

DAGSTER

A Parallel Structured SAT Solver Final Report Against Project Activities

MARK BURGESS*, CHARLES GRETTON,
JOSH MILTHORPE, MARSHALL CLIFTON, LUKE CROAK

November 16, 2022

ABSTRACT

We describe a novel solver for the Boolean satisfiability (SAT) problem for use in high-performance computing (HPC) environments. Our solver is hybrid, incorporating both systematic backtracking search and (stochastic) local search processes, and providing flexibility regarding the specific arrangement used for a given search exercise. Our objectives are twofold: (i) engineer a SAT solver that seamlessly scales to very large CNF formulae, and (ii) enable practitioners to solve relatively challenging problems in HPC environments by efficiently distributing search effort across computing cores. Our solver takes as input a SAT problem and a decomposition of it into parts, with each part a conjunctive normal form formula corresponding to an abstract subproblem connected together in a directed acyclic graph (DAG). As search progresses and primitive subproblem solutions are computed, our solver uses the labelled DAG to schedule subsequent search activities for execution on the available processing resources. Through the creation of this tool and subsequent demonstrating experiments, we are able to show how our tool meets our objectives in the context of the various problems. Additionally we outline several demonstration runs of DAGSTER to indicate specific qualities and functions.

All Authors: *School of Computing, Australian National University, Canberra, Australia.*

*Corresponding Author: mark.burgess@anu.edu.au

CONTENTS

1	Introduction	3
2	Background	5
3	Decomposing SAT Problems	6
3.1	The ‘Divide and Conquer’ Approach	6
3.2	The ‘Divide and Unify’ Approach	7
4	DAGSTER Components	9
4.1	Conflict-Driven Clause Learning Solver	9
4.2	Stochastic Local Search	10
4.3	Clause Simplification	14
5	DAGSTER in more detail	15
5.1	DAG File Specification	15
5.2	DAG Decompositions and Parallel Search with DAGSTER	16
5.3	Scheduling Search Processes	17
6	Empirical Studies	20
6.1	Model Counting in Parallel - Hard Satisfiable Random Formulae	20
6.2	A Large Satisfiable Problem - Pentominoes	21
6.3	Counting Models in Parallel - Costas Arrays	25
6.4	Accelerating Unsatisfiability Proofs via Parallelisation	27
6.5	Model Counting in Parallel - Easy Large Satisfiable Random Formulae	29
6.6	Model Counting in Parallel - Easy Connected Satisfiable Random Formulae	30
6.7	Decomposing Software Verification Problems for DAGSTER	32
7	Conclusion and Future Work	35
8	Assets used in Empirical Studies	36

LIST OF FIGURES

Figure 1	CDCL branching process with SLS depths	14
Figure 2	Hand-worked CNF and DAG file examples	16
Figure 3	Messages between components in DAGSTER	19
Figure 4	An example DAG for conjoined small-hard problems	21
Figure 5	Runtime relative acceleration of parallel small-hard problems	21
Figure 6	Absolute execution time wrt. number small-hard subproblems	22
Figure 7	A Pentomino puzzle featured on youtube channel ‘Cracking the Cryptic’	22
Figure 8	An example 3x3 pentomino superproblem	23
Figure 9	An example DAG for a 4x4 pentomino superproblem	23
Figure 10	Runtime performance for Pentomino problems	24
Figure 11	Runtime performance for Pentomino problems with and without strengthener	24
Figure 12	A simplified DAG structure for the Costas decomposition	26
Figure 13	Runtime to solve Costas arrays	26
Figure 14	Runtime performance on Costas problems with/without strengthener	27
Figure 15	Scatterplot of TiniSat against Dagster for random unsatisfiable 3Sat instances	28
Figure 16	DAGSTER memory consumption for conjunctions of large-easy problems against core count.	29
Figure 17	DAG for 7 conjoined random-5-SAT problems	30
Figure 18	Runtime performance for random 5-SAT formulas	31
Figure 19	Example code in C.	33
Figure 20	Conversion of code into SSA form.	33

Figure 21	Resultant formula from conversion.	34
Figure 22	DAG for example code.	34
Figure 23	3 independent conditionals.	34
Figure 24	DAG for the problem with 3 independent conditionals.	34

LIST OF TABLES

Table 1	Process table of CNF subproblems solved by cubes	7
Table 2	Solving CNF subproblems independently and merging solutions	9
Table 3	Solving CNF subproblems in sequence	9
Table 4	An example 6x6 Costas array	25
Table 5	The displacements of the example 6x6 Costas array 4, organised by horizontal distance in rows, by horizontal offset in columns	25
Table 6	The runtimes (in seconds) for evaluated solvers on CNFs associated with Independent AES programs.	35
Table 7	CPU information of the Etna machine	36
Table 8	CPU information of the Whale machine	36
Table 9	CPU information of the Goedel machine	37
Table 10	CPU information of the Gadi machine	37
Table 11	CPU information of the Luke machine	37

1 INTRODUCTION

This report describes the design and experimental evaluation of DAGSTER, a Boolean satisfiability problem (SAT) solver we have built to solve difficult and/or large SAT problems using HPC (High Performance Computing) infrastructure. DAGSTER takes as input a sequence of disjunctive logical clauses defining a SAT problem and also a graph structure that describes its components (or subproblems) and defines the way in which they should be solved in parallel and/or in sequence. In this way information about how to solve a given problem in a distributed computing environment is specified. DAGSTER proceeds by solving independent subproblems in parallel in an HPC environment, so that eventually the satisfiability of the entire SAT problem is determined. In the case that the SAT problem is satisfiable, our DAGSTER tool can also be used to count the number of satisfying valuations efficiently using the parallel computing environment.

The SAT problem of interest and its compositional structure of subproblems is described according to a labelled Directed Acyclic Graph (DAG) which is input into the program. The labelled vertices of the DAG represent distinct subproblems and each edge between represents a set of variables that is shared between two subproblems. At each edge, the variable assignments that are present in solutions generated by the preceding subproblems are used to constrain the search in subsequent subproblems. Where multiple paths exist through the DAG these paths can be processed in parallel. Additionally, any subproblem can potentially generate multiple solutions that place different constraints on subsequent subproblems, which can then also be executed in parallel; thus there are multiple means of instantiating parallel search within DAGSTER. A detailed explanation of the structure of the relevant DAGs and how they relate to SAT subproblems is provided the subsequent sections.

In summary, the way the search is distributed across multiple processes is described by the DAG structure, with partial valuations and sets of logically consistent partial valuations determining what search tasks need to be scheduled. This approach provides several benefits:

- DAGSTER is designed to take advantage of known substructure within SAT problems, with different logical components of the problem solved in parallel.
- Search can take advantage of distributed processing cores with minimal overhead.
- Large problems do not need to be handled in-memory at-once, allowing for calculation of solutions to SAT problems too big to fit in RAM.

There are also several weaknesses of the approach to be aware of:

- The performance of the search is highly sensitive to the quality of the provided DAG problem decomposition, and finding a ‘good’ decomposition is not necessarily easy.
- The parallel infrastructure that the tool provides has some computational overhead, such that the tool shows worse performance compared to serial solves on some very small and/or easy problems.

In this report we outline some of the features of the DAGSTER system:

- A simple input file format for DAG decomposition, which is used in addition to a standard DIMACS CNF representation of the SAT problem.
- Support for solving CNF problems which are larger than available system memory.

- An optional hybrid solving mode between CDCL—particularly the search implemented in the `TINISAT` system—and SLS—particularly derived from the `cNOVELTY+` system—allowing `DAGSTER` to combine the benefits of these varied procedures.
- An optimal parallel clause strengthening module, actively simplifying learnt clauses within the CDCL routine.
- A distributed solver framework using MPI for communication between processes, including a Master scheduler with optional modes tailored to enumerating all solutions of a SAT problem, or racing to a first solution.
- An optional Binary Decision Diagram (BDD) module storing and compiling knowledge about the solutions to subproblems. This can minimise computational and communication overhead where there might otherwise be combinatorial explosion in memory.
- A testing suite for verification and validation of the correct execution of these functions.

Within this document, we detail the background, design, and evaluation of `DAGSTER` in the following sections:

- Section 2 gives a brief background to the SAT problem and the state of SAT solving processes;
- Section 3 gives an outline of two conceptual ways of decomposing SAT problems into parts for parallel computation, and indicates that `DAGSTER` is a tool which can handle both.
- Section 4 describes the components of the `DAGSTER` system in isolation, introducing each with some background in turn.
- Section 5 describes how these subsystems communicate together to solve a problem, and specify the input/output of the `DAGSTER` program that results.
- Section 6 considers some sources of DAG decomposed SAT problems, and introduces a range of `DAGSTER` experiments with their results, particularly:
 - Section 6.1: parallel solution of small hard problems.
 - Section 6.2: parallel solution of larger hard problems, particularly Pentomino problems.
 - Section 6.3: a divide-and-conquer solution to the Costas array problem.
 - Section 6.4: a two part decomposition of random SAT problems to accelerate their solution process, with SLS assistance.
 - Section 6.5: solving easy large problems in parallel, demonstrating how `DAGSTER` can solve problems in sequence where an explicit representation of that set of problems exceeds the computer memory.
 - Section 6.6: connected easy-5-SAT problems, showing that SLS assistance can speed up solution process.
 - Section 6.7: software verification problems.

2 BACKGROUND

Modern satisfiability solving has a long history of development and research, and currently there are many well respected and powerful solvers which can tackle many distinct problems from a range of disciplines. The Boolean satisfiability problem (SAT) is the problem of determining whether a formula in propositional logic is *satisfiable*; in other words, whether there exists an assignment of the propositional variables under which the formula evaluates to *true*. If no such assignment exists, we say the formula is *unsatisfiable*. SAT problems are often specified in conjunctive normal form (CNF), where the problem is rendered as being a conjunction over a disjunctions of literals (or a conjunction over ‘clauses’, where a literal is a variable, negated or not). The related #SAT problem is that of counting the number of distinct satisfying assignments associated with a formula. The Boolean SAT problem is the canonical problem of the NP-complete class [?, ?]. This report outlines the development and functionality of the DAGSTER tool which supports solving both the SAT and #SAT problems.

There are two dominant paradigms for solving SAT problems using serial processing: (i) *stochastic local search* (SLS) such as investigated in [?, ?, ?], and (ii) the systematic backtracking search of *conflict-driven clause learning* (CDCL), as studied in [?, ?]. The latter are descendants of the Davis, Putnam, Logemann, and Loveland (DPLL) procedure [?, ?]. Both local and systematic searches operate by computing a satisfying valuation should one exists. Systematic searches are sound and complete procedures – I.e., they are guaranteed to find a satisfying assignment should one exist, and if they declare that none exists, that declaration is valid. In case the formula at hand is unsatisfiable, some modern structured SAT solvers are able to emit UNSAT proof certificates in a canonical format [?]. Conversely, Local search procedures are sound but not complete. These treat the SAT problem as an optimisation problem in which the objective is to compute an assignment that minimises the number of unsatisfied Boolean clauses. If a satisfying assignment is reported, then it is valid, and the formula at hand is satisfiable. However, because local search procedures aren’t systematic in their search, they are unable to prove that a formula is unsatisfiable.

In modern systematic SAT solvers, the implemented search procedure is augmented with additional rules and dynamics, such as restarts, sophisticated heuristics regarding what variables and values to explore during search and mechanisms to acquire succinct representations of acquired/learnt knowledge about the problem at hand. Notably, most of these procedures are sequential, and not typically designed to be parallelised across multiple compute cores. Additionally, there is no obvious way to benefit substantially from parallel computing environments in the case of local search procedures. Some challenges of accelerating and scaling SAT solving in parallel computing environments are discussed in [?]. Parallelism can be exploited in the tuning of algorithm configurations [?], *search-space splitting*, or *portfolio approaches* [?, ?, ?]. And in these cases, identifying and sharing good clauses is key to efficient parallelism [?], however, [?] identify bottlenecks in the structure of resolution refutations which place hard limits on parallelism in search-space splitting for CDCL solvers. Finally, in case the domain of interest is *bounded model checking* (BMC), see [?], parallelisation is raised as a topic when considering the search horizon at which to pose queries [?, ?]. How to leverage parallelisation in this setting has not been explored satisfactorily in the existing literature.

In our work, we pursue scaling and accelerating SAT solving in HPC environments supposing we have a decomposition of the problem for parallel satisfiability. We typically break the SAT formula itself into different subproblems, such that a conjunction of subproblems corresponds to the original problem itself. Solving each of the subproblems and resolving any differences between the solutions to the parts of the problem results in a solution to the original problem. Our open-source solver is called DAGSTER and includes state-of-the-art components including SLS report-

ing and recommendations, BDD simplification between subproblems, and clause simplifying, and subproblem scheduling techniques. The effectiveness of the decomposition of the problem into subproblems determines the effectiveness of the resultant search procedure, and hence the effectiveness of the solver. However, for problems that are naturally decomposed into subproblems our solver can exceed the walltime performance of sequential solvers, as the results in this report will demonstrate.

3 DECOMPOSING SAT PROBLEMS

The fundamental problem of breaking a SAT problem into subproblems for parallel computing involves a couple of considerations, particularly what structure the subproblems should have and how they should be solved together. We focus on two uniquely distinct processes of breaking down a problem, which we broadly call the ‘divide and conquer’ approach, and the ‘divide and unify’ approach. In light of this we consider building a tool which can be adapted to handle both approaches.

The ‘divide-and-conquer’ approach partitions the search space into disjoint sets by constraining some of the variable values; each of these subspaces are then searched in parallel. In this way, a solution to any of the parts is a solution to the original problem, and therefore the original problem is satisfied the moment that any process finds a solution. Alternatively the ‘divide and unify’ approach breaks the problem apart by considering different subsets of the clauses that define the original problem. Because each part has a subset of the clauses of the original problem, it is thus under-constrained, and hence a unification of the solutions of the parts together defines the solutions of the original problem.

These approaches define distinct approaches to SAT parallelisation.

3.1 The ‘Divide and Conquer’ Approach

Various authors have experimented with the process of breaking SAT problems into subproblems arranged in various structures. One of the most direct ways of breaking a SAT problem into subproblems is to consider the resulting subproblems after certain variables have been assigned. If a single variable is assigned to be one way, then the choice of values over the remaining variables reduces the dimension of the search space. This resulting search space is also disjoint from the resulting search space if the variable was assigned the other way. These subproblems can also potentially be broken down into yet smaller subproblems by selection of further variables, and those subproblems will also be disjoint in the same way. This process of generating subproblems we call the ‘divide-and-conquer’ strategy to parallel SAT solving [?], as different problems will be guaranteed to be searching for solutions in different and disjoint parts of the problem search space. One noteworthy example of this process is the so-called ‘cube-and-conquer’ algorithm [?] which breaks the problem into many subproblems by first using a ‘lookahead’ process [?] and breaking the search into subproblems at different junctions of that search. In the ‘divide-and-conquer’ technique, each subproblem encodes a portion of the search space which is then searched in parallel. In this scheme, a solution to any subproblem is also a solution to the original problem, as each subproblem is describing a problem that is more constrained than the original problem.

These ideas can be made concrete using a simple example.

Example 1. Let f be the following CNF formula over 4 propositional variables.

$$(\neg 1 \vee 2 \vee 3) \wedge (1 \vee \neg 2 \vee \neg 3) \wedge (2 \vee \neg 3) \wedge (3 \vee \neg 4) \wedge (2 \vee \neg 3 \vee \neg 4) \wedge (\neg 2 \vee 3 \vee 4)$$

The set of 3 possible satisfying assignments to f is:

$$\{1 \wedge 2 \wedge 3 \wedge 4; 1 \wedge 2 \wedge 3 \wedge \neg 4; \neg 1 \wedge \neg 2 \wedge \neg 3 \wedge \neg 4\}$$

The set of clauses in the formula f set can be considered after the following 4 disjoint assignments (or ‘cubes’) of variables 1 and 4: $1 \wedge 4$, $1 \wedge \neg 4$, $\neg 1 \wedge 4$, and $\neg 1 \wedge \neg 4$. For example, the solutions associated with cube $1 \wedge 4$ are a satisfying assignment to $f \wedge 1 \wedge 4$. In Table 1 we tabulate the solutions of f associated with each cube. Note that the cubes induce a partition of the satisfying assignments of f .

■

The ‘divide-and-conquer’ approach is identified as being quite effective at parallel SAT solving, particularly with hard or UNSAT problems where there is no easy structure inherent to the problem that can be exploited. However, where there does exist inherent structure to the problem, another kind of decomposition may be useful. Particularly where a SAT problem is decomposable into multiple distinct parts which could be solved together; and this is the inspiration for the ‘divide-and-unify’ approach.

	Cubes			
	$1 \wedge 4$	$1 \wedge \neg 4$	$\neg 1 \wedge 4$	$\neg 1 \wedge \neg 4$
Clauses	$2 \vee 3$	$2 \vee 3$	$\neg 2 \vee \neg 3$	$\neg 2 \vee \neg 3$
	$2 \vee \neg 3$	$2 \vee \neg 3$	$2 \vee \neg 3$	$2 \vee \neg 3$
	3		3	
	$2 \vee \neg 3$	$\neg 2 \vee 3$	$2 \vee \neg 3$	$\neg 2 \vee 3$
Solutions	$1 \wedge 2 \wedge 3 \wedge 4$	$1 \wedge 2 \wedge 3 \wedge \neg 4$	\emptyset	$\neg 1 \wedge \neg 2 \wedge \neg 3 \wedge \neg 4$

Table 1: An instance of a problem from Example 1 is decomposed into four independent parts based on the possible assignments of variables 1 and 4 in the ‘divide-and-conquer’ approach. Each independent part has a reduced clause set and possible solutions, and the solutions of each part match the solutions of the full problem.

3.2 The ‘Divide and Unify’ Approach

In the ‘divide-and-unify’ approach, each subproblem is less-constrained than the original problem; a union of the subproblem clauses/solutions solves the original problem. Thus, there are more solutions to the subproblems than there are solutions to the whole problem itself.

One possible way of solving subproblems together is to run a process to generate all possible solutions to each subproblem, and then afterwards go through a process of enumerating compatible subproblem solutions together to resolve solutions that satisfy the original problem. This process of solving each subproblem independently and then merging their solutions is one way of solving many subproblems together, and a simple example of this is shown in Table 2. In Table 2 we have two subproblems (part1 and part2), that are composed of separate clauses on the first row. The possible solutions to each of the subproblems are shown on the second row, the merger of compatible pairs of solutions of these subproblems is shown on the bottom row. Note that each of the merged solutions on the bottom row is exhaustive in satisfying all the clauses of both the subproblems together.

Another way of solving subproblems together is that the solutions of one subproblem can be fed as constraints into the solving procedure of a second subproblem. In this way the solutions of the second subproblem will tacitly be compatible with (at least one) solution of the first subproblem. It is obviously quite possible to string this process together in a line, in this way the n^{th} subproblem will be solved with the constraints that satisfy a solution of the $(n - 1)^{\text{th}}$ subproblem, which was solved with constraints that satisfy a solution of the $(n - 2)^{\text{th}}$ subproblem, and so on. In this manner, if there is a solution to the final subproblem in such a line, then it

will satisfy all the subproblems and hence be a valid solution to the original problem. This line between subproblems is just one example of a directed acyclic graph (DAG) between subproblems.

A simple example of this is shown in Table 3. In Table 3 are two subproblems (similar to that shown in Table 2), but the relevant variables to subproblem 1 are fed as constraints into subproblem 2. Here, the subproblem 1 has four unique solutions, however only the variables 2 and 3 are relevant to the second subproblem, and the solutions from subproblem 2 in response to the possible values of 2 and 3 from subproblem 1 are shown in the second column; note that this process creates the same outputs as shown in Table 2, hence a complete solution set over both the subproblems.

By these two ways, we can see techniques of solving subproblems together (all independently, and in a sequence) and these are extreme examples of possible ways of solving subproblems together. However, it is easy to imagine alternatives, for instance, if there are three subproblems, it is possible to solve the first two independently and resolve their solutions together before feeding those resolved solutions as constraints into the third subproblem. In such a way, it is possible to consider the structure of the ordering of subproblems into a directed acyclic graph – i.e., an arbitrary DAG.

This raises the question of when and where it is preferable to solve subproblems independently or sequentially. Particularly it is seen that it is preferable to solve subproblems independently when the number of variables shared between the subproblems is small, such that the solutions of one subproblem are largely independent of those generated for the other. Conversely it is better to arrange subproblems in a sequence where there is a large variable overlap between them, such that the solutions of one subproblem are expected to constrain the possible compatible solutions of the other subproblem.

In light of this flexibility, we developed a DAG file format to specify the possible decomposition of the SAT problem to be used in a solving routine as given in Section 5.1.

It is also worth pointing out that there are several explicit examples of the ‘divide-and-unify’ approach used in wider literature. One simple instance of this approach is considered in [?] where the ‘Joining and model Checking scheme’ or JaCk-SAT, where the set of all variables is divided into two subsets of similar size, and the set of clauses is divided into three groups, one which features the variables of one variable subset, one which features the variables of the other subset, and one which features the variables of both subsets. JaCk-SAT then proceeds by generating the first and second clause sets, before resolving the solutions together with the third clause set.

Another example of our SAT decomposition is found in [?] which considers a much more general tree-based decomposition of problem into multiple parts based on a hypergraph of the relationship between variables and clauses; they define and describe their tree-based decomposition and consider a DPLL inspired solver for different subproblems. An alternative approach to subproblem generation and solving is given in [?] where they decompose a SAT problem into parts based on clause/variable ratio, that are solved in a series where the solutions of one part of the problem are passed on to the next part of the problem as a constraint; this process directly mirrors our elucidation in this section.

	part1	part2
Clauses	$\neg 1 \vee 2 \vee 3$ $1 \vee \neg 2 \vee \neg 3$ $2 \vee \neg 3$	$3 \vee \neg 4$ $2 \vee \neg 3 \vee \neg 4$ $\neg 2 \vee 3 \vee 4$
Solutions	$1 \wedge 2 \wedge 3$ $1 \wedge 2 \wedge \neg 3$ $\neg 1 \wedge 2 \wedge \neg 3$ $\neg 1 \wedge \neg 2 \wedge \neg 3$	$2 \wedge 3 \wedge 4$ $2 \wedge 3 \wedge \neg 4$ $\neg 2 \wedge 3 \wedge \neg 4$ $\neg 2 \wedge \neg 3 \wedge \neg 4$
Merged Solutions	$1 \wedge 2 \wedge 3 \wedge 4$ $1 \wedge 2 \wedge 3 \wedge \neg 4$ $\neg 1 \wedge \neg 2 \wedge \neg 3 \wedge \neg 4$	

Table 2: An instance of a problem decomposed into two parts with different clauses, the solution to both subproblems is the merger of compatible pairs of solutions of the subproblems

	part1	part2
Clauses	$\neg 1 \vee 2 \vee 3$ $1 \vee \neg 2 \vee \neg 3$ $2 \vee \neg 3$	$3 \vee \neg 4$ $2 \vee \neg 3 \vee \neg 4$ $\neg 2 \vee 3 \vee 4$
Solutions	$1 \wedge 2 \wedge 3 \Rightarrow$ $1 \wedge 2 \wedge \neg 3 \Rightarrow$ $\neg 1 \wedge 2 \wedge \neg 3 \Rightarrow$ $\neg 1 \wedge \neg 2 \wedge \neg 3 \Rightarrow$	$1 \wedge 2 \wedge 3 \wedge 4$ $1 \wedge 2 \wedge 3 \wedge \neg 4$ \emptyset $\neg 1 \wedge \neg 2 \wedge \neg 3 \wedge \neg 4$

Table 3: An instance of a problem decomposed into two parts with different clauses, the solutions to the first part constrain the solutions generated by the second (blue indicates redundant literals into the second part)

4 DAGSTER COMPONENTS

4.1 Conflict-Driven Clause Learning Solver

A central part of the DAGSTER program is the choice of SAT solver that generates solutions to any subproblem, and there is a wide range of potential SAT solving algorithms available for this purpose. The canonical approach is the conflict-driven clause learning (CDCL) backtracking search. Within this class, there exist a range of options regarding the choice of clause-learning process, heuristics for variable selection, and restart schedules. We took the choices embedded in the GPL licenced `TINISAT` solver coded by Huang et al. [?] as our starting point.

In the backtracking process, when a contradiction between variable assignments and clauses is reached, it is possible for the procedure to learn additional information about the ‘reason’ why the conflict occurred and to append an additional clause to the problem to avoid the same conflict at different points in the search. In this way, each additional clause learnt reduces the size of the remaining search space.

CDCL solvers utilise clause learning processes, and the 1-UIP procedure is almost universally employed, although there are other possible clause learning algorithms which can potentially be chosen [?, ?]. The basis of many of the clause learning processes is well documented and involve creating a ‘cut’ in the graph of implications that directly lead to the occurrence of a conflict. A cut in the implication graph (sometimes called the ‘I-graph’) divides the conflict itself (as a conflicting implication) on the one side, from a set of intermediary variable valuations which directly lead to it, on the other. These variable assignments are collected, and then a clause is added to the CDCL procedure to avoid these variable assignments in future; particularly the variable values are negated and then added as an additional disjunctive clause in the CDCL procedure. In this way CDCL procedures have the

ability to learn new clauses with each conflict encountered, which raises questions about how these many clauses are managed; where many SAT solvers have heuristics to identify and keep the most valuable clauses.

CDCL solvers also differ depending on what heuristic is used to select the variables on which the process branches. A particularly famous heuristic is called VSIDS, which was introduced with the CHAFF solver [?]. The VSIDS selection rule assigns each variable with a score called the *activity* (initially zero) and then increments each variable's activity by 1 (the additive 'bump') each time it appears (or is associated with) the learning of a conflict clause. Then at regular intervals in the solving procedure, the activity of all variables is multiplied by some constant $0 < \alpha < 1$ (called the multiplicative *decay factor*), the VSIDS heuristic selects variables with the highest activity for branching. The VSIDS heuristic is just one example of a popular SAT heuristic algorithm, with some investigation into why it is effective in practice [?]. Different heuristics are possible, and for DAGSTER we considered an entirely different process to variable selection guided by Stochastic Local Search, with VSIDS as a fallback, as considered in Section 4.2.2.

Another consideration in the selection of a CDCL procedure is in relation to the restart schedule employed. CDCL procedures scan through the problem space in a branching process in order to find satisfying solutions to the problem. However it is identified that it is advantageous for the CDCL procedure to occasionally restart itself to avoid being locked in unproductive areas of the problem space. A restart in CDCL resets the values of all the variables and starts a fresh branching process with the assistance of all of the learnt clauses it has previously derived. The hope is that the restarted CDCL process will then use its learnt knowledge to quickly come to a solution in a different part of the search space. CDCL restarts are an effective component of solving relevant and industrial problems, and the choice of different restart procedures can be instrumental [?]. There also exists the possible effectiveness of partial restarts and choice of different restart schedules etc. [?] For DAGSTER we utilised the Luby restart policy [?] inbuilt into the TINTSAT solver coded by Huang et al. [?].

4.2 Stochastic Local Search

In order to solve different subproblems in an effective way, different solution methodologies need to be employed, and there is a particularly interesting division between backtracking search (such as CDCL) and those searches which don't use backtracking, particularly stochastic local search (SLS). Stochastic local search procedures have been the topic of investigation for SAT problems since at least the early 90's, when [?, ?] introduced stochastic local hill-climbing procedures which outperformed more traditional backtracking procedures at the time. From this time there has been much interest, development and experimentation with these procedures which can be loosely described using the pseudocode in Algorithm 1.

Algorithm 1: loose pseudocode of SLS procedure

Input: CNF formula Φ , max steps m

Output: satisfying assignment of Φ or nosolution.

```

for  $i \leftarrow 1$  to  $m$  do
   $s \leftarrow \text{initAssign}(\Phi)$ ;
  if  $s$  satisfies  $\Phi$  then
    return  $s$ 
  else
     $x \leftarrow \text{chooseVariable}(\Phi, s)$ ;
     $s \leftarrow s$  with truth of  $x$  flipped;
  end
end
return no solution

```

From this pseudocode, it is seen that the SLS algorithm is primarily dependent on what method is used to choose the variable to flip in each iteration, and there exist a range of possibilities. One of the very first SLS algorithms was GSAT [?] where `initAssign` function is a randomisation of all problem variables, and `chooseVariable` method selects a variable which, if flipped, would minimise the number of clauses of the CNF Φ that are unsatisfied (and random selection between variables which are equally good). This GSAT algorithm encodes the classic ‘greedy’ SLS step, of trying to select a variable that greedily attempts to satisfy CNF clauses.

From inspection, there are several things worth noting about SLS procedures, particularly that SLS procedures can fail to find satisfying assignments in reasonable time as they are fundamentally stochastic in their performance. Furthermore, it is quite apparent that SLS procedures (particularly GSAT) may become caught in ‘local minima’ of the search space, where the search procedure enters into loops of variable flips and/or becomes locked into a set of assignments which satisfy some but not all clauses of the CNF [?].

Over time there have been multiple variants of the SLS procedure with different elements in the algorithm:

- The incorporation of algorithm restarts, where periodically the algorithm is restarted with some randomisation with the intention that it may avoid being trapped in the same sets of local minimum, and potentially find global minima (satisfying all clauses); restarts are a feature in the original GSAT procedure, and most others.
- A probabilistic random step in an iteration adjacent to the greedy step. This is most notably embodied in the ‘GSAT with random walk’ or GWSAT algorithm [?] where, every step with probability p a randomly selected variable from an unsatisfied clause is flipped.
- There is the possibility of adding a bias to search towards variables that are least recently flipped, such as historically encoded in the HSAT algorithm [?] where the variable that was least recently selected is selected among equally greedy candidates.
- By incorporating a Tabu into the search that prohibits flipping of recently flipped variables, such as encoded in the TSAT algorithm [?]
- By introducing a ‘walk’ step, which is similar to the random step, in that a random unsatisfied clause is selected, and a variable of that clause is chosen to be flipped. Particularly the variable can be selected in a greedy manner (to minimise the total clauses unsatisfied) or such as to minimise breaking existing satisfied clauses, or to prioritise those flips which ensure that no existing satisfied clause is made unsatisfied [?]. see [?].
- By incorporating clause weighting techniques, such as to make persistently unsatisfied clauses more important in the selection process and bias the search away from local minima. For instance, each time step, an unsatisfied clause can increase its weight linearly or multiplicatively [?].

Over time, many different SLS procedures have been developed and implemented, and many of their elements are compatible with each other. Particularly we implemented the `GNOVELTY+` algorithm [?], as shown in Algorithm 2. In this algorithm a variable is *promising* if its flip satisfies more clauses than it breaks, and more promising the greater/lesser the difference. Also, the adaptive noise update step is

inherited from AdaptNovelty+ [?], and the clause weight update scheme is inherited from the PAWS algorithm [?].

Algorithm 2: GNOVELTY+ algorithm without Tabu

Input: CNF formula Φ , max steps m , probabilities p, s, w

Output: satisfying assignment of Φ or nosolution.

Initialise the weight of each clause to 1;

randomly generate an assignment A ;

for $i \leftarrow 1$ **to** m **do**

if A satisfies Φ **then**

return A

else

if within a walking probability w_p **then**

 randomly select a variable x that appears in an unsatisfied clause;

else if If there exist promising variables **then**

 greedily select promising variable x , breaking ties by selecting the least recently flipped;

else

 greedily select the most promising variable x from a random unsatisfied clause c , breaking ties by selecting least recently flipped;

if x is the most recently flipped variable in c AND within a noise probability p **then**

 re-select x as the second most promising variable in c

 update the weights of unsatisfied clauses;

 with probability $s p$ smooth the weights of all *weighted* clauses;

end

 update A with the flipped value of x ;

 adaptively adjust the noise probability p ;

end

end

return *no solution*

4.2.1 Stochastic Local Search - Solution Reporting

One of the primary differences between SLS and CDCL is the types of steps that SLS can perform, which CDCL cannot. Particularly the *greedy* step—in which a variable is flipped that directly increases the number of clauses satisfied even if it makes other clauses unsatisfied—is an example of a step which is available in SLS which is not part of CDCL. The greedy step is not incorporated into the CDCL procedure primarily as it is not easily compatible with a stack-based backtracking process that maps uniquely over the search space.

Steps such as the greedy step make the search space under SLS much more connected, and thus it is more likely that the SLS will race to a solution more quickly than CDCL. The primary cost of this quality is that SLS does not do backtracking, and therefore will also potentially fail to find any solution and/or fail to terminate if the original problem is UNSAT. Whereas by contrast, the CDCL procedure will methodologically scan through the entire solution space, and can therefore be implemented to report the entire set of solutions to the problem. In this way, SLS is understood to be generally more effective at non-deterministically finding solutions to particular classes of problems, whereas CDCL is more effective at deterministically finding solutions in other classes of problems, and particularly problems that are UNSAT.

A trivial way of integrating the advantages of SLS and CDCL in solving a problem is to run them both in parallel (i.e., in a portfolio arrangement), and detect which finishes first in finding a solution. This integration inherits the advantages of both algorithms, and DAGSTER was programmed with this flexibility in mind.

Particularly in Section 5.3 we identify a mode of DAGSTER’s operation where each CDCL process is run alongside k other SLS processes, and the solutions which these processes find are recorded and accounted for in real time.

However, another way of integrating the advantages of SLS and CDCL is in using the information gathered in the SLS process to assisting the CDCL procedure, and *vice versa*, in making better decisions about what part of the search space to explore.

4.2.2 Stochastic Local Search - Suggestions

One of the ways in which SLS can assist in the CDCL procedure is in the selection of what variables the CDCL procedure should branch upon.

Within CDCL procedures, there is a choice of appropriate rules/heuristics which are used to select variables, and VSIDS is an example of a classic heuristic - see Section 4.1. VSIDS attempts to determine and preferably select the most influential variables in the learnt clauses of the problem, however other heuristics are possible.

SLS processes tend to occasionally get stuck in local minima or otherwise directly solve the problem by encountering a global minimum with all clauses satisfied - see Section 4.2. Insofar as the SLS process actually solves the problem, trivially this solution should be accounted for and the CDCL process should be updated to avoid resolving the same solution. However, even if the SLS only manages to get stuck in a local minimum it is still potentially of interest for the SLS to direct the search of the CDCL process towards these local minima so that the CDCL procedure can eliminate them from its search. After eliminating a local minimum communicated by an SLS instance, the CDCL process can then also potentially direct that SLS process away from that local minimum, to search for other minima. In this way the CDCL procedure can detect and direct the SLS away from the local minimum which it finds, and the SLS can direct the CDCL procedure towards fruitful (or potentially fruitful) areas of the search space; and thus the SLS and the CDCL procedure can mutually assist each other in finding a solution.

The variable suggestions which the SLS gives to CDCL procedure will potentially be relevant depending on where in the search space the CDCL search is, particularly what variables it has assigned and what depth it is at in its branching process. The depth and the variables assigned by the CDCL also constrain the SLS search for local minima within that remaining space and thus determine the appropriateness of the suggestions it offers to back to the CDCL.

As the SLS and the CDCL procedures execute, it is advantageous for the CDCL to advise the SLS process of the variables which the CDCL has assigned (i.e. its *prefix*) and then take variable suggestions from the SLS, after the SLS has had some time to resolve into a local minimum. Thus, the SLS is constrained to search in a solution space that is consistent with the most recent assignment prefix it has received from the CDCL process. The SLS keeps a buffer of variable assignments that are most likely to characterise the local minimum in which it is stuck, which it then sends to the CDCL procedure as variable suggestions.

In our DAGSTER system, a CDCL search is coupled to a pool of local searches, and communicates with them in a *round robin* fashion, sending new prefixes when it encounters evenly-spaced depths in its decision tree. When the CDCL procedure backtracks, it communicates with and takes advice from the local search that is most relevant to the backtrack depth.

An illustration of this schema is shown in Figure 1, where we have illustrated the branching process of a CDCL procedure, with historically investigated branches shown in grey. The active branch is shown in black. Conflicts that the CDCL procedure has encountered on the active branch are shown as crosses. Horizontal blue lines indicate where each of 4 different local searches have received a prefix corresponding to the decisions on the branch up to that line.

The use of assisting SLS processes to report solutions and provide suggestions to the CDCL procedure has been identified with increased performance of the CDCL

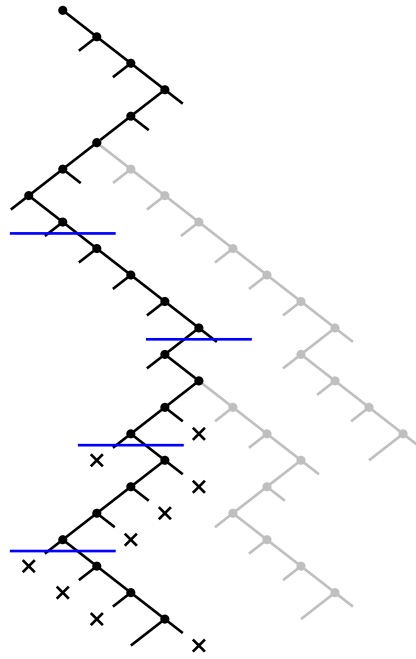


Figure 1: Illustration of CDCL branching process with SLS suggestion processes working on various branch cuts at various depths (blue)

procedure, but the optimal configuration may be problem-dependent. DAGSTER supports configuration parameters for the number of SLS processes per CDCL process, the size of the suggestion buffer being provided to the CDCL procedure, and the search depth interval at which the SLS processes should be allocated.

4.3 Clause Simplification

In the CDCL procedure new clauses are learnt and added to the SAT problem when the solver reaches a conflict. Depending on the problem, the CDCL procedure can unfortunately become slowed as it becomes bloated with storing and considering new conflict clauses. In the worst case, the number of conflict clauses generated will be exponential [?, ?]. Thus it is advantageous to involve a management process of purging or simplifying these learnt clauses.

Some heuristics and measures have been investigated to judge the quality of conflict clauses in order to determine and subsequently purge the least valuable clauses [?, ?]. These measures score the value of the learnt clauses and the solving procedure periodically purges a proportion of the lowest scoring conflict clauses. The frequency of the purging events and the proportion of clauses that are purged are important to the efficiency of the solver. (For instance, well-known SAT solvers MiniSAT and Glucose delete half of the learnt clauses at each purge event.)

In addition to dropping irrelevant or weak clauses, there is the question of when and/or how learnt clauses might be simplified, and different techniques have been proposed for this purpose [?]. One of the most basic approaches to simplifying learnt clauses is clause ‘minimisation’ or ‘strengthening’, where different techniques are used to identify and remove redundant literals in the learnt clauses.[?]

One of the more basic approaches of identifying and removing redundant literals from learnt clauses is considered by [?] where learnt clauses are passed to a separate strengthener process where any redundant variables are identified and removed by a separate unit-propagating SAT solving process, before being passed back and reintegrated into the original CDCL procedure. Particularly the procedure is inspired by the “vivification” clause simplification procedure [?] where for any learnt clause $c = \{l_0, l_1, \dots\}$ each of the literals l_i are in turn assigned to be false, and a unit-

propagation SAT procedure is run to determine if the resulting problem is UNSAT. if the resulting problem is determined to be UNSAT, then any literals not assigned to be false are identified to be redundant and can be removed.

Minimising and simplifying conflict clauses improves the performance of the CDCL process, particularly as smaller clauses are smaller in memory and enhance the likelihood of unit propagation. However, this clause minimisation process consumes computing power and thus competes with the computing resources that might otherwise be allocated to CDCL itself. One of the most natural ways of avoiding this tradeoff is to perform the minimisation process in parallel to the CDCL on a separate compute core (such as implemented by [?]). It has also been shown that clause minimisation can be effective even in serial with the CDCL procedure as well.[?]

Within the design of DAGSTER, it is considered as an option whether or not to have a separate clause simplification process in conjunction with the CDCL procedure. Particularly, we incorporated code developed by the authors of [?] the reducing process called MINIREP/GLUCORED.¹

5 DAGSTER IN MORE DETAIL

The specification and relationship between subproblems and how they are connected is specified in a DAG file, which is then input into the solver which then attempts to solve each subproblem in an effective way.

5.1 DAG File Specification

The specification of what parts of a SAT problem should be sectioned into subproblems and what variables should be passed between these subproblems is specified in a DAG file. An example of a DAG file is shown next to its associated DIMACS CNF file in Figure 2, where the DAG file has a header and multiple sections.

1. The DAG file begins with a header "DAG-FILE"
2. then the next line identifies the number of nodes (or subproblems) in the dag, in this case 4 nodes indexed 0, 1, 2, 3.
3. The next part begins with a header "GRAPH:", and then identifies what links there exist between the nodes one per line, and what variables get communicated along those links. For instance, the first link is between node 0 and node 2, where the only variables of interest in the solutions of node 0 that get passed to node 2 are the values of variables 1, 3, as the variable 2 in the clauses of node 0 no longer belong to the rest of the program. Other links include: from node 1 to 2 and 2 to 3, where variables 3, 4 and 4 get passed respectively.
4. Then there is a section (beginning with "CLAUSES:") which identifies what clauses belong to which nodes (or subproblems), the clauses are indexed in the order that they appear in the CNF. The example indicates that clauses indexed 0 and 1 belong to node 0 (ie. the clauses $2 \vee 1, \neg 2 \vee 3$) etc.
5. and finally there is a "REPORTING:" section, which identifies what variables we are interested in a satisfying valuation. In our example we are reporting variables 4, 6, 7.

In this example program it is possible to walk-through the execution of the DAGSTER program:

- We consider the possible solutions from node 0 (with clauses $2 \vee 1, \neg 2 \vee 3$) as $1 \wedge \neg 2 \wedge 3, \neg 1 \wedge 2 \wedge 3, 1 \wedge 2 \wedge 3, 1 \wedge \neg 2 \wedge \neg 3$, however since variable 2 is irrelevant to the rest of the problem then the messages that get passed to node 2 are: $1 \wedge 3, 1 \wedge \neg 3, \neg 1 \wedge 3$.

¹ Bronze winner of international SAT competition 2013 open track competition

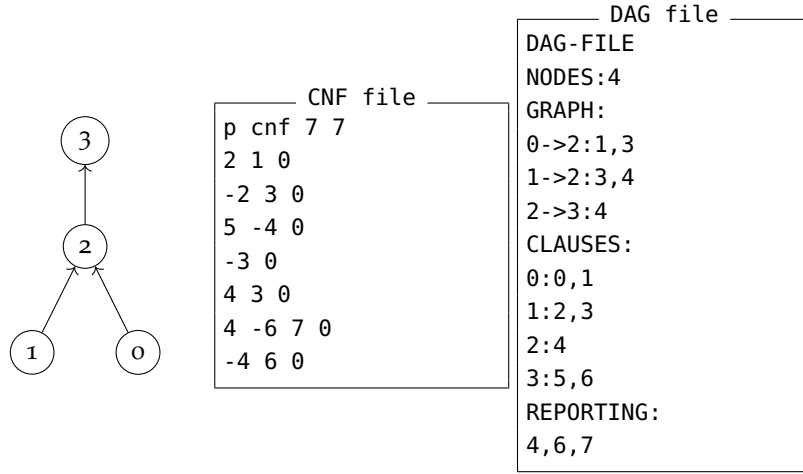


Figure 2: Example DAGSTER inputs of a CNF and DAG file. The directed graph described in the DAG file is depicted graphically on the left-hand side.

- We consider the possible solutions from node 1 (with clauses $5 \vee \neg 4, \neg 3$) as $\neg 3 \wedge 5 \wedge 4, \neg 3 \wedge 5 \wedge \neg 4, \neg 3 \wedge \neg 5 \wedge \neg 4$, however since variable 5 is irrelevant to the rest of the problem then the messages that get passed to node 2 are: $\neg 3 \wedge 4, \neg 3 \wedge \neg 4$.
- We now consider the resolution between the messages from nodes 0 and 1 into node 2, as $1 \wedge \neg 3 \wedge 4, 1 \wedge \neg 3 \wedge \neg 4$.
- We now consider the output from node 2 (with clauses $4 \vee 3$) to the inputs given to it resolved from nodes 0 and 1, in this case input $1 \wedge \neg 3 \wedge 4$ satisfies the clause and is a solution, however since variables 1 and 4 are discarded, the message 4 is passed onto node 3; the input $1 \wedge \neg 3 \wedge \neg 4$ does not satisfy the node's clauses and has no solution resulting in no other messages passed onto node 3.
- We now consider the output from node 3 (with clauses $4 \vee \neg 6 \vee 7, \neg 4 \vee 6$) to the inputs given to it from node 2, in this case the input 4 is given, and for this input the solutions $4 \vee 6 \vee \neg 7$ and $4 \vee 6 \vee 7$ are its output, which in turn are the solution of the entire problem; and those values are reported as output.

It is also possible to confirm that these outputs are appropriate solutions to the original CNF in Figure 2 where the possible solutions (with all variables included) of the original CNF are $1 \wedge \neg 2 \wedge \neg 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7$ and $1 \wedge \neg 2 \wedge \neg 3 \wedge 4 \wedge 5 \wedge 6 \wedge \neg 7$.

This minimal worked example provides a concrete illustration of the essential components of the DAGSTER algorithm, particularly the importance of an algorithm to generate solutions for a specific node, and the method of resolving solutions together for input into another node.

5.2 DAG Decompositions and Parallel Search with DAGSTER

To aid understanding, it may be helpful to provide an informal illustration of the kinds of problems and how they might best be related to an appropriate DAG. The motivation of the DAGSTER tool was to solve planning-type problems and this is a natural illustration of the kinds of problems which DAGSTER is designed to solve.

Particularly we might understand a DAG structure between nodes 0, 1, 2, 3 as might be graphed in Figure 2. In this figure, we might consider that nodes 0 and 1 are parts of the problem that are naturally solved in parallel and are predominantly independent of each other. The intersection of their solutions are then passed to node 2, whose outputs are then passed as constraints to node 3 whereby we might imagine nodes 1 and 2 are problems which are naturally solved in sequence.

An illustration of the kind of problem that these nodes might correspond to is parts of a planning procedure, particularly we might imagine a procedure of travelling

to another country being composed of subtasks which bear relations between them. We might imagine that nodes 0 and 1 relate to the relatively independent subtasks of how to pack your luggage, and problem of choosing modes of transport to the airport. In this way the problems of nodes 0 and 1 relate to subproblems that are relatively independent, in that the way you pack your luggage would (most often) be rather independent of the possible ways that you could navigate to the airport. The intersection of solutions to nodes 0 and 1 feeds into node 2, which may be the subtask of choosing an airline flight, which would depend on the result of a choice of transport (node 1) and be dependent on the way in which the luggage was packed (output from node 0); perhaps the transport constrains your arrival time at the airport, and the luggage packing constrains your flight by luggage weight and dimensional restrictions. Consequently the output of node 2, informs which flight you can take, and subsequently constrains the solutions of node 3 which might be the subtask of choosing your in-flight meal.

Although this illustration is simplistic, it illustrates the fact that problems can have a natural structure and how this might correspond to elements of a DAG decomposition of the problem.

Some general directions are:

1. In a sequence it is better for more constrained problems to occur first, in that the output from the more constrained problem should bind the solutions from the less constrained problem
2. subproblems that are mostly (or entirely) independent should occur in parallel
3. partitions between the clauses that define the nodes of the problem should correspond with meaningful subproblems - which are somewhat difficult (not trivially easy), generate few solutions (as opposed to voluminously many) and naturally constrain further subproblems.

5.2.1 Where can I find DAGs?

There are a range of schemes from the literature which provide recipes for computing a decomposition given the object to be decomposed [?, ?]. We also note that lookahead mechanisms, for example as described for cube and conquer approaches to Boolean SAT [?], provide a decomposition that can be represented using a DAG schematic. However the most natural source of DAGs for specific problems arises from considering the structure that is naturally inherent and recognised in the problem itself; rather than generated via recipe.

An example class of problem that has always been a focus of this initiative is bounded model checking of safety properties. In this setting, a range of literature has already contemplated and described dependency structures between problem variables that can be expressed using a DAG. In particular we have the concept of a *dependency graph*, also sometimes called *causal graph* [?, ?]. A dependency graph expresses dependencies between problem variables in terms of the ability of one variable to impact the value of another in time – I.e., in a discrete event system setting. For some classes of problems such graphs expose small subproblems with few interconnections, with the subproblems' size independent of parameters determining the overall problem size. These graphs can be computed quickly (in linear time), and provided exactly the schematic we desire to represent a larger overall problem about a transition system model in terms of a series of abstract subproblems and their refinements.

5.3 Scheduling Search Processes

Fundamentally DAGSTER is a parallel SAT solver, and the compute cores allocated to DAGSTER are each given a specific role. DAGSTER obeys a Master/Worker dichotomy, where the Master keeps track of the work that needs to be done and

allocates appropriate work to the Workers, who report back to Master the results of their computations.

The first and most natural description of the process occurs from the Master's perspective. The Master coordinates what workers should be working on what parts of the DAG. When DAGSTER begins it sets the workers onto the initial leaf nodes of the DAG (for instance nodes 0 and 1 in Figure 2) and these workers will gradually generate solutions for those nodes, the Master will then record these solutions and from them generate new messages that will be given to workers to initiate and constrain work on further nodes (for instance node 2). In this way, solutions to earlier nodes seed computational runs on further nodes, and the Master process coordinates and keeps track of all the work that is yet to be done and has been done. It does this through a compact representation of the logic of these solutions embedded in Binary Decision Diagrams (BDDs) particularly using the CUDD library² for BDD manipulation and generation.

DAGSTER is allowed to run in multiple 'modes' identifying the resources which the workers are allocated. Worker processes are grouped in worker groups, which are given work from Master. In the simplest operation of DAGSTER, mode $m = 0$, each worker group is a single core running a CDCL process, and when the Master passes a message to the worker process it loads the CNF associated with the node it is to work on. Any solutions generated by the CDCL search are then reported back to Master and serve to seed computation on further nodes. When the CDCL procedure has finished generating all solutions for a message it then requests further work from the Master process.

In more complicated operational mode $m = 1$, each worker group is a single core running a CDCL process and an additional k processes running the SLS gNOVELTY+ procedure. The Master gives the CDCL process the message identifying what node of the DAG it is to be working on and what constraints it is to have, and the CDCL procedure passes the information to the SLS processes which then initialise and process the problem as well. The gNOVELTY+ processes give variable suggestions to the CDCL procedure as well as report solutions as they discover them in conjunction to the CDCL procedure. As before, when the CDCL procedure has verified that all solutions have been generated, it stops the work of the SLS processes and requests further work from the Master.

In a yet more complicated operational mode, $m = 2$, each worker group is a single running CDCL process, an additional k SLS processes, and an additional clause strengthener process. As before, the Master's message is given to the CDCL procedure, the SLS(s) and the strengthener process, and the strengthener process assists the CDCL procedure by strengthening the conflict clauses that the CDCL procedure generates.

There is yet another mode $m = 3$ where each Worker is a single CDCL process with an associated clause strengthener process.

In these three operational modes $m = 0, 1, 2, 3$ each worker group has different numbers of processes in a worker group that is visible to the Master process and which the Master interacts with

1. Mode zero: each worker group is a single CDCL process (particularly TIN-SAT)
2. Mode one: each worker group is composed of a single CDCL process, and k SLS processes
3. Mode two: each worker group is composed of a single CDCL process, assisted by another strengthener process, and k SLS processes
4. Mode three: each worker group is composed of a single CDCL process, assisted by another strengthener process

² see: <https://davidkebo.com/cudd>

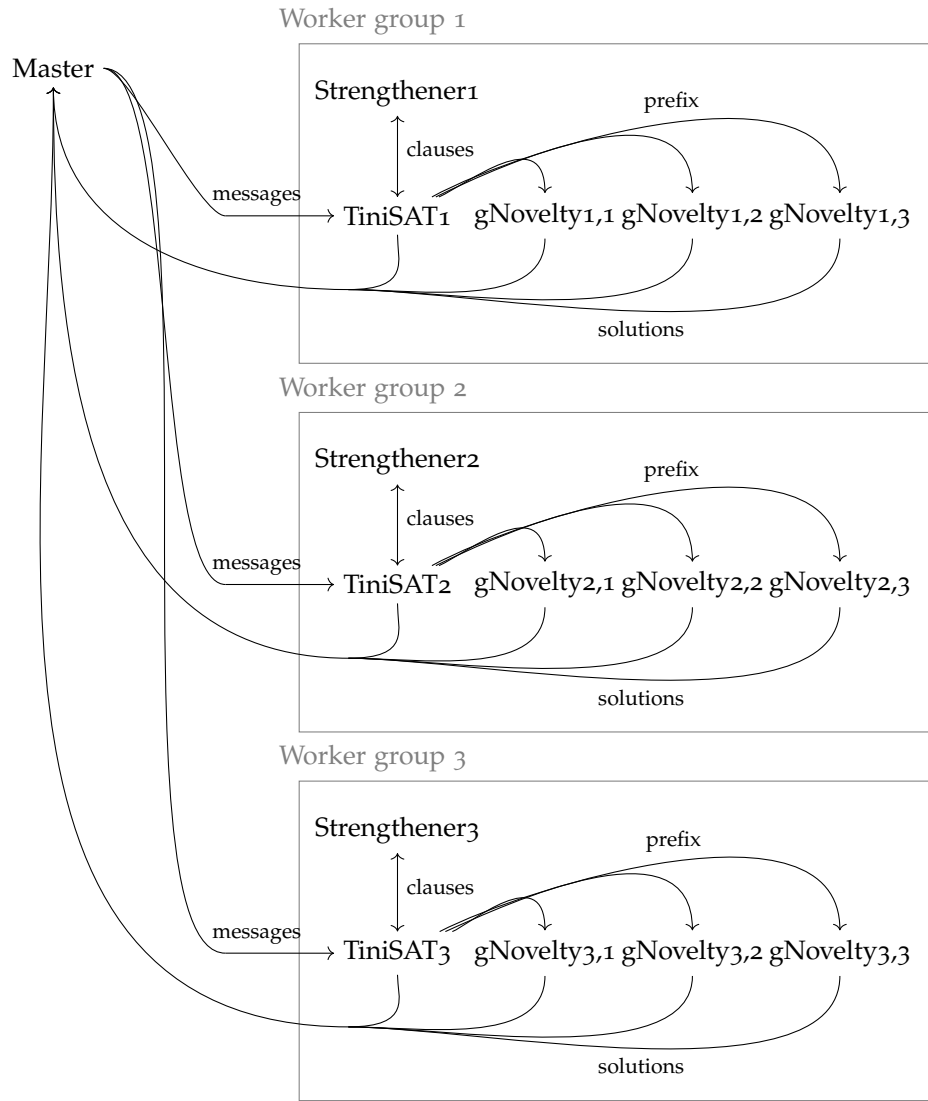


Figure 3: Relationship and messages between and within the Master and worker groups within DAGSTER mode2 operation

An illustration of the elements of the modes are depicted in Figure 3.

The choice of operational mode for DAGSTER is selected by command line argument, with mode zero as the default. Additionally, DAGSTER has command line configuration identifying the way in which the Master will coordinate the worker groups. Particular configuration specifies whether Master will preferentially allocate messages that are depth-first in the DAG in an attempt to race to a solution for the problem, or alternatively to allocate breadth-first and solve all layers of the DAG structure more systematically. In this context there is also additional configuration regarding whether DAGSTER will terminate as soon as it finds a solution to the whole problem, or if it should attempt to generate all solutions to the problem. The optimal choice of these options will be expected to depend on the particular problem which DAGSTER is presented with.

6 EMPIRICAL STUDIES

There are many other sources of decomposed problems, particularly where the process of generating the problem also gives its structure. An example of this is the Pentomino problems considered in Section 6.2, which is an example of a tiling problem where the problem is naturally distinct between different tiling regions. Different problems naturally give rise to different methods of decomposing them into structures, and it is possible to experiment with possible decompositions for any given problem. In the following sections we report on various experimental results which has been generated with DAGSTER for different problems.

Particularly we consider the following experiments:

- Section 6.1: parallel solution of small hard problems
- Section 6.2: parallel solution of larger hard problems, particularly Pentomino problems
- Section 6.3: a divide-and-conquer solution to the Costas array problem
- Section 6.4: a two part decomposition of random SAT problems to accelerate their solution process, with SLS assistance
- Section 6.5: solving easy large problems in parallel, demonstrating how DAGSTER can solve problems in sequence that are bigger than the computer has memory to hold together
- Section 6.6: connected easy-5-SAT problems, showing that SLS assistance can speed up solution process
- Section 6.7: software verification problems

6.1 Model Counting in Parallel - Hard Satisfiable Random Formulae

Here, we examine the performance of DAGSTER at counting models in a hard synthetic example that exhibits a single solution. One simple case of verifying the parallel performance of DAGSTER is to test the performance solving small-hard SAT problems which are entirely disjoint. Particularly it is expected that as the number of small-hard problems increases then the time DAGSTER should take in solving these should be a function of the time it takes to solve any single one of the subproblems, and the ratio of the number of multiprocessing cores to the number of subproblems which need to be solved.

The SGEN1 script³ distributed with DAGSTER was used to generate small-hard subproblems comprising 95 variables and featuring one unique solution. Such problems take our baseline CDCL procedure, TINI_{SAT}, about half a second to solve and also prove that no further solutions exists. Our repository includes a custom script for conjoining subproblems of that type into one super-problem described by a accompanying DAG. Figure 4 gives a depiction of this type of super-problem. The performance of DAGSTER at solving this arrangement of subproblems was measured. We examined the wall-time performance of DAGSTER as we scale the number of cores as a fraction of the time it took a vanilla CDCL procedure in TINI_{SAT} to solve the problem without having to also prove that no further solution exists.

The runtime of search on super-problems with varying numbers of subproblems, in different computational environments where we vary the number of available cores, is shown in Figures 5 and 6. From Figure 5 we can see that as the number of subproblems increases the time it takes DAGSTER to solve the problem as a fraction of the time it takes the baseline CDCL procedure, TINI_{SAT}, to solve the problem decreases – i.e., DAGSTER is significantly faster. Particularly, it is possible to note

³ SGEN1: <http://www.cs.qub.ac.uk/~i.spence/sgen/>

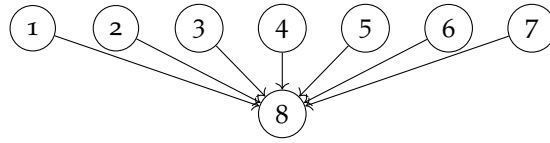


Figure 4: An example DAG for conjoined small-hard problems, where disjoint subproblems are processed in parallel and their solutions joined together at the final node

that the trend-line is approximately hyperbolic, indicating that the computational overhead of using DAGSTER goes to zero as the number of subproblems increases. Also, the speed of DAGSTER is pseudo-linearly increasing for large numbers of subproblems from Figure 6 which is expected. The reader should also not that here, DAGSTER is solving a more difficult problem compared to the CDCL baseline. It has to find the one satisfying valuable, and then further prove that no further solutions exists.

We also notice that increasing the number of cores available to DAGSTER solving the problem increases performance, but only upto a certain point. Particularly it was noticed that 64 cores performed significantly worse than 32 cores.

The invocation to run the experiments is to be found within the code repository:

```
/experiment_scripts/conjoined_experiment/runme.sh
```

Experimenting with with small-hard problems in parallel we were able to verify that DAGSTER offers significant parallel advantage, as is expected by its design - these experiments were performed on the ‘Etna’ machine - see Section 8.

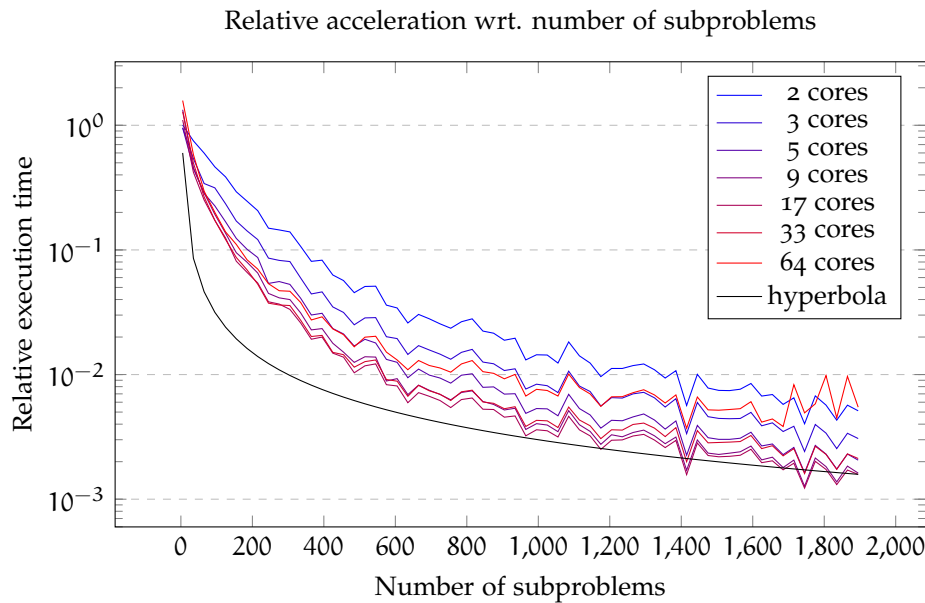


Figure 5: Runtime relative acceleration on parallell small-hard subproblems, relative to TINSAT time, with hyperbolic trendline, for different numbers of cores and subproblems

6.2 A Large Satisfiable Problem - Pentominoes

A tiling problem was extended as a benchmark for DAGSTER from a recreational puzzle solving YouTube channel⁴, as shown in Figure 7, where the challenge is to

⁴ Cracking the Cryptic, YouTube video: [youtube.com/watch?v=S2aN-s3hG6Y](https://www.youtube.com/watch?v=S2aN-s3hG6Y)

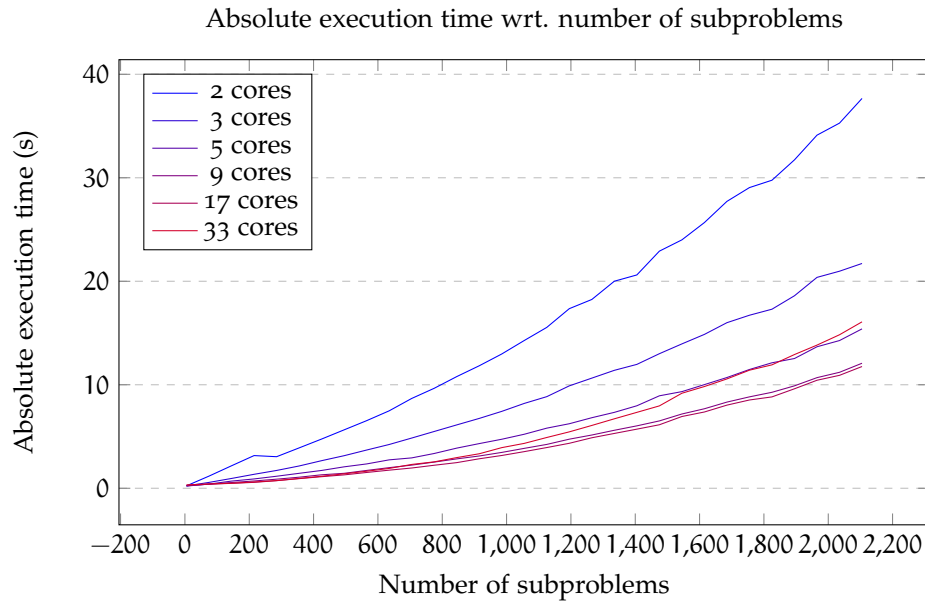


Figure 6: Runtime on parallel small-hard subproblems, for different numbers of cores and subproblems

fill a grid with Pentominoes (5 connected blocks) such that no pentomino crosses a boldened black line, and that no pentominoes of the same shape (counting reflections/rotations) touch each other.

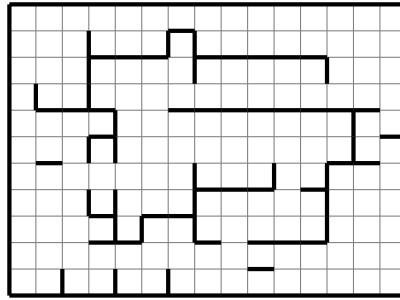


Figure 7: A Pentomino puzzle featured on youtube channel 'Cracking the Cryptic'

A generator of these kind of problems was coded to randomly generate hard 15x15 pentomino problems, involving a process of:

1. Randomly filling a 15x15 grid with pentominoes
2. Bolden the outline of those pentominoes, and removing them
3. Iteratively removing a random boldened line segments while the puzzle is uniquely solvable, until no more such removals are possible.

This process was shown to become slow to generate problems larger than 25x25, and so larger pentomino problems were generated by cascading these 15x15 subproblems side-by-side together in a grid pattern, such as shown in Figure 8. In this way, the grid of pentomino problems constitutes a larger problem which has logically distinct parts and where each subproblem is logically related only to its immediate neighbours: above, below, left and right; and because every pentomino subproblem is uniquely solvable then also this larger pentomino problem is also uniquely solvable.

For these large pentomino problems a DAG would be generated embodying a solution process of solving from the top left diagonally through to the bottom right,

as shown in Figure 9. In these particular problems the size of the grid of 15×15 pentomino subproblems would determine the maximum branching of the parallel process and thus the efficiency of the solving in parallel.

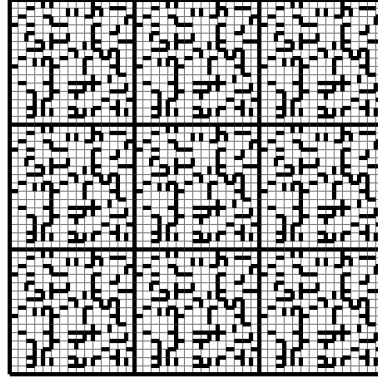


Figure 8: An example 3×3 pentomino superproblem

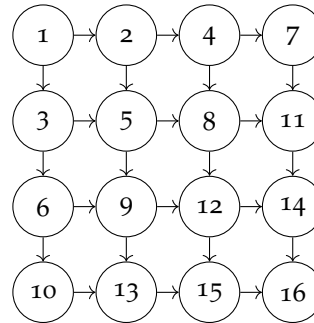


Figure 9: An example DAG for a 4×4 pentomino superproblem

For these problems we tested the performance of DAGSTER for different numbers of processor cores, against the TINI SAT and LINGELING CDCL baselines for different sized problems, the results are shown in Figure 10. In this figure, we see that DAGSTER is up to an order of magnitude faster (in the median) for these problems than serial sat solvers, but that increasing the number of cores does not necessarily improve performance. Specifically we suspect that this is because the DAGs of these problems (such as per instance in Figure 9) do not support sufficiently many parallel processing streams to take advantage of higher parallel processing cores. The Pentomino problems were also run with the strengthener module enabled, and the results show little difference in runtime on these problems as shown in Figure 11 - in this context enabling strengthener module slowed the runtime by 4.8% across the data. We note that the enabling of different modules within the DAGSTER tool gives variable results depending on the structure and kind of problem.

These Pentomino problems verified the functioning of DAGSTER (particularly in Figure 10) in providing speedup due to parallelisation in solving larger structured problems with coupled subproblems. The invocation to run these experiments is to be found by executed the following commands within the code repository:

```
experiment_scripts/Pentomino/runme.sh
experiment_scripts/Pentomino2/runme.sh
```

These experiments were performed on the ‘Etna’ and ‘Whale’ machines respectively - see Section 8.

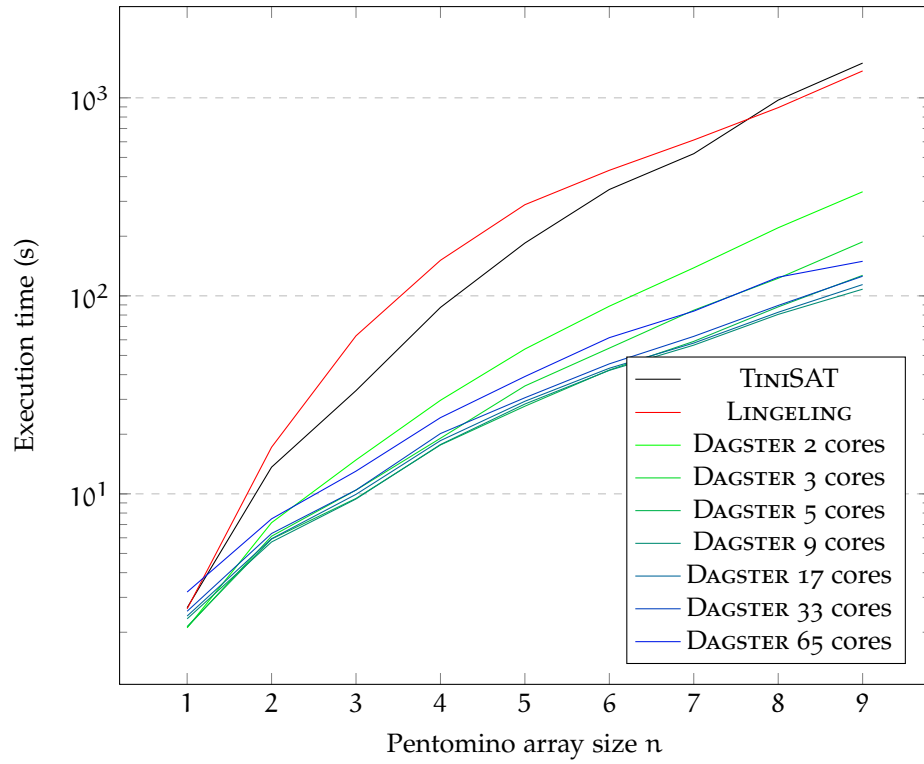


Figure 10: Runtime performance (specifically medians) of DAGSTER against TINI SAT and LINGELING solvers for different numbers of cores, across n , for $n \times n$ grid of 15×15 pentomino problems.

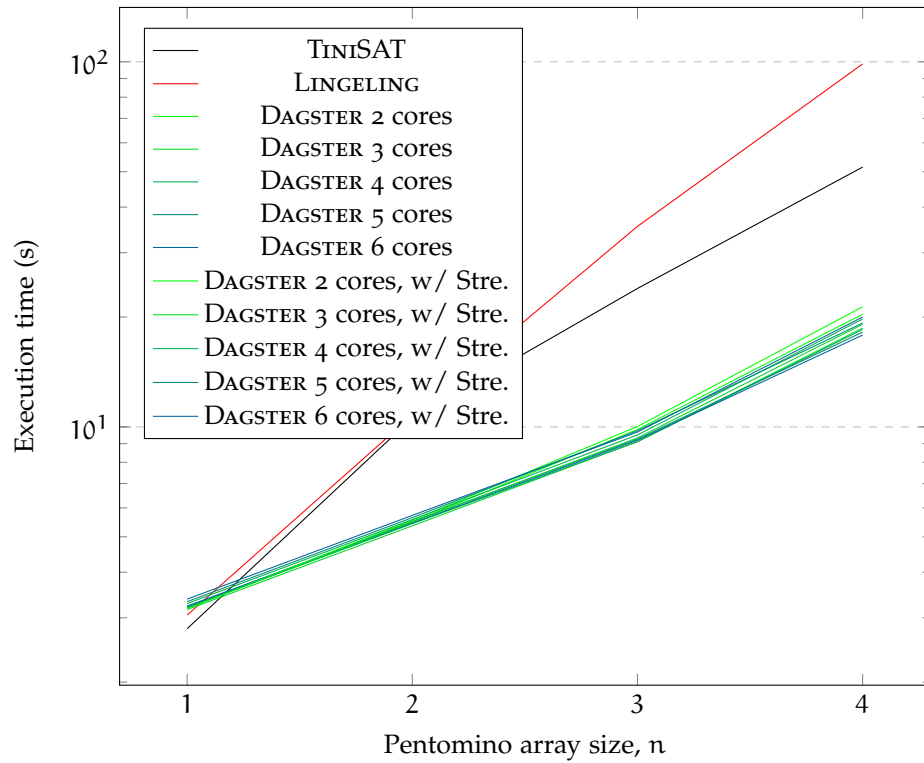


Figure 11: Runtime performance (specifically medians) of DAGSTER against TINI SAT and LINGELING solvers for different numbers of cores, across n , for $n \times n$ grid of 15×15 pentomino problems, with and without additional strengthening cores.

6.3 Counting Models in Parallel - Costas Arrays

A Costas array is a set of n points in an $n \times n$ array such that each column and row contains exactly one point and each of the $n(n-1)/2$ displacement vectors between the points are distinct; Costas arrays are well known and have various applications, and the process of using search techniques to solve for them is known to be challenging. Specifically there is an open question about whether any Costas arrays exist notably for sizes 32×32 and 33×33 . Searching processes have revealed that there are none of specific sub-classes of Costas arrays of those sizes [?], which has invited some to predict that Costas arrays of those sizes exist [?]. Notwithstanding, searching processes has been conducted at least upto size 29×29 [?].

As an example, consider the following 6×6 Costas array in Table 4:

		■			
				■	
					■
■					
			■		
	■				

Table 4: An example 6×6 Costas array

From this example Costas array we can see that there is exactly one filled-in cell for every column and row; additionally we can see that the vector displacement between any of the filled-in cells is unique and that there are no two sets of cells that have the same spacing between them. Particularly if we tabulate all of the $n(n-1)/2$ displacements between pairs of nodes in Table 5 we can see that all the displacements values along each of the rows are unique.

	1	2	3	4	5
1	+2	-5	+4	-3	+1
2	-3	-1	+1	-2	
3	+1	-4	+2		
4	-2	-3			
5	-1				

Table 5: The displacements of the example 6×6 Costas array 4, organised by horizontal distance in rows, by horizontal offset in columns

Costas arrays are known to exist for many sizes, and for every Costas array, there are potentially 8 symmetry mappings of the same array that are also Costas arrays (by rotations and flip, ie. the Dihedral group), Costas arrays have been enumerated by size such as given by the online encyclopedia of integer sequences (OEIS): <https://oeis.org/A008404> and <https://oeis.org/A001441>.

The investigative question therefore is: if we can encode the Costas problem into SAT, and then decompose the resulting SAT problem for accelerated solving using DAGSTER. Particularly the SAT encoding of the Costas problem was produced with optional symmetry breaking constraints to break Dihedral mappings, particularly with lex-leader symmetry breaking, and a more simplified and less total symmetry breaking as found in [?]. From this SAT problem we considered primarily two different ways of decomposing the SAT problem.

1. Inspired by the construction of such tables as Table 5, we considered a decomposition of iteratively constructing Costas arrays from the bottom of the table up. This decomposition proved to be poor performing as a combinatorial number of intermediate solutions between depths were created
2. Taking a much simpler approach of decomposing the problem into two parts, the first being placing the first three columns of the Costas array, and the

second part of the problem being that the rest of the Costas array would be filled in. This proved to be more effective decomposition of the problem. See Figure 12.

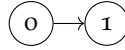


Figure 12: A simplified DAG structure for the Costas decomposition

The performance of DAGSTER at solving Costas problems for different sized Costas arrays is given in Figure 13. In this figure the time taken to count all the Costas arrays of a given size is plotted against the size of the problem for different numbers of cores against the time taken to do the same thing running the serial baseline CDCL procedure implemented in TINISAT (shown in black). From this figure we can see that for larger sized Costas arrays that DAGSTER shows significantly improved performance in solving the Costas model counting problem, where the performance increases with the number of cores. However it is also noticed that for smaller and easier Costas problems (of size $n \times n$ where $n < 10$) that the parallel overhead of using DAGSTER is the primary determinant of the solution time, where the greater the number of cores the larger the overhead and the slower the process.

With this simulation we can verify that DAGSTER can be used to speedup solving performance on the types of problems which are geometrically difficult and resemble research-interesting questions.

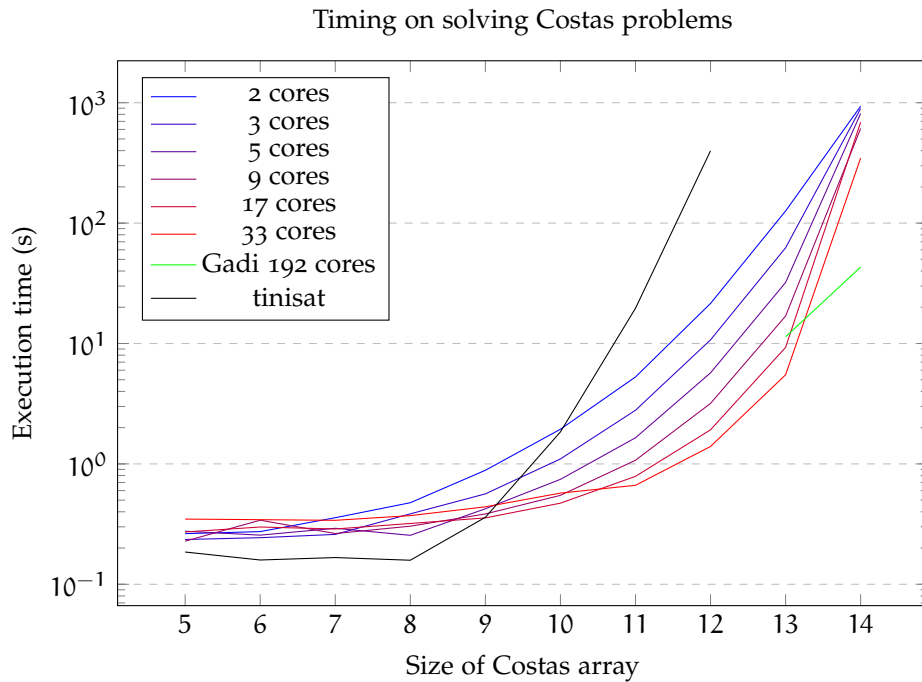


Figure 13: Runtime to solve Costas arrays, by core count and Costas array size with 1800s timeout.

It was also interesting whether the strengthener module would improve performance on Costas array solving, and to investigate this the strengthener module was added for different sized Costas array problems for different core counts. The resulting performance is shown in Figure 14. where we can see that the strengthener module does not notably improve performance, and slows the execution by about 4.8 percent.

The Costas simulations that produced the data shown in Figures 13 and 14 can be invoked from the DAGSTER repository by executing:

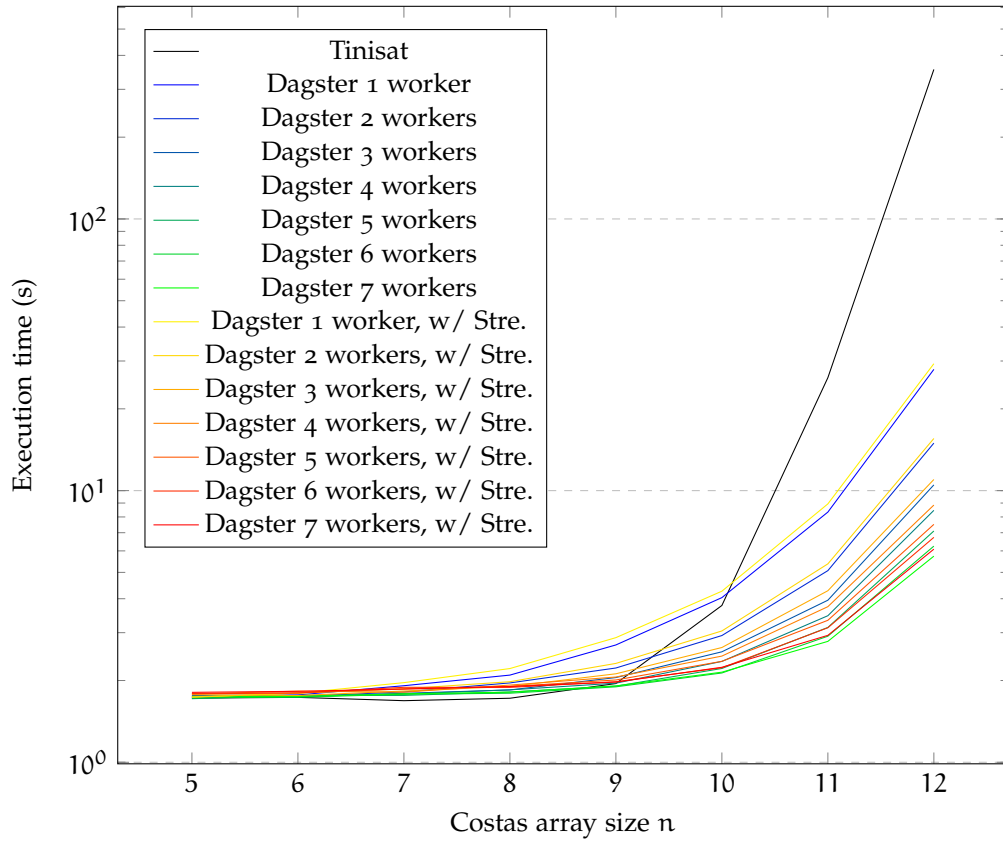


Figure 14: Runtime performance on Costas problems with and without strengthening processes, for different numbers of cores on costas problems of different size

```
experiment_scripts/costas/run.sh
experiment_scripts/costas2/run.sh
```

And particularly from these figures, we can see that DAGSTER can accelerate the solution of Costas model counting particularly by parallelism. These experiments were performed on the ‘Etna’ machine by default, as well as the ‘Gadi’ machine where noted - see Section 8.

6.4 Accelerating Unsatisfiability Proofs via Parallelisation

Empirical studies of the Boolean satisfiability problem late last century identified and studied a notion of empirically *hard* problems. The earliest works studied formulae occurring in conjunctive normal form with all disjunctive clauses having a fixed length k . A range of studies of this “ k -Satisfiability” problem have been undertaken treating different values of k , and other more flexible concepts of structural invariants. Taking $k = 3$ and studying sets of pseudo-random problems researchers found that so-called “hard” problems occur when the ratio of the count of clauses to the count of problem variables is approximately 4.26 [?]. Here we present a small study of pseudo-random 3-Satisfiability using the problem distributions associated with Dubois et al. (2000)[?].

Somewhat trivially—i.e., primarily due to a portfolio effect—the hybrid search implemented in DAGSTER outperforms CDCL baselines in hard satisfiable random instances. For satisfiable 3-Satisfiability the best solution procedure is a local search [?]. Our hybrid configuration of DAGSTER allows any number of such searches to be scheduled to run more-or-less independently in parallel on your cluster. Thus, not

140 unsatisfiable random 3-SAT problems - runtime comparison with baseline

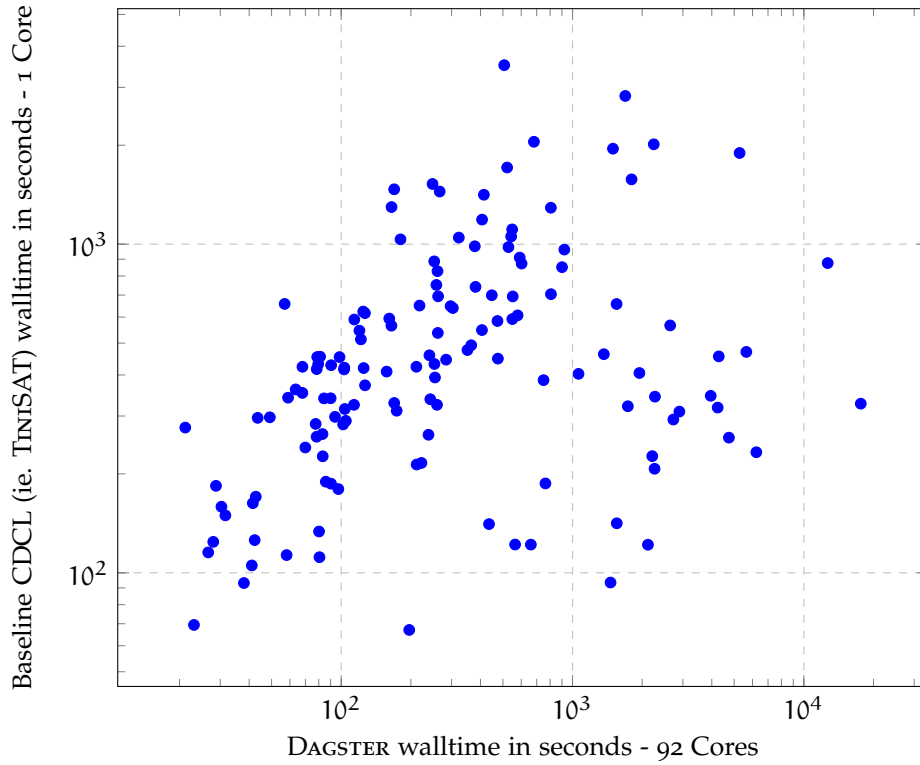


Figure 15: Logscale scatter plot of runtime distributions of TiniSAT (vertical) and DAGSTER using 92 cores (horizontal) on pseudo-random unsatisfiable 3-Satisfiability problem instances.

only is this parallel system better than CDCL here, courtesy of implementing a local search, it is also faster than running a serial local search procedure, because the expected walltime to a solution being emitted is at the lower end of the runtime distribution of that stochastic procedure. Here, we focus on the more interesting setting, accelerating search in a family of unsatisfiable 3-Satisfiability problems.

In our first result, we accelerate the walltime performance of the default implementation of CDCL in TiniSAT, by leveraging the hybrid search mode, using the local search to generate a complete set of relatively easy UNSAT subproblems that can be solved independently in parallel. A comparison of runtime of DAGSTER as compared with the serial CDCL procedure in TiniSAT is given in Figure 15. Specifically, we can decompose the problem using a 2 node DAG in which the “source” node indexes 94% of clauses, and this communicates solutions to a “sink” node that includes all the clauses from the concrete problem at hand. What is communicated is a small prefix of the total subproblem valuation. The problem at the source node, with only 94% of clauses, is easily satisfiable using a local search, and indeed the hybrid procedure can quickly enumerate all solutions to that subproblem and subsequently the CDCL procedure can prove that no further solutions exists. The problem of proving that the concrete problem is unsatisfiable, for each of the valuations associated with the source node subproblem, is extremely easy.

In conclusion, our hybrid solver can, by virtue of local search, easily enumerate the solutions to an underconstrained subproblem, and by virtue of parallelism quickly eliminate those solution candidates using systematic search.

6.5 Model Counting in Parallel - Easy Large Satisfiable Random Formulae

It is also possible to verify that DAGSTER can solve problems which are large in comparison with available memory, particularly we consider solving large-easy SAT problems which are entirely disjoint and in parallel, similar to the scheme implemented in Section 6.1. It is verified that as the number of large-easy problems increases the memory that DAGSTER should take in solving these should be less than the size of the CNF file of all those problems combined; and in this way it is verifiable that DAGSTER can solve problems which are larger than the amount of memory on the machine. This is accomplished by splitting the CNF of the larger problem into parts corresponding to subproblems, which are loaded sequentially and solved in turn in accordance with the DAG.

Particularly a 5MB random-7-SAT problem was generated, and multiple copies of the same problem were concatenated to form a parallel DAG - as previously shown in Figure 4). In this way the maximum size of any subproblem was 5MB, whereas the CNF of the problem would be many multiples of 5MB.

The memory consumed by DAGSTER running these problems against the size of the CNF of these problems for different numbers of cores is shown in figures 16. From Figure 16 we can see that the memory used is a barely a fraction of the size of the CNF, which doesn't vary considerably with the number of cores. From this experiment it can be verified that DAGSTER can tackle problems which are large with respect to the available memory. The experiment was performed on the 'Etna' machine - see section 8.

The invocation to run these experiments is to be found by the following command within the code repository:

```
experiment_scripts/sequential_random_sat/runme.sh
```

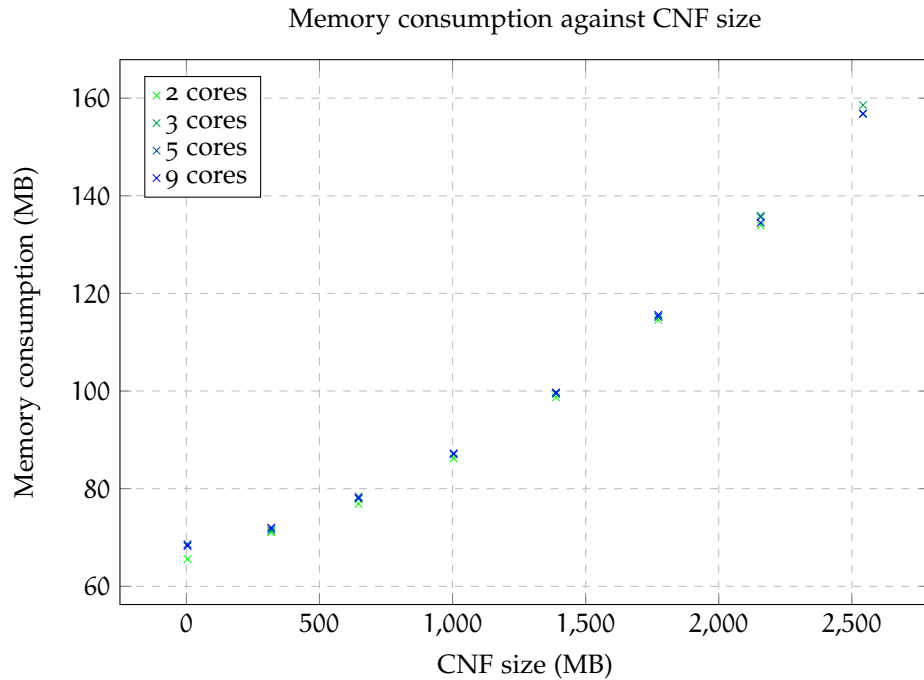


Figure 16: DAGSTER memory consumption for conjunctions of large-easy problems against core count.

6.6 Model Counting in Parallel - Easy Connected Satisfiable Random Formulae

We have already investigated the potential of using DAGSTER to solve “hard” random disjoint problems in parallel, however we can also investigate the possibility of non-disjoint problems being solved in parallel. For this purpose we built a monolithic formula by connecting a series of easy random-5-SAT formulae, with each subformula being randomly generated with a clause-to-variable ratio of 10/1. Each adjacent pair in the series of subformulae have some variables overlapping. Specifically, the monolithic formula was generated with 5 variables in common between any successive pair of subformulae. The DAG relating subformulae is as per Figure 17, thus giving a chain of subformulae. For this case we considered the runtime of DAGSTER both with and without gNOVELTY+ assistance.

The resulting runtime (first quartile, median and third quartile) was compared with and without the gNOVELTY+ assistance against the number of subformulae in Figure 18. The gNOVELTY+ assistance does provide some minor acceleration of walltime performance in these instances.

The invocation to run this experiment is to be found by the following command within the code repository:

```
experiment_scripts/sequential_random_sat2/runme.sh
```

The simulation was performed with a maximum of 14 cores on the ‘Whale’ machine - see Section 8. In addition to the results of Section 6.4, this experiment also demonstrated possible speedup utilising gNOVELTY+ assistance in DAGSTER.

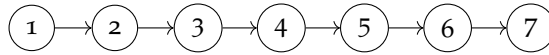


Figure 17: DAG for 7 conjunctioned random-5-SAT problems, where the set of preceding subproblem solutions constrain solutions to succeeding problems.

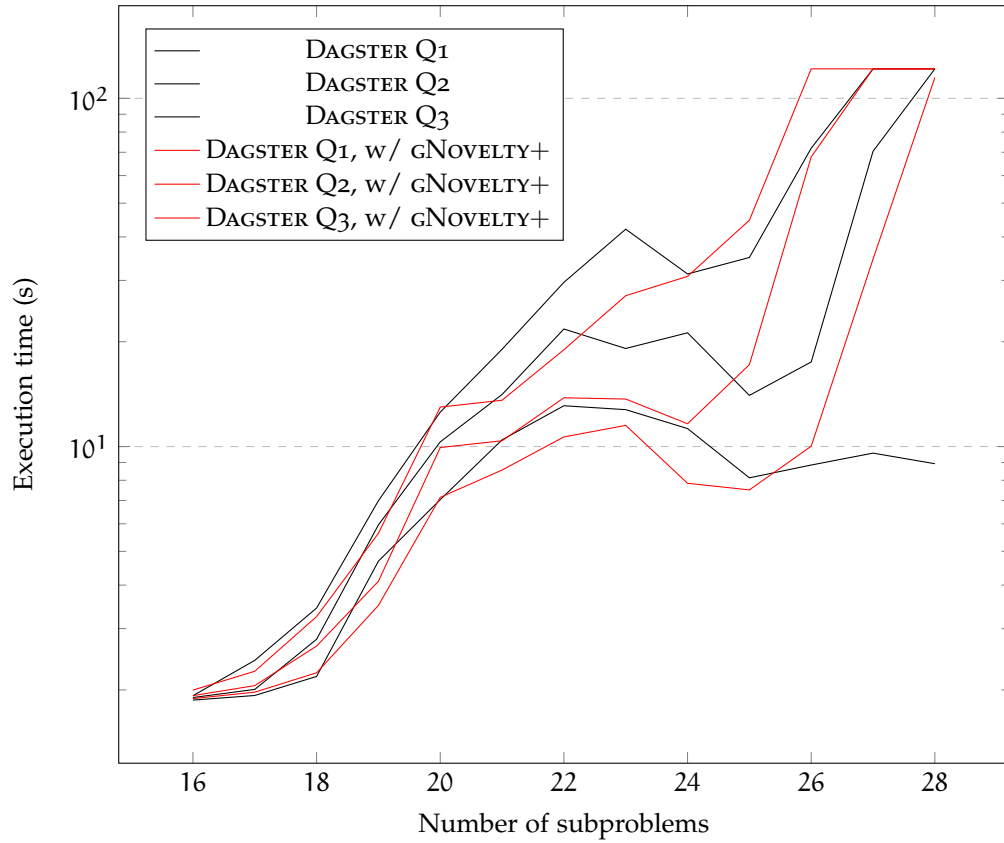


Figure 18: Runtime performance—specifically first quartile (Q1), median (Q2) and third quartile (Q3)—of DAGSTER on monolithic SAT problems comprised of n “easy” random 5-SAT problems. We contrast DAGSTER performance both with and without gNOVELTY+ assistance. For each value of n we ran DAGSTER 28 times, and the reported statistics are based on that empirical distribution. Each DAGSTER run was with an 1800s timeout.

6.7 Decomposing Software Verification Problems for DAGSTER

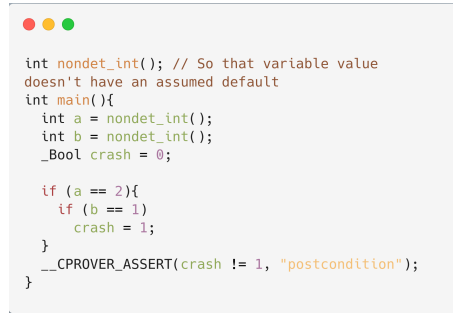
In this section we present results for a recipe which produces a decomposition of a SAT formula associated with a software security problem. We show that by using this decomposition DAGSTER is able to find a solution to the problem faster than our familiar serial baseline, `TINISAT`, or the *de facto* SAT solver for this setting, namely `MiniSAT`. Prior to presentation of our experimental results, we give context on the tool used to generate the SAT formulae and the actual security problem we have examined.

Software is pervasive and critical in the functioning of our modern society, both corporately and personally. Motivated by this, the field of software security has become essential. Many different techniques exist to find bugs in programs, one of which frames this task as a model checking problem. Model checking formalises the problem as determining whether a system, our program in this case, satisfies a set of requirements – E.g., no null dereferences. The bounded model checking (BMC) problem can be used to address some software security problems, and it does so by looking for violations of the property within a certain limited search *depth*. In the case of system *safety properties*, if an upper bound—called a *completeness threshold*—on the required depth is derived and used, then this method can demonstrate that no violations of the property will ever occur. Otherwise this method is only able to find the presence of violations, and not prove their absence. BMC operates by translating the statement of a security problem about a system into a (Boolean) SAT problem, and so, a SAT solver is used as the solution engine.

We use the tool CBMC to translate the problem of finding bugs into a SAT problem. CBMC implements BMC for C and C++ programs, and generates a corresponding SAT query associated with the software verification task at hand. It takes a program, which we want to verify certain properties about, and broadly speaking it passes it through 3 stages to result in the SAT formula. We will work through the example code in Figure 19 to illustrate the process.⁵

1. In the first stage, the given C/C++ program is converted into a GOTO program, which consists of equivalent goto statements for any code that performs a jump; E.g., loops, if statements, jumps etc. This results in a GOTO program that only has: (i) guarded goto instructions, (ii) goto instructions, (iii) assignments, and (iv) assertions and labels.
2. In the second stage, the loops in the GOTO program are unwound and the program is converted into single static assignment (SSA) form. The loop unwinding corresponds with bounding the depth of the system execution. In SSA form, variables can only be assigned once, and so-called ϕ functions are inserted eagerly at merge points in the control flow graph. As a result, numerous copies/instances of the same variable in the program are created for each step that the variable is re-assigned. Additionally, variables are created for the ϕ functions that determine which of the previous instances of the variable are now being referred to – I.e., what control flow is being reasoned about. An example of this conversion can be seen in Figure 20, where ϕ nodes have been expanded into a form that uses the C ternary operator "?:".
3. In the final stage, the SSA form is converted into a set of conjoined constraints, by turning assignments into equalities and forming the conjunction of all them. The result of running this final stage in our example is depicted in Figure 21. Observe that the property to be checked is negated, which in this case means we check to see if `crash_5 = 1`. The conjoined constraints are then converted into a SAT formula by a process known as bit-blasting.

⁵ Our provided example is by no means exhaustive of the capability CBMC. That tool can also perform other safety checks in relation to memory and array bounds etc. Our example will only consider an assertion that the user wants to be satisfied.



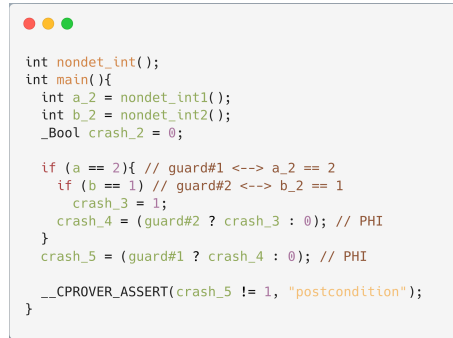
```

int nondet_int(); // So that variable value
doesn't have an assumed default
int main(){
    int a = nondet_int();
    int b = nondet_int();
    _Bool crash = 0;

    if (a == 2){
        if (b == 1)
            crash = 1;
    }
    __CPROVER_ASSERT(crash != 1, "postcondition");
}

```

Figure 19: Example code in C.



```

int nondet_int();
int main(){
    int a_2 = nondet_int1();
    int b_2 = nondet_int2();
    _Bool crash_2 = 0;

    if (a == 2){ // guard#1 <--> a_2 == 2
        if (b == 1) // guard#2 <--> b_2 == 1
            crash_3 = 1;
        crash_4 = (guard#2 ? crash_3 : 0); // PHI
    }
    crash_5 = (guard#1 ? crash_4 : 0); // PHI
    __CPROVER_ASSERT(crash_5 != 1, "postcondition");
}

```

Figure 20: Conversion of code into SSA form.

We are able to produce a decomposition for this program by using a recipe that considers the formula as it is represented in SSA form. The resulting decomposition that we create will have four DAG nodes, as seen in Figure 22. The first Node contains clauses that relate to calculating the goal and determining the values that guard#1 and guard#2 should be assigned, in order to set the crash flag to *true*. When DAGSTER is operating given this DAG, the guard values will be communicated as messages to search scheduled at subproblems at Nodes two and three. Those subproblems are characterised by clauses that described the required values for *a* and *b*, separately. Note that these variable values are independent of one another, and that the DAGSTER search will calculate the required variable assignment independently in parallel. The final Node will merge the results of Nodes two and three, and confirm that they cohere to form a solution.

Using the same recipe, we can produce a decomposition for a program with three independent conditionals depicted in Figure 23, which will result in three separate sub-problems, depicted in Figure 24. In this program there are three possible places the program can crash, which occur if any of the flag variables *a*, *b*, or *c*, are set to 1. The DAG produced by our recipe has been represented in Figure 24. Node 0 contains the assertion that one of the crashes must occur. Nodes 1 and 2 contain the computation done to set one of the flags. Node 3 contains the same clauses⁶ as Node 0, and thus is equivalent to Node 0.

We performed experiments on programs which had an identical structure to the independent program in Figure 23. One example had three independent conditionals, and the other two independent conditionals. In the case of two independent conditionals, the Node 3 in Figure 24 was removed by the recipe. To make the problem more difficult, we modified the example to use an implementation of the AES-128 encryption scheme. In this modification each of the flags could only be set if the correct message was found so that the encryption of the message resulted in an identical cipher text. This amounts to reversing the AES encryption, a difficult problem to solve. This resulting programs had an identical DAG structure to Fig-

⁶ This is due to the assertion being a logical disjunction, and how the recipe works.

```

a_2 = nondet_int1      AND
b_2 = nondet_int2      AND
guard#1 <--> a_2 = 2    AND
guard#2 <--> b_2 = 1    AND
crash_3 = 1            AND
crash_4 = (guard#2 ? crash_3 : 0) AND
crash_5 = (guard#1 ? crash_4 : 0) AND
crash_5 = 1

```

Figure 21: Resultant formula from conversion.

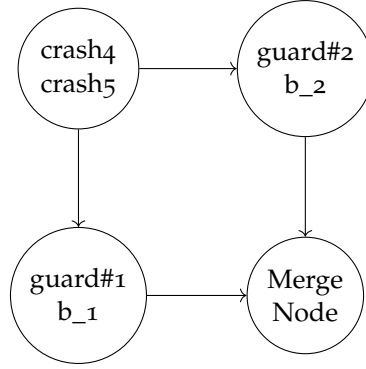


Figure 22: DAG for example code.

ure 23. We limited our recipe to only exposing the structure from the main function, thus do not expose any decompositional structure that exists in the implementation of AES.

```

int main(){
  int a, b, c, crash1, crash2, crash3;
  crash1 = crash2 = crash3 = 0;

  if (a == 1)
    crash1 = 1;
  if (b == 1)
    crash2 = 1;
  if (c == 1)
    crash3 = 1;

  __ASSERT(crash1 != 1 && crash2 != 1
           && crash3 != 1)
}

```

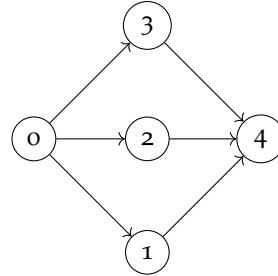


Figure 24: DAG for the problem with 3 independent conditionals.

Figure 23: 3 independent conditionals.

To produce the below results, DAGSTER was run in the default mode zero with no enumeration, and with restarts turned off in the CDCL searches. We found that with restarts on, DAGSTER performed poorly for these programs that used AES. We compared our results with TINI_{SAT}⁷, the CDCL procedure that DAGSTER uses in mode 0, and with MiniSAT which has an entirely different backtracking procedure than TINI_{SAT}. Our comparison with TINI_{SAT} provides a good platform to judge the decomposition since DAGSTER uses TINI_{SAT}, and the main difference is that TINI_{SAT} is working on the monolithic SAT formula but DAGSTER is able to distribute the search.

⁷ Restarts remained on for TiniSAT, since all runs without restarts for this problem resulted in a runtime of over five minutes.

The results for the AES program with two crashes, and also with three crashes, are in Table 6. This experiment was run on the ‘Luke’ machine described in Section 8. Anything that went over five minutes was recorded as a *time out*, denoted “T.O.” in the table. Both TINI_{SAT} and DAG_{STER} were executed on identical input CNFs. There seems to be an ideal number of processors for DAG_{STER} to use when distributing the search according to this decomposition; 4 processors for two crashes, and 6 processors for three crashes. The results showcase a significant improvement when using DAG_{STER}, where multiple CDCL processes are running independently in parallel, instead of using TINI_{SAT}, which is just working on the monolithic CNF. As can be seen in Table 6, TINI_{SAT} *timed out* on both runs, whereas DAG_{STER} took 21.9 seconds (two crashes) and 24.2 seconds (three crashes), respectively. Furthermore, DAG_{STER} outperformed MiniSAT, only slightly with two crashes, but using only a third of the reported MiniSAT runtime for three crashes (using 6 processors).

Solver	DAG _{STER}				TINI _{SAT}	MiniSAT
	8 Proc	6 Proc	4 Proc	2 Proc	1 Proc	1 Proc
Two Crashes	26.6	21.9	20.2	T.O	T.O	22.8
Three Crashes	30.4	24.2	40.8	T.O	T.O	78.5

Table 6: The runtimes (in seconds) for evaluated solvers on CNFs associated with Independent AES programs.

Consequently, the multiple focused searches of DAG_{STER} perform better than the one unfocused TINI_{SAT} search and indicate that decomposition is a fruitful path to improve the runtime of solving SAT problems related to software security.

7 CONCLUSION AND FUTURE WORK

The central software artefact produced in our project is a tool we have called DAG_{STER}. Over the next 12 months we shall support the continued development and expanded the features and use cases of the tool.

The DAG_{STER} tool implements a number of search-based solution procedures for the Boolean SAT problem, including both systematic and local search procedures. These have been integrated to operate in tandem, as a parallel hybrid search. The DAG_{STER} tool takes as input a Boolean SAT problem, and a schematic description of the compositional structure of the SAT. The tool operates in an HPC environment, and distributes the search effort to solve the given problem according to the provided compositional structure. It is able to operate on very large problems, and has been shown to accelerate Boolean SAT queries and model counting on a number of scenarios.

Particularly possibilities of continuing and future investigations include:

- Using machine learning and probing in parallel computing environments to produce variable and value selection rules for systematic search.
- Using machine learning for portfolio optimisation specifically in the DAG_{STER} setting, where we would locally optimise portfolios with respect to features of subproblems.
- Parallel distributed Property Directed Reachability (PDR) using the DAG_{STER} architecture. This substantial subproject is being pursued in tandem to the reported exercises, and will be reported on separately in December 2021.
- Design, implementation, and release of a checkpointing functionality in the DAG_{STER} tool. For context, at present a search exercise in an HPC environment orchestrated by DAG_{STER} cannot be interrupted, paused, and then restarted. Checkpointing will allow for progress to be saved, interrogated, and for a search activity to be paused and recommenced at some later time.

- Adding a feature to DAGSTER which can emit UNSAT certificates in a standard format. This is a familiar functionality of many serial SAT tools that run on PCs.
- Design, implementation, and testing of incremental search functionality. When a search process is repeatedly invoked on a family of related formulae, there are opportunities to re-use knowledge accumulated in successive searches. Presently that knowledge is being discarded by DAGSTER, and moreover the practicality of DAGSTER is limited in this setting by a substantial startup cost associated with commencing and recommencing searches.

8 ASSETS USED IN EMPIRICAL STUDIES

The following details the CPU information and available RAM of the machines used to run the experiments.

vendor id	AuthenticAMD
cpu family	23
model	49
model name	AMD Ryzen Threadripper 3990X
stepping	0
microcode	0x8301039
cpu MHz	2198.798
cache size	512 KB
siblings	128
cpu cores	64
Memory RAM	131856952 kB

Table 7: CPU information of the Etna machine

vendor id	GenuineIntel
cpu family	6
model	85
model name	Intel(R) Xeon(R) Gold 6134 CPU @ 3.20GHz
stepping	4
microcode	0x2006b06
cpu MHz	1200.170
cache size	25344 KB
siblings	16
cpu cores	8
Memory RAM	97506008 kB

Table 8: CPU information of the Whale machine

vendor id	GenuineIntel
cpu family	6
model	85
model name	Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz
stepping	7
microcode	0x5003102
cpu MHz	2836.313
cache size	36608 KB
siblings	48
cpu cores	24
Memory RAM	196464272 kB

Table 9: CPU information of the Goedel machine

vendor id	GenuineIntel
cpu family	6
model	85
model name	Intel(R) Xeon(R) Platinum 8274 CPU @ 3.20GHz
stepping	7
microcode	0x5002f01
cpu MHz	3799.866
cache size	36608 KB
siblings	48
cpu cores	24
Memory RAM	197733300 kB

Table 10: CPU information of the Gadi machine

vendor id	GenuineIntel
cpu family	6
model	158
model name	Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz
stepping	13
microcode	0xea
cpu MHz	2300
cache size	16384 KB
siblings	16
cpu cores	8
Memory RAM	32649348 kB

Table 11: CPU information of the Luke machine

REFERENCES

- [1] Stephen A. Cook. *The Complexity of Theorem-Proving Procedures*. STOC '71. Association for Computing Machinery, New York, NY, USA, 1971.
- [2] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [3] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *JSAT*, 4(2-4):149–172, 2008.
- [4] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building structure into local search for SAT. In *IJCAI 2007, Proceedings of the 20th International Joint*

- Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2359–2364, 2007.
- [5] A. Balint and U. Schöningh. probsat and pprobsat. In *Proceedings of SAT Competition 2014 Solver and Benchmark Descriptions*, page 63, 2014.
 - [6] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.
 - [7] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
 - [8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.
 - [9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.
 - [10] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.
 - [11] Youssef Hamadi and Christoph Wintersteiger. Seven challenges in parallel SAT solving. *AAAI Conference on Artificial Intelligence*, 2012.
 - [12] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-race and iterated f-race: An overview. In *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
 - [13] Marius Lindauer, Frank Hutter, Holger H. Hoos, and Torsten Schaub. Autofolio: An automatically configured algorithm selector (extended abstract). In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 5025–5029, 2017.
 - [14] Marius Thomas Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *J. Artif. Intell. Res.*, 53:745–778, 2015.
 - [15] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Intell. Res.*, 36:267–306, 2009.
 - [16] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, AAAI’13*, 2013.
 - [17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
 - [18] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI’2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 682–687, 2004.
 - [19] Matthew J. Streeter and Stephen F. Smith. Using decision procedures efficiently for optimization. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, pages 312–319, 2007.

- [20] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Mag.*, 34(2):99–106, 2013.
- [21] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference 2011*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65, 2012. Best Paper Award.
- [22] Marijn J.H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Frontiers in Artificial Intelligence and Applications*, volume 185, chapter 5, pages 155–184. IOS Press, Amsterdam, February 2009.
- [23] Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In Roman Wyrzykowski, Jack J. Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*, volume 4967 of *Lecture Notes in Computer Science*, pages 249–258. Springer, 2007.
- [24] Djamel Habet, Lionel Paris, and Cyril Terrioux. A tree decomposition based approach to solve structured SAT instances. In *2009 21st IEEE International Conference on Tools with Artificial Intelligence*, pages 115–122. IEEE Computer Society, 2009.
- [25] Eyal Amir and Sheila McIlraith. Solving satisfiability using decomposition and the most constrained subproblem (preliminary report). *Electron. Notes Discret. Math.*, 9:329–343, 2001.
- [26] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2318–2323, 2007.
- [27] Nick Feng and Fahiem Bacchus. Clause size reduction with all-UIP learning. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 28–45, Cham, 2020. Springer International Publishing.
- [28] Lintao Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, pages 279–285, 2001.
- [29] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference, DAC '01*, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery.
- [30] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers - 1506.08905. *arXiv*, 2015.
- [31] Antonio Ramos, Peter van der Tak, and Marijn Heule. Between restarts and backjumps. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 216–229. Springer, 2011.
- [32] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

- [33] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI'92*, page 440–446. AAAI Press, 1992.
- [34] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bull.*, 3(1):8–12, January 1992.
- [35] Ian P. Gent and Toby Walsh. An empirical analysis of search in GSAT. *J. Artif. Intell. Res. (JAIR)*, 1:47–59, 1993.
- [36] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, pages 337–343. AAAI Press / The MIT Press, 1994.
- [37] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proc. of AAAI-93*, pages 28–33, Washington, DC, 1993.
- [38] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Tabu search for SAT. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 281–285. AAAI Press / The MIT Press, 1997.
- [39] David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 321–326. AAAI Press / The MIT Press, 1997.
- [40] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 191–196. AAAI Press / The MIT Press, 2004.
- [41] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *J. Satisf. Boolean Model. Comput.*, 4(2-4):149–172, 2008.
- [42] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 655–660. AAAI Press / The MIT Press, 2002.
- [43] Jerry Lonlac and Engelbert Mephu Nguifo. Towards learned clauses database reduction strategies based on dominance relationship. *CoRR*, abs/1705.10898, 2017.
- [44] L. Guo, S. Jabbour, J. Lonlac, and L. Saïs. Diversification by clauses deletion strategies in portfolio parallel SAT solving. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 701–708, 2014.
- [45] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 703–711, 2017.

- [46] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 237–243, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [47] Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8–12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2013.
- [48] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21–25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008.
- [49] Hongteng Xu, Dixin Luo, and Lawrence Carin. Scalable gromov-wasserstein learning for graph partitioning and matching. In H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.
- [50] Craig A. Knoblock. Automatically generating abstractions for planning. *Artif. Intell.*, 68(2):243–302, 1994.
- [51] Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23–29, 1997, 2 Volumes*, pages 1178–1185, 1997.
- [52] Ivor Spence. Sgen1: A generator of small but difficult satisfiability benchmarks. *ACM J. Exp. Algorithmics*, 15, March 2010.
- [53] C.P. Brown, M. Cenk, R.A. Games, J.J. Rushanan, and O. Moreno. New enumeration results for costas arrays. In *Proceedings. IEEE International Symposium on Information Theory*, pages 405–405, 1993.
- [54] Jon Carmelo Russo, Keith G. Erickson, and James K. Beard. Costas array search technique that maximizes backtrack and symmetry exploitation. In *2010 44th Annual Conference on Information Sciences and Systems (CISS)*, pages 1–8. IEEE, 2010.
- [55] Konstantinos Drakakis, Francesco Iorio, Scott Rickard, and John Walsh. Results of the enumeration of costas arrays of order 29. *Adv. Math. Commun.*, 5(3):547–553, 2011.
- [56] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI’91*, page 331–337, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.
- [57] Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In David B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9–11, 2000, San Francisco, CA, USA*, pages 126–127. ACM/SIAM, 2000.