Dagster: Parallel SAT

Checkpointing and Performance Progress Against Project Activities

MARK BURGESS*, CHARLES GRETTON,
LUKE CROAK, TOM WILLINGHAM

November 16, 2022

ABSTRACT

Dagster is a parallel SAT solver that implements a new approach to scheduling interdependent (Boolean) SAT search activities in high-performance computing (HPC) environments. In this report we outline new usability features that have been added to Dagster in 2022, including a *checkpointing* functionality that allows users to suspend a distributed search, and then later resume that seamlessly. We also describe recent performance enhancements that have been integrated in Dagster, including the integration of the MiniSAT algorithm as a choice of CDCL procedure, and a range of algorithm optimisations. We also outline new user interface modules to configure and monitor Dagster processes. The performance enhancements and algorithm integration are showcased via a series of case studies, particularly, in updated pentomino and Costas array problems, where we compare the performance of the latest version of Dagster against an array of state-of-the-art systems. We also consider a new case study, showing how Dagster can be used to solve a Bounded Model Checking (BMC) problem using a new abstraction hierarchy.

All Authors: *School of Computing, Australian National University, Canberra, Australia.*
*Corresponding Author: mark.burgess@anu.edu.au
We would also like to credit Josh Milthorpe, who was previous a core part of the Dagster project

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

DAGSTER is a parallel structured high-performance computing (HPC) Boolean SAT solver that implements a new approach to scheduling interdependent search activities in HPC environments. This system allows practitioners to solve challenging problems by efficiently distributing search effort across computing nodes in a customizable way. Our solver takes as input a set of disjunctive clauses (i.e., DIMACS CNF) and a labelled directed acyclic graph (DAG) structure describing how the clauses are decomposed into a set of interrelated search problems. Component problems are solved using standard systematic backtracking search, which may optionally be coupled to (stochastic dynamic) local search and/or clause-strengthening processes. A number for performance and usability improvements have been made to DAGSTER in recent times, and in this report we examine the performance gains realised in combinatorial case study examples, particularly the model counting of Costas arrays, and in finding solutions in large pentomino tiling problems. We also explore a case study that develops a novel workflow for Bounded Model Checking using the compositional approach to distribute search using DAGSTER.

We outline a range of changes and contributions to the DAGSTER project, before considering the case studies. We conclude with a summary of future work and directions.

1.1 Summary of New Contributions

Since the last report on the state of the DAGSTER system we have implemented a range of new features and performance improvements that we summarise below. In Section 3 we evaluate the performance of the latest version of DAGSTER, contrasting the performance of our parallel system with state-of-the-art parallel SAT tools described in recent scientific literature.

Primary improvements to the DAGSTER system include:

- Detailed in Section 2.1, to improve usability we have developed a *Text User Interface* (TUI) wizard for configuring DAGSTER.
- Detailed in Section 2.2, to improve usability with regard to monitoring search performance, we developed a new TUI monitoring tool for interpreting the logs generated by DAGSTER. Logging information is now displayed by the TUI in a human readable graphical format. Using the TUI users can monitor search progress and performance of DAGSTER.
- Detailed in Section 2.3, is the introduction of checkpoint functionality, whereby DAGSTER takes regular checkpoints of its search progress. A DAGSTER search can now be resumed in the event of a search being suspended, or the event of a system crash.
- Detailed in Section 2.4, we developed a new modular interface for integrating systematic search procedures. We have used that interface to integrate the MINISAT CDCL procedure into DAGSTER.
- Detailed in Section 2.5, we developed an operational mode, called *negative literal purging*, whereby only positive literals are communicated between subprocesses. When admissible given the compositional structure of the problem at hand, this provides an important performance optimisation, reducing the size of solutions recorded and messages between subprocesses.
- Detailed in Section 2.6, we created a new solution-length reporting optimisation in DAGSTER, to improve performance. Solutions recorded and communicated between processing elements are now shortened, by workers, in a more effective manner. Workers remove literals that are are logically redundant

to satisfying the target subproblem, with an explicit bias towards reporting positive literals.

- Detailed in Section 2.7, we developed a new geometric restarting strategy in the default CDCL procedure, implemented to accelerate the performance of Dagster search processes in model counting workflows. This new strategy improves solver performance by reducing the time the solver spends proving that no further model exist in counting scenarios, and has minimal impact on the efficiency of enumerating solutions.
- Detailed in Section 2.8, we developed a new (and optional) memory optimisation mode, whereby representations of subproblems are generated on demand by subprocesses, rather than all being represented explicitly in memory.
- Detailed in Section 2.9, we developed some additional performance gains achieved in the variable neighbourhood calculation and lookup processes in the SLS modules.

2 NEW CONTRIBUTIONS

2.1 Usability Feature: TUI Initialisation Interface

The Dagster system is initialised by command line, with arguments detailing the inputs and configuration options adopted by the solver. Currently there are many options supported in the Dagster system, 24 in all, and these options are specified by alphabetic command line switches. Of the 26 letters in the alphabet, all but 2 are now assigned, and the relationship between what letters match to what configuration option is not user friendly. A more user friendly approach, is to design an interface that guides any new users through the process to configure Dagster appropriately, and explains the uses and abuses of each configuration option.

A guided interface has now been implemented as a separate program from Dagster proper, and is bundled with the latest release in order to guide users through the configuration options available, and to output the appropriately formatted command line string to initiate a desired Dagster run. Thus, Dagster can be compiled and run irrespective of whether users wish to employ our new optional guided interface module. An emitted command line string can be copied and re-input into the command line, to kick-off the program with those options at a desired time (such as may be shared with colleagues for experimental reproducibility), or the interface can directly invoke a Dagster run in the users environment.

There were several options available for the coding of a guided interface module and a Text User Interface (TUI) was deemed appropriate. This was particularly because a TUI is able to be run in a console, and thus able to run on local computers with and without graphical operating systems, it is also highly compatible with running on remote systems via remote shell. The TUI interface was configured to display a nested set of configuration options that are topically described in a hierarchical fashion. As the Dagster system continues to mature, the command line interface and options available are expected to change and grow, and so the interface module was coded quickly and for ease of modification and adaptation using Python programming language, utilising the UrWID widget library.[1]

The resulting TUI configuration wizard interface is shown in Figure 1. In this interface there are check boxes and nested windows of configuration options, that enable and explain the various options which are possible in Dagster. The user uses the keyboard and arrow keys to navigate around the interface, and input appropriate arguments. Additionally, a submemu can be expanded by hitting the

1 https://urwid.org/

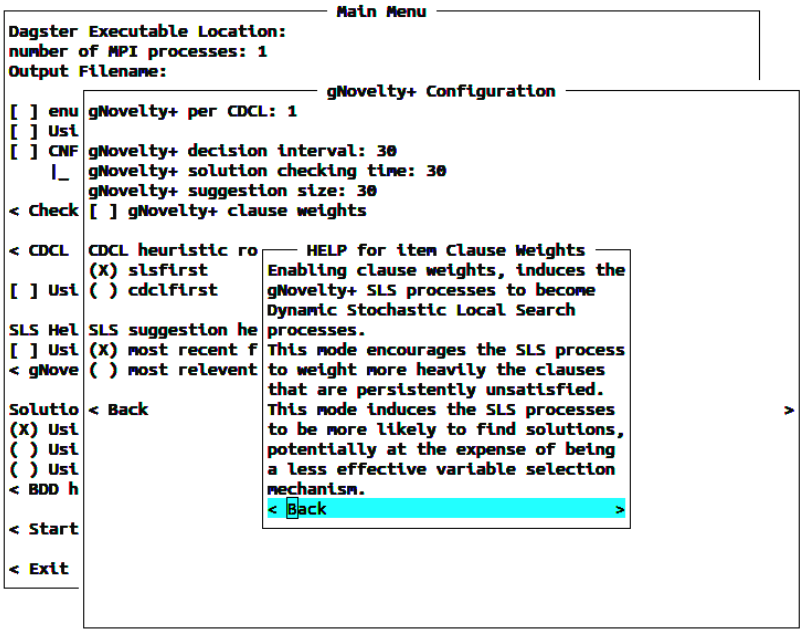


Figure 1: The DAGSTER TUI configuration wizard interface. This demonstrates the description of the Dynamic Local Search option the user can select, for the local search helper processes.

'enter' key on the appropriate menu button. The '?' key can be hit to bring up a dialogue box explaining any highlighted item.

The TUI configuration interface must be run in an environment with an appropriate version of Python, with appropriate libraries installed (particularly the UrWid library), and can be initialised from the command:

```
python ./wizard.py
```

This TUI configuration wizard allows a more user friendly way of beginning and configuring a DAGSTER run.

2.2 Usability Feature: TUI Monitoring Interface

We provide an new interface that allows users to monitor a DAGSTER run, and to assist in diagnosing any issue that might arise. DAGSTER emits log messages while it is running. For example, logs can be to standard error (*stderr*), standard output (*stdout*) and/or to appropriate log files (as configured by Google Logging library (GLOG) environment variables). The verbosity of logging is defined by an environment variable (`GLOG_v`), and the log messages hold many kinds of messages, such as: timing information, debugging hints, message counts, worker process status updates, and various possible error messages. At verbosity 2 or higher, the log messages are sufficient to give all the most relevant information about the progress of a run. In principle, parsing these logs allows users to check on the health of a DAGSTER solution process, and allows inspection and diagnoses of bottlenecks, or of stalling between subproblem nodes of the DAG. However, scanning through log files to ascertain the overall state of the DAGSTER system dynamics is not a user friendly process. Thus, a program to automate this task was seen to be desirable, and is now provided with the latest DAGSTER release.

In order to diagnose the overall state of the DAGSTER system progress in solving a problem, the most relevent information is about the progress of the workers in generating/processing the messages passed between DAG nodes. The depiction

of the DAG with messages flowing between nodes, and the progress workers have made on nodes, constitutes a summary of overall DAGSTER progress that is amenable to graphical representation. For the reasoning as given in Section 2.1, we developed a separate graphical TUI module that parses the relevant log files, and pulls out the most relevant information for graphical display. It is able to provide a graphical view of the DAG, along with statistics about worker progress and any relevant error messages.

A section of the TUI is shown in Figure 3. From this interface, we can see on the left hand side a text based graphical viewer of the DAG nodes, by their index, and the connections between them. Next to that, we can see that there are three worker processes actively working on various DAG nodes. Also rendered is the exact number of messages/solutions passed between each of the nodes of the DAG – I.e., specifically the number of solutions ingested, completed, and output to subsequent nodes for processing. For each node, the TUI also reports the timing statistics for how long it takes for each worker process to complete a message, including the average (and standard deviation) of seconds it takes to find one (or any) solutions for each of the messages, along with how long it takes for workers to prove that there are no further messages before completing the message. At the bottom of the TUI we render the miscellaneous log messages, which can be useful for error diagnosis.

Figure 3 depicts the interface when rendering runtime information related to the DAG shown in Figure 2. Here, Node 0 connects to Nodes 1 and 3, Node 1 connects to Nodes 2 and 4, etc. The reader can see that there are three worker processes working concurrently on Nodes 1, 2 and 6. Node 0 has fully completed one message inwards, and has completed giving 3 messages of output. These output messages have been given to Nodes 1 and 3, with Node 3 taking one of these to produce an output message that has been given to Node 6 – I.e., which is currently being worked upon, etc. Below this we can see an update from the worker logs, identifying themselves as working on Nodes 1,2 and 6. The master process also periodically logs timing information related to workers on assigned nodes, the interface here reporting their performance when working on Nodes 1 and 3. The TUI also renders a Master health-status field, which is currently blank, indicating it is normal; and an 'Exit' button. The 'unparsed' console message field will contain messages that have not been interpreted and rendered elsewhere in the TUI. Serious error messages will be rendered here, in case a serious error occurs.

We note that the DAG shown in Figure 2, and the problem that is being run in this instance, as shown in Figure 3, is actually a pentomino problem whose generation is explained in later Section 3.2.

The sequence of commands used to generate the screenshot shown in Figure 3 was as follows. First, there was Google logging library (GLOG) environment variables specifying that it was to output logs to STDERR, and at log level 3. DAGSTER was initialised with 4 processes (1 master and 3 workers), and to pipe standard error to a file named 'log_output.txt':

```
export GLOG_logtostderr=true
export GLOG_v=5
mpirun -n 4 ./dagster <DAG> <CNF> 2> log_output.txt
```

Secondly, in a separate shell, the following call was used to initialise the TUI monitoring program to parse the DAGSTER generated log file.

```
python ./viewer.py <DAG> log_output.txt
```

The particular CNF and DAG file which was used was a pentomino problem generated as per Section 3.2, consisting of solving an 3x3 array of pentomino tiles.

The monitoring process allows a visualisation of how DAGSTER is progressing throughout the problem solution process. However, the interface is primarily passive, consisting of monitoring the logs and outputting relevant information to the screen for users to see. In order to change the way the DAGSTER program is running, pausing it and/or halting it, it is necessary to store the progress and/or potentially reinitialise the DAGSTER system with that progress established. To do this, there is a necessity of being able to store the progress that DAGSTER has completed, via 'checkpointing'.

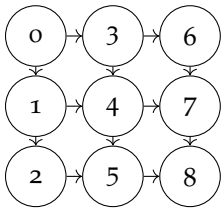


Figure 2: DAG for a 3x3 pentomino superproblem - Decomposition A, further described in Section 3.2

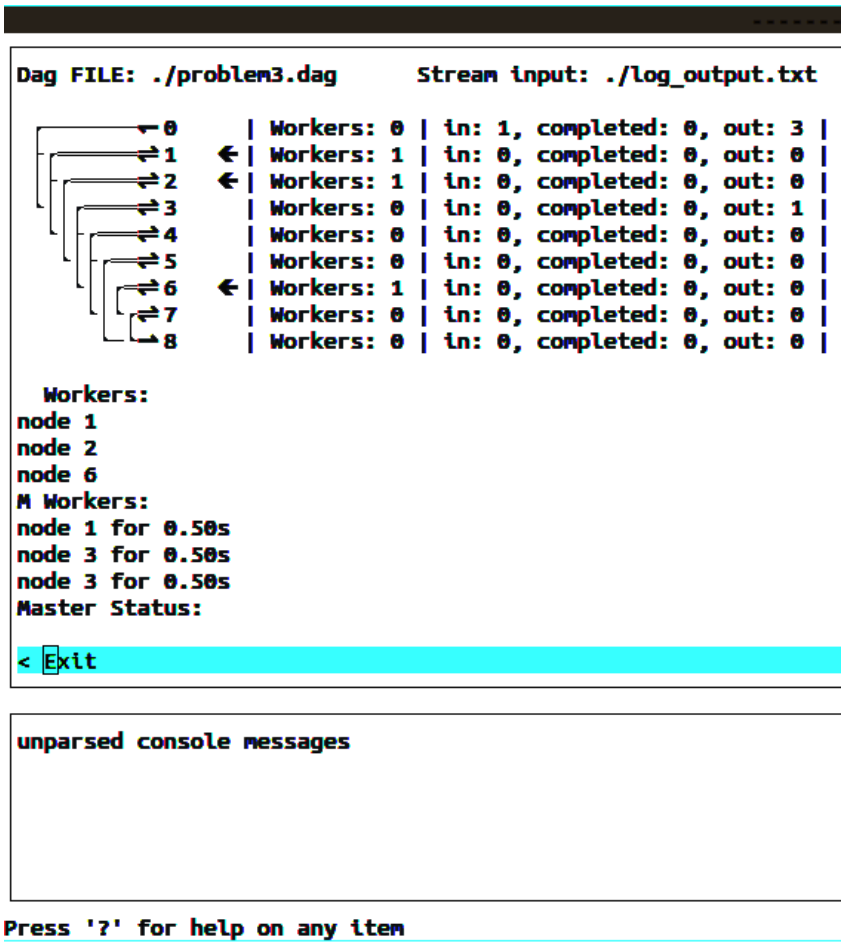


Figure 3: The DAGSTER TUI monitoring interface

2.3 Usability and Performance Feature: Checkpointing

One primary design aspect of Dagster is the ability to initialise, monitor, and process hard and/or difficult SAT problems. In the process of solving long-running SAT problems, the priorities for HPC usage can motivate the user to interrupt the Dagster run, also the system can potentially crash. This motivates the need for a backup functionality, which is coded in Dagster as a checkpoint functionality. The checkpointing functionality allows Dagster to store its progress, thereby enabling reinitialisation from a checkpoint, as desired.

The checkpointing functionality is initialised in the Master process, as the master not only holds the details of what work needs to be (and has been) done, but is therefore also most able to have responsibility for monitoring and backing up (and/or reloading) that work. The checkpointing functionality of Dagster is optional, and configurable. Master can be configured to dynamically format and dump all the relevant progress data to a series of custom files, with dumps occurring at equal time increments. These files are written, and then periodically overwritten, in a round-robin fashion by Dagster, so that there always is a sequence of recent checkpoints of progress. These checkpoints that Dagster produces can be then loaded back up into Dagster, from the command line, in the event that a search is interrupted to be later resumed.

The checkpoints hold all of the logical information relevant to a solution process, and are not tied to any specific configurations of the Dagster system. In this way, a checkpoint generated on one system can be loaded into Dagster on a different system with a completely different configuration of search processes, and types of search. The primary constraint on the flexibility of the checkpoint functionality, is that checkpoints generated when the Master is using BDD representation of system progress, are incompatible with checkpoints generated when Master process is using tables to represent messages and system progress - and vice versa. This incompatibility arises because these two representations are fundamentally different, and not directly compatible.

The new checkpointing functionality is exposed by command line options, where the user is able to specify: *(i)* where the checkpoint files are to be placed and named, *(ii)* how often checkpoint files should be generated (if at all), and *(iii)* what (if any) checkpoint file should be loaded on initialisation.

2.4 Performance Feature: Interface for Integrating Systematic Search Algorithms(e.g., CDCL)

Dagster was originally built with all the CDCL worker processes implementing a systematic search based on TiniSAT [**?**, **?**]. Modifications to the TiniSAT algorithm were made to support the interaction with local search processes, that provide search guidance and solutions, and with possible strengthener processes that dynamically simplified TiniSAT's learned clauses. Recent experimentation with pentomino problems—see results in Section 3.2—motivated us to enable Dagster to more easily support different search algorithms. For example, TiniSAT is less effective than an alternative CDCL solver, namely MiniSAT [**?**], at pentomino problems. Therefore, being able to parametrically select to use MiniSAT, rather than TiniSAT, as the base systematic search procedure is motivated.

Dagster was modified to incorporate MiniSAT as an option for the worker processes, and the modular interface implemented for these purposes can also support the integration of other CDCL solvers. The modular interface more readily allows any CDCL solver to now be integrated with Dagster, by inheriting a virtual class in C++ with appropriately overloaded methods. In this way we anticipate that it should be easily possible to incorporate further CDCL solvers as options into the Dagster system. We have yet to support all the integrations between subsolver

types for such interactions, such as local search based guidance and strengthening of learnt clauses.

2.5 Performance Feature: Negative Literal Purging

Many SAT problems, and subproblems, feature solutions that are entirely (and immediately) derivable by an assignment over positive literals only. Such a fact is evident in our experiments, particularly the Costas and pentomino problems in Sections 3.1 and 3.2, where the relevant information between subproblems consists entirely of where the points/pentominoes are, and thus from the positive literals in a solution valuation, all the negative information about where the points/pentominoes are not is then immediately implied. By having workers eliminate negative literals from solution records, the messages between processing elements can be greatly minimised. In order to experiment and take advantage of this fact, we created a new Dagster mode for having workers purge negative literal information from messages and solution records. In this way a lot of communication overhead in these problems is reduced, resulting in a material runtime performance enhancement.

A consequence of this option is that Dagster is potentially unsound, because purging of negative variable information along the arcs of the DAG may mean a loss of solution information in specific problem cases where negative information is necessary. In principle, binary variables and their participation in a SAT problem can be coded negatively, or positively, just as effectively. This new mode of operation effectively primes Dagster to take advantage of a particular bias in the way that people tend to code problems into SAT - particular that positive information is usually coded positively, and that negative variable evaluations are often consequences of that.

The unsafe purging just described was seen to be effective at improving solution times in pentomino problems (see Section 3.2), however a safer alternative approach was sought. Described next in Section 2.6, an approach compliant with soundness and completeness is described, which is computationally more expensive than the negative literal purging approach just described.

2.6 Performance Feature: Literal Trimming

CDCL solvers by default (such as TiniSAT and MiniSAT) report a valuation over all the variables when they find a satisfying valuation. However, this full valuation is potentially overspecified, as a subset of those variables may be able to satisfy the CNF. For example, consider the CNF shown in Figure 4. The valuation $[1, -2, 3, -4, 5, 6, 7]$ solves this CNF, however there is redundancy as more variables are assigned than is necessary to satisfy the CNF. Specifically, the valuation $[1, 3, 5, 6, 7]$ satisfies the CNF without specifying the truth values of variables 2 and 4, and a shortened valuation such as this encompases a solution to the CNF with thoes variables specified any way. In this way, the shorter the subproblem solutions encountered by Dagster the less work Dagster potentially has to do in enumerating the span of subproblem solutions. Thus, with shorter valuations, Dagster is able to compute and record a smaller and more succinct representation of a set of solutions, thereby performing less search and using less system memory.

The Dagster workers (optionally) implement a solution trimming procedure, where after a worker generates a solution to a given subproblem, it then analyses what variables can be removed from that solution such as to leave the SAT problem immediately satisfied irrespective of which way those trimmed variables would be assigned. Previously, Dagster workers implemented a more rudimentary trimming system, eliminating all variables which were not present in the first satisfied variable of every CNF clause (considered in turn). To contrast the new approach with

CNF file

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

Figure 4: An example CNF file

our historical trimming approach, consider the valuation $[1, -2, 3, -4, 5, 6, 7]$ to the problem in Figure 4. Previously, a DAGSTER worker would take the first clause, $(2\ 1\ 0)$, and conclude that 1 must not be trimmed because it is the first literal that satisfies this first clause. Then, that worker would take the second clause, $(-2\ 3\ 0)$, and conclude that -2 could not be trimmed because it was the first variable which satisfied the second clause. The worker would continue thusly, concluding that the valuation $[1, -2, 3, -4, 5, 7]$ was sufficiently trimmed with the only exclusion being the assignment to 6. Our new trimming algorithm prioritises trimming of negative literals in a valuation, owing to the bias of human SAT encodings, that positive literals are generally more informative. With reference to the total valuation provided by the underlying search procedure, the new algorithm scans through each clause in turn, maintaining a monotonically increasing list of literals in the solution valuation it will report, as follows:

1. If the clause is not already satisfied by an element in the reporting list, then
2. If there is a positive literal in the total valuation which satisfies the clause, then the worker adds the positive literal with the lowest absolute value to the reporting list.
3. Otherwise, the worker adds the lowest absolute value indexed negative literal to the reporting list.

After iterating over all clauses, only the literals in the reporting list are recorded as the discovered solution. Continuing with the example from Figure 4, the run of this trimming algorithm, given a total valuation $[1, -2, 3, -4, 5, 6, 7]$, would take the first clause $(2\ 1\ 0)$, and see that 1 would be the lowest absolute value positive valuation in the solution to keep to satisfy the clause. Processing the second clause, $(-2\ 3\ 0)$, and it finds that 3 is the smallest absolute value positive literal satisfying the clause, which also satisfies clause $(4\ 3\ 0)$. For clause $(5\ -4\ 0)$, literal 5 is added to the reporting list, etc. The reported solution by a worker is $[1, 3, 5, 6, 7]$, which is the total valuation excluding the assignments to the two variables 2 and 4. It is both more effective as a trimming algorithm, and also has a direct bias towards keeping positively valued literals.

The optional inclusion of this trimming algorithm, is marginally more computationally expensive than the negative literal purging process described in Section 2.5, but unlike that approach is provably sound and complete.

2.7 Performance Feature: Geometric Restarting Strategy

The efficiency of systematic backtracking search to solve the SAT decision problem is premised on a restarting strategy **[?]**. A backtracking decision procedure is said to restart when the current variable assignments being investigated are abandoned, and the search directed to start again with zero assignments made. The benefit of restarting has been demonstrated theoretically, with restarting noted as the feature

that enables a clause learning search to polynomially simulate the powerful general resolution procedure [?, ?, ?]. In practice, mechanisms for restarting are also recognised as important [?, ?, ?, ?], with adaptive search policies characterised by restarting featuring in all performant algorithms. The default CDCL algorithm implemented by Dagster systematic search processes—which is based on the TiniSAT algorithm—was indeed motivated by the recognition of the importance and the study of restarting [?]. That base procedure offers a great deal of flexibility, regarding what restarting mechanisms researchers can implement, investigate and study. We discuss a new restarting mechanism we have implemented in that default systematic search of Dagster.

Prior to describing our new restarting mechanism, it is worth motivating the need for our new approach. When using Dagster to solve #SAT problems, search processes use the same CDCL algorithm to drive both: *(i)* model enumeration, and *(ii)* the exercise of proving that no further models exists – i.e., performing an UNSAT proof, which is ideally done using a distinct restarting mechanism [?]. We note that, even when the underlying Dagster query is to solve a SAT decision problem, and not a #SAT counting problem, due to the compositional structure of the associated DAG the query can still pose subproblems that require subproblem model enumeration. Our new restarting mechanism adapts the frequency of restarting, so that when enumeration becomes difficult the search rapidly shifts towards an aggressive restarting approach that is relatively efficient at proving UNSAT. We find that our dynamic approach dramatically improves the efficiency of proving UNSAT, and of discovering "hard-to-find" models in a counting problem. Specifically, our new "geometric" restarting approach is very important for the best results we document for the Costas case study below.

With geometric restarting, the search implements restarts based on a dynamic probability-of-not-restarting (PnR) value. Whenever a search discovers a new satisfying model to the subproblem it is assigned, the PnR value is set to its initial value, 1. Whenever n conflicts are encountered,[2] the PnR is updated to be its current value multiplied by a discount factor $0 \leqslant \alpha < 1$.[3] Thus, if the search is not fruitful the probability of restarting approaches 1, geometrically, and whenever the search yields a satisfying model this is reset to 0. Prior to the PnR being updated, a restart event is triggered with probability $\rho = (1 - \text{PnR})$.

To our knowledge we are the first to consider this dynamic restarting mechanism, and note that it is peculiar to our setting, where a systematic search is assigned to enumerate models to subformulae in the context of a larger monolithic search exercise. Our new search mechanism is motivated to accelerate the enumeration process, and to free up search processes relatively quickly when no more subproblem models are available.

2.8 Performance Feature: Alternative Modes of Memory Operation

Dagster takes a CNF and DAG file as input, specifying the problem and its decomposition respectively. The system can then handle this CNF file in different ways depending on configuration.

Currently there are three configurations:

1. The input CNF is loaded from file into the memory of every process, with all subproblems specified in the input DAG stored in memory.

2. The input CNF is split into an array of CNF files, each corresponding to a subproblem from the input DAG. Workers only input and hold the CNF file of the subproblem they are working on.

2 Value n is a hyperparameter set to 800 by default.
3 Value α is a hyperparameter set to .95 by default.

3. ⋆ **N**EW⋆ The input CNF is scanned by all workers, with those storing only the clauses associated with the currently assigned subproblem.

Each of these modes strikes a balance between computing speed, and memory requirements for larger CNF problem instances. SAT problems in CNF can range from Kilobytes to Gigabytes, and when there are many workers each having copies and working on larger CNF subproblems, this can quickly exhaust the available memory.

For the smallest CNF files, it is most efficient to choose Option 1 of these configurations. The whole CNF is loaded into the memory of every worker, and each worker does all possible preprocessing on their own copy of the CNF, involving splitting it into CNFs corresponding to all subproblems stored in memory. In this way, each worker has maximal speed in directly accessing and manipulating its own local information in memory, involving the full and split problem CNFs. Option 1 maximises speed, but comes at the cost of having a greater memory requirement for each worker process, and therefore the Dagster system overall.

For the very largest CNF files, it is better to use the Option 2 configuration, where at no point is the whole problem CNF itself ever loaded into memory, but the Dagster system leans on file storage for the storage and manipulation of CNF parts. In this way CNF files potentially larger than the computer has memory can be handled and solved. This configuration option is ideal for larger CNF files, but comes at the cost of having to load and reload CNF subproblem files as the Dagster run progresses, requiring greater amount of runtime consumed by file IO operations.

The new Option 3 configuration is a middle ground between the Options 1 and 2, where each worker parses and holds the original input CNF, but only the required target subproblem CNF is generated and stored in the worker's memory. In this way each worker operates in its own local memory space, but there is larger amount of IO and memory manipulation as the Dagster run progresses.

2.9 Performance Feature: SLS Neighbourhood Calculations

Part of the startup process of the local search module, based on gNovelty+, is the process of calculating the sets of clauses connected to each variable, and the variables that are thereby neighbours in these clauses. In the process of the SLS operation, variable values are flipped, and the scores associated with each of the variables are updated. The score associated with a variable during search reflects the net number of clauses satisfied by a variable if it were to be flipped. When a variable is flipped this causes some clauses to be satisfied and others to become unsatisfied, thus influencing the score of the flipped neighbourhood of the flipped variable. For this process of updating the score of variables that are induced by a flip of one of them, it is expedient for the computer to have a store of what variables are a neighbour of each other. However, there are several issues creating this data store of variable neighbours, particularly as larger CNF files often feature several millions of variables, many of which may be neighbours of each other; and thus raw storage of this information can constitute Gigabytes of memory, consequently the process of calculating and/or accessing this information can become prohibitive. The creation of the variable neighbourhood data structure involves the insertion of many pairwise variable integers into a larger data set which preserves these pairs uniquely, and this information is then read many times over as the SLS progresses. For this purpose there is a range of possible data structures which can can be employed, each with potential tradeoffs. The following functions are included in the Dagster project which outlay different ways of calculating variable neighbourhood information.

PREVIOUS IMPLEMENTATION:

1. Create an array with a Boolean flag for each variable.
2. Then iterate over all the variables, one by one:
   - Scan through the clauses in which the variable occurs, and for each of the variable's neighbours, set the neighbour's flag in the array to be true.
   - Scan through the array for all variables whose flag is set, to determine the neighbourhood of the variable, which is then stored more compactly as a static list.
   - Clear the array of flags

This algorithm was previously implemented in the DAGSTER SLS routine, and while it is has constant insertion time (array look up and setting a flag), it also suffers from quadratic complexity in terms of the total number of variables, as each variable's neighbourhood is resolved by scanning over the array of flags, which has the size of the number of variables itself. This quadratic complexity was seen to be an immediate bottleneck on larger problems. However at the end of the algorithm, each variable has a static list of its neighbours, which is compact with a fast lookup. To ameliorate the issue of quadratic complexity in computation, we considered a different data structure to hold variable neighbourhood information, particularly we used the C++ template library SET to hold candidate variable neighbours and to purge duplicates, The first of two new implementations is as follows:

NEW IMPLEMENTATION 1:

1. Then iterate over all the variables, one by one:
   - Initialise an empty SET structure
   - For each clause in which the variable occurs, insert its neighbours into the set.
   - Scan through the set for all the neighbours of the variable, and then store the neighbours more compactly as a static list.

This algorithm improves the quadratic complexity over total variables in the computation, with a logarithmic insertion cost (over the size of the set) throughout the iteration. This logarithmic insertion cost is an inherent part of the C++ standard template SET structure, which stores and manipulates data in red-black trees. This much improved algorithm made it possible to run DAGSTER's SLS modules with larger SAT problems, with many more variables.

Unfortunately it was noted that the memory consumption of the SET datastructure, and the logarithmic calculation time became prohibitive on larger problem instances. To remedy these observed problems, a second alternative algorithm was implemented, using duplicate data structure to improve speed and memory performance. The second new implementation is as follows:

NEW IMPLEMENTATION 2:

1. Create an array with a Boolean flag for each variable.
2. Then iterate over all the variables, one by one:
   - Initialise an empty list structure
   - For each clause in which the variable occurs, and for each neighbour in that clause, check if the neighbours flag is set in the array, if it is not then set its flag, and add it to the list.
   - The list stores the neighbourhood of the variable.
   - Clear the array of flags

This implementation uses two data structures, a list and an array of flags, to hold neighbourhood information as it is being compiled, using the constant lookup time of the array of flags, to check uniqueness, and the list to store unique entries. The neighbourhood information is compiled directly as a list which is then used in the SLS routine. This implementation, uses linear operations for lookup, and insertion and compilation of information, without being as memory expensive as the SET data structures. Using this optimised implementation, the DAGSTER's SLS routine was able to work for larger SAT instances.

3 CASE STUDIES

In the previous sections we highlighted all the newer features and changes to DAGSTER since the last report deliverable. The contributions include a range performance optimisations, new functionalities, operating modes, and user interfaces. Using these feature additions, we were able to improve the runtime performance on several problems documented in the previous report, particularly Costas array counting and pentomino problems. In these case studies, some of the features which we introduced proved to be useful in providing performance gains relative to other solvers. A range of other solvers were newly considered as performance comparisons against DAGSTER. These performance comparisons are primarily shown in Figures 7 and 10. In addition to extending these two familiar case studies, we also introduce a new case study not included in the previous report, particularly using DAGSTER in Bounded Model Checking (BMC) between different variable fidelities using abstraction invariant variable values.

We consider three case studies:

- In Section 3.1, we consider performance in parallel model counting of Costas arrays, contrasting DAGSTER performance against the state-of-the-art DMC parallel model counter, showing performance enhancement due to geometric restarting, and SLS & Strengthener integration.
- In Section 3.2, we consider performance in solving generated pentomino problems, showing the performance enhancement provided by different decompositions, and the choice of DAGSTER's underlying CDCL solver against a wider range of state-of-the-art distributed and parallel SAT tools.
- In Section 3.3, we show the performance enhancement in a BMC case study, using DAGSTER's decomposition approach to carry abstraction invariant variable values between variable-fidelity models.

3.1 Costas Arrays

NEW FEATURES of DAGSTER that are being used when generating the experimental data reported on for this case study are: *(i)* new efficient search neighbourhood representations as described in Section 2.9, *(ii)* in Table 1 specifically, the reported runtime data is gathered with DAGSTER using the geometric restarting policy described in Section 2.7, and *(iii)* we use the sound and complete approach to trimming solution literals described in Section 2.6. □

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. An example Costas array is shown in Figure 5. Using search to solve for Costas arrays is known to be challenging, and searching processes have been conducted at least up to size $n = 29$ [**?**, **?**]. However, whether arrays exist at $n \in \{32, 33\}$ is an open problem. As a number of Costas array subclasses at those sizes have been eliminated [**?**], it is conjectured (but not confirmed) that Costas arrays of those sizes do not exist [**?**].

For each size n, we can directly synthesise a CNF formula whose models are in one-to-one correspondence with the set of Costas arrays of size n, and Lex-leader constraints can be added to break the dihedral group symmetries (reflection and rotation) [**?**]. For the purpose of using SAT-search to count unique arrays for values of n using DAGSTER, we employ a simple DAG structure, Figure 6, with two connected vertices. The first node is the placing of the first m columns of the Costas array, and the second part is the placement of the remainder.

The performance of DAGSTER at solving Costas problems for different sized Costas arrays n, and for different numbers of columns m solved in the first node, for different numbers of cores and DAGSTER modes, is given in Figure 7. Here, we also compare DAGSTER against the distributed model counter DMC [**?**], and against the performance of model counting by repeatedly calling TINISAT. This figure shows the runtime results of including clause strengthening and/or local search helper processes, and also shows the effect of two different decompositions on the performance of Costas model counting.

In these figures, the time taken to count all the Costas arrays of a given size is plotted, and we can see that for larger sized Costas arrays, DAGSTER is significantly faster than TINISAT at solving the Costas model counting problem, and that performance can be improved further using DAGSTER's additional feature modes. For smaller and easier Costas problems (of size $n \leqslant 12$) the parallel overhead of using DAGSTER is the primary determinant of the solution time, and DAGSTER performs worse than TINISAT which has minimal overhead; this overhead is most pronounced with the more granular decompositions that include more columns (e.g. for Costas-10, 3-column is worse than 2-column). However, for larger and harder Costas problems ($n > 12$), it is seen that DAGSTER consistently outperforms TINISAT - the CDCL procedure which DAGSTER is employing; as well as outperforming the DMC model counter. Additionally, for large problems, having one local search (modes denoted with 'L') and/or clause strengthening process (modes denoted with 'S') per worker group complements the CDCL procedure to yield improved runtime.

Another advantage of using DAGSTER is that it allows easy experimentation about the decomposition employed. In our results, we have considered decomposition

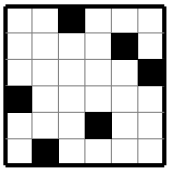


Figure 5: An example Costas array

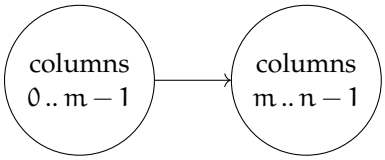


Figure 6: A DAG decomposing Costas problem into two parts.

Size (n)	Cores	Columns	S	Runtimes
9	48	{4}		5.527
10	2	{2,4}		1.356
10	48	{2,4}		5.191
11	2	{2,4}		4.155
11	48	{2,4}		5.642
12	48	{2,4}		21.234
12	48	{2,4}		7.267
12	48	{2,4}	✓	7.419
13	2	{2,4,6}		91.22
13	48	{2,4,6}		8.427
13	48	{2,4,6}	✓	10.242
14	2	{5,7,9}		524.414
14	384	{5,7,9}		16.562
14	383	{5,7,9}	✓	12.118
15	384	{5,7,9,11}		118.133
15	383	{5,7,9,11}	✓	86.125
16	528	{5,7,9,11}		275.127
16	527	{5,7,9,11}	✓	235.25

Table 1: Runtimes for different Costas problems with decompositions with columns in the first DAG node, with and without Strengthener (S) - i.e. modes C and CS (w/ TiniSAT cores); runtime in seconds, against cores and sizes.

into contiguous blocks of columns (where the first node has the first m columns, and the second node has the remainder), but we can also consider decompositions with interleaved columns. There are also many configurable variables related to restarting policy and variable selection controls. In particular, we considered a vanilla VSIDS heuristic [**?**] and a fixed geometrically increasing restart policy (as introduced in Section 2.7). In Table 1, we can see how changing the index of the columns that Dagster decomposes the problem by can increase the runtime performance.

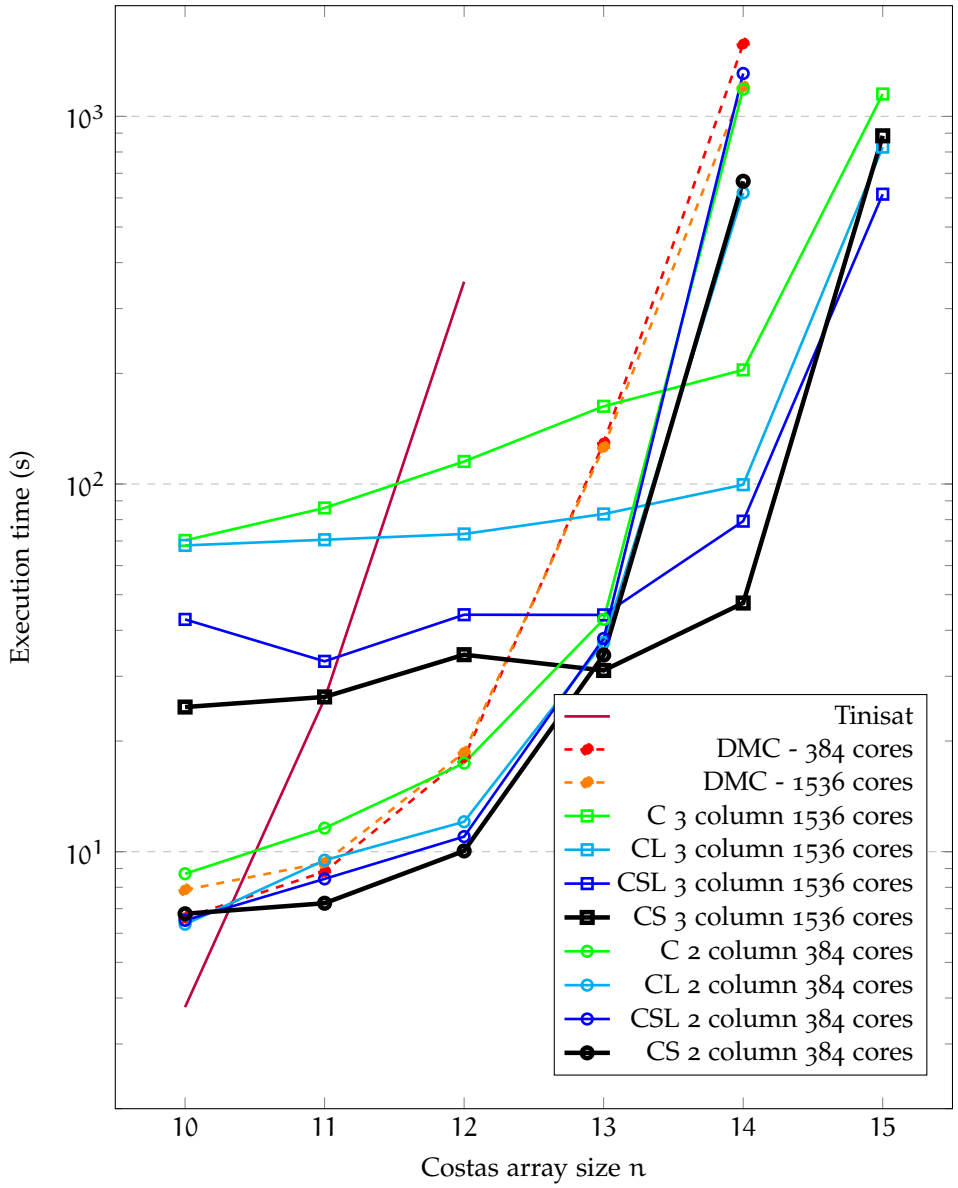


Figure 7: Runtime performance of DAGSTER against model counting with TINISAT and DMC on Costas problems with different number of columns in the decomposition and processor cores. C = CDCL only; CL = CDCL + local search; CS = CDCL + strengthener; CSL = CDCL + strengthener + local search; DAGSTER running with TINISAT CDCL core. One local search process per worker group.

3.2 Pentominoes

NEW FEATURES of DAGSTER that are being used when generating the experimental data reported on for this case study are: *(i)* we use the sound and complete approach to trimming solution literals described in Section 2.6, and *(ii)* we contrast the runtime performance of DAGSTER using the systematic search algorithm based on TiniSAT with the performance using the algorithms from MiniSAT, thus using the new search interface described in Section 2.4. □

We considered pentomino tiling problems where different tiling regions correspond to different sub-problems. The problems are to fill a grid area with pentominoes such that no pentomino crosses a bolded wall and no two pentominoes of the same shape (counting reflections/rotations) touch each other, an example pentomino problem is shown in Figure 8. We created a program to randomly generate hard 15×15 pentomino problems by:

1. Randomly filling a 15×15 grid with pentominoes,
2. Outlining those pentominoes with walls, and
3. Iteratively remove a random wall segment such that the puzzle is still uniquely solvable, until no further removals are possible.

We used this process to generate large pentomino problems by cascading 15×15 compatible sub-problems side-by-side together in a grid pattern. In this way, the grid of pentomino problems constitutes a larger problem with logically distinct parts, where each sub-problem is only constrained by its immediate neighbours. As every pentomino sub-problem is uniquely solvable, this larger pentomino problem is also uniquely solvable.

We considered two DAG structures (A and B, in Figures 9a and 9b) as possible processes of solving these pentomino problems: (A) from the top left diagonally through to the bottom right, and (B) solving the sub-problems in parallel by columns, and a final verification node.

We measured the performance of DAGSTER (with TINISAT and MINISAT CDCL cores, and both decompositions) against a range of solvers, including TINISAT , LINGELING and MINISAT serial CDCL baselines, and some parallel baselines including PAINLESS-MCOMSPS [?], PARACOOBA [?], DMC [?] and D-SYRUP [?] solvers[5]. The results for different sized pentomino problems are shown in Figure 10 where we see a range of different performances, with DAGSTER (Decomposition B and MINISAT core) outperforming other solvers.

DAGSTER demonstrates a speedup due to parallelization by solving a larger structured problems with coupled sub-problems, and that the strength of this effect depends on the decomposition and the number and type of CDCL cores used. The coupling between sub-problems creates leverage which DAGSTER exploits to provide

4 youtube.com/watch?v=S2aN-s3hG6Y

5 Some solvers we compare with here are not model counting tools, and thus not reasonably used in Costas results.

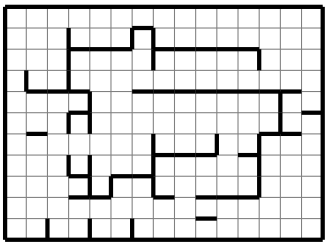
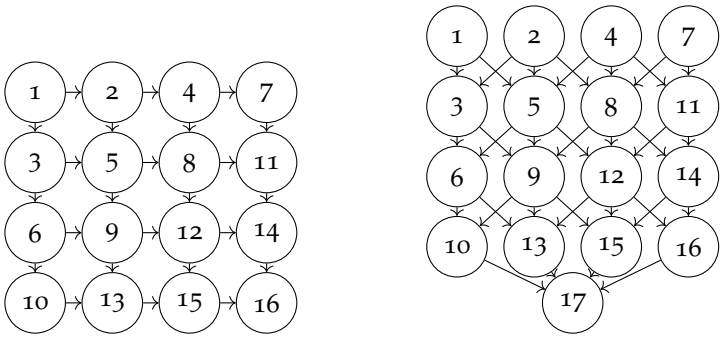


Figure 8: Pentomino puzzle featured on Youtube channel *Cracking the Cryptic* [4]



(a) DAG for a 4x4 pentomino superproblem - Decomposition A

(b) DAG for a 4x4 pentomino superproblem - Decomposition B

Figure 9: Two DAG arrangements for solving a cascaded grid of connected sub-problems

the witnessed speedup, and the structure and arrangement of the solving process between the sub-problem elements (between Decompositions A and B) can create a large difference in the resulting performance.

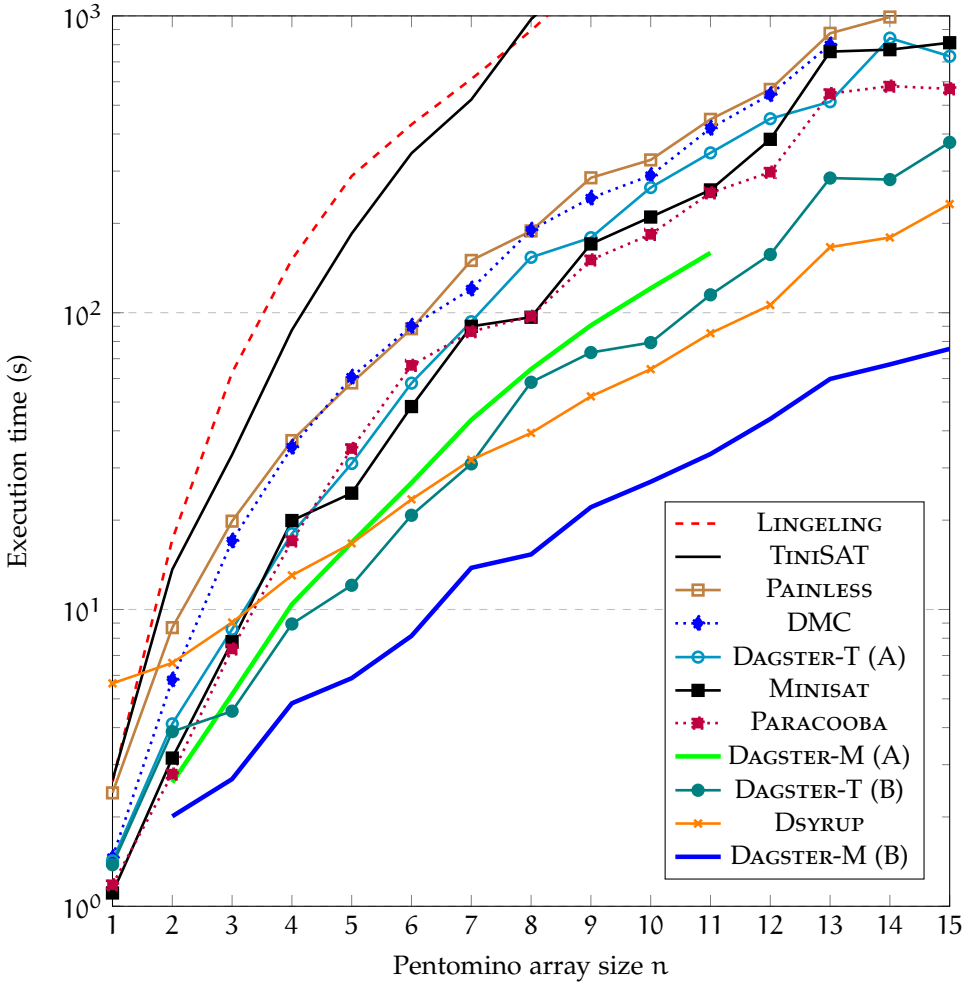


Figure 10: Runtime model counting performance (medians across 86 samples) of Dagster (w/ decompositions A and B, w/ TiniSAT (T) and Minisat (M) cores) against TiniSAT, Lingeling, Minisat, Painless-Mcomsps, Paracooba, DMC and Dsyrup solvers, for $n \times n$ arrays of 15×15 pentomino subproblems. all paralell algorithms use 17 cores.

3.3 Bounded Model Checking with Abstraction Invariants

THE NEW FEATURE of DAGSTER that is is used when generating the experimental data reported on for this case study is the sound and complete approach to trimming solution literals described in Section 2.6. □

We now describe a model checking case study, showing how DAGSTER can be used with existing tools to interrogate the functioning of finite-state-machines and circuits, and in particular to verify that a particular error state of the machine can be reached. Our checking processes will be based on search performed by SAT reasoning, as exemplified in [**?**]. A survey of approaches to model checking software systems is in [**?**], and we note a wide range of systems exist in this setting, including CBMC [**?**, **?**], F-SOFT [**?**], ESBMC [**?**], LLBMC [**?**], and ESBMC [**?**]. In our case study, we shall be using CBMC as the basis for generating structured SAT queries for DAGSTER.

Our case study considers the wireless security protocol for communication with a implantable low-power medical device described in [**?**]. *Alwen Tiu* determined *a priori* and by manual inspection that this protocol has a potential issue. The protocol is based on encrypted communication using a 32bit secret key K, shared between an implantable medical device (IMD) and a base-station (BASE). The IMD has a 32bit serial number S that uniquely identifies it among other devices. Both the IMD and BASE have a 32bit message counter, A for the devise, and B for BASE, with both counters initially set to zero. We use notation X&Y to denote bit string concatenation, and $\mathsf{Split}(A)$ to denote splitting a bit string A into two halves, and $\mathsf{Interleave}(A, B)$ to denote the result of interleaving A and B bit strings. We also write $\{A\}_K$ to denote the bit string A encrypted with K. For a message transmitted from the BASE to the IMD, and be accepted (i.e., not "dropped"), the following is required:

1. BASE has a 64bit message X (larger messages either chunked and/or padded into 64bits)
2. BASE adds one to its message counter B
3. BASE produces a message $M_1, M_2 = \mathsf{Split}(\mathsf{Interleave}(X, S\&B))$
4. BASE sends the message $\{M_1\}_K\&\{M_2\}_K$ to IMD
5. IMD receives $\{M_1\}_K\&\{M_2\}_K$ and decrypts each part with K then joins and de-interleaves to find X, S, B
6. IMD checks compatible S, if not match then drop message, it then checks message counter B against its own counter A. If $B > A$ it accepts the message and sets A to be equal to B, otherwise it drops the message

From examining this protocol, we note there is a weakness. Particularly, an adversary can witnesses a message $\{M_1\}_K\&\{M_2\}_K$ from the BASE to the IMD, and then subsequently send a message $\{M_1\}_K\&\{M_1\}_K$ to the IMD. This message from the adversary will then causing the IMD's message counter A to be incremented to S (perhaps a large number), consequently causing the IMD to cease accepting legitimate messages from the BASE. This error state in the protocol is subject to model checking, which can be done using DAGSTER.

We approach model checking this protocol compositionally, using DAGSTER, by appealing to a notion of process abstraction. Specifically, intending to proceed with CBMC, we faithfully describe the protocol in the C programming language. State variables describing the evolution of the protocol–e.g., whether an attacker or BASE is sending a message at timestep i—are of a fixed type. Variables encoding protocol registers, such as A, B, X, etc., being of a range of types, depending on where we are

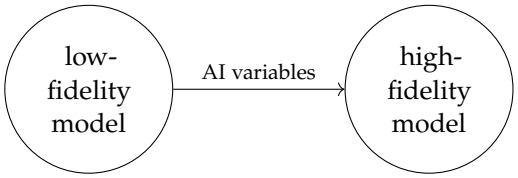


Figure 11: DAG for bounded model checking problem, showing AI variables being passed from the solution of low fidelity model to higher fidelity model

in an abstraction hierarchy. State variables of a fixed type we call *abstraction invariant* (AI). Our abstraction hierarchy then considers the other variables at a range of fidelities, with 8-bit registers modelling protocol instructions at the highest level of abstraction (lowest fidelity) and 64-bit registers at the lowest level (highest fidelity). We see that the protocol is much easier to model check, in practice, at a high abstraction level, and so our approach takes assignments to AI variables from satisfying assignments to highly abstract models, and uses those to inform search at lower levels of abstraction. A simulated run of the bidirectional communication from the base-station (BASE) to the medical-device (IMD) was written in C programming for different fidelities of their variables, and passed to the CBMC software to generate corresponding SAT instance problems with annotations for AI variables' bit values.[6]

We use CBMC to generate CNF representations of the protocol at different fidelities, producing annotated formulae that identify AI variables and the relationships between AI variables in different fidelity models. Dagster is used to automate the workflow, of solving the lower fidelity problem/s and then carrying across the AI variable values as constraints to the higher fidelity models - as indicated in the DAG shown in Figure 11. Here, we document the observed improvement in performance of this process, over running the higher fidelity models directly in a SAT solver. Our results are in Figure 12, where we can see that solving the 64 bit model using AI solutions from lower fidelity models results in a improvement in search performance. Particularly, in Figure 12 we can see that using AI variable information saves an order of magnitude on the number of conflicts encountered as well as a reduction of ~5 times fewer variable assignments required. In this way AI information can be used to accelerate bounded model checking. The results presented here were achieved using Dagster in mode C, with one CDCL TiniSAT core.

4 CONCLUSIONS AND FUTURE WORK

We have outlined our new tool, Dagster, summarising some capabilities using two case studies: Costas arrays and pentomino tiling problems, and also demonstrating its use in bounded model checking. The Dagster 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 Dagster 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. We have outlined a range of new features of the Dagster system since the last report deliverable. We plan to support the continued development and expanded the features of the tool.

6 Sourcecode:
https://github.com/ThomWillingham/bmc-summer2122

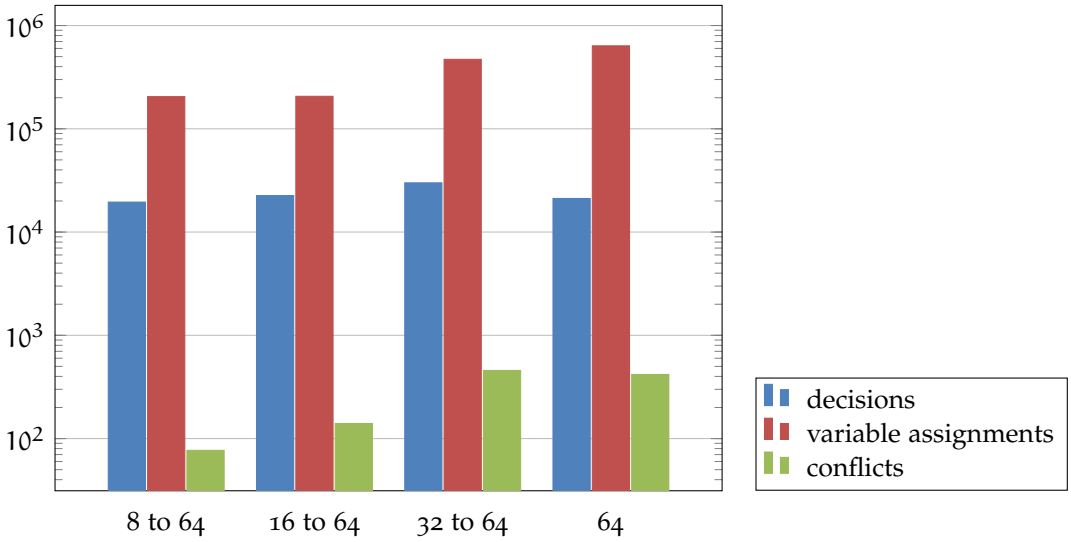


Figure 12: The SAT performance of EMD model checking with propagation AIs between different fidelities of the problem, between 8/16/32 bit instances and the 64 bit instance, with the 64 bit instance without propagation for comparison

REFERENCES

[1] Jinbo Huang. A case for simple SAT solvers. In *International Conference on Principles and Practice of Constraint Programming*, pages 839–846. Springer, 2007.

[2] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *20th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2318–2323, 2007.

[3] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[4] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021.

[5] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[6] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 283–290. AAAI Press, 2008.

[7] Sam Buss and Jakob Nordström. Proof complexity and sat solving. *Handbook of Satisfiability*, 336:233–350, 2021.

[8] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In Jack Mostow and Chuck Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 431–437. AAAI Press / The MIT Press, 1998.

[9] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiabil-*

ity Testing – SAT 2008, pages 28–33, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[11] Armin Biere and Andreas Fröhlich. Evaluating cdcl restart schemes. *Proceedings of Pragmatics of SAT*, pages 1–17, 2015.

[12] 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.

[13] Chanseok Oh. Between sat and unsat: the fundamental difference in cdcl sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 307–323. Springer, 2015.

[14] 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.

[15] Konstantinos Drakakis, Scott Rickard, James K. Beard, Rodrigo Caballero, Francesco Iorio, Gareth O'Brien, and John Walsh. Results of the enumeration of Costas arrays of order 27. *IEEE Transactions on Information Theory*, 54(10):4684–4687, 2008.

[16] Curtis P Brown, Michal Cenkl, Richard A Games, Joseph J Rushanan, Oscar Moreno, and Pei Pei. New enumeration results for Costas arrays. In *IEEE International Symposium on Information Theory*, 1993.

[17] 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.

[18] Toby Walsh. General symmetry breaking constraints. In *Principles and Practice of Constraint Programming - CP 2006*, pages 650–664. Springer Berlin Heidelberg, 2006.

[19] Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. DMC: A distributed model counter. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1331–1338. ijcai.org, 2018.

[20] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[21] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. PaInleSS: A framework for parallel SAT solving. In *Theory and Applications of Satisfiability Testing - SAT 2017*, 2017.

[22] Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with Paracooba. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020*, volume 12178 of *Lecture Notes in Computer Science*, pages 114–122. Springer, 2020.

[23] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. A distributed version of Syrup. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 215–232. Springer, 2017.

[24] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[25] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[26] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[27] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[28] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[29] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[30] Stephan Falke, Florian Merz, and Carsten Sinz. The bounded model checker LLBMC. In *ASE*, pages 706–709. IEEE, 2013.

[31] Mikhail R. Gadelha, Rafael S. Menezes, and Lucas C. Cordeiro. Esbmc 6.1: Automated test case generation using bounded model checking. *International Journal on Software Tools for Technology Transfer*, 23(6):857–861, dec 2021.

[32] Saied Hosseini-Khayat. A lightweight security protocol for ultra-low power asic implementation for wireless implantable medical devices. In *2011 5th International Symposium on Medical Information and Communication Technology*, pages 6–9, 2011.