DAGSTER

A Parallel Structured SAT Solver
Midterm Progress Report Against Project Activities

MARK BURGESS, CHARLES GRETTON,
JOSH MILTHORPE, LUKE CROAK

15 March 2021

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 formulae, and *(ii)* enable practitioners to solve relatively challenging problems easily in HPC environments by efficiently distributing search effort. Our solver takes as input a SAT problem given in parts, with each part a conjunctive normal form formula corresponding to an abstract subproblem. The concrete problem at hand is a conjunction of parts. The dependency relationships between subproblems are given according to a labelled directed acyclic graph (DAG) structure. 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.

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

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

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 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*. 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 [1, 2]. The DAGSTER tool reported on here 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 [3, 4, 5], and *(ii)* the systematic backtracking search of *conflict-driven clause learning* (CDCL), as studied in [6, 7]. The latter are descendants of the Davis, Putnam, Logemann, and Loveland (DPLL) procedure, originally described in [8]. 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 [9]. Conversely, Local search procedures are sound but not complete. These treat the SAT problem as an optimization problem in which the objective is to compute an assignment that minimizes the number of unsatisfied Boolean "constraints". 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, the procedures are sequential, and not typically designed to be parallelized across multiple compute cores. As is the case in modern sequential solvers, 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 [10]. Parallelism can be exploited in the tuning of algorithm configurations [11], *search-space splitting*, or *portfolio approaches* [12, 13, 14]. Identifying and sharing good clauses is key to efficient parallelism [6], however, [15] 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 [16], parallelization is raised as a topic when considering the search horizon at which to pose queries [17, 18]. How to leverage parallelization 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 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 reporting 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.

2 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 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, as 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 parallelization.

2.1 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 entirely 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 is sometimes called the 'divide-and-conquer' strategy to parallel SAT solving [19], 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 [20] which breaks the process into many subproblems by first using a 'lookahead' process [21] 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 operating on 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.

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

The set of 3 satisfying assigments to f is:

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

This set can be decomposed according to the following 4 assignments (or 'cubes') of variables 1 and 4: $1 \wedge 4$, $1 \wedge \neg 4$, $\neg 1 \wedge 4$, and $\neg 1 \wedge \neg 4$. The solutions associated with a cube are those associated with f conjoined with the conjunctive cube. For example,

the solutions associatd with cube $1 \wedge 4$ are 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	$2 \vee 3$ $2 \vee \neg 3$ 3 $2 \vee \neg 3$	$2 \vee 3$ $2 \vee \neg 3$ $\neg 2 \vee 3$	$\neg 2 \vee \neg 3$ $2 \vee \neg 3$ 3 $2 \vee \neg 3$	$\neg 2 \vee \neg 3$ $2 \vee \neg 3$ $\neg 2 \vee 3$
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.

2.2 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 merging these subproblem solutions together to generate a solution that satisfies the original problem. The merging process is necessary because the solutions of different subproblems can share variables (even though different subproblems may have disjoint clauses, those disjoint clauses can have variables in common, hence solutions with incompatible variable assignments in common). Hence a merging/resolution process must be done to select subproblem solutions that are compatible with each other, thereby solving the full original problem. This process of solving each subproblem independently and then merging their solutions is one way of solving many subproblems together.

A simple example of this is shown in Table 2. In Table 2 we have two subproblems (part1 and part2), that is composed of separate clauses on the first row, the possible solutions to each of the subproblems are shown on the second row and 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 nth subproblem will be solved with

the constraints that satisfy a solution of the $(n-1)$th subproblem which was solved with constraints that satisfy a solution of the $(n-2)$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 generates 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 subproblem 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 specified a DAG file format to specify the possible decomposition of the SAT problem to be used in a solving routine as given in section 4.1.

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 [22] 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 solving the first to clause sets, before resolving the solutions together with the third clause set.

Another example of our SAT decomposition is found in [23] 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 [24] 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	$1 \wedge$ $2 \wedge 3$	$\Rightarrow$	$1 \wedge 2 \wedge 3 \wedge 4$ $1 \wedge 2 \wedge 3 \wedge \neg 4$
	$1 \wedge$ $2 \wedge \neg 3$ $\neg 1 \wedge$ $2 \wedge \neg 3$	$\Rightarrow$	$\emptyset$
	$\neg 1 \wedge$ $\neg 2 \wedge \neg 3$	$\Rightarrow$	$\neg 1 \wedge \neg 2 \wedge \neg 3 \wedge \neg 4$

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)

3 DAGSTER COMPONENTS

3.1 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 *TiniSAT* solver coded by Huang et al. [25] 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 utilize 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 [26, 27]. 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 clause in the CDCL procedure. In this way CDCL procedures have the ability to learn new

clauses with each conflict encountered, which leads to additional questions about how these many clauses are managed. Many SAT solvers have heuristics to keep the most valuable clauses and/or simply learnt clauses into stronger ones - see section 3.3.

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 [28]. 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*), at every junction, 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 [29]. 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 3.2.2.

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 [25]. There also exists the possible effectiveness of partial restarts and choice of different restart schedules etc. [30] For DAGSTER we utilized the Luby restart policy [31] inbuilt into the *TiniSAT* solver coded by Huang et al. [25].

3.2 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 [32, 33] 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 ← 1 to m do
    s ← initAssign(Φ);
    if s satisfies Φ then
        return s
    else
        x ← chooseVariable(Φ, s);
        s ← 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 chooses the variable to flip in each iteration, and there exist a range of possibilities. One of the very first SLS algorithms was GSAT [32] where `initAssign` function is a randomization of all problem variables, and `chooseVariable` method selects a variable which, if flipped, would minimize 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 fundmentally 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 becomes locked in to a set of assignments which satisfy some but not all clauses of the CNF [34].

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 randomization and hopefully avoids being trapped in the same local minimum; restarts are a feature in the original GSAT procedure, and most others.
- The inclusion of 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 [35] where, every step with probability p a random variable from an unsatisfied clause is randomly selected and 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 [36] 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 [37]
- 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 non-random variable can be selected in a greedy manner (to minimize the total clauses unsatisfied) or such as to minimize breaking existing satisfied clauses, or to prioritize those flips which ensure that no existing satisfied clause is made unsatisfied [35]. see [38].

- 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 [39].

Over time, many different SLS procedures have been developed and implemented, and many of their elements are compatible with each other. Particularly we modified and incorporated code for the gNovelty+ algorithm [40], 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+ [41], and the clause weight update scheme is inherited from the PAWS algorithm [39].

Algorithm 2: gNovelty+ algorithm without Tabu

```
Input: CNF formula Φ, max steps m, probabilities p, s, w
Output: satisfying assignment of Φ or nosolution.
Initialize the weight of each clause to 1;
randomly generate an assignment A;
for i ← 1 to m do
    if A satisfies Φ then
        return A
    else
        if within a walking probability wp 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 unsatisifed clauses;
            with probability sp 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
```

3.2.1 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 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 commonly 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 enhancement 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 pro-

cedure will methodically scan through the entire solution space and will always report the total set of solutions to the problem. In this way, SLS is understood to be more effective at non-deterministically finding solutions to problems which are under-or-moderately constrained, whereas CDCL is more effective at deterministically finding solutions to problems that are over-constrained.

A trivial way of integrating the advantages of SLS and CDCL in solving a problem is to run them both in parallel 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 we identify a mode of DAGSTER's operation where each CDCLs process is run alongside k other SLS processes, and the solutions which these processes find are recorded and accounted for.

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 in making better variable selection decisions.

3.2.2 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 a classic heuristic - see section 3.1. Particularly VSIDS which attempts to determine and preferably select the most influential variables in 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 will all clauses satisfied - see section 3.2. Insofar as the SLS process actually solves the problem, trivially the CDCL should adopt the SLS assignment as a solution and therefore this can be communicated. 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 it from its search. After eliminating the local minimum from its search, the CDCL process can then direct the SLS to search for other minima. 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. In this way, 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 depend 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 constrain the SLS search for local minima within that space and determine the appropriateness of the suggestions it offers to back to the CDCL.

As coupled local search 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. *prefix*) and then take variable suggestions from the SLS, after the SLS has had some time to resolve a local minimum. The SLS is constrained to search in a solution space that is consistent with the most recent assignment prefix is has received from the CDCL process. The SLS keeps a buffer of recent variable assignments that are most likely to characterize the local minimum in which it is stuck, which it then sends to the CDCL procedure as variable suggestions.

When should the CDCL procedure communicate a new prefix to an SLS procedure and consequently take suggestions from it? In our system, a CDCL search is coupled to pool of local searches, and communicates with them in a *round robin* fashion, sending new prefixes when it encounters evenly-spaced depths in the decision tree. When the CDCL procedure backtracks, it communicates with and takes

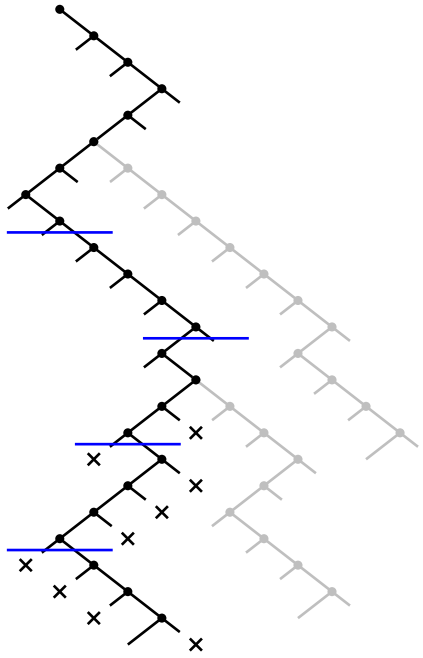


Figure 1: Illustration of CDCL branching process with SLS suggestion processes working on various branch cuts at various depths (blue)

advise from the local search that is most relevant to the backtrack depth. The fact that the local searches have been kicked off well in advance to a backtrack means they have had time to develop advise, in the sense that they will likely be exploring a local minimum.

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 gray. 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 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 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.

3.3 Clause Simplification

In the process of 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 [42, 43]. Thus it is advantageous to involve a 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.[42, 43] 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.[44] One of the most basic approaches to simplifying learnt clauses is clause 'minimization' or 'strengthening', where different techniques are used to identify and remove redundant literals in the learnt clauses.[45]

One of the more basic approaches of identifying and removing redundant literals from learnt clauses is considered by [46] 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 [47] where for any learnt clause $c = \{l_0, l_1, \dots\}$ each of the literals l_i are in turn are 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.

Minimizing 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 minimization 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 conflict is to perform the minimization process in parallel to the CDCL on a separate compute core (such as implemented by [46]), however it has also been shown that clause minimization can be effective even in serial with the CDCL procedure as well.[44]

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 [46] the reducing process called MINIRED/GLUCORED.

4 STRUCTURE OF DAGSTER

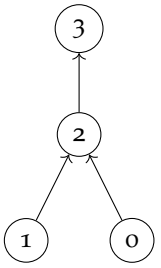
The relationship between what clauses were associated with which subproblem, and what subproblems should feed their solutions into other subproblems is specified in a DAG file which is then input into the solver, which then attempts to solve each subproblem in an effective way; and the beginning of this process is the specification of what clauses belong to the subproblems, as specified in a DAG file.

4.1 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.

Bronze winner of international SAT competition 2013 open track competition



CNF file

```
p cnf 7 7
2 1 0
-2 3 0
5 -4 0
-3 0
4 3 0
4 -6 7 0
-4 6 0
```

DAG file

```
DAG-FILE
NODES:4
GRAPH:
0->2:1,3
1->2:3,4
2->3:4
CLAUSES:
0:0,1
1:2,3
2:4
3:5,6
REPORTING:
4,6,7
```

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.

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: $1 \wedge 3$, $1 \wedge \neg 3$, $\neg 1 \wedge 3$.
- 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.

4.2 An Intuitive View of DAG Decomposition

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 1, whose outputs are then passed as constraints to node 2 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 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 nodes 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.

4.3 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 [23, 48]. We also note that lookahead mechanisms, for example as described for cube and conquer approaches to Boolean SAT [20], provide a decomposition that can be represented using a DAG schematic.

One 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[49, 50]. 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. A number of ancillary student projects are progressing exactly these ideas, both in a classical planning setting and also in the setting of software/protocol verification.

5 SCHEDULING PROCESS

Fundamentally DAGSTER is a parallel SAT solver, and the compute cores allocated to DAGSTER are each given a specific role. Particularly 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, particularly the Master coordinates what Workers should be working on what parts of the DAG. Initially 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 for 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 library for BDD manipulation and generation.

DAGSTER is allowed to run in multiple 'modes' identifying the resources which the workers are allocated. Particularly 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 TiniSAT 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 initialize and process the problem aswell. 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 gnovelties 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 gnovelty processes and an additional clause strengthener procedure. As before the Master's message is given to the CDCL procedure, the gnovelties and the strengthener process, and the strengthener

see: https://davidkebo.com/cudd

process assists the CDCL procedure by strengthening the conflict clauses that the CDCL procedure generates.

In these three operational modes $m = 1, 2, 3$ 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 Tinisat)
2. Mode one: each worker group is composed of a single CDCL process, and k gnovelty processes
3. Mode two: each worker group is composed of a single CDCL process, assisted by another strengthener process, and k gnovelty processes

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 SOURCES OF DECOMPOSED PROBLEMS

There are a range of schemes from the literature which provide recipes for computing a decomposition given the object to be decomposed [23, 48]. We also note that lookahead mechanisms, for example as described for cube and conquer approaches to Boolean SAT [20], provide a decomposition that can be represented using a DAG schematic.

One 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[49, 50]. 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. A number of ancillary student projects are progressing exactly these ideas, both in a classical planning setting and also in the setting of software/protocol verification.

7 PERFORMANCE DEMONSTRATION

In order to attempt to characterize the DAGSTER engine, we considered the execution time across a family of stochastically generated problems. Particularly we

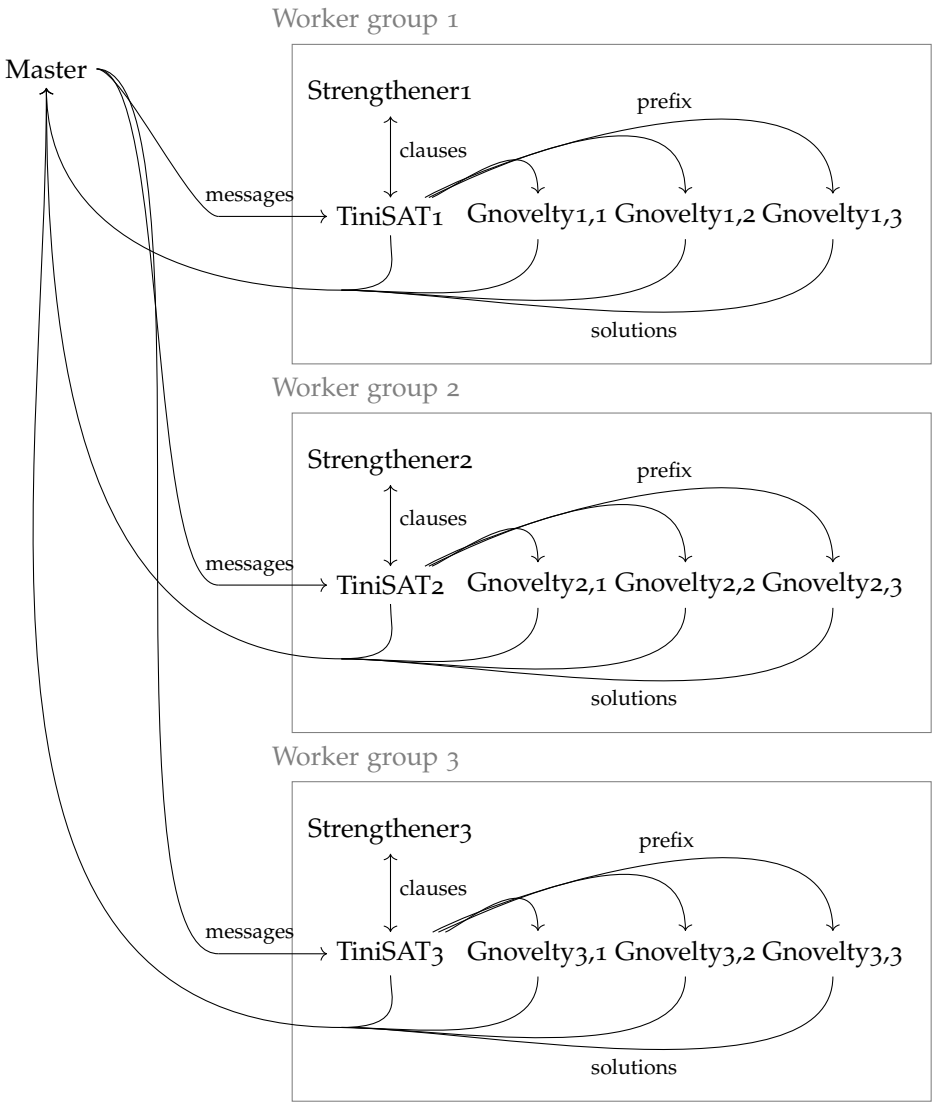


Figure 3: Relationship and messages between and within the master and worker groups within DAGSTER mode2 operation

considered the generation of random problems reminiscent of [51]. In [51], random SAT problems are generated by selecting in-advance a satisfying assignment to the problem, and then clauses are constructed within the problem which always include at-least one literal from the satisfying assignment. In this way, each additional clause that is added to the problem reduces the space of satisfying solutions, but is guaranteed to preserve the pre-selected satisfying solution; and thus the solution of the CNF is guaranteed to be SAT. The difficulty of the resulting problem is characterized by the total number of variables and the number of clauses added, additionally by how likely each clause is likely to have different numbers of literals coincident with the original satisfying assignment.

Taking inspiration from this approach, we constructed a series of these subproblems which each having the same number of variables and clauses, and sharing some variables in common with the one previous. Thus we were able to generate a family of random structured problems, as given by algorithm 3.

Algorithm 3: Random structured problem construction

Input: subproblem variable size s, subproblem variable overlap p, number of subproblems n, clause size c, clause multiplier m
Output: a set of clauses of a CNF
Consider subproblem i has variable set $V_i = \{(s-p)i+1, \ldots, (s-p)i+s\}$;
Create a random solution $S = \{\text{Random}(a, \neg a) : a \in \cup_i V_i\}$;
Start with an empty CNF;
for $i \leftarrow 1$ **to** s **do**
  for $j \leftarrow 1$ **to** m **do**
    for $v \in V_i$ **do**
      Construct a subset H of size $c-1$ which does not include v, from variables in V_i;
      Construct a clause $C = \{v \text{ if } v \in S \text{ else } \neg v\} \cup \{\text{Random}(a, \neg a) : a \in H\}$;
      Append C to CNF;
    end
  end
end

The generated problems naturally have a structure which is amenable to sequential solving, and DAG files were generated to solve each of the subproblems in sequence. The CNF of all these subproblems together can also directly be passed to traditional SAT solvers for resolution. In this way the execution time required to solve the subproblems together can be measured by traditional SAT solvers against our Dagster engine. and the results of some of these runs are shown in figures 4 and 5. In these figures is shown the execution time required for Dagster and Minisat to come to a solution of the generated problems for different clause multiplier m in the generation procedure per Algorithm 3. The graphs show the median solving time (with a 60s timeout) together with the inter-quartile range (between the first and third quartiles) on the solution times. Figure 4 shows the execution times when the number of overlapping variables between the subproblems is 5, and Figure 5 shows the execution times when the number of overlapping variables between the subproblems is 15.

We notice significant differences between the performance of Minisat and Dagster depending on the clause multiplier m, particularly that when m is small, all of the subproblems are markedly under-constrained and it is easy and quick for both Minisat and Dagster to generate a solution to the problem. However as m increases, and the problems become marginally more constrained, then the performance of Dagster suffers particularly for an interface size of 15. This is primarily because an under-constrained problem generates a large number of assignments between the interfaces of the problem (particularly when the interface size is large) , and when the problem is sufficiently constrained such that most of these are ul-

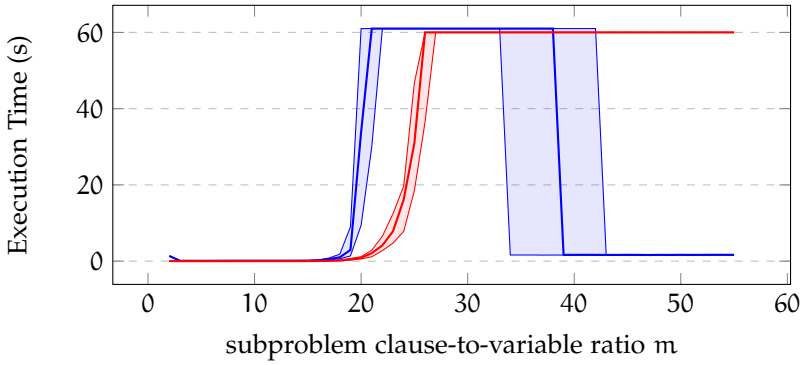


Figure 4: Runtime performance (medians and interquartile ranges) of DAGSTER (as blue) and MINISAT (as red) for structured problems given by algorithm 3 with few overlapping variables ($s = 100, p = 5, s = 4, c = 5$)

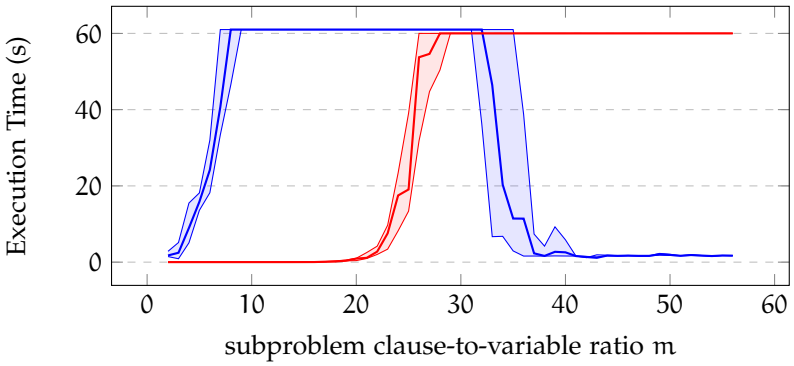


Figure 5: Runtime performance (medians and interquartile ranges) of DAGSTER (as blue) and MINISAT (as red) for structured problems given by algorithm 3 with many overlapping variables ($s = 100, p = 15, s = 4, c = 5$)

timately discarded in the subsequent processing it is unsurprising that DAGSTER suffers. However in this region MINISAT still has performance, as it tackles the problem holistically, and opportunistically solves particular parts of the problem in an order that suits it.

There is a middle region where both MINISAT and DAGSTER struggle, where m is high enough for the problem to be intractable for MINISAT, but not yet over-constrained enough for DAGSTER to be advantageous. And there is a final region of higher m where the subproblems begin to be more constrained, and where solutions of one part of the problem begin to be materially constraining on the solution to other subproblems, and in this region DAGSTER has the opportunity to occasionally pull ahead and solve the problem again. Ultimately in the limit where m is very large, and all subproblems are over-constrained then a solution of one subproblem effectively solves all subsequent subproblems (perhaps even by unit-propagation in the extreme case), and in this case DAGSTER pulls ahead decisively as it solves all subproblems in order, whereas MINISAT stalls as it considers all subproblems together holistically.

These figures constitute an example of a family of structurally generated random SAT problems in which DAGSTER has demonstrated performance improvement against MINISAT.

8 CONCLUSION AND FUTURE WORK

We have presented Dagster, a Boolean SAT solving tool that can distribute and schedule search activities in HPC environments according to a given problem decomposition. The focus of our activities moving forward is to exploit this tool in a range of settings. Working collaboratively with students who have already begun their ancillary projects, we will be developing a test suite and evaluation of this approach in the setting of bounded model checking of safety properties, and in particular such problems as they occur in software verification and protocol analysis settings. We will be exploring decompositions of structured (from industry and applications) SAT benchmarks according to published schemes, and examining the effectiveness of those in our setting.

REFERENCES

[1] Stephen A. Cook. The complexity of theorem-proving procedures. STOC '71, New York, NY, USA, 1971. Association for Computing Machinery.

[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. Sch oning. 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] 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.

[10] Youssef Hamadi and Christoph Wintersteiger. Seven challenges in parallel sat solving. 2012.

[11] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. *F-Race and Iterated F-Race: An Overview*, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[17] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 682–687, 2004.

[18] 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.

[19] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Mag.*, 34(2):99–106, 2013.

[20] 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.

[21] Marijn J.H. Heule and Hans van Maaren. *Look-Ahead Based SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 5, pages 155–184. IOS Press, Amsterdam, handbook of satisfiability edition, February 2009.

[22] 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.

[23] Djamal Habet, Lionel Paris, and Cyril Terrioux. A tree decomposition based approach to solve structured sat instances. In *ICTAI*, pages 115–122. IEEE Computer Society, 2009.

[24] Eyal Amir and Sheila Mcllraith. Solving satisfiability using decomposition and the most constrained subproblem (preliminary report). *Electron. Notes Discret. Math.*, 9:329–343, 2001.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers, 2015.

[30] Antonio Ramos, Peter van der Tak, and Marijn Heule. Between restarts and backjumps. In Karem A. Sakallah and Laurent Simon, editors, *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 216–229. Springer, 2011.

[31] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[32] 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.

[33] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bull.*, 3(1):8–12, January 1992.

[34] Ian P. Gent and Toby Walsh. An empirical analysis of search in gsat. *J. Artif. Intell. Res. (JAIR)*, 1:47–59, 1993.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Jerry Lonlac and Engelbert Mephu Nguifo. Towards learned clauses database reduction strategies based on dominance relationship. *CoRR*, abs/1705.10898, 2017.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Craig A. Knoblock. Automatically generating abstractions for planning. *Artif. Intell.*, 68(2):243–302, 1994.

[50] 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.

[51] Haixia Jia, Cristopher Moore, and Doug Strain. Generating hard satisfiable formulas by hiding solutions deceptively. *J. Artif. Intell. Res.*, 28:107–118, 2007.