Incorporating dilemma reasoning into modern SAT Solvers

Yue Chen Li

NUS High School of Mathematics and Science
Singapore, Singapore
h1610073@nushigh.edu.sg

Kian Ming A. Chai, Vincent Boon Chin Seng
INFO Division
DSO National Laboratories
Singapore, Singapore
ckianmin@dso.org.sg, sboonchi@dso.org.sg

***Abstract*—The Boolean SATisfiability Problem (SAT) is an important intractable problem in computer science with numerous theoretical and industrial applications. With the growing availability of multi-core and multi-processor systems, parallel algorithms are increasingly important and useful. Yet, state-of-the-art parallel SAT solvers are mostly portfolio solvers that exchange learnt clauses to achieve knowledge sharing. We describe some novel extensions to using Stalmårck's method to create a parallel SAT solver. This includes two new knowledge sharing methods across solvers, t-BFD and c-BFD merging. We prove that t-BFD and c-BFD merging are distinct, and experimentally show that they occur significantly. In fact, t-BFD merging occurs more frequently at deeper levels of the problem decomposition tree. Within the problem subtree, t-BFDs merging gives unit clauses, and c-BFD merging gives clauses that are empirically short. Such short clauses are likely useful in simplifying the problem. Also, we introduce a more flexible problem decomposition method than naive splitting on N variables to form 2^N subproblems. Based on the Dilemma Rule, it picks branching literals independently at every vertex in the decomposition tree. In addition, we engineered a method to avoid processor starvation in the parallelisation. We benchmarked a preliminary implementation of our modified solver against Glucose-Syrup 4.1 and show that certain configurations of restart and runtime budgets allow our solver to outperform Glucose-Syrup 4.1 on SAT instances. Furthermore, t-BFD merging significantly improved our solver's performance on UNSAT benchmarks, suggesting that t-BFD and c-BFD merging are effective.**

Keywords–SAT solver; boolean satisfiability problem; parallel algorithms; Stålmarck's method; dilemma rule; problem decomposition

I. Introduction

The *Boolean SATisfiability Problem* (SAT) is a classic intractable (NP-Complete) problem that is important to many computer scientists and industries as it can be used to model many real-world problems. Currently, state-of-the-art solvers are mostly based on the *Davis-Putnam-Logemann-Loveland* (DPLL) and *Conflict-Driven Clause Learning* (CDCL) algorithms [1], and they are significantly faster than *Look-Ahead* solvers [2].

Sequential solvers can take a long time to tackle difficult mathematical problems expressed as SAT. Hence, parallel SAT solvers are useful to trade computing resources for a shorter solving time [3]. The vast majority of parallel SAT solvers presently are *portfolio solvers*, which are solvers that run multiple instances of a sequential solver with different configurations on the same problem concurrently, hoping that one of them will be able to solve the problem quickly as different configurations can have significantly varying performance on different problems [4]. This is because DPLL + CDCL is not inherently parallelisable. Efforts to improve parallel solvers have resulted in the sharing of learnt clauses across solvers, for instance in Glucose-Syrup [5].

Stalmårck's method [6], an alternative algorithm to the DPLL + CDCL algorithm, is better able to lend itself to parallelisation as its *Dilemma Rule* involves partitioning of the search space. Recently, a solver based on it, *Dissolve* [7], was created. In this paper, we present another approach inspired by this relatively new development for *decomposition* and *knowledge sharing*, two challenges in parallel SAT solving suggested by Hamadi and Wintersteiger [8].

This paper has three contributions. First, we give a novel way to approach problem decomposition inspired by the Stålmarck's method and *Dissolve*, which provides greater flexibility than homogeneous decomposition. Second, we present two additional ways of merging information from splits (t-BFD and c-BFD merging) and prove that they are different. Third, we also engineer a method to avoid *starvation*, a challenge encountered when designing parallel algorithms. An implementation of our algorithm is benchmarked against Glucose-Syrup 4.1, showing that some configurations of our solver are able to outperform Glucose-Syrup in certain SAT instances. We find that t-BFD and c-BFD merging occur to a significant extent, especially when we are deeper in the search tree, to derive potentially useful clauses. Furthermore, we show experimentally that t-BFD merging improves the performance on certain UNSAT benchmarks for our solver.

II. Preliminaries

Boolean Satisfiability Problem We begin with definitions. A **boolean variable**, x, is a variable which can take on the values *True* and *False*. For simplicity, we refer to boolean variables as

This is an Author Accepted Manuscript version of the following chapter: Yue Chen Li and Kian Ming A. Chai and Vincent Boon Chin Seng, Incorporating dilemma reasoning into modern SAT Solvers, published in IRC-Set 2020 - Proceedings of the 6th IRC Conference on Science, Engineering and Technology, July 2020, Singapore, edited by Guo,Huaqun and Ren, Hongliang and Kim, Noori, 2021, Springer, reproduced with permission of Springer Nature Singapore Pte Ltd. The final authenticated version is available online at: http://dx.doi.org/10.1007/978-981-15-9472-4.

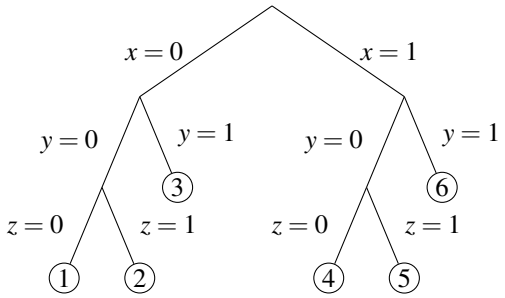
variables. A **literal**, l, is a variable x (positive literal) or its negation $\neg x$ (negative literal). A **clause** C of length N is a disjunction of N literals: $C = l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_N$. A **Conjunctive Normal Form (CNF)** formula F of size M is a conjunction of M clauses: $F = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_M$. In propositional logic, the Boolean Satisfiability Problem (SAT Problem) asks whether it is possible to assign a *True/False* value to all variables in a formula so that it will evaluate to *True*. A formula for which that is possible is termed *satisfiable*, otherwise it is termed *unsatisfiable*.

DPLL+CDCL Solvers The DPLL+CDCL algorithm can be summarised as follows. The solver guesses the value of a variable (such a guess is termed a *decision*) and simplifies the formula as much as possible using **Boolean Constraint Propagation (BCP)** after each guess, that is, if a literal is true, then all clauses containing it are satisfied and its negation can be removed from all clauses. If any clause ends up with only one literal, a similar simplification is applied. This is repeated until either there is no clause containing only one literal, or the solver arrives at a contradiction, which is termed a *conflict*. If the latter occurs, the solver analyses the conflict to determine the reason and creates a *learnt clause*, which expresses the condition that must be true to avoid that conflict — this is the CDCL algorithm. By doing so, the CDCL algorithm is able to prune the search space to avoid arriving at the same conflict. After that, it backtracks to a state before the conflict. In either case, it then continues to guess the values of other variables. Eventually, if the solver is able to guess the values of all the variables without arriving at a contradiction, the formula is satisfiable. In contrast if the solver arrives at an inherent contradiction in the formula, the formula is unsatisfiable.

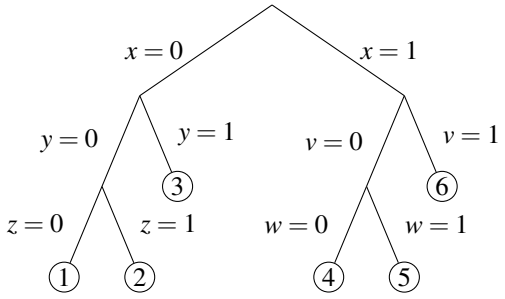
Stalmårck's Method Stalmårck's method [6] is a system using deductions on parts of the formula to prove whether a propositional logic formula is a tautology (a formula which is true under every possible assignment of values to its variables). The key idea involved is termed the *Dilemma Rule*. In brief, two parts of the formula are first assumed to either have the same or distinct truth values. Next, the relationships between the parts are simplified based on a set of *propagation rules*. Finally, the two sets of relationships are intersected: if a relationship is true in both cases, it is true all the time since the two cases form a partition of the entire search space. This can be repeatedly applied to simplify the formula to eventually arrive at a conclusion.

III. Parallel SAT-Solvers with Dilemma Reasoning

In this section, we describe the three contributions of problem decomposition, knowledge sharing and starvation avoidance.



(a) Homogeneous Splitting



(b) Heterogeneous Splitting

Figure 1. Problem decomposition with $N = 6$ solvers

A. Decomposition

To create new subproblems for each sequential solver to work on, one straight-forward way is to pick k decision variables and split on them as done in *Dissolve*, which is shown in Fig. 1a. In contrast, to increase the flexibility of our search, we pick the decision variables to split on independently and randomly at each node. This way, we build an incomplete binary tree which is as balanced as possible that has the same number of leaves as solvers we have, with the root-to-leaf path representing a set of assumptions for each solver, as shown in Fig. 1b. Extensions to this idea is part of our future work as we did not fully exploit this flexibility presently.

Since each split forms a partition of the current search space, then for any chosen vertex all the leaf vertices in its subtree form a partition of its search space by induction. This is an important property that will be required in the next section.

B. Knowledge Sharing

Given a certain set of assumptions, let us first introduce the following definitions: The **trail-before-first-decision**, denoted *t-BFD* is the set of literals that are implied to be true by BCP on the assumptions before any decisions are made. The **clauses-before-first-decision**, denoted *c-BFD* is the set of clauses that are simplified and not yet satisfied (have literals deleted) due to BCP on the assumptions before any decisions are made. These different types of derived facts are summarised in Fig. 2. With that, we present the following two observations.

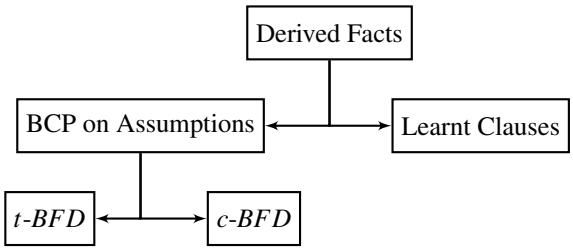


Figure 2. Types of facts derived in solvers

Observation 1. If any implication is derived from BCP on the assumptions in all the solvers represented by a set of leaves in the subtree of a vertex, it must be true for the set of assumptions represented by this vertex. (Intersection, analogous to the Dilemma Rule)

Thus, we see that for a vertex in the decomposition binary tree, we can take the intersection of the *t-BFD* and *c-BFD* each of all the solvers represented by its children to be true in this subtree. For the *c-BFD*, we can get rid of any clause that it subsumes (derived or otherwise) since for any three clauses A, B and C, if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$. However, if we obtain an UNSAT result after clause simplification, we cannot determine whether it is inherent or a consequence of the assumptions. This is as the CDCL algorithm will give a top-level conflict in both cases.

Observation 2. Since learnt clauses are independent of the assumptions [7], we can bring any of learnt clauses up the tree. (Union, similar to Glucose-Syrup)

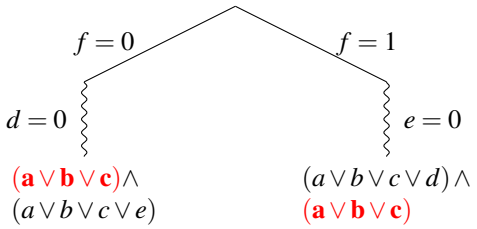
Observation 2 allows all the solvers to share learnt clauses just like in portfolio solvers. Together, these observations should help to prune the search space further. Merging the *t-BFD*s is especially useful since we learnt the equivalent of unit clauses, which we can employ BCP on. Ideally, longer *t-BFD*s are desired as they are more likely to overlap. This applies similarly to *c-BFD*s. To achieve that, we postulate the following:

Hypothesis 1. The deeper we are in the binary tree, the more literals we will be able to merge from our children's *t-BFD*s.

Hypothesis 2. The deeper we are in the binary tree, the more clauses we will be able to merge from our children's *c-BFD*s.

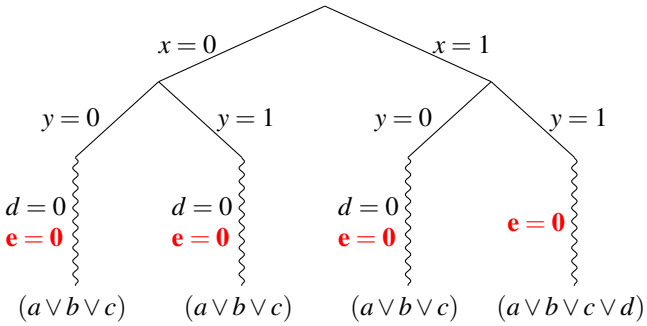
Hypothesis 1, but not hypothesis 2, is supported by our experimental observations. Hence, for problems that give long trails, this may be a useful heuristic, especially since trail merging is a relative cheap operation that can be done in $O(N)$ time and $O(N)$ space, where N is the number of variables in the problem. On the other hand, merging the *c-BFD*s is an expensive process if we loop through all the modified clauses naively, which can take up to $O(NM)$ time, where N is number of variables and M is the number of clauses. However, with efficient data structures to identify the modified clauses and determine whether clauses are identical, perhaps through hashing, we may be able to reduce the overhead. Yet, one would have hoped that merging *t-BFD* is sufficient to derive any facts derivable by merging *c-BFD*, but that is unfortunately incorrect.

$$(a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee e) \wedge (f \vee \neg d) \wedge (\neg f \vee \neg e)$$



(a) Counterexample ($A \not\subseteq B$)

$$(a \vee b \vee c \vee d \vee e) \wedge (x \vee \neg d) \wedge (y \vee \neg d) \wedge (x \vee \neg e) \wedge (y \vee \neg e) \wedge (\neg x \vee \neg y \vee \neg e)$$



(b) Counterexample ($B \not\subseteq A$)

Figure 3. Counterexamples showing that merging *t-BFD* and *c-BFD* give different results

Proposition 1. *Both the sets of learnt clauses from merging t-BFD and merging c-BFD do not imply each other.*

Proof. Suppose otherwise, that at least one of the sets implies the other. Let A be the set of learnt clauses from merging clauses and B be the set of learnt clauses from merging trails. Then either $A \subseteq B$ or $B \subseteq A$. Fig. 3a and Fig. 3b are two counterexamples to each of the case. This is a contradiction. □

C. Maximising Computing Resources

After the problem is decomposed, each solver is assigned a subproblem to work on for a limited amount of wall-clock time, termed a *budget*. This allows us to both merge and use the information

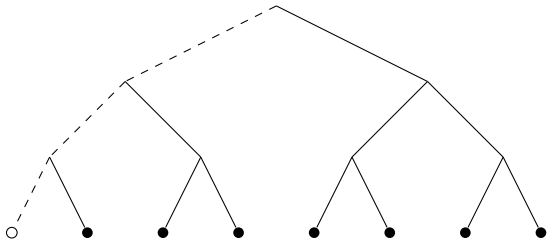


Figure 4. Dashed edges show the uncompleted path; Solid edges show the completed and idle paths.

derived in the subproblems from a split as well as recover from possibly ineffective splits of the problem. However, if we wait for both children to terminate and return the derived information before commencing the merging, we may encounter *starvation* [8]. That is, if the solver of one leaf vertex continues running for an extended period of time until it reaches the maximum allowed running time (time budget) but all other solvers have terminated earlier, we may have up to $N-1$ idle solvers. An illustration is given in Fig. 4.

To remedy this, we first build a tree which is equivalent to the recursion tree created by problem decomposition. On this shared tree data structure, we introduce a *hurry* flag at each vertex, which will cause the vertex to terminate as soon as possible and return the information derived so far.

When one vertex terminates, it will reset its *hurry* flag and set the *hurry* flag of its sibling (if said sibling has not terminated) and propagate the flag down the sibling's subtree. This way, we are able to propagate the termination signal down to all solvers blocking a branch. In the implementation, it is important to check the status of the sibling in a thread-safe manner (e.g. by using mutexes/locks) to avoid race conditions. Alternatively, the tree can be implemented as a lock-free data structure to improve performance.

This effectively solves the issue of many solvers idling for a possibly long time (if the time budget given is long).

D. Pseudocode

Algorithm 1 shows the pseudocode for our idea. It takes the following as parameters: N, the corresponding node in tree data structure; P, the set of unused variables; A, the set of assumptions for this subtree; C, the set of merged clauses for this subtree; R, the number of restarts allowed at each depth; T, the time budget allowed for a solver. In the pseudocode, the **threadSafe** keyword denotes the need to guard against race conditions when processing the statement (e.g. by using mutexes), the **parallel do ... done** construct denotes that the statements within the block are executed in parallel until both are done, and the **break** statement denotes exit of the innermost loop containing the statement.

Algorithm 1 Dilemma Main Procedure

```
1:  procedure DILEMMA(N,P,A,C,R,T)
2:      threadSafe N.done ← false
3:      learnts ← {trail : {}, clauses : {}, result : undefined}
4:      if N.solversAssigned > 1 then
5:          for i = 0 to R[N.depth] − 1 do
6:              if N.hurry then break
7:              x ← PICKRANDOMVAR(P)
8:              P ← P \ {x}
9:              parallel do
10:                 r1 ← DILEMMA(N.l,P,A ∪ {¬x},C,R,T)
11:                 r2 ← DILEMMA(N.r,P,A ∪ {x},C,R,T)
12:             done
13:             r ← COMBINE(r1,r2)
14:             if job is done by any solver then
15:                 r.result ← jobStatus
16:             if r.result ≠ undefined then
17:                 learnts.result ← r.result
18:             P ← P ∪ {x}
19:             add r.trail to learnts.trail
20:             add r.clauses to learnts.clauses
21:     else
22:         Add C to N.solver
23:         for i = 0 to R[N.depth] − 1 do
24:             if N.hurry then break
25:             learnts.result ← N.solver.SOLVEWITHBUDGET(A,N,T)
26:             add N.solver.t-BFD to learnts.trail
27:             add N.solver.modifiedClauses to learnts.clauses
28:         Remove C from N.solver
29:     threadSafe N.done ← true
30:     if N.sibling exists then
31:         threadSafe N.sibling.PROPAGATEHURRYIFNOTDONE
32:     threadSafe N.hurry ← false
33:     return learnts
```

TABLE 1. COMPARISON OF SOLVERS

Heuristics	Glucose-Syrup	Dissolve	Our Solver
Problem decomposition	Yes	No	No
Branch literal heuristic	Yes	No	Yes[a]
Clause sharing	No	No	No
t-BFD merging	Yes	Yes	No
c-BFD merging	Yes	Yes	No

[a] See future work

IV. METHODS AND RESULTS

Combining the ideas in Section III, we arrive at our modified algorithm. Table 1 compares our solver with *Dissolve* and the unmodified Glucose-Syrup 4.1. Our solver is implemented in C++ based on Glucose-Syrup 4.1, a portfolio solver with good performance in recent SAT competitions. To evaluate our modified solver, we selected the 23 largest benchmarks from the Agile Track of the SAT 2017 Competition and all 26 benchmarks related to van der Waerden numbers[1] from the SAT 2011 Competition. The solver is run against each benchmark for a maximum of 5,000 seconds and 64 GB memory with 8 threads on an Intel(R) Xeon(R) computing system (CPU: E5-2620 v4, 2.10 GHz; 64 hyper-threaded cores), exceeding which it is prematurely terminated. Statistics of the solver are logged and the run-times are represented in a cactus plot [9]. Fig. 5 shows the performance of our solver compared to Glucose-Syrup, while Fig. 6a and Fig. 6b show the normalised number of literals and clauses merged (from the *t-BFD*s and *c-BFD*s of the children vertices) respectively with respect to the number of merges at each depth — this is important as the number of merges are greater at greater depths. Regression lines are plotted to show the difference in gradients.

A. Results and Discussion

There is evidence that intersections of *t-BFD*s and *c-BFD*s do derive a significant number of facts. In fact, Hypothesis 1 is supported by the graph shown in Fig. 6a. Interestingly, the average merged clause length appears to be rather short (≈ 4) compared to the maximum clause lengths (> 20) (refer to Table 2), suggesting that merging *c-BFD* may allow us to obtain useful clauses.

Overall, our solver performed worse than Glucose-Syrup by a constant time factor as seen in Fig. 5. This is likely due to the greater overhead from the splitting and merging, which appears to be especially significant for UNSAT instances, particularly for hard UNSAT instances (related to van der Waerden numbers). In fact, with only our modified decomposition, some configurations of our solver solve SAT instances faster than Glucose-Syrup. With *t-BFD* merging turned on, our solver solves an additional van der Waerden instance and the Agile instances faster.

Although these are only preliminary tests with only a small number of benchmarks, we believe that these results show that these new heuristics are worth investigating further.

[1] A van der Waerden number $W(c,k)$ is the minimum natural number such that if the integers $\{1,2,\ldots,W(c,k)\}$ are assigned colours from the set $\{1,2,\ldots,c\}$, then there are at least k integers in an arithmetic progression which all have the same colour. Finding such numbers is an unsolved problem in Ramsey Theory in general.

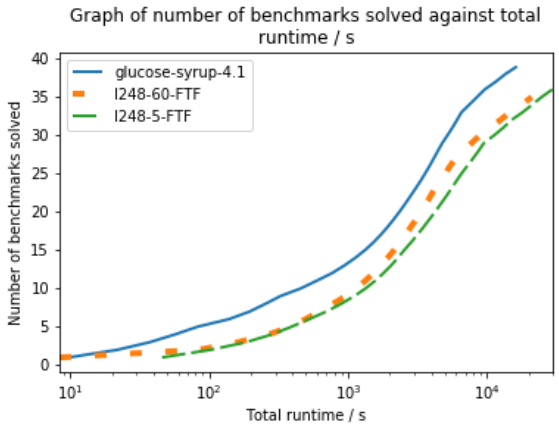


Figure 5. Comparison of overall performance of our solver with trail merging and glucose-syrup

TABLE 2. LENGTH OF MERGED CLAUSES FOR THE CASES (#) WHICH OUR SOLVER SOLVED. ALL MINIMUM LENGTHS ARE 0.

(A) BUDGET = 60

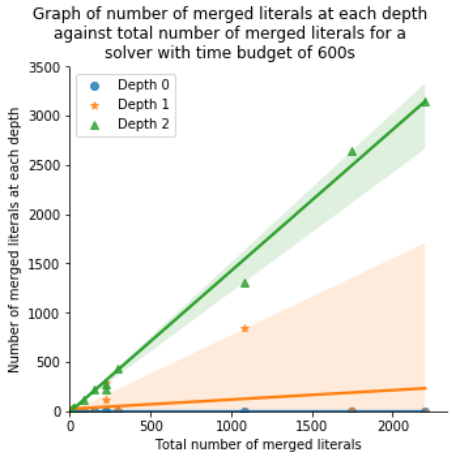
#	Average	Maximum
0	3.958164	12
1	3.758052	19
2	3.746337	20
3	3.798731	20
4	3.980227	21
5	3.751489	21
6	3.790726	21
7	4.024393	22
8	4.268452	22
9	3.843114	23

(B) BUDGET = 600

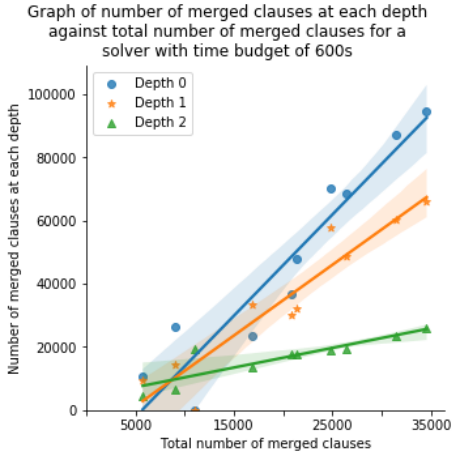
#	Average	Maximum
0	3.981017	12
1	3.934512	13
2	3.721468	19
3	3.718266	20
4	3.843215	20
5	3.935537	21
6	3.759932	21
7	3.995705	21
8	3.896167	22
9	3.899929	22
10	3.866584	23
11	4.157838	24

V. CONCLUSION AND FUTURE WORK

We described a novel way to approach problem decomposition inspired by the Stålmarck's method and *Dissolve*. Beyond merely splitting the problem on N variables at a time with Dilemma rule, we explored heterogeneous splitting for about k times instead, which



(a) Graph showing a greater number of merged literals at greater depths.



(b) Graph showing a smaller number of merged clauses at greater depths.

Figure 6. Graph of number of merged *t-BFD* and *c-BFD* against total number of merges for tests with ∞-2-4-8 restart pattern and a time budget of 600s.

provides greater flexibility. Furthermore, we presented two more ways of merging information from splits (*t-BFD* and *c-BFD* merging) in addition to the commonly used sharing of learnt clauses. Moreover, we proved that *t-BFD* and *c-BFD* are distinct. Finally, we discussed a method to avoid *starvation*, a challenge encountered when designing parallel algorithms.

We implemented our modified algorithm and ran a set of tests to benchmark its performance against an existing state-of-the-art solver, Glucose-Syrup 4.1, as well as against one another with different configurations. We showed that some configurations of our solver are able to outperform Glucose-Syrup in SAT instances and (a) *t-BFD* and *c-BFD* merging occur to a significant extent and (b) $t-BFD$ merging occurs more frequently at greater depths in the decomposition tree. Finally, we showed that the *c-BFD* merged are rather short.

In the future, we hope to optimise the implementation of our algorithm to minimise the overheads, as well as implement a heuristic for the picking of decision variables like *Dissolve*. In addition, we hope to exploit the flexibility of heterogeneous splitting. Furthermore, we hope to fully implement and optimise our *c-BFD* merging idea, after working out how to distinguish between cases of UNSAT due to assumptions and UNSAT due to the problem itself as explained after Observation 1. Finally, due to the promising preliminary results, we hope to run more detailed tests with more benchmarks.

Acknowledgment

Yue Chen thanks DSO National Laboratories for hosting her for this Young Defence Programme (2019) internship project.

References

[1] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506 – 521, May 1999.

[2] M. J. H. Heule and H. Maaren, "Look-ahead based SAT solvers," in *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, Amsterdam, The Netherlands, The Netherlands: IOS Press, Jan. 2009.

[3] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: guiding CDCL SAT solvers by lookaheads," in *Hardware and Software: Verification and Testing*, (Berlin, Heidelberg), pp. 50 – 65, Springer Berline Heidelberg, Dec. 2011.

[4] A. E. J. Hyvärinen and N. Manthey, "Designing scalable parallel SAT solvers," in *Theory and Applications of Satisfiability Testing – SAT 2012* (A. Cimatti, ed.), (Berlin, Heidelberg), pp. 214 – 227, Springer Berlin Heidelberg, Jun. 2012.

[5] G. Audemard and L. Simon, "On the Glucose SAT solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

[6] J. Nordström, "Stålmarck's method versus resolution: A comparative theoretical study," Master's thesis, Stockholm University, Stockholm, Sweden, Sep. 2001.

[7] J. Henry *et al.*, "Dissolve: A distributed SAT solver based on Stålmarck's method," tech. rep., University of Wisconsin-Madison, Madison, Wisconsin, United States of America, Mar. 2017.

[8] Y. Hamadi and C. Wintersteiger, "Seven Challenges in Parallel SAT Solving," *AI Magazine*, vol. 34, Jan. 2012.

[9] M. N. Brain, J. H. Davenport, and A. Griggio, "Benchmarking solvers, SAT-style," in *Satisfiability Checking and Symbolic Computation* (M. England and V. Ganesh, eds.), (Kaiserslautern, Germany), Springer Berlin Heidelberg, Jul. 2017.

[10] C. Sinz and T. Balyo, "Practical SAT solving." `https://baldur.iti.kit.edu/sat/`, 2019. Accessed on Jul. 2019.

[11] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502 – 518, Springer Berlin Heidelberg, 2003.