that we use are not strong enough to provide such theoretical lower bound guarantees. Instead, we hope that the formulas are "morally close enough" to [AJPU07] to yield interesting instances in practice. We remark that, to the best of our knowledge, sparse stone formulas have not been studied before and nothing is known about lower bounds for them.

1.5 Subset Cardinality Formulas

These formulas are variations of benchmarks suggested in [Spe10, VS10]. To generate subset cardinality formulas, consider a $0/1$ $n \times n$ matrix $A = (a_{i,j})$ and identify positions $a_{i,j} = 1$ with variables $x_{i,j}$. If we write $R_i = \{j \mid a_{i,j} = 1\}$ and $C_j = \{i \mid a_{i,j} = 1\}$ to denote the non-zero positions in row i and column j, respectively, the subset cardinality formula over A consists of the cardinality constraints $\sum_{j \in R_i} x_{i,j} \geq |R_i|/2$ and $\sum_{i \in C_j} x_{i,j} \leq |C_i|/2$ for all $i, j \in [n]$ encoded into CNF in the natural way (without auxiliary variables). In the case when all rows and columns have $2k$ variables, except for one row and column that have $2k+1$ variables, the formula is unsatisfiable but is hard for resolution if the positions of the variables are "well spread-out" (formally, if the matrix is *expanding*) as shown in [MN14]. It can be shown (using techniques in [MN15]) that this lower bound holds even if we strengthen the cardinality constraints to equalities $\sum_{j \in R_i} x_{i,j} = \lceil |R_i|/2 \rceil$ and $\sum_{i \in C_j} x_{i,j} = \lfloor |C_i|/2 \rfloor$. The clauses in the latter formula are a strict superset of the clauses in the former. We refer to these two variants of the formulas as "greater-than-or-equal (geq)" and "equal (eq)" versions, respectively. Formula sizes and number of variables scale linearly with n.

In order to obtain theoretically easy instances that are nevertheless challenging in practice, instead of considering expanding matrices we generate matrices by fixing a $0/1$ pattern for the first row and shifting this pattern down the diagonal. For such matrices it is not hard to show (as observed in [VS10]) that the resulting "fixed bandwidth" subset cardinality formulas have linear-size resolution proofs. The formulas remain easy even for tree-like resolution, albeit with a polynomial blow-up. We use two fixed bandwidth patterns: 11010001 (i.e., with 1s in positions $1, 2, 4, 8$) as suggested in [VS10], and 11010010 (i.e., with 1s in positions $1, 2, 4, 7$); and then flip the 0 in the top-right corner of the matrix to 1 to obtain an unsatisfiable instance.

1.6 Even Colouring Formulas

Even colouring formulas [Mar06] are defined on connected graphs $G = (V, E)$ with all vertices of constant, even degree and with an odd total number of edges. The edges $e \in E$ correspond to variables x_e, and for all vertices $v \in V$ we have constraints $\sum_{e \ni v} x_e = \deg(v)/2$ asserting that there is a $0/1$-colouring such that each vertex has an equal number of incident 0- and 1-edges. The formula is unsatisfiable since the total number of edges is odd. For suitably chosen graphs, such as random graphs, these formulas are empirically hard for CDCL. It seems plausible that there should be exponential lower bounds on proof size in resolution for formulas obtained from graphs with good enough expansion (such as random graphs). We are not aware of any such formal resolution size lower bounds having been proven, however, and a naive application of the standard lower bound techniques from [BW01] does not work (although this does not seem a priori impossible to achieve with some variation of this approach).

We generate our even colouring formulas not for hard graphs, however, but for the same kind of long, narrow rectangular grids as those used for the Tseitin formulas in Section 1.1, where we then subdivide one edge into a degree-2 vertex to get an odd number of edges. We also generate these formulas for versions of the grids with diagonal edges. It does not seem unreasonable to expect that for grids with similar parameters as those in [BNT13] one should obtain strong size-space trade-offs, just as for Tseitin formulas, but we have no formal proof for this. As for our Tseitin benchmarks, we instead fix the number of rows to be a small constant and then scale the formulas by varying the number of columns. Formula sizes and number of variables scale linearly with n. Again, the formulas have linear-size resolution proofs and polynomial-size tree-like resolution proofs.

1.7 Relativized Pigeonhole Principle Formulas

Relativized pigeonhole principle (RPHP) formulas, which have been studied in [AMO15, ALN16], are a variant of the well-known pigeonhole principle (PHP) formulas with a twist to scale down the hardness from exponential to polynomial. These formulas claim that k pigeons (where we let k be a small constant) can fly into $k-1$ holes via n "resting places," where n is the parameter used to scale the formulas. There are clauses enforcing that pigeons fly into the resting places in a one-to-one fashion and continue from resting places to holes in a one-to-one fashion. For constant k the formula size and number of variables scale like n^2. It was shown in [ALN16] that these formulas require resolution proofs of size roughly n^k, and such proofs can be found even in tree-like resolution. For suitably chosen constant k this is thus an example of a family of formulas that have proofs of polynomial but superlinear size.

The lower bound on size holds even if the formulas are converted to 3-CNF using extension variables. It is clear that if a CNF formula over n variables has a resolution refutation in width w, then it can also be refuted in size $n^{\mathrm{O}(w)}$, and RPHP formulas converted to 3-CNF are extremal in the sense that they show this counting argument to be essentially tight.

1.8 Dominating Set Formulas

Dominating set formulas (as proposed by Atserias [Ats15]) are defined in terms of an undirected graph G and a parameter k, and encode the claim that G has a dominating set of size k (i.e., a set of k vertices such that all other vertices are connected to some vertex in this set by an edge). Since this is an NP-complete problem, these formulas should be expected to be very hard in the worst case. However, we generate formulas for the trivial case of graphs $G = (V, E)$ consisting of $k+1$ disjoint cliques of equal size, i.e., such that $V = \bigcup_{i=1}^{k+1} V_i$ for $|V_i| = |V|/(k+1)$ and $E = \{(u, v) \mid u, v \in V_i \text{ for some } i,\ u \neq v\}$, where k is constant and $n = |V|$ is our scaling parameter.

Even for this trivial case, a resolution proof size lower bound $n^{\Omega(k)}$ can be proven for these formulas [Ats15] using the same kind of approach as in [AMO15, ALN16] for relativized PHP formulas. Hence, this is another example of a family of formulas with proofs of polynomial but superlinear size.

It is worth noting, though, that this lower bound depends on the way the problem is encoded into CNF (as does any lower bound in proof complexity). The particular encoding we use has not been chosen to make the dominating set instance maximally easy to solve, but to ensure that it is possible to prove lower bounds on resolution proof size.

1.9 Clique Formulas

Clique formulas (as discussed in, e.g., [BIS07, BGLR12, BGL13]) are defined in terms of an undirected graph $G = (V, E)$ and a parameter k, and encode the claim that G has a clique of size k. This is again an NP-complete problem, and for constant k it seems reasonable to expect that the worst-case lower bound on proof size should be $n^{\Omega(k)}$, where $n = |V|$ is the number of vertices. Interestingly no such lower bound is known for constant k, however. As for the dominating set formulas in Section 1.8, we consider a family of graphs on which the problem instead becomes very easy, namely complete $(k-1)$-partite graphs $G = \left(\bigcup_{i=1}^{k-1} V_i, E\right)$ where $|V_i| = n/(k-1)$ and $E = \{(u, v) \mid u \in V_i, v \in V_j, i \neq j\}$.

As in Section 1.8, the exact properties of the k-clique formulas depend on exactly how the problem is encoded into CNF, and again we do not optimize the encoding for SAT solver performance but rather choose an encoding for which we can expect the formulas to have interesting properties. For the encoding that we use it can be shown that k-clique formulas for complete $(k-1)$-partite graphs require tree-like resolution proofs of size roughly n^{k-1} and are thus fairly hard for DPLL, but they have very short proof not only in general resolution but even in the subsystem regular resolution (see [BGL13] for more details).

2 Description of CDCL Parameter Settings and Batches of Experiments

To run our experiments, we have instrumented the Glucose SAT solver [AS09, Glu] with "knobs" to analyse the interactions among the following major features of the CDCL algorithm:

1. restart policy,
2. branching,
3. clause database management,
4. clause learning.

Although it is strictly speaking not part of the basic CDCL search algorithm, we have also studied the effects of turning standard preprocessing on or off.

Below, we describe the parameter settings we have investigated, where we use a naming convention [⟨PARAM⟩:⟨setting⟩] to compactly label the different parameters and possible settings for them. We also discuss some potentially interesting settings which we were not able to study, and explain our reasons for prioritizing the way we did.

We want to emphasize that what follows below is not intended as a full technical specification of how our instrumented CDCL solver has been coded up, but rather as an overview of the most important aspects. It should be noted that implementing the key different features in solvers such as MiniSat [Min], Glucose [Glu], and MapleSAT [Map], and doing it is such a way that different combinations of options can be combined in a way that makes sense, poses nontrivial challenges. When implementing features distinctive for a solver, as a general rule we have tried to stick with how these features are implemented in the respective solvers. We have spent a significant effort on verifying the instrumented solver by comparing the solver run with "MiniSat-like settings" to original MiniSat, with "Glucose-like settings" to original Glucose, et cetera, checking that the results were as expected. However, sometimes we have been forced to deviate in certain aspects, in order to get a more uniform and clean code base and/or to avoid undesirable side effects for certain combinations of settings. When this is the case, we have tried to add a comment below, but the reader who desires to know the full technical details will have to look at the source code.[1]

Let us remark right away that looking at the list of CDCL options presented below, ideally we would have liked to run experiments on the full Cartesian product of all possible combinations of settings. This is simply not feasible, however. Already the more limited set of experiments carried out for the present work required a total computation time of on the order of hundreds of years, and generated massive amounts of data to analyse.

2.1 Restart Policy

We consider the following restart policies:

1. [RE:no] No restarts.

[1] The source code will be made available in connection with the publication of the paper.

2. [RE:lbd] LBD-style restarts (as in Glucose; frequent and adaptive).

3. [RE:luE2] Luby restarts with multiplicative factor 100 (frequent; non-adaptive), so that the solver restarts after each $100, 100, 200, 100, 100, 200, 400, 100, 100, 200, 100, 100, 200, 400, 800, \ldots$ conflicts. This is the default restart strategy in Minisat.

4. [RE:luE3] Luby restarts with multiplicative factor 1000 (less frequent; non-adaptive), so that the solver restarts after each $1000, 1000, 2000, 1000, 1000, 2000, 4000, \ldots$ conflicts. This setting was chosen to get a clear difference compared to RE:luE2.

The experiments with very infrequent restarts or with restarts turned completely off are mainly aimed at trying to study the power and limitation of CDCL without restarts, and whether it correlates with different theoretical properties of the combinatorial benchmark formulas (such as being hard or easy for so-called *regular resolution*).

We have found that on average RE:lbd and RE:luE2 generate restarts roughly equally often for the benchmarks that we are considering, and the purpose of comparing these two settings is to see whether the adaptive restarts used by Glucose makes any significant difference. It would also have been interesting to study very frequent Luby restarts RE:luE1 with multiplicative factor 10, but we will have to leave this as future work.

One other set of experiments that could be interesting, but that we were not able to perform on top of all other experiments, would be to have a CDCL solver doing reasonably frequent restarts but chosen at random points in time, to check whether state-of-the-art restart heuristics are doing something clever or whether the frequency of the restarts is what mainly counts.

2.2 Branching

For variable selection we explore two main options: fixed-order branching (chosen to be good for the benchmark in question when a good fixed order clearly exists) and VSIDS branching with different settings of the decay factor, where a larger decay factor corresponds to remembering more of the conflict history when choosing which variable to branch on next. We also experiment with learning rate-based branching (LRB) as described in [LGPC16b]. Finally, we have done some limited experiments with a solver choosing variables to decide on at random, with the purpose of verifying that the different heuristics described above are doing something clearly better than this. The full list of variable decision options is as follows:

1. [VD:fix] Fixed order (chosen to be good for the benchmark when possible).

2. [VD:rnd] Randomly chosen variable decision.

3. [VD:df99] VSIDS with decay factor 0.99.

4. [VD:df95] VSIDS with decay factor 0.95.

5. [VD:df80] VSIDS with decay factor 0.80.

6. [VD:df65] VSIDS with decay factor 0.65.

7. [VD:lrb] Learning rate-based branching.

By “random decisions” as in VD:rnd we mean the following: At the outset all variables are given random “activity scores.” These numbers are then just treated as standard VSIDS activity scores in the sense that the next decision will always be made on the unset variable with highest core. There is no exponential decay of the scores, but at every backjump all variables removed from the heap get a new random activity score.[2]

The reason to investigate several different VSIDS decay factors is that some limited (and unpublished) previous experiments have indicated that some combinatorial benchmarks are quite sensitive to this setting. Also, for some other benchmarks VSIDS seems to perform very poorly [MN14], and it is interesting to study whether this is due to a very particular setting of the decay factor or whether it is a problem with the VSIDS heuristic as such.

We remark that for all our VD:df⟨NN⟩ options the VSIDS decay factor is fixed once and for all, and is not gradually modified as in, e.g., Glucose. This is in order to get cleaner experiments with results that are easier to analyse. Another important aspect to note is that as the VSIDS decay factor is chosen to be smaller and smaller, more and more time will be spent on moving elements in the heap maintaining variables enqueued in order of decreasing VSIDS score, and this time is significant [Bie16]. One interesting direction for future work would be to run experiments on a carefully implemented version of *variable move to front (VMTF)*, which is essentially the same heuristic as VSIDS with decay factor 0.5 but can be much more efficient since there is no need to maintain a heap. Another interesting future work to research is the heuristic conflict history-based (CHB) branching described in [LGPC16a].

We also want to explore different settings of the literal phase saving option:

1. [PH:dynrnd] Phase chosen independently at random for every new variable assignment ("dynamic random phase," meaning no phase at all).

2. [PH:fix0] Phase fixed to all false at start of execution (as the "classic" MiniSat default) and then kept fixed.

3. [PH:fixrnd] Phase fixed randomly at beginning to an independently chosen random polarity for each variable, and then kept fixed ("fixed random phase").

4. [PH:std] Standard phase saving with initial phase set to all false at start of execution and with phases of individual variables updated for every unit propagation.

5. [PH:ctr] Branching against the phase, setting variables to the opposite value to which they were last propagated.

We want to point out that for the options PH:std and PH:ctr we define the "phase" of a variable to be the value it was set to *when it was last unit propagated.* For the option PH:std, whenever a variable decision is made the variable is set to the same value as the phase. For PH:ctr, the variable is instead consistently set to the opposite value of the phase (but the phase itself does not change until the variable is unit propagated to the opposite value).[3]

[2] We remark that although decisions made as outlined here are certainly "random" in some sense, this is not the only possible solution. Indeed, an a priori even more natural interpretation of "random decision" would be that the decision is always made on a variable chosen uniformly and independently at random among all variables. It can be noted that our implementation does *not* achieve a uniform distribution, since variables that have not been decided on for a long time are less likely to be chosen next than variables that were decided on more recently (if a variable has not been decided on for a long time, then it is likely that its randomly chosen activity was quite low). However, while a uniformly random distribution would have been more pleasing from a theoretical point of view, it is not clear how to implement it without a significant penalty in performance due to that it would take time to find a random variable that has not already been set. This is not the effect that we want to measure. Instead, our solution reuses the standard data structures employed for VSIDS, in this way avoiding a performance hit, and since variables are randomly rescored after being removed from the trail after a backjump the actual distribution should hopefully be at least not too far from uniformly random in practice.

[3] It should be emphasized that in this definition the phase of a variable does *not* change when the variable is decided, but only when it is propagated. Another possible definition of phase would have been to say that the phase is updated every time a variable is assigned (by a decision or propagation). For the PH:std this modification is immaterial, since any variable decision is always made in accordance with the phase, and so we can adopt either definition without changing the standard meaning of the phase. For the PH:ctr

As a possible direction for future work, we note that the VSIDS version we are using now bumps variables resolved in the conflict analysis as well as variables in the learned clause. In some early CDCL solvers only variables in the learned clause were bumped. It could be interesting to experiment with the three different variants of bumping only learned clause variables, only resolved variables, or both, to see if one could separate the effects and determine what is most important.

2.3 Clause Database Management

When it comes to managing the database of learned clauses, we explore different policies for clause erasure (how often to delete learned clauses and how many clauses to erase) and for clause assessment (which learned clauses are deemed to be more desirable to keep or throw away when it is time for clause erasure).

We also want to explore the effects of turning clause learning on or off, switching between DPLL-style and CDCL-style search. This is particularly interesting for us since many of the combinatorial benchmarks that we study actually have short proofs even in tree-like resolution, which means that they can in principle be solved by simple DPLL backtrack search with the right variable order. Implementing such a switch between CDCL and DPLL is easier said than done, however. It is not entirely clear what it would mean to "turn off" clause learning while "keeping fixed" other parameter settings such as branching heuristic, since VSIDS-based variable selection is intimately linked with clause learning. Also, it would seem to handicap DPLL unnecessarily if we only backtrack and do not backjump even when it is immediately obvious from the current conflict that such backjumping could be made.

A first, naive approach would be to simply implement "true" DPLL but with backjumps instead of simple backtracking. What we mean by this is running DPLL to conflict, performing conflict analysis (from the clauses in the formula), finding the asserting literal, backjumping to the corresponding level and making a decision setting the asserting literal to the right value. This would involve deriving the 1UIP clause, which would also be used to compute VSIDS scores, but this clause would then immediately be forgotten after the backjump.

This approach would yield a DPLL solver with VSIDS of sorts. However, as the DPLL search continued, the conflict analysis and VSIDS bumping would diverge more and more from what we usually mean by these concepts. In a standard CDCL solver, the assignment of the asserting literal after a backjump, when this literal is flipped, does not count as a decision affecting the decision level but is treated as a propagation, which it what it is. This means that if a new conflict is immediately reached, then the ensuing conflict analysis can potentially go further back on the trail beyond the assignment to the asserting literal. However, in the DPLL version described here the assertion would in fact be a decision, which means that any future conflict analysis would have to stop at this point. In this way, the conflict analysis and VSIDS scoring would fail to see that this assignment is something that follows from previous conflicts rather than being an independent decision.

Therefore, we would like the asserted literal to be a unit propagation with a reason clause, so that future conflict analysis steps will treat the variable assignment correctly. However, this would require us to save a reason clause for this propagation, which would be the clause just learned during conflict analysis. This also seems a bit problematic, since we just said that we wanted to switch clause learning off.

However, some careful thought reveals that one can envision a hybrid DPLL/CDCL algorithm that only does the bare minimum of clause learning needed to get VSIDS scores that make sense, and which would work as follows:

1. Run DPLL search to conflict, do conflict analysis, and find the 1UIP clause C with asserting literal a.
2. Backjump and assert the literal a, storing C as the reason clause.

option, however, if decisions are allowed to affect the phase, then the assignments to a variable would flip-flop if it were assigned multiple times without being propagated in between. We feel it is more interesting to explore "branching against the phase" as we first described, where the phase is affected by unit propagations only.

3. Immediately before any further unit propagation has been done, erase all learned reason clauses beyond the backjump point that are not active reason clauses of propagated assignments currently on the trail. Then resume DPLL search as in step 1.

For the next conflict we will now get a proper conflict analysis that takes into consideration that the literal a is *not* a decision but was propagated by the learned clause C. Thus, by performing the search in this way, we can ensure that all measures and heuristics such as VSIDS, LBD, et cetera maintain their standard meaning.

One argument against this approach is that it is no longer seems to be a DPLL-style algorithm, since we will have up to n learned clauses in memory, where n is the number of variables in the formula. However, some further careful thought reveals that if we learn the "vanilla" 1UIP clause without recursive clause minimization, so that during conflict analysis only reason clauses at the conflict level are touched, then in fact this procedure does generate a tree-like resolution proof, and is thus a form of DPLL search![4] For practical reasons, the DPLL-like option that we actually implement deviates slightly from the above description for the special cases of unary and binary clauses. When a CDCL solver learns such clauses, they are typically not put in the general clause database but are handled separately. Undoing this special handling without breaking other parts of the CDCL solver seems complicated, and therefore the DPLL option in our instrumented CDCL solver implements a CE:dpll option generating resolution proofs that are tree-like except for that unary and binary clauses can be reused several times (thus making our DPLL-like version slightly stronger than actual DPLL).

Summarizing, for clause erasure we consider the following settings:

1. [CE:no] No clause erasure (keep all learned clauses).

2. [CE:lin] Essentially as in Glucose, but modified so that the clause database size is increased not by an additive term but by a multiplicative factor. This means that after N conflicts the clause database will contain $\Theta(N)$ clauses.

3. [CE:glu] standard Glucose-style removal: Erase half of the clauses; increase clause database size by an additive term. After N conflicts the clause database contains $\Theta\left(\sqrt{N}\right)$ clauses.

4. [CE:min] MiniSat-style removal that keeps roughly $O\left(N^{0.24}\right)$ clauses after N conflicts.[5]

5. [CE:dpll] Absolutely minimal clause learning keeping only currently active reason clauses on the trail as explained above, yielding essentially DPLL-like behaviour.[6]

For clause assessment we consider the standard settings used in MiniSat and Glucose, and also do limited comparisons with a totally random heuristic just to check how well the state-of-the-art heuristics identify important clauses.

1. [CA:none] No clause assessment at all (used together with CE:no and CE:dpll).

2. [CA:rnd] Randomly choose which clauses to erase or keep.

3. [CA:vsids] Keep clauses with a good (high) VSIDS score à la MiniSat.

[4] This requires a careful argument, of course, but it is not too hard to argue formally. Very briefly, the reason that this works is that we guarantee that a learned clause is only used (at most) once in conflict analysis before being thrown away, and therefore the resolution proof found by the CDCL solver will have a tree-like structure.

[5] We deviate from MiniSat behaviour in the following important way. When pruning the clause database, MiniSat calculates how many clauses should be removed but on top of this unconditionally erases any clause that has a VSIDS activity score below a certain threshold. This could lead to much more aggressive clause removal than described above. Since we want to compare how different ways of assessing clause quality affect performance when the total number of clauses in the database is the same, we have turned off this feature.

[6] Note that, for reasons as described above, for CE:dpll we switch *off* recursive clause minimization, which is otherwise turned on as per the Glucose default setting.

4. [CA:lbd] Keep clauses with a good (low) LBD score à la Glucose (and always keep "glue clauses" with LBD score at most 2).[7]

2.4 Clause Learning

For clause learning, our main focus is on the 1UIP clause learning scheme, which seems to be the one used in all state-of-the-art CDCL solvers. In order to have at least one point of comparison, however, we also study the *last (decision) UIP* learning scheme. Since it turns out that the description of kUIP clauses for $k > 1$ is not quite standardized in the SAT solving literature, let us give a precise description what we mean.

When doing conflict analysis, we start with the clause falsified by the current assignment and perform a trivial resolution derivation with the reason clauses on the trail in reverse chronological order up until some point where we stop. This termination point is what defines the clause learning scheme. To obtain the 1UIP clause, we carry out the resolution derivation up until the first time when the derived clause contains only a single literal from the final decision level. When this clause has been derived, we also perform standard clause minimization as in MiniSat and Glucose. To obtain the *last (decision) UIP clause*, we instead perform the resolution derivation all the way to the final decision on the trail (and then minimize the clause as before). Thus, we consider the following two learning schemes:

1. [CL:1uip] Standard 1UIP clause learning with standard clause minimization (except when combined with the CE:dpll version as noted above).

2. [CL:Luip] Last (decision) UIP clause learning (with standard clause minimization).

We mention that other variants, such as learning multiple clauses per conflict as explored in [MS99], or performing the conflict analysis all the way back to the beginning of the trail (referred to as the *all UIP* learning scheme in [ZMMM01]), remain as future possibilities to explore.

2.5 Preprocessing

Our focus in this work is on understanding basic CDCL proof search. We are of course aware of the immense importance of different preprocessing techniques, but have chosen not to go into a deeper investigation of these techniques for two reasons:

- Firstly, one important reason for preprocessing seems to be to "re-encode" CNF instances into a form that makes them more amenable to CDCL proof search, compensating for choices made in the initial encoding, as it were. For our combinatorial instances we believe that such considerations should not be too relevant, since the CNF formulas are quite straightforward encodings of the underlying combinatorial principles.

- Secondly, preprocessing is much less well understood from a theoretical point of view, and use techniques that if deployed optimally are too strong to allow satisfactory analysis by theoretical means. (In particular, they have the power to solve almost all CNF benchmark formulas discussed in the literature extremely efficiently—since they are strong enough to capture reasoning in the *extended Frege proof system*—but this seems less interesting from a practical point of view since it clearly does not correspond to what can be observed in reality.)

[7] It is important to note that because of the fact that glue clauses are always kept, the option CA:lbd might keep more clauses than other clause assessment settings CA:xxx for an identical clause erasure setting CE:yyy, and for some benchmarks this difference can be noticeable. In this sense, the comparison of different clause assessment settings might not be completely fair, since CA:lbd gets to save more clauses. One way around this would have been to force the option CA:lbd to erase also glue clauses to reach the correct clause database size bound. Since previous experimental work indicates that the saving of glue clauses is an important part of the good performance of Glucose, however, we felt that this change would have been to drastic and instead decided to let CA:lbd keep slightly more clauses than other clause assessment settings.

Nevertheless, since preprocessing is an integral part of successful implementations of the CDCL paradigm, we include some preprocessing in our experiments to see if and how it affects our overall conclusions. We therefore explore two options:

1. [PR:off] Preprocessing turned off.
2. [PR:on] Standard preprocessing as in Minisat/Glucose turned on.

2.6 Some General Comments on CDCL Parameter Settings

A CDCL solver with some configuration of settings as described above can be encoded by a character string with capital letters for each parameter followed by letters specifying the setting. For instance, the string

$$\texttt{RE:lbd-VD:df80-PH:std-CE:glu-CA:vsids-CL:1uip-PR:on} \tag{2.1}$$

encodes a solver with the following settings:

- Glucose LBD-style restarts.
- VSIDS variable decision heuristic with decay factor 0.80.
- Standard phase saving on.
- Glucose-style removal keeping a number of clauses scaling roughly like the square root of the number of conflicts seen so far.
- Priority given to clauses with a good (high) VSIDS score when deciding which clauses to keep during clause database reduction.
- 1UIP clause learning scheme.
- Standard Glucose preprocessing turned on.

A more careful study of all the different options described above reveals that not all combinations of settings make sense. For instance, if we are not going to erase any learned clauses, then we do not need different ways of assessing which clauses to keep. As another example, if the variable decision order is fixed, then we do not need different settings for computing the VSIDS scores that will never be used anyway. However, even limiting the parameter settings to only combinations that make sense yields a huge number of different variants to try out. If in addition we want to run the solvers with these settings on close to a thousand different benchmarks, which we do, and if we want to use a large solver time-out on the order of an hour or two, which preliminary experiments show that we have to do, then even just a quick back-of-the-envelope calculation reveals that the total computation time will be completely unrealistic.

Since we cannot explore all combinations of settings, what should we do? A natural proposal might seem to be that the number of combinations could be decreased, taking into consideration that certain settings are known according to SAT community folklore to work well only in combination with certain other settings, and so there is no need to test them with all other combination. But appealing to such folklore knowledge is precisely what we do *not* want to do! Indeed, we would expect that the results of our experiments will show that certain settings only work well together, but this is the kind of “received wisdom” that we would want to confirm (or refute) as a result of our experiments, and not to take as a starting point for the experiments. We do not want to exclude the possibility that we will find pieces of conventional wisdom that the results of our experiments will flag as questionable, or at least as meriting a closer study.

These consideration lead us to the experiments outlined in the next section.

2.7 Batches of Experiments

Let us now list the batch of experiments that we have run. Let us first recall the different parameters we want to vary and the different settings that we want to explore for these parameters. Below, each setting of a parameter is marked as follows:

- A superscript r denotes that this is an a priori relevant setting that we want to explore in combination with all other settings with which it can be combined.
- A superscript $*$ denotes that this is a non-standard setting that will only be investigated together with a limited number of other combinations.
- A superscript D denotes the "default" combinations considered for such non-standard experiment combinations, intended to give reasonable but not full coverage of other combinations together with non-standard parameter settings.

This results in the following list of parameters and settings:

Restart policy: RE:no^r, RE:lbd^D, RE:luE2^r, RE:luE3^D.

Variable decisions: VD:fix*, VD:rnd*, VD:df99*, VD:df95^D, VD:df80^D, VD:df65*, VD:lrb^r, VD:chb*.

Phase saving: PH:dynrnd*, PH:fix0^D, PH:fixrnd*, PH:ctr*, PH:std^D.

Clause erasure: CE:no^r, CE:lin^r, CE:glu^D, CE:min^D, CE:dpll*.

Clause assessment: CA:none*, CA:rnd*, CA:vsids^D, CA:lbd^D.

Clause learning: CL:1uip^D, CL:Luip*.

Preprocessing: PR:off^D, PR:on^D.

We remark that the reason to pick RE:lbd and RE:luE3 as our "default" restart combinations is that we want to have one setting for "fast restarts" and one for "slow restarts" to combine with the other settings. In preliminary experiments we were not able to see any significant difference between LBD-based restart and fast Luby restarts, and so the choice between RE:lbd and RE:luE2 for our "fast restarts" did not seem to make too much of a difference.

It is important to note that when we want to analyse certain (sub)combinations of settings, we always need to study full Cartesian (sub)products of these settings. Schematically, if we want to study how the settings AA:uuu and AA:vvv interact with BB:xxx and BB:yyy, then it is important that we consider all four combinations AA:uuu-BB:xxx, AA:uuu-BB:yyy, AA:vvv-BB:xxx, and AA:vvv-BB:yyy. Otherwise the conclusions we reach could be misleading. The way we achieve large subsets of full Cartesian products is firstly by having one large batch (Batch 1) that contains most of the standard combinations we are interesting in studying, and secondly by making a number of smaller batches for non-standard settings that can be combined with subsets of Batch 1 to yield full Cartesian products for these settings.

2.7.1 Batch 1 (Main Configuration — 336 Combinations)

(RE:no, RE:lbd, RE:luE2, RE:luE3) × (VD:df95, VD:df80, VD:lrb) × (PH:fix0, PH:std) × (CE:no,[8] CE:lin, CE:glu, CE:min) × (CA:vsids, CA:lbd) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

[8] Note here that since CE:no keeps all learned clauses, there is no need to combine it with different clause assessment strategies (used to choose which clauses to erase), and so the actual number of combinations is slightly smaller than what the full Cartesian product would suggest.

Note that this batch explores all restart policies we want to study, so no special dedicated batch below is needed to cover additional possible settings for restarts. We also run experiments with preprocessing turned both on and off, and so no special batch is needed for that either.

2.7.2 Batch 2 (Variable Decisions — 96 Combinations)

(RE:lbd, RE:luE3) × (VD:rnd, VD:df99, VD:df65) × (PH:fix0, PH:std) × (CE:glu, CE:min) × (CA:vsids, CA:lbd) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

Results from the above experiments can be studied together with the corresponding combinations for the settings VD:df95, VD:df80, and VD:lrb from Batch 1 to obtain a broad picture of the behaviour of different variable decision heuristics.

2.7.3 Batch 3 (Good Fixed Variable Decision Order — 40 Combinations)

(RE:no, RE:lbd) × (VD:fix) × (PH:fix0, PH:std) × (CE:no, CE:glu, CE:min) × (CA:vsids, CA:lbd) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

This batch is mostly intended to obtain benchmark values to see how fast the solver could run if it performed proof search extremely efficiently (which will be the results for most of our combinatorial benchmarks when using the fixed order). However, it might also be possible to combine it with experiments from the main batch 1 and the variable-decision batch 3.

2.7.4 Batch 4 (Phase Saving — 96 Combinations)

(RE:lbd, RE:luE3) × (VD:df95, VD:df80) × (PH:dynrnd, PH:fixrnd, PH:ctr) × (CE:glu, CE:min) × (CA:vsids, CA:lbd) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

Results from the above experiments can be studied together with the corresponding combinations for the settings PH:fix0 and PH:std from Batch 1 to obtain a broad picture of the behaviour of different phase saving (or non-saving) heuristics.

2.7.5 Batch 5 (DPLL-like — 16 Combinations)

(RE:lbd, RE:luE3) × (VD:df95, VD:df80) × (PH:fix0, PH:std) × (CE:dpll) × (CA:none) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

The main purpose of this batch is to study what happens when clause learning is turned off. The expectation is that this should be very bad for most other combinations of settings.

Note that all other clause erasure options are studied in Batch 1. Results from that batch can be compared to the results for the DPLL-like option in this batch, but since DPLL does not use any clause assessment settings it is not possible to obtain a full Cartesian product of settings.

2.7.6 Batch 6 (Clause Assessment — 24 Combinations)

(RE:lbd, RE:luE3) × (VD:df95, VD:df80) × (PH:std) × (CE:lin, CE:glu, CE:min) × (CA:rnd) × (CL:1uip) × (PR:off, PR:on) × (SH:off)

Results from the above experiments, choosing clauses to erase completely at random, can be studied together with the corresponding combinations for the standard clause assessment settings CA:vsids and CA:lbd in Batch 1 to see if the latter settings are better at identifying which learned clauses are useful and should be kept.

2.7.7 Batch 7 (Clause Learning Schemes — 64 Combinations)

(RE:lbd, RE:luE3) × (VD:df95, VD:df80) × (PH:fix0, PH:std) × (CE:glu, CE:min) × (CA:vsids, CA:lbd) × (CL:Luip) × (PR:off, PR:on) × (SH:off)

These experiments with non-standard last UIP clause learning can be compared with the same configurations using standard clause learning CL:1uip from Batch 1 instead.

3 Brief Discussion of Tools Used to Analyse the Results

3.1 Web-interface for Interactive Visualization

We have implemented a web application to interactively explore and visualize the data. This application allows the user to represent three kind of plots, which are described in this section. We emphasize that we plan to make public this application and all this data; we could not do it in this submission for anonymity restriction.

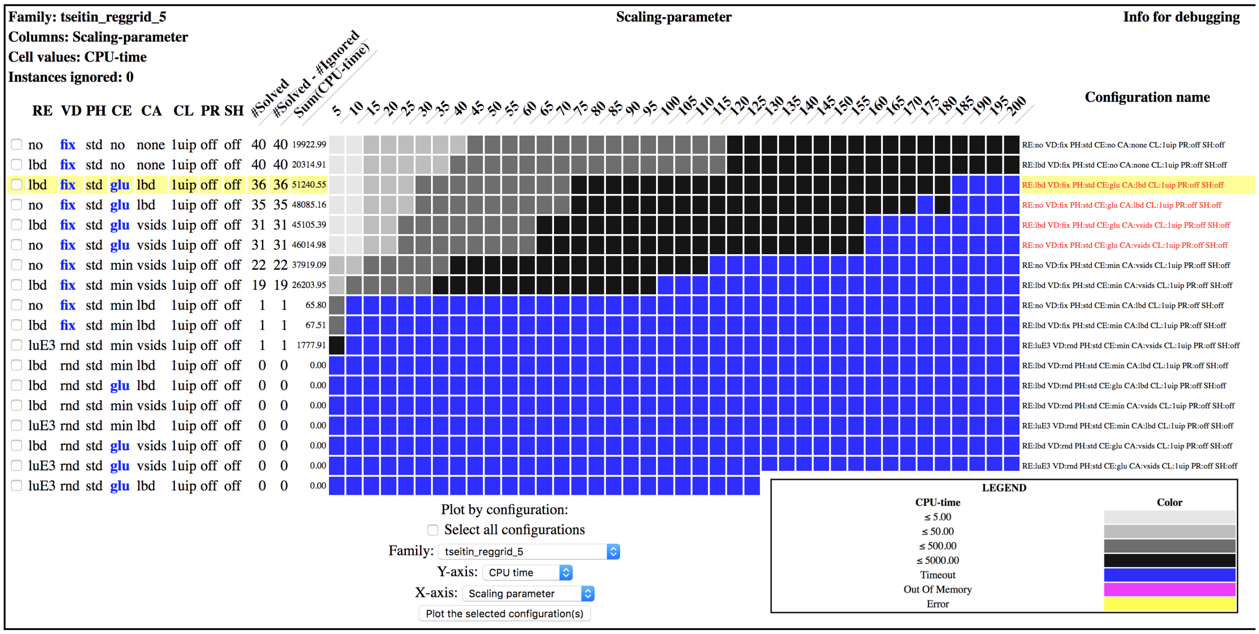
First, we have *heatmaps* to represent the performance of many configurations, and compare them. For a set of selected configurations and a given family, it creates a table where each row represents each of these configurations, and the columns represent the instances of the family (e.g., their scaling parameter) –there will be as many columns as instances in the selected family–. The cells of this table are just colors in a grey-scale, which represent different solver performances in a selected metric (e.g., solving time): from short solving times (in light grey) to large solving times (in black). We distinguish 4 categories (colors) for these states (the user can change the intervals of these four categories; by default these categories represent different orders of magnitude). The application also allows to represent other metrics, as number of conflicts or number of propagations. Instances not solved (TIMEOUT) are represented in blue, as OUTOFMEMORY's are represented in purple. Finally, the rows (CDCL configurations) are sorted by the number of solved instances, breaking ties by the sum of solving times (or number of conflicts). The application has some extra function, like highlighting certain parameter values, or sorting rows alphabetically by some parameter.

Therefore, *good* configurations are positioned in the top of these heatmaps, and are characterized by a row with cell-colors in the grey scale (i.e., most instances of the family were solved), whilst the bottom of these plots shows the *bad* configurations, which are probably characterized by many blue cells (TIMEOUT's).

In Figure 1, we show an example of these heatmaps on Tseitin formulas, where it is shown some configurations with [VD:fix] (i.e., good configurations, hence in the top of the heapmap with cells in the grey-scale) and [VD:rnd] (i.e., bad configurations, hence in the bottom of the heatmap with most of the cells in blue representing TIMEOUT's).

The second kind of plots for the *comparison of two parameters*, in order to compare two values of the same parameter (e.g., [RE:lbd] versus [RE:luE2]). For two given parameter values [PAR:a] and [PAR:b], a given family, and a given set of configurations, the application looks for all pairs of existing configurations (in the given set) such that the first element of the pair is a configuration with [PAR:a], and the second element is the same configuration but with [PAR:b] instead. Then, it is plotted a table, where every row corresponds to each of these pairs, and similarly to heatmaps, every column represents each of the instances of the family (e.g., their scaling parameter). Every cell represents the ratio of the two configurations in each pair, on a given metric, (e.g., solving time), i.e., each cell would represent $\frac{CPUtime(configuration\text{[PAR:a]})}{CPUtime(configuration\text{[PAR:b]})}$ when the selected metric is the solving time. We use a red-grey-green scale to represent this ratio, where a light grey is a neutral color to represent that both configurations perform similarly, and the darker red (resp. green) represents a big difference between them, being [PAR:b] (resp. [PAR:a]) much better.

In Figure 2, we represent an example of these comparison plots on ordering principle formulas, for which the value of the VSIDS decay factor is crucial: formulas becomes very hard for configurations with a high decay factor, whilst they are solvable when decreasing such a value. Therefore, most of the cell in this plots are dark red.



Family: tseitin_reggrid_5
Columns: Scaling-parameter
Cell values: CPU-time
Instances ignored: 0

	RE	VD	PH	CE	CA	CL	PR	SH	#Solved	#Solved - #Ignored	Sum(CPU-time)	Configuration name
□	no	fix	std	no	none	1uip	off	off	40	40	19922.99	RE:no VD:fix PH:std CE:no CA:none CL:1uip PR:off SH:off
□	lbd	fix	std	no	none	1uip	off	off	40	40	20314.91	RE:lbd VD:fix PH:std CE:no CA:none CL:1uip PR:off SH:off
□	lbd	fix	std	glu	lbd	1uip	off	off	36	36	51240.55	RE:lbd VD:fix PH:std CE:glu CA:lbd CL:1uip PR:off SH:off
□	no	fix	std	glu	lbd	1uip	off	off	35	35	48085.16	RE:no VD:fix PH:std CE:glu CA:lbd CL:1uip PR:off SH:off
□	lbd	fix	std	glu	vsids	1uip	off	off	31	31	45105.39	RE:lbd VD:fix PH:std CE:glu CA:vsids CL:1uip PR:off SH:off
□	no	fix	std	glu	vsids	1uip	off	off	31	31	46014.98	RE:no VD:fix PH:std CE:glu CA:vsids CL:1uip PR:off SH:off
□	no	fix	std	min	vsids	1uip	off	off	22	22	37919.09	RE:no VD:fix PH:std CE:min CA:vsids CL:1uip PR:off SH:off
□	lbd	fix	std	min	vsids	1uip	off	off	19	19	26203.95	RE:lbd VD:fix PH:std CE:min CA:vsids CL:1uip PR:off SH:off
□	no	fix	std	min	lbd	1uip	off	off	1	1	65.80	RE:no VD:fix PH:std CE:min CA:lbd CL:1uip PR:off SH:off
□	lbd	fix	std	min	lbd	1uip	off	off	1	1	67.51	RE:lbd VD:fix PH:std CE:min CA:lbd CL:1uip PR:off SH:off
□	luE3	rnd	std	min	vsids	1uip	off	off	1	1	1777.91	RE:luE3 VD:rnd PH:std CE:min CA:vsids CL:1uip PR:off SH:off
□	lbd	rnd	std	min	lbd	1uip	off	off	0	0	0.00	RE:lbd VD:rnd PH:std CE:min CA:lbd CL:1uip PR:off SH:off
□	lbd	rnd	std	glu	lbd	1uip	off	off	0	0	0.00	RE:lbd VD:rnd PH:std CE:glu CA:lbd CL:1uip PR:off SH:off
□	lbd	rnd	std	min	vsids	1uip	off	off	0	0	0.00	RE:lbd VD:rnd PH:std CE:min CA:vsids CL:1uip PR:off SH:off
□	luE3	rnd	std	min	lbd	1uip	off	off	0	0	0.00	RE:luE3 VD:rnd PH:std CE:min CA:lbd CL:1uip PR:off SH:off
□	lbd	rnd	std	glu	vsids	1uip	off	off	0	0	0.00	RE:lbd VD:rnd PH:std CE:glu CA:vsids CL:1uip PR:off SH:off
□	luE3	rnd	std	glu	vsids	1uip	off	off	0	0	0.00	RE:luE3 VD:rnd PH:std CE:glu CA:vsids CL:1uip PR:off SH:off
□	luE3	rnd	std	glu	lbd	1uip	off	off	0	0	0.00	

Plot by configuration:
□ Select all configurations
Family: tseitin_reggrid_5
Y-axis: CPU time
X-axis: Scaling parameter

LEGEND

CPU-time	Color
≤ 5.00	⬜
≤ 50.00	▒
≤ 500.00	▓
≤ 5000.00	⬛
Timeout	🟦
Out Of Memory	🟪
Error	🟨

Figure 1: Example of heatmap on Tseitin formulas on regular grids, showing *good* configurations with [VD:fix] and *bad* configurations with [VD:rnd].

Finally, the application allows to represent *X-Y axes plots*, which are useful to show the exact performance of a reduced number of configurations. These plots are an alternative to represent the same results that are shown in the heatmaps. Notice that in heatmaps results are aggregated on the selected metric (e.g., solving time) into four categories. In these plots, however, we represent the actual value of this metric in the Y-axes (and, as usual, the scaling parameter, or any other metric representing each formula of the family, like formula size, in the X-axes). However, representing a large number of configurations makes these plots very messy. Therefore, to represent a large number of configurations, heatmaps seem to be the most suitable candidate (although some information may be lost in the aggregation of the results).

In Figure 3, we represent an example of these plots on clique formulas, where we represent several configurations with and without restarts. We can observe than disabling restarts outperforms all the other configurations with restarts.

We finally emphasize that we have carried out some preliminary experiments for analyzing the results using other *more automated* methods, like aggregating results using PAR2. However, such a huge aggregation of the results must be done carefully, since some results may not represent the performance of the solver accurately. For instance, in Tseitin formulas, configurations with (CE:glu × CA:lbd) perform, in general, better than configurations with [CE:lin] (regardless of the CA value). However, these last configurations generally perform better than configurations with (CE:glu × CA:vsids). Therefore, aggregating the PAR2 results of [CE:glu] configurations and compare it to the same aggregation on [CE:lin] configurations may lead to erroneous conclusions. In particular, the problem is that sometimes pairs or triples of settings are more important than a single setting, and this kind of pairwise or triplewise correlations are very challenging to find.

In what follows, we present a list of examples –as screenshots of these tools– of (some) of the observations stated in the paper:

- In Figure 4, we represent some configurations with different restarts strategies on pebbling formulas on pyramid graphs, to show that, even if LBD restarts are less frequent than Luby-2 restarts, they are more efficient in practice.

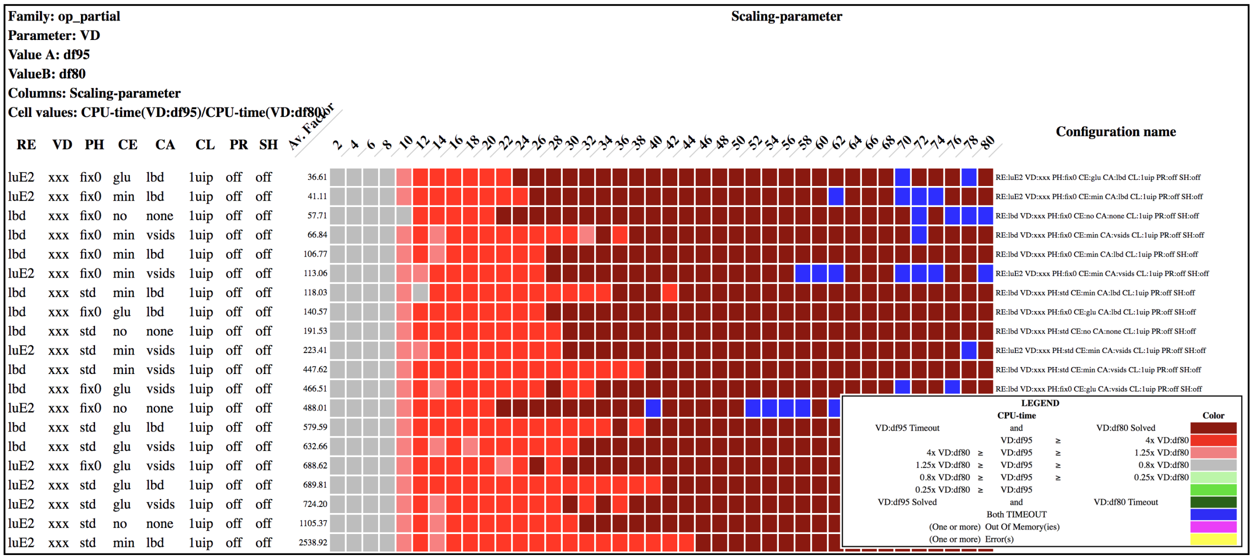


Figure 2: Example of two parameters comparison plot on partial ordering principle formulas, showing that a low value on the VSIDS decay factor is much better than a high value.

- In Figure 5, we represent the comparison between standard phase saving and branching with always negative phase in sparse stone formulas, to show that phase saving is actually crucial in this family.
- In Figure 6, we represent the heatmap of different configurations varying the clause erasure policy on even colouring formulas on regular grids. We show that in these formulas with time-space trade offs, a very aggressive clause erasure policy hurts a lot.
- In Figure 7, we represent the comparison of some configurations with and without preprocessing on ordering principle formulas with partial ordering. We show that in this family, preprocessing has no real effect on the performance of the solver.

3.2 Automated Data Exploration

The goal of the automated data exploration is to figure out which settings are important and significantly different to other settings. As the different benchmark families are expected to have different behavior the results are computed for each subfamily. Only the scaling parameter changes within the subfamily and therefore it is expected that a good configuration performs good on all instances. Therefore, we will only consider instances of one subfamily in the following paragraphs.

Let $\mathcal{I}$ be the set of all instances and $\mathcal{C}$ the set of all configurations. The performance score $s(c, i)$ is given for each configuration $c \in \mathcal{C}$ and instance $i \in \mathcal{I}$. The performance score can be either a par-x score to measure runtime or the rank of the configuration when compared to all other configurations on this instance, which can be used for runtime or number of conflicts.

The first step is to aggregate the performance score for each configuration, which is done by taking the average: $agg(c) := \text{avg}\{s(c, i) | i \in \mathcal{I}\}$.

By selecting all configurations that contain a certain setting of a parameter p, for example all configurations containing CE:no, we can construct a set $C_p \subseteq \mathcal{C}$. This set is again associated with an average score: $agg(C_p) := \text{avg}\{agg(c) : c \in C_p\}$. The measure $agg(C_p)$ can now be compared to the total average, i.e. $\mu := \text{avg}\{agg(c) : c \in \mathcal{C}\}$ to figure out if the chosen setting p performs better or worse than average.

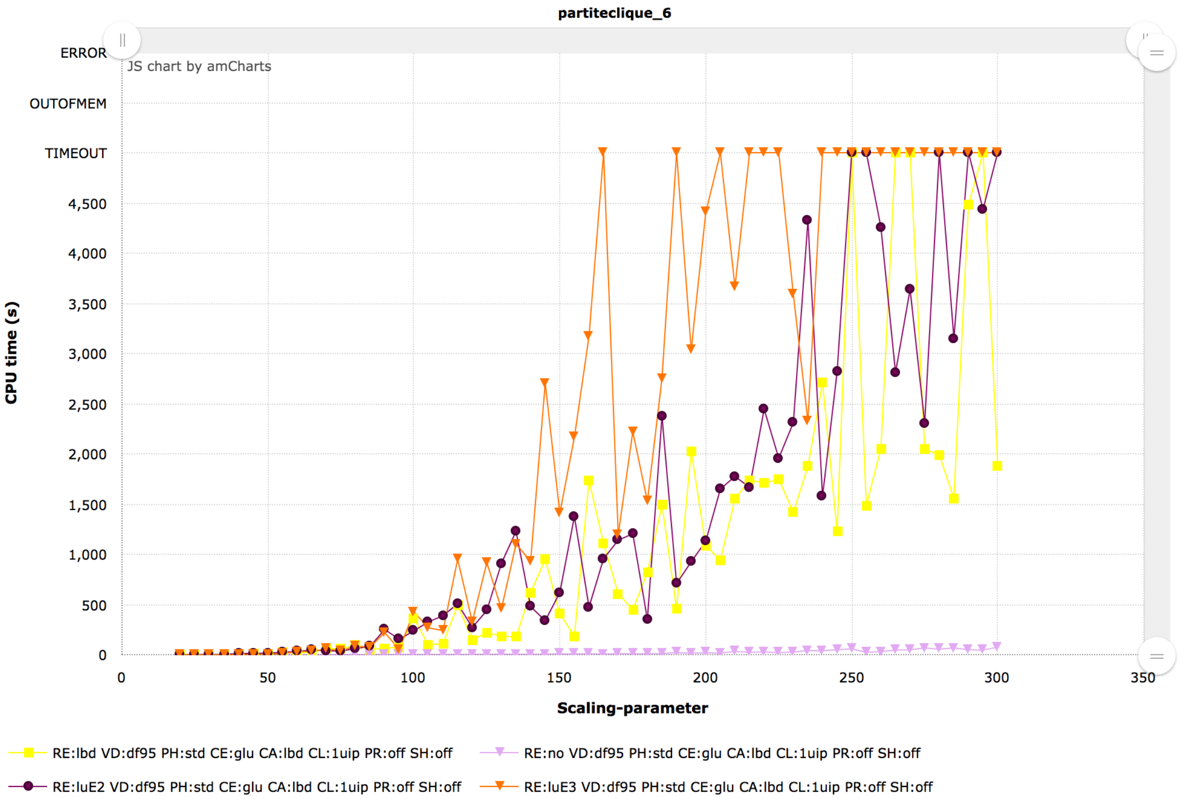


Figure 3: Example of X-Y plot on clique formulas, where it can be seen that using restarts hurts a lot.

However, this does not tell us how significant the discrepancy to the average is. Especially, even when choosing a set $C \subseteq_r \mathcal{C}$ completely at random we would expect to see some deviation from the average. To evaluate this we sample the standard deviation $\sigma(|C_p|) := \text{stdDev}\{agg(C) : C \subseteq_r \mathcal{C} : |C| = |C_p|\}$. And can now compute how many standard deviations $agg(C_p)$ is away from the total average: $\delta(C_p) := |agg(C_p) - \mu|/\sigma(|C_p|)$. Assuming that $\{agg(C) : C \subseteq_r \mathcal{C} : |C| = |C_p|\}$ is normal distributed it holds that $\Pr[\delta(C) \geq 2] < 0.05$.

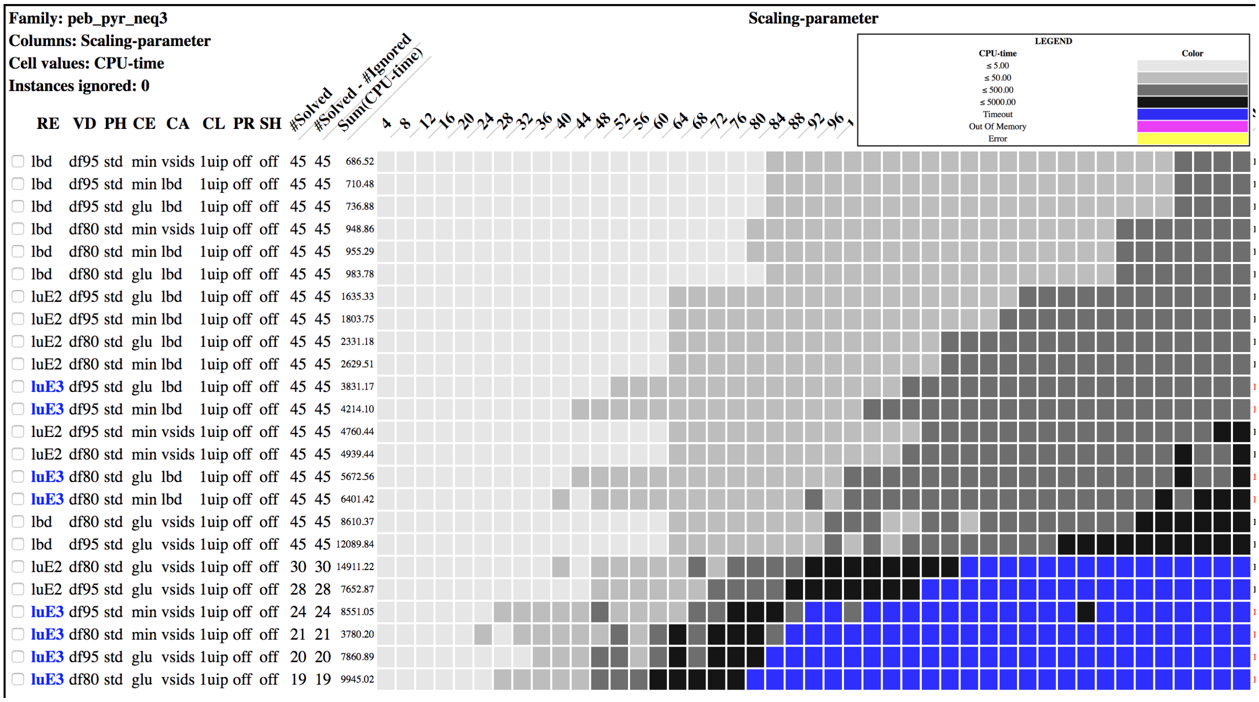
It might be that the aggregation contains very bad settings that let smaller differences for other parameters vanish. To counter this effect we consider the parameter, p.a. CE, with maximal $\delta(C_p)$ for a value p, p.a. CE:no, such that $agg(C_p)$ is below average. If $\delta(C_p) \geq 4$ then all values of that parameter, which are below average, are dropped. This is done by removing all configurations from $\mathcal{C}$ that contain such a value for the considered parameter.

A Technical Specification of Benchmark Encodings

In this section we give a precise specification of how our benchmark formulas are encoded in CNF. In what follows, we use the standard notation $[n] = \{1, 2, 3, \ldots, n\}$. For some of the technical descriptions it will also be useful to use the literal notation x^b, $b \in \{0, 1\}$, where $x^1 = x$ and $x^0 = \overline{x}$ (i.e., the literal x^b is satisfied by assigning $x = b$).

Unless explicitly specified otherwise below, all graphs $G = (V, E)$ are assumed to be undirected and have no multiple edges or self-loops. For such graphs we write $E(v)$ to denote the set of edges incident to the vertex v and $N(v)$ to denote the set of neigbours of v, i.e., the set of other endpoints of the edges in $E(v)$.

When G is specified to be a directed acyclic graph (DAG), then we write $\mathit{pred}(v)$ to denote the immediate



	RE	VD	PH	CE	CA	CL	PR	SH	#Solved	#Solved - #Ignored	Sum(CPU-time)
□	lbd	df95	std	min	vsids	1uip	off	off	45	45	686.52
□	lbd	df95	std	min	lbd	1uip	off	off	45	45	710.48
□	lbd	df95	std	glu	lbd	1uip	off	off	45	45	736.88
□	lbd	df80	std	min	vsids	1uip	off	off	45	45	948.86
□	lbd	df80	std	min	lbd	1uip	off	off	45	45	955.29
□	lbd	df80	std	glu	lbd	1uip	off	off	45	45	983.78
□	luE2	df95	std	glu	lbd	1uip	off	off	45	45	1635.33
□	luE2	df95	std	min	lbd	1uip	off	off	45	45	1803.75
□	luE2	df80	std	glu	lbd	1uip	off	off	45	45	2331.18
□	luE2	df80	std	min	lbd	1uip	off	off	45	45	2629.51
□	**luE3**	df95	std	glu	lbd	1uip	off	off	45	45	3831.17
□	**luE3**	df95	std	min	lbd	1uip	off	off	45	45	4214.10
□	luE2	df95	std	min	vsids	1uip	off	off	45	45	4760.44
□	luE2	df80	std	min	vsids	1uip	off	off	45	45	4939.44
□	**luE3**	df80	std	glu	lbd	1uip	off	off	45	45	5672.56
□	**luE3**	df80	std	min	lbd	1uip	off	off	45	45	6401.42
□	lbd	df80	std	glu	vsids	1uip	off	off	45	45	8610.37
□	lbd	df95	std	glu	vsids	1uip	off	off	45	45	12089.84
□	luE2	df80	std	glu	vsids	1uip	off	off	30	30	14911.22
□	luE2	df95	std	glu	vsids	1uip	off	off	28	28	7652.87
□	**luE3**	df95	std	min	vsids	1uip	off	off	24	24	8551.05
□	**luE3**	df80	std	min	vsids	1uip	off	off	21	21	3780.20
□	**luE3**	df95	std	glu	vsids	1uip	off	off	20	20	7860.89
□	**luE3**	df80	std	glu	vsids	1uip	off	off	19	19	9945.02

Figure 4: Example of heatmap on pebbling formulas on pyramid graphs, where it can be seen that adaptive restarts [RE:lbd] perform better than frequent restarts [RE:luE2].

predecessors of v, i.e., all vertices u for which there is an edge (u, v). We say that vertices without incoming edges are *sources* and that vertices without outgoing edges are *sinks*. The DAGs we consider will always have a unique single sink.

A.1 Tseitin Formulas

Given a graph $G = (V, E)$ and a function $\chi : V \to \{0, 1\}$ such that $\sum_{v \in V} \chi(v) \not\equiv 0 \pmod 2$, we introduce a variable x_e for every edge $e \in E$. For every vertex $v \in V$ we encode the parity constraint $\sum_{e \in E(v)} x_e \equiv \chi(v)$ $\pmod 2$ as the set of clauses

$$\left\{ \bigvee_{e \in E(v)} x_e^{1-b_e} \;\middle|\; \sum_{e \in E(v)} b_e \not\equiv \chi(v) \pmod 2 \right\} , \tag{A.1}$$

and the *Tseitin formula* over G consists of the union of all these clauses. See Figure 8 for a small example.

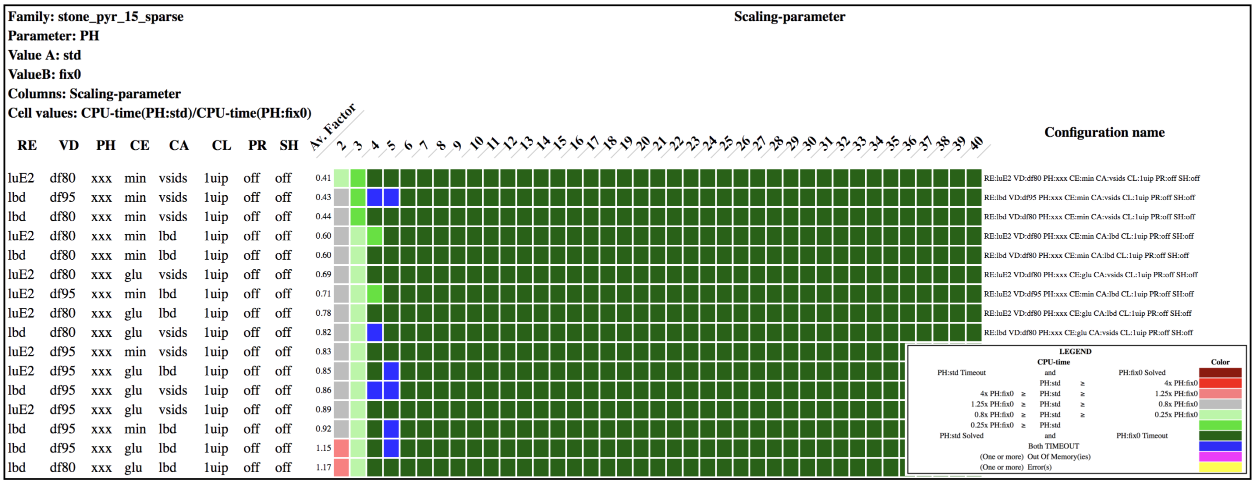
A.2 Ordering Principle Formulas

Ordering principle formulas have variables $x_{i,j}$ for $i, j \in [n]$, $i \neq j$, where any assignment to these variables is meant to specify an ordering on the set $\{e_1, \ldots, e_n\}$ under the interpretation that $x_{i,j}$ is true if e_i is smaller than e_j. The *partial ordering principle formulas* consist of the clauses

$$\overline{x}_{i,j} \vee \overline{x}_{j,i} \qquad \text{[anti-symmetry; not both } e_i < e_j \text{ and } e_j < e_i\text{]} \tag{A.2a}$$

$$\overline{x}_{i,j} \vee \overline{x}_{j,k} \vee x_{i,k} \qquad \text{[transitivity; } e_i < e_j \text{ and } e_j < e_k \text{ implies } e_i < e_k\text{]} \tag{A.2b}$$

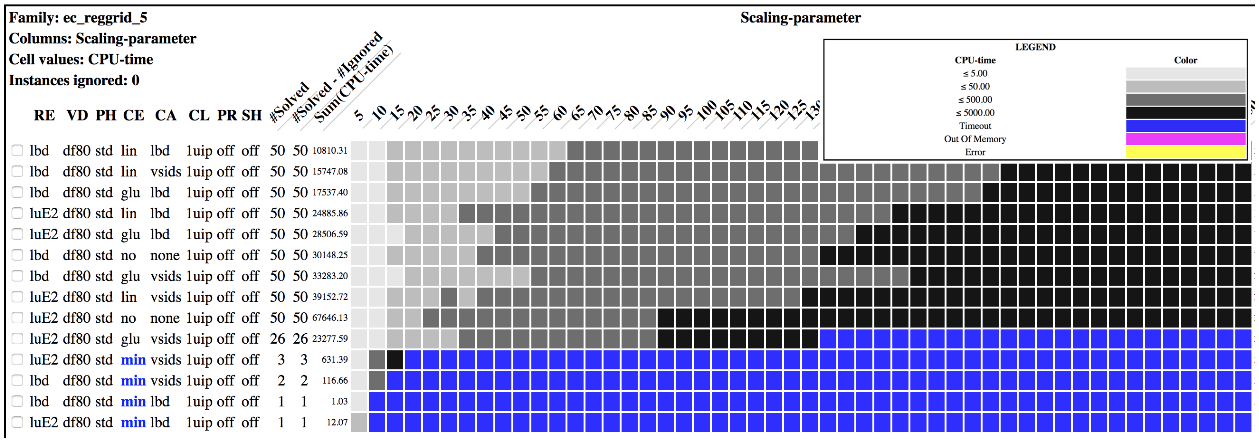
$$\bigvee\nolimits_{1 \leq i \leq n,\, i \neq j} x_{i,j} \qquad [e_j \text{ is not a minimal element}] \tag{A.2c}$$



Family: stone_pyr_15_sparse
Parameter: PH
Value A: std
ValueB: fix0
Columns: Scaling-parameter
Cell values: CPU-time(PH:std)/CPU-time(PH:fix0)

RE	VD	PH	CE	CA	CL	PR	SH	Av. Factor	Configuration name
luE2	df80	xxx	min	vsids	1uip	off	off	0.41	RE:luE2 VD:df80 PH:xxx CE:min CA:vsids CL:1uip PR:off SH:off
lbd	df95	xxx	min	vsids	1uip	off	off	0.43	RE:lbd VD:df95 PH:xxx CE:min CA:vsids CL:1uip PR:off SH:off
lbd	df80	xxx	min	vsids	1uip	off	off	0.44	RE:lbd VD:df80 PH:xxx CE:min CA:vsids CL:1uip PR:off SH:off
luE2	df80	xxx	min	lbd	1uip	off	off	0.60	RE:luE2 VD:df80 PH:xxx CE:min CA:lbd CL:1uip PR:off SH:off
lbd	df80	xxx	min	lbd	1uip	off	off	0.60	RE:lbd VD:df80 PH:xxx CE:min CA:lbd CL:1uip PR:off SH:off
luE2	df80	xxx	glu	vsids	1uip	off	off	0.69	RE:luE2 VD:df80 PH:xxx CE:glu CA:vsids CL:1uip PR:off SH:off
luE2	df95	xxx	min	lbd	1uip	off	off	0.71	RE:luE2 VD:df95 PH:xxx CE:min CA:lbd CL:1uip PR:off SH:off
luE2	df80	xxx	glu	lbd	1uip	off	off	0.78	RE:luE2 VD:df80 PH:xxx CE:glu CA:lbd CL:1uip PR:off SH:off
lbd	df80	xxx	glu	vsids	1uip	off	off	0.82	RE:lbd VD:df80 PH:xxx CE:glu CA:vsids CL:1uip PR:off SH:off
luE2	df95	xxx	min	vsids	1uip	off	off	0.83	
luE2	df95	xxx	glu	lbd	1uip	off	off	0.85	
lbd	df95	xxx	glu	vsids	1uip	off	off	0.86	
luE2	df95	xxx	glu	vsids	1uip	off	off	0.89	
lbd	df95	xxx	min	lbd	1uip	off	off	0.92	
lbd	df95	xxx	glu	lbd	1uip	off	off	1.15	
lbd	df80	xxx	glu	lbd	1uip	off	off	1.17	

Figure 5: Example of comparison plot on sparse stone formulas, where it can be seen that standard phase saving is crucial.



Family: ec_reggrid_5
Columns: Scaling-parameter
Cell values: CPU-time
Instances ignored: 0

	RE	VD	PH	CE	CA	CL	PR	SH	#Solved	#Solved - #Ignored	Sum(CPU-time)
□	lbd	df80	std	lin	lbd	1uip	off	off	50	50	10810.31
□	lbd	df80	std	lin	vsids	1uip	off	off	50	50	15747.08
□	lbd	df80	std	glu	lbd	1uip	off	off	50	50	17537.40
□	luE2	df80	std	lin	lbd	1uip	off	off	50	50	24885.86
□	luE2	df80	std	glu	lbd	1uip	off	off	50	50	28506.59
□	lbd	df80	std	no	none	1uip	off	off	50	50	30148.25
□	lbd	df80	std	glu	vsids	1uip	off	off	50	50	33283.20
□	luE2	df80	std	lin	vsids	1uip	off	off	50	50	39152.72
□	luE2	df80	std	no	none	1uip	off	off	50	50	67646.13
□	luE2	df80	std	glu	vsids	1uip	off	off	26	26	23277.59
□	luE2	df80	std	**min**	vsids	1uip	off	off	3	3	631.39
□	lbd	df80	std	**min**	vsids	1uip	off	off	2	2	116.66
□	lbd	df80	std	**min**	lbd	1uip	off	off	1	1	1.03
□	luE2	df80	std	**min**	lbd	1uip	off	off	1	1	12.07

Figure 6: Example of heatmap on even colouring formulas on regular grids, where it can be seen that a very aggressive clause erasure policy hurts the solver performance a lot.

for $i \neq j \neq k \neq i$ ranging over $1, \ldots, n$, and for the *total* or *linear ordering principle formulas* we also add clauses

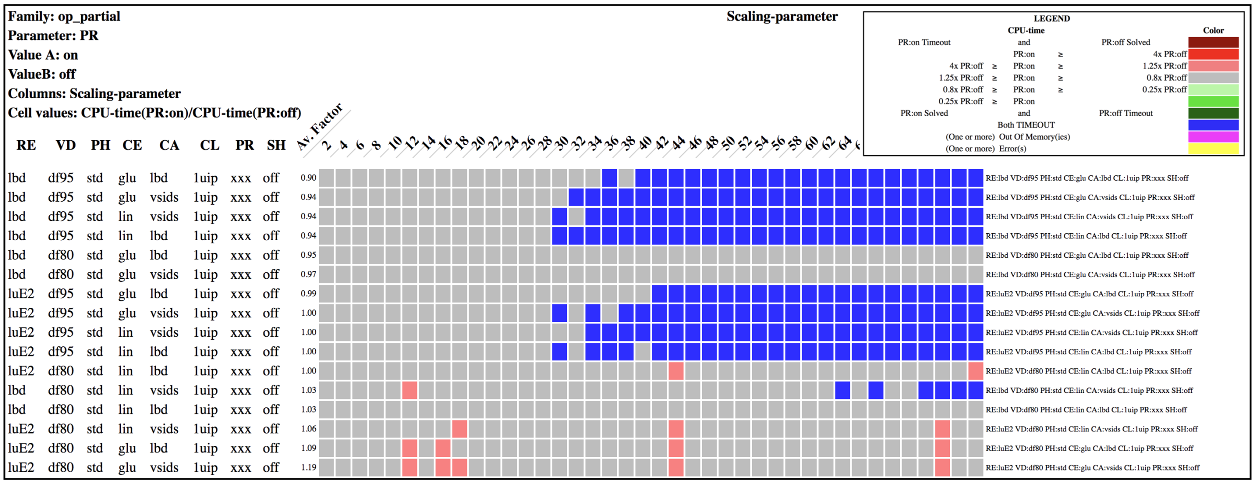
$$x_{i,j} \vee x_{j,i} \qquad \text{[totality; either } e_i < e_j \text{ or } e_j < e_i\text{]} \qquad \text{(A.2d)}$$

to specify that any two elements in the set are comparable.

A.3 Pebbling Formulas

Given a DAG G with source vertices S and a unique sink vertex z, and with all non-source vertices having fan-in 2, we identify every vertex $v \in V(G)$ with a propositional logic variable v. The *pebbling contradiction* over G, denoted Peb_G, is the conjunction of the following clauses:

- for all $s \in S$, a unit clause s (*source axioms*),
- For all non-source vertices w with immediate predecessors $pred(w) = \{u, v\}$, the clause $\overline{u} \vee \overline{v} \vee w$ (*pebbling axioms*),



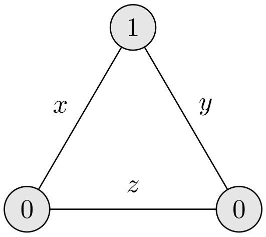
Family: op_partial
Parameter: PR
Value A: on
ValueB: off
Columns: Scaling-parameter
Cell values: CPU-time(PR:on)/CPU-time(PR:off)

RE	VD	PH	CE	CA	CL	PR	SH	Av. Factor	
lbd	df95	std	glu	lbd	1uip	xxx	off	0.90	RE:lbd VD:df95 PH:std CE:glu CA:lbd CL:1uip PR:xxx SH:off
lbd	df95	std	glu	vsids	1uip	xxx	off	0.94	RE:lbd VD:df95 PH:std CE:glu CA:vsids CL:1uip PR:xxx SH:off
lbd	df95	std	lin	vsids	1uip	xxx	off	0.94	RE:lbd VD:df95 PH:std CE:lin CA:vsids CL:1uip PR:xxx SH:off
lbd	df95	std	lin	lbd	1uip	xxx	off	0.94	RE:lbd VD:df95 PH:std CE:lin CA:lbd CL:1uip PR:xxx SH:off
lbd	df80	std	glu	lbd	1uip	xxx	off	0.95	RE:lbd VD:df80 PH:std CE:glu CA:lbd CL:1uip PR:xxx SH:off
lbd	df80	std	glu	vsids	1uip	xxx	off	0.97	RE:lbd VD:df80 PH:std CE:glu CA:vsids CL:1uip PR:xxx SH:off
luE2	df95	std	glu	lbd	1uip	xxx	off	0.99	RE:luE2 VD:df95 PH:std CE:glu CA:lbd CL:1uip PR:xxx SH:off
luE2	df95	std	glu	vsids	1uip	xxx	off	1.00	RE:luE2 VD:df95 PH:std CE:glu CA:vsids CL:1uip PR:xxx SH:off
luE2	df95	std	lin	vsids	1uip	xxx	off	1.00	RE:luE2 VD:df95 PH:std CE:lin CA:vsids CL:1uip PR:xxx SH:off
luE2	df95	std	lin	lbd	1uip	xxx	off	1.00	RE:luE2 VD:df95 PH:std CE:lin CA:lbd CL:1uip PR:xxx SH:off
luE2	df80	std	lin	lbd	1uip	xxx	off	1.00	RE:luE2 VD:df80 PH:std CE:lin CA:lbd CL:1uip PR:xxx SH:off
lbd	df80	std	lin	vsids	1uip	xxx	off	1.03	RE:lbd VD:df80 PH:std CE:lin CA:vsids CL:1uip PR:xxx SH:off
lbd	df80	std	lin	lbd	1uip	xxx	off	1.03	RE:lbd VD:df80 PH:std CE:lin CA:lbd CL:1uip PR:xxx SH:off
luE2	df80	std	lin	vsids	1uip	xxx	off	1.06	RE:luE2 VD:df80 PH:std CE:lin CA:vsids CL:1uip PR:xxx SH:off
luE2	df80	std	glu	lbd	1uip	xxx	off	1.09	RE:luE2 VD:df80 PH:std CE:glu CA:lbd CL:1uip PR:xxx SH:off
luE2	df80	std	glu	vsids	1uip	xxx	off	1.19	RE:luE2 VD:df80 PH:std CE:glu CA:vsids CL:1uip PR:xxx SH:off

LEGEND

	CPU-time			**Color**
PR:on Timeout	and		PR:off Solved	🟥
	PR:on	≥	4x PR:off	🟥
4x PR:off ≥	PR:on	≥	1.25x PR:off	🟥
1.25x PR:off ≥	PR:on	≥	0.8x PR:off	⬜
0.8x PR:off ≥	PR:on	≥	0.25x PR:off	🟩
0.25x PR:off ≥	PR:on			🟩
PR:on Solved	and		PR:off Timeout	🟩
	Both TIMEOUT			🟦
(One or more)	Out Of Memory(ies)			🟪
(One or more)	Error(s)			🟨

Figure 7: Example of comparison plot on ordering principle formulas with partial ordering, where it is compared [PR:off] against [PR:on], to show that preprocessing does not make almost any difference on this family.



(a) Triangle graph with odd labelling.

$$\begin{aligned} & (x \vee y) \\ \wedge\ & (\overline{x} \vee \overline{y}) \\ \wedge\ & (x \vee \overline{z}) \\ \wedge\ & (\overline{x} \vee z) \\ \wedge\ & (y \vee \overline{z}) \\ \wedge\ & (\overline{y} \vee z) \end{aligned}$$

(b) Corresponding Tseitin formula.

Figure 8: Example Tseitin formula.

- for the sink z, the unit clause $\overline{z}$ (*target* or *sink axiom*).

See Figure 9 for a small example.

Such pebbling formulas are uninteresting for experiments, however, since it is easy to see that they are immediately decided by unit propagation. We therefore make them (moderately) harder by substituting some suitable Boolean function $f(x_1, \ldots, x_d)$ over new variables $\{x_1, \ldots, x_d\} = \vec{x}$ for each variable x and expanding to get a new CNF formula. Every function $f(\vec{x})$ is equivalent to a CNF formula over $x_1, \ldots, x_d$ with at most 2^d clauses. We write $Cl[f_d(\vec{x})]$ to denote the set of clauses representing $f(\vec{x})$ and $Cl[\neg f_d(\vec{x})]$ to denote the set of clauses encoding the negation of f_d applied on $\vec{x}$. Just to give a simple illustration of this, if we want to do substitution with standard logical or of arity 2, then we have

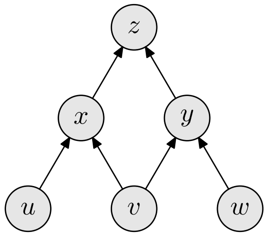
$$Cl[\vee_2(\vec{x})] = \{x_1 \vee x_2\} \quad \text{and} \quad Cl[\neg\vee_2(\vec{x})] = \{\overline{x}_1, \overline{x}_2\} \tag{A.3}$$

and for substitution with exclusive or of arity 2 we have

$$Cl[\oplus_2(\vec{x})] = \{x_1 \vee x_2, \overline{x}_1 \vee \overline{x}_2\} \quad \text{and} \quad Cl[\neg\oplus_2(\vec{x})] = \{x_1 \vee \overline{x}_2, \overline{x}_1 \vee x_2\} \ . \tag{A.4}$$

To obtain the substituted CNF formula, we replace every positive literal x by the set of clauses $Cl[f_d(\vec{x})]$ and every negative literal $\overline{y}$ by the set of clauses $Cl[\neg f_d(\vec{y})]$, and then apply the rewriting rule

$$F \vee G = \{C \vee D \,\big|\, C \in F,\ D \in G\} \tag{A.5}$$



(a) Pyramid graph Π_2 of height 2.

$$\begin{aligned}
& u \\
\land\ & v \\
\land\ & w \\
\land\ & (\overline{u} \lor \overline{v} \lor x) \\
\land\ & (\overline{v} \lor \overline{w} \lor y) \\
\land\ & (\overline{x} \lor \overline{y} \lor z) \\
\land\ & \overline{z}
\end{aligned}$$

(b) Pebbling contradiction Peb_{Π_2}.

Figure 9: Example pebbling contradiction for the pyramid of height 2.

repeatedly until the formula has been expanded out to CNF. Substituting with the functions (A.3) in the formula in Figure 9b yields the formula in Figure 10a and substitution with (A.4) yields the formula in Figure 10b.

For our experiments we do not use standard or $\lor$ or exclusive or $\oplus$, but in order to get more interesting results we instead use the not-all-equal function neq with arity 3, which is encoded in CNF as

$$Cl[\mathrm{neq}_3(\vec{x})] = \{x_1 \lor x_2, \lor x_3,\ \overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3\} \tag{A.6a}$$

and

$$Cl[\neg\mathrm{neq}_3(\vec{x})] = \{x_1 \lor \overline{x}_2,\ x_1 \lor \overline{x}_3,\ x_2 \lor \overline{x}_1,\ x_2 \lor \overline{x}_3,\ x_3 \lor \overline{x}_1,\ x_3 \lor \overline{x}_2\}\ . \tag{A.6b}$$

Applying this substitution to Figure 9b yields a formula that is a bit too large to fit nicely into an example, however, which is why we illustrated substituted pebbling formulas with the simpler functions above.

A.4 Stone Formulas

To generate *stone formulas*, we start with a DAG G over a vertex set V of size $|V| = n$ which has sources S and a unique sink vertex z, and where all non-source vertices have fan-in 2, as for the pebbling formulas in Section A.3. In contrast to pebbling formulas, however, we also have a set of stones or markers M of size $|M| = m$. For each vertex $v \in V$ we specify a set of available stones $N(v) \subseteq M$. We can represent this collection of sets compactly as a bipartite graph $H = (V \,\dot{\cup}\, M, E)$. For the standard stone formulas in the literature we have $m = 3n$ and $N(v) = M$ for all $v \in V$, so that H is the complete bipartite graph $K_{n,m}$, but for our experiments we will set m to be n or $n/2$ to get formula instances of manageable size. For the same reason we also generate formulas with marker specification bipartite graphs H of constant left degree. To the best of our knowledge, such *sparse stone formulas* have not been considered in the literature before.

Given a DAG G with sources S and sink z and a bipartite graph $H = (V \,\dot{\cup}\, M, E)$ as above, the stone formulas consist of the following clauses (where $p_{v,m}$ is true if marker m is on vertex v and r_m is true if marker m is red):

$$\bigvee_{m \in N(v)} p_{v,m} \qquad v \in V, \tag{A.7a}$$

$$\overline{p}_{s,m} \lor r_m \qquad s \in S, m \in N(s), \tag{A.7b}$$

$$\overline{p}_{z,m} \lor \overline{r}_m \qquad m \in N(z), \tag{A.7c}$$

$$\overline{p}_{u,m_u} \lor \overline{r}_{m_u} \lor \overline{p}_{v,m_v} \lor \overline{r}_{m_v} \lor \overline{p}_{w,m_w} \lor r_{m_w} \qquad \mathit{pred}(w) = \{u, v\},\ m_x \in N(x) \text{ for } x \in \{u, v, w\}. \tag{A.7d}$$

$$
\begin{aligned}
& (u_1 \lor u_2) \\
\land\ & (v_1 \lor v_2) \\
\land\ & (w_1 \lor w_2) \\
\land\ & (\overline{u}_1 \lor \overline{v}_1 \lor x_1 \lor x_2) \\
\land\ & (\overline{u}_1 \lor \overline{v}_2 \lor x_1 \lor x_2) \\
\land\ & (\overline{u}_2 \lor \overline{v}_1 \lor x_1 \lor x_2) \\
\land\ & (\overline{u}_2 \lor \overline{v}_2 \lor x_1 \lor x_2) \\
\land\ & (\overline{v}_1 \lor \overline{w}_1 \lor y_1 \lor y_2) \\
\land\ & (\overline{v}_1 \lor \overline{w}_2 \lor y_1 \lor y_2)
\end{aligned}
$$

$$
\begin{aligned}
\land\ & (\overline{v}_2 \lor \overline{w}_1 \lor y_1 \lor y_2) \\
\land\ & (\overline{v}_2 \lor \overline{w}_2 \lor y_1 \lor y_2) \\
\land\ & (\overline{x}_1 \lor \overline{y}_1 \lor z_1 \lor z_2) \\
\land\ & (\overline{x}_1 \lor \overline{y}_2 \lor z_1 \lor z_2) \\
\land\ & (\overline{x}_2 \lor \overline{y}_1 \lor z_1 \lor z_2) \\
\land\ & (\overline{x}_2 \lor \overline{y}_2 \lor z_1 \lor z_2) \\
\land\ & \overline{z}_1 \\
\land\ & \overline{z}_2
\end{aligned}
$$

(a) Substitution pebbling contradiction $Peb_{\Pi_2}[\lor_2]$ with respect to binary logical or.

$$
\begin{aligned}
& (u_1 \lor u_2) \\
\land\ & (\overline{u}_1 \lor \overline{u}_2) \\
\land\ & (v_1 \lor v_2) \\
\land\ & (\overline{v}_1 \lor \overline{v}_2) \\
\land\ & (w_1 \lor w_2) \\
\land\ & (\overline{w}_1 \lor \overline{w}_2) \\
\land\ & (u_1 \lor \overline{u}_2 \lor v_1 \lor \overline{v}_2 \lor x_1 \lor x_2) \\
\land\ & (u_1 \lor \overline{u}_2 \lor v_1 \lor \overline{v}_2 \lor \overline{x}_1 \lor \overline{x}_2) \\
\land\ & (u_1 \lor \overline{u}_2 \lor \overline{v}_1 \lor v_2 \lor x_1 \lor x_2) \\
\land\ & (u_1 \lor \overline{u}_2 \lor \overline{v}_1 \lor v_2 \lor \overline{x}_1 \lor \overline{x}_2) \\
\land\ & (\overline{u}_1 \lor u_2 \lor v_1 \lor \overline{v}_2 \lor x_1 \lor x_2) \\
\land\ & (\overline{u}_1 \lor u_2 \lor v_1 \lor \overline{v}_2 \lor \overline{x}_1 \lor \overline{x}_2) \\
\land\ & (\overline{u}_1 \lor u_2 \lor \overline{v}_1 \lor v_2 \lor x_1 \lor x_2) \\
\land\ & (\overline{u}_1 \lor u_2 \lor \overline{v}_1 \lor v_2 \lor \overline{x}_1 \lor \overline{x}_2) \\
\land\ & (v_1 \lor \overline{v}_2 \lor w_1 \lor \overline{w}_2 \lor y_1 \lor y_2) \\
\land\ & (v_1 \lor \overline{v}_2 \lor w_1 \lor \overline{w}_2 \lor \overline{y}_1 \lor \overline{y}_2)
\end{aligned}
$$

$$
\begin{aligned}
\land\ & (v_1 \lor \overline{v}_2 \lor \overline{w}_1 \lor w_2 \lor y_1 \lor y_2) \\
\land\ & (v_1 \lor \overline{v}_2 \lor \overline{w}_1 \lor w_2 \lor \overline{y}_1 \lor \overline{y}_2) \\
\land\ & (\overline{v}_1 \lor v_2 \lor w_1 \lor \overline{w}_2 \lor y_1 \lor y_2) \\
\land\ & (\overline{v}_1 \lor v_2 \lor w_1 \lor \overline{w}_2 \lor \overline{y}_1 \lor \overline{y}_2) \\
\land\ & (\overline{v}_1 \lor v_2 \lor \overline{w}_1 \lor w_2 \lor y_1 \lor y_2) \\
\land\ & (\overline{v}_1 \lor v_2 \lor \overline{w}_1 \lor w_2 \lor \overline{y}_1 \lor \overline{y}_2) \\
\land\ & (x_1 \lor \overline{x}_2 \lor y_1 \lor \overline{y}_2 \lor z_1 \lor z_2) \\
\land\ & (x_1 \lor \overline{x}_2 \lor y_1 \lor \overline{y}_2 \lor \overline{z}_1 \lor \overline{z}_2) \\
\land\ & (x_1 \lor \overline{x}_2 \lor \overline{y}_1 \lor y_2 \lor z_1 \lor z_2) \\
\land\ & (x_1 \lor \overline{x}_2 \lor \overline{y}_1 \lor y_2 \lor \overline{z}_1 \lor \overline{z}_2) \\
\land\ & (\overline{x}_1 \lor x_2 \lor y_1 \lor \overline{y}_2 \lor z_1 \lor z_2) \\
\land\ & (\overline{x}_1 \lor x_2 \lor y_1 \lor \overline{y}_2 \lor \overline{z}_1 \lor \overline{z}_2) \\
\land\ & (\overline{x}_1 \lor x_2 \lor \overline{y}_1 \lor y_2 \lor z_1 \lor z_2) \\
\land\ & (\overline{x}_1 \lor x_2 \lor \overline{y}_1 \lor y_2 \lor \overline{z}_1 \lor \overline{z}_2) \\
\land\ & (z_1 \lor \overline{z}_2) \\
\land\ & (\overline{z}_1 \lor z_2)
\end{aligned}
$$

(b) Substitution pebbling contradiction $Peb_{\Pi_2}[\oplus_2]$ with respect to binary exclusive or.

Figure 10: Examples of substitution pebbling formulas for the pyramid graph Π_2.

$$\begin{pmatrix}
1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1\\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1\\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1\\
1 & 0 & 1 & 0 & 0 & 0 & 1 & \mathbf{1} & 0 & 0 & 1
\end{pmatrix}$$

(a) Matrix encoding row and column constraints.

$$\begin{aligned}
&(x_{1,1} \lor x_{1,2} \lor x_{1,4})\\
\land\ &(x_{1,1} \lor x_{1,2} \lor x_{1,8})\\
\land\ &(x_{1,1} \lor x_{1,4} \lor x_{1,8})\\
\land\ &(x_{1,2} \lor x_{1,4} \lor x_{1,8})\\
&\vdots\\
\land\ &(\overline{x}_{4,11} \lor \overline{x}_{8,11} \lor \overline{x}_{10,11})\\
\land\ &(\overline{x}_{4,11} \lor \overline{x}_{8,11} \lor \overline{x}_{11,11})\\
\land\ &(\overline{x}_{4,11} \lor \overline{x}_{10,11} \lor \overline{x}_{11,11})\\
\land\ &(\overline{x}_{8,11} \lor \overline{x}_{10,11} \lor \overline{x}_{11,11})
\end{aligned}$$

(b) Example cardinality constraints in CNF.

Figure 11: Matrix and (fragment of) corresponding subset cardinality formula.

The clauses in (A.7a) say that every vertex has marker. Clauses (A.7b) say that sources have red markers and clauses (A.7c) that the sink has a blue marker. Finally, the clauses in (A.7d) specify that if both immediate predecessors u and v of a vertex w have blue markers, then w must also have a blue marker.

A.5 Subset Cardinality Formulas

We generate *subset cardinality formulas* from $n \times n$ $0/1$-matrices M with exactly $2k$ ones per row and column, except that we add an extra one somewhere so that one row and one column has $2k+1$ ones. In the formulas we generate we fix $k = 2$, so that all rows and columns have 4 ones except one row and column that have 5 ones, and this is how we describe the formulas below.

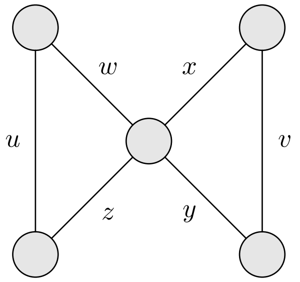
Recall that we identify positions $a_{i,j} = 1$ with variables $x_{i,j}$ and let $R_i = \{j \mid a_{i,j} = 1\}$ and $C_j = \{i \mid a_{i,j} = 1\}$. Then the "geq" version of the subset cardinality formula over M contains the following clauses:

- for every row i, the set of clauses $\left\{\bigvee_{j \in R^*} x_{i,j} \,\middle|\, R^* \subseteq R_i,\ |R^*| = 3\right\}$;
- for every column j, the set of clauses $\left\{\bigvee_{i \in C^*} \overline{x}_{i,j} \,\middle|\, C^* \subseteq C_j,\ |C^*| = 3\right\}$.

See Figure 11 for an example for the fixed bandwidth patterns 11010001 (with 1s in positions $1, 2, 4, 8$) that is one of the patterns used in our experiments, where in this example we have also added an extra one in the bottom row that is presented in bold face (the placement of the extra 1 in our actual generated instances is slightly different).

For the "eq" version of the formulas we also add clauses to encode the exact equalities $\sum_{j \in R_i} x_{i,j} = \lceil |R_i|/2 \rceil$ and $\sum_{i \in C_j} x_{i,j} = \lfloor |C_i|/2 \rfloor$, namely the additional clauses:

- for every row i with 4 ones, the set of clauses $\left\{\bigvee_{j \in R^*} \overline{x}_{i,j} \,\middle|\, R^* \subseteq R_i,\ |R^*| = 3\right\}$;
- for the row i' with 5 ones, the set of clauses $\left\{\bigvee_{j \in R^*} \overline{x}_{i',j} \,\middle|\, R^* \subseteq R_{i'},\ |R^*| = 4\right\}$;
- for every column j with 4 ones, the set of clauses $\left\{\bigvee_{i \in C^*} x_{i,j} \,\middle|\, C^* \subseteq C_j,\ |C^*| = 3\right\}$;
- for the column j' with 5 ones, the set of clauses $\left\{\bigvee_{i \in C^*} x_{i,j'} \,\middle|\, C^* \subseteq C_{j'},\ |C^*| = 4\right\}$.



(a) Graph with even vertex degrees.

$$
\begin{array}{ll}
(u \lor w) & \land\, (w \lor x \lor y) \\
\land\, (\overline{u} \lor \overline{w}) & \land\, (w \lor x \lor z) \\
\land\, (u \lor z) & \land\, (w \lor y \lor z) \\
\land\, (\overline{u} \lor \overline{z}) & \land\, (x \lor y \lor z) \\
\land\, (v \lor x) & \land\, (\overline{w} \lor \overline{x} \lor \overline{y}) \\
\land\, (\overline{v} \lor \overline{x}) & \land\, (\overline{w} \lor \overline{x} \lor \overline{z}) \\
\land\, (v \lor y) & \land\, (\overline{w} \lor \overline{y} \lor \overline{z}) \\
\land\, (\overline{v} \lor \overline{y}) & \land\, (\overline{x} \lor \overline{y} \lor \overline{z})
\end{array}
$$

(b) Corresponding CNF formula.

Figure 12: Example of even colouring formula (satisfiable instance).

A.6 Even Colouring Formulas

Let $G = (V, E)$ be a graph with all vertices of constant, even degree and with an odd total number of edges, and identify every edge $e \in E$ with a variable x_e. Then the *even colouring formula* for G consists of the constraint $\sum_{e \in E(v)} x_e = \deg(v)/2$ for every vertex $v \in V$, which we encode in CNF as

$$\left\{ \bigvee_{e \in E^*} x_e\,,\ \bigvee_{e \in E^*} \overline{x}_e, \;\middle|\; E^* \subseteq E(v),\ \left|E^*\right| = \frac{1}{2}\deg(v) + 1 \right\} \tag{A.8}$$

as illustrated in Figure 12 (though this particular instance in satisfiable, because we wanted to keep the graph and the example CNF formula as small as possible).

A.7 Relativized Pigeonhole Principle Formulas

Formally, the *relativized pigeonhole principle (RPHP) formulas* encode the contradictory statement that there are (partial) functions $p : [k] \to [n]$ and $q : [n] \to [k-1]$ such that p is one-to-one and defined on $[k]$, and q is one-to-one and defined on the range of p. The RPHP formula with parameters k and n has variables $p_{u,v}$ that encode the function p, $q_{v,w}$ that encode the function q, and r_v that encode a superset of the range of p. It consists of the following collection of clauses:

$$
\begin{array}{lll}
p_{u,1} \lor p_{u,2} \lor \cdots \lor p_{u,n} & u \in [k], & \text{(A.9a)} \\
\overline{p}_{u,v} \lor \overline{p}_{u',v} & u, u' \in [k],\, u \neq u',\, v \in [n], & \text{(A.9b)} \\
\overline{p}_{u,v} \lor r_v & u \in [k],\, v \in [n], & \text{(A.9c)} \\
\overline{r}_v \lor q_{v,1} \lor \cdots \lor q_{v,k-1} & v \in [n], & \text{(A.9d)} \\
\overline{r}_v \lor \overline{r}_{v'} \lor \overline{q}_{v,w} \lor \overline{q}_{v',w} & v, v' \in [n],\, v \neq v',\, w \in [k-1]. & \text{(A.9e)}
\end{array}
$$

The clauses in (A.9a)–(A.9b) say that p maps $[k]$ injectively into $[n]$; clauses (A.9c) encode the range of p; and clauses (A.9d)–(A.9e) force q to be defined and injective on this range.

A.8 Dominating Set Formulas

The *dominating set formula* for a graph $G = (V, E)$ with parameter $k \in \mathbb{N}^+$ encodes the claim that G has a dominating set of size at most k. We have variables x_v and $g_{v,i}$ for all $v \in V$ and $i \in [k]$, with the intended meaning that x_v is true if v is a member of the dominating set, and if so $g_{v,i}$ is true if v is the ith member of the dominating set. The formula consists of the following clauses:

$$x_v \lor \bigvee_{u \in N(v)} x_u \qquad v \in V, \tag{A.10a}$$

$$\overline{x}_v \lor \bigvee_{i \in [k]} g_{v,i} \qquad v \in V, \tag{A.10b}$$

$$\overline{x}_v \lor \overline{g}_{v,i} \lor \overline{g}_{v,j} \qquad v \in V,\, i,j \in [k],\, i \neq j \tag{A.10c}$$

$$\overline{x}_u \lor \overline{x}_v \lor \overline{g}_{u,i} \lor \overline{g}_{v,i} \qquad u,v \in V,\, u \neq v,\, i \in [k]. \tag{A.10d}$$

The clauses (A.10a) specify that each vertex is dominated. Clauses (A.10b) and (A.10c) say that the variables $g_{v,i}$ define a total map from $\{v \in V \mid x_v = 1\}$ to $[k]$. The clauses in (A.10d) then add the constraint that this mapping is injective on $\{v \in V \mid x_v = 1\}$ (which is possible only if G has a dominating set of size k).

A.9 Clique Formulas

The *clique formula* over a graph $G = (V, E)$ with parameter $k \in \mathbb{N}^+$ encodes the claim that G has a k-clique in the following way. The variables are $x_{i,v}$ for $i \in [k]$ and $v \in V$, where the intended meaning is that $x_{i,v}$ is true if vertex v is the ith member of the k-clique, and the clauses are

$$\textstyle\bigvee_{v \in V} x_{i,v} \qquad i \in [k]; \tag{A.11a}$$

$$\overline{x}_{i,u} \lor \overline{x}_{i,v} \qquad i \in [k] \text{ and } u,v \in V,\, u \neq v; \tag{A.11b}$$

$$\overline{x}_{i,u} \lor \overline{x}_{j,v} \qquad i,j \in [k],\, i \neq j, \text{ and } u,v \in V,\, (u,v) \notin E; \tag{A.11c}$$

where clauses (A.11a) and (A.11b) say that the ith member of the clique is some unique vertex $v_i \in V$ for $i \in [k]$ and clauses (A.11c) say that two vertices u and v cannot both be members of the clique if there is no edge between them.

References

[AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011. Preliminary version in *SAT '09*.

[AJPU07] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory of Computing*, 3(5):81–102, May 2007. Preliminary version in *STOC '02*.

[ALN16] Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. *ACM Transactions on Computational Logic*, 17(3):19:1–19:30, May 2016. Preliminary version in *CCC '14*.

[AMO15] Albert Atserias, Moritz Müller, and Sergi Oliva. Lower bounds for DNF-refutations of a relativized weak pigeonhole principle. *Journal of Symbolic Logic*, 80(2):450–476, June 2015. Preliminary version in *CCC '13*.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 399–404, July 2009.

[Ats15] Albert Atserias. Personal communication, 2015.

[BBI16] Paul Beame, Chris Beck, and Russell Impagliazzo. Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space. *SIAM Journal on Computing*, 45(4):1612–1645, August 2016. Preliminary version in *STOC '12*.

[BG01] María Luisa Bonet and Nicola Galesi. Optimality of size-width tradeoffs for resolution. *Computational Complexity*, 10(4):261–276, December 2001. Preliminary version in *FOCS '99*.

[BGL13] Olaf Beyersdorff, Nicola Galesi, and Massimo Lauria. Parameterized complexity of DPLL search procedures. *ACM Transactions on Computational Logic*, 14(3):20:1–20:21, August 2013. Preliminary version in *SAT '11*.

[BGLR12] Olaf Beyersdorff, Nicola Galesi, Massimo Lauria, and Alexander A. Razborov. Parameterized bounded-depth Frege is not optimal. *ACM Transactions on Computation Theory*, 4(3):7:1–7:16, September 2012. Preliminary version in *ICALP '11*.

[Bie16] Armin Biere. Personal communication, 2016.

[BIS07] Paul Beame, Russell Impagliazzo, and Ashish Sabharwal. The resolution complexity of independent sets and vertex covers in random graphs. *Computational Complexity*, 16(3):245–297, October 2007. Preliminary version in *CCC '01*.

[BIW04] Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, September 2004.

[BK14] Samuel R. Buss and Leszek Kołodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10(2):16:1–16:22, June 2014.

[BN08] Eli Ben-Sasson and Jakob Nordström. Short proofs may be spacious: An optimal separation of space and length in resolution. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*, pages 709–718, October 2008.

[BN11] Eli Ben-Sasson and Jakob Nordström. Understanding space in proof complexity: Separations and trade-offs via substitutions. In *Proceedings of the 2nd Symposium on Innovations in Computer Science (ICS '11)*, pages 401–416, January 2011.

[BNT13] Chris Beck, Jakob Nordström, and Bangsheng Tang. Some trade-off results for polynomial calculus. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC '13)*, pages 813–822, May 2013.

[BW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, March 2001. Preliminary version in *STOC '99*.

[CLNV15] Siu Man Chan, Massimo Lauria, Jakob Nordström, and Marc Vinyals. Hardness of approximation in PSPACE and separation results for pebble games (Extended abstract). In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS '15)*, pages 466–485, October 2015.

[CNF] CNFgen: Combinatorial benchmarks for SAT solvers. `https://github.com/MassimoLauria/cnfgen`.

[CS76] Stephen A. Cook and Ravi Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences*, 13(1):25–37, 1976. Preliminary version in *STOC '74*.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[Glu] The Glucose SAT solver. `http://www.labri.fr/perso/lsimon/glucose/`.

[JMNŽ12] Matti Järvisalo, Arie Matsliah, Jakob Nordström, and Stanislav Živný. Relating proof complexity measures and practical hardness of SAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 316–331. Springer, October 2012.

[Kri85] Balakrishnan Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, August 1985.

[Kul99] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), 1999.

[LENV17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. CNFgen: A generator of crafted benchmarks. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, August 2017.

[LGPC16a] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI '16)*, pages 3434–3440, 2016.

[LGPC16b] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, July 2016.

[Map] MapleSAT. `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`.

[Mar06] Klas Markström. Locality and hard SAT-instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):221–227, 2006.

[Min] The MiniSat page. `http://minisat.se/`.

[MN14] Mladen Mikša and Jakob Nordström. Long proofs of (seemingly) simple formulas. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 121–137. Springer, July 2014.

[MN15] Mladen Mikša and Jakob Nordström. A generalized method for proving polynomial calculus degree lower bounds. In *Proceedings of the 30th Annual Computational Complexity Conference (CCC '15)*, volume 33 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 467–487, June 2015.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD ’96*.

[Nor12] Jakob Nordström. On the relative strength of pebbling and resolution. *ACM Transactions on Computational Logic*, 13(2):16:1–16:43, April 2012. Preliminary version in *CCC ’10*.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, February 2011. Preliminary version in *CP ’09*.

[Spe10] Ivor Spence. sgen1: A generator of small but difficult satisfiability benchmarks. *Journal of Experimental Algorithmics*, 15:1.2:1–1.2:15, March 2010.

[Stå96] Gunnar Stålmarck. Short resolution proofs for a sequence of tricky formulas. *Acta Informatica*, 33(3):277–280, May 1996.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, January 1987.

[VS10] Allen Van Gelder and Ivor Spence. Zero-one designs produce small hard SAT instances. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT ’10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 388–397. Springer, July 2010.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD ’01)*, pages 279–285, November 2001.