Guiding SAT solving by formula partitioning*

Zoltán Ádám Mann, Pál András Papp

Abstract

When deciding the satisfiability of a Boolean formula, one promising approach is to split the formula into two smaller independent sub-formulas. While many studies report encouraging early results with such methods, the approach is rarely used in state-of-the-art solvers. In this paper, we present a technique that uses formula partitioning to guide the solution of the SAT problem through providing initialization values for the VSIDS heuristic. Our results on a large number of benchmark instances show that the method can notably improve the performance of modern solvers, especially if the time available for solving is short. We also present some findings in the area of hypergraph partitioning, which is used as a tool for our technique.

1 Introduction

Boolean satisfiability is one of the central problems of computer science. It is not only relevant from a theoretic point of view, but also because of its applications in a wide range of different areas, including automated reasoning, circuit design, formal verification, and artificial intelligence. Due to this, the study and enhancement of SAT solvers is of key importance.

Deciding satisfiability is NP-complete. Although modern SAT solvers can sometimes solve extremely large formulas in reasonable time, they may need orders of magnitude more time to solve other, much smaller formulas [39]. Therefore, further research is still needed. From an application point of view, it is especially important to devise methods that can recognize the structure inherently present in problem instances encoding real-world problems and exploit that structure to speed up the search.

Partitioning the formula into smaller subformulas is a natural idea in this direction, since reducing a problem to a set of smaller independent subproblems is a successful technique for a variety of hard problems [17]. There are a number of studies in the literature that aim to introduce some form of this idea into an algorithm for SAT solving [46] [19] [1] [37] [29] [49] [26]. Most of these studies suggest that the approach indeed yields promising results.

Nonetheless, techniques based on partitioning are rarely present in today's state-of-the-art SAT solvers. One possible explanation is that most of these studies focused only on a specific family of formulas, and methods that are successful on such a family may not bring similar results on the more heterogeneous set of problems that solvers usually encounter. Another possible reason may be that some of these experiments were carried out a longer time ago, and as both the performance of leading solvers and the size of the considered input formulas have drastically increased since then, methods that were once beneficial may not be effective anymore.

In our previous work [41], we investigated the use of various versions of the Fiduccia-Mattheyses (FM) heuristic for SAT partitioning on a diverse set of benchmark formulas. Our findings indicated that, although each considered partitioning method gave promising results on some problem instances, but none of them was consistently good on the set of benchmarks as a whole. In this paper, we continue this research by evaluating two further, more advanced hypergraph partitioning algorithms: the state-of-the-art *multilevel* Fiduccia-Mattheyses heuristic, and a novel algorithm

*This paper has been published in *International Journal on Artificial Intelligence Tools*, volume 26, issue 4, article 1750011. DOI: `http://dx.doi.org/10.1142/S0218213017500117`

that we specifically developed for finding small cuts in the hypergraph of real-world formulas. These two algorithms perform indeed better than the simpler heuristics considered previously, but there is no clear winner from the two approaches. We show that based on simple properties of the formula, we can give an efficient prediction as to which of our partitioning algorithms will find the best cut, and thus we can develop a heuristic which automatically chooses the most promising partitioning algorithm for a given input formula.

Using this partitioning heuristic, we present two different methods to introduce the divide-and-conquer approach into modern SAT solvers. Both techniques can be incorporated into any SAT solver based on the DPLL algorithm, and only require minimal modification of the underlying solver. One of the two methods, called the *hard partitioning* approach, uses the concept of priority levels to divide the formula, and was already investigated to some extent previously. The other technique, called *soft partitioning*, uses the information obtained from partitioning to guide the variable selection heuristic of the underlying solver, with its impact decreasing over time. The second approach is, to our knowledge, new.

To evaluate the methods, we introduced them into a leading solver and validated the resulting algorithms on the large and heterogeneous set of benchmark problems used in the SAT competitions of recent years. Our experiments show that the soft partitioning approach is indeed capable of notably increasing the number of successfully solved problem instances, especially if the time available for solving is relatively small. Furthermore, we argue that the success rate of the different solver variants is strongly related to some properties of the input formula, and therefore we can also develop a heuristic that automatically chooses the most promising solver version to use for a specific formula. Our results show that for smaller time limits, this combined algorithm can even outperform the soft partitioning solver.

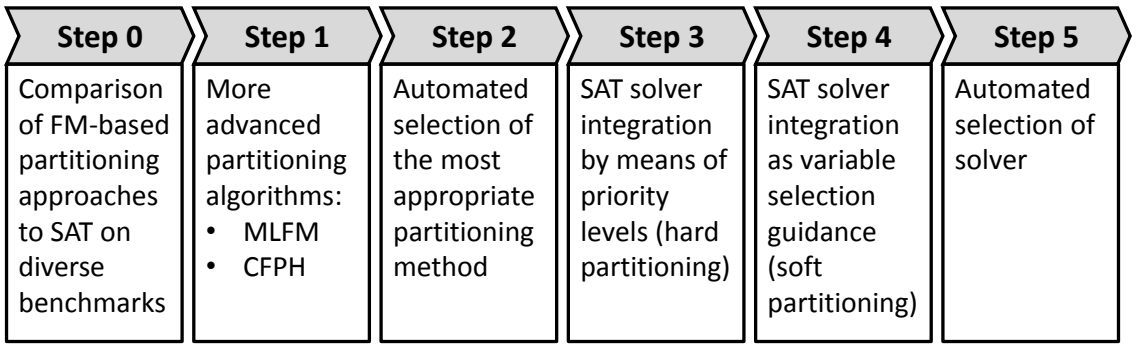


Figure 1: Overview of the presented research

Fig. 1 shows an overview of the presented research. Step 0 was covered in our earlier paper [41]; Steps 1 to 5 are the contributions of this paper.

1.1 Paper organization

The following section summarizes previous results in related areas. Section 3 defines the concepts and notation used throughout the paper. Section 4 describes different models to reduce the splitting of the formula to a hypergraph partitioning problem. Section 5 discusses the partitioning algorithms used for splitting the hypergraph, including state-of-the-art methods, our novel algorithm, and their relationships to the properties of the input hypergraph. Section 6 describes the hard partitioning and soft partitioning techniques to guide the SAT solver. Section 7 presents our empiric results with the different algorithm versions, while Section 8 shows additional details about the way the soft partitioning technique works, when it is useful and what the limits of its applicability are. Finally, Section 9 concludes the paper.

2 Related work

Various studies considered some form of partitioning to accelerate the solution of a SAT problem. The validity of the partitioning approach was demonstrated by the work of Biere and Sinz [11], who investigated the case when during the process of solving, the formula falls into multiple independent parts. They show that a minor modification in the solver to recognize these situations can notably increase the efficiency of the solving process. This indicates that developing and applying partitioning techniques may be a promising way to improve solver performance.

A number of different papers investigate partitioning-based methods. Several of these studies experiment with applying a partitioning algorithm to the dual hypergraph of the formula, that is, finding a set of variables that separate the formula when removed. Park and Gelder were among the first to show results with this approach [46]; however, since their work, both the performance of solvers and the size of the formulas they can address increased dramatically. Similarly promising results were reported with this method by Durairaj and Kalla [19], Li and van Beek [37], and Huang and Darwiche [29]. All these studies examine the technique that we will refer to as hard partitioning, and they evaluate it on a set of relatively small formulas (on today's scale). However, in our experiments, we have found that the hard partitioning technique performs weakly on a heterogeneous set of formulas of large size.

Instead of looking for a set of splitting variables, several studies were conducted to search for a separating set of clauses in the formula (that is, partitioning the primal hypergraph). Amir and McIlraith present such an approach, but the nature of their work is mostly theoretic, without a thorough empiric evaluation of their technique [1]. Similar methods are investigated by Monnet and Singer [49], by Monnet and Villemaire [43] and by Torres-Jimenez et al. [50]; however, their results are preliminary in the sense that their algorithms are only tested on a very small number of instances.

Another natural idea is to carry out the partitioning of the input formula recursively, creating a so-called tree decomposition of the original formula. Multiple studies aimed to improve solver performance by generating an ordering of decision variables based on such a decomposition [12] [29] [43]. However, one drawback of this technique is that finding the optimal tree decomposition is a very challenging task itself.

Most studies in the area concentrate on the application of only one specific partitioning method. One exception to this is the work of Heule and Kullmann [26], who presented a comprehensive survey of a wide variety of partitioning methods. However, while they mostly consider the approach from a theoretic point of view, our aim is to evaluate these techniques according to their efficiency in practice.

Another related area is parallel SAT solving. As multicore architectures become more and more dominant, researchers put more emphasis on designing solution algorithms that can exploit this parallelism, leading to a variety of parallel SAT solvers [14, 42]. These algorithms also reduce the input problem to a set of smaller problems that are solved independently, and as such, also execute some form of partitioning. An example is the cube-and-conquer algorithm of Heule et al. [27], which combines CDCL and lookahead solvers by first executing some branching steps according to carefully chosen decision variables, and then parallelly solving the resulting reduced formulas with CDCL solvers. In contrast to these methods, we focus on applying the partitioning approach in the context of sequential SAT solvers. A more fundamental difference is that parallel SAT solvers *partition the search space* (e.g.: one solver instance searches the part of the search space where $x_1 = true$, whereas the other works in the part of the search space where $x_1 = false$); in contrast, we are dealing with approaches that *partition the formula*, which is a much more costly operation, but leads to much smaller subproblems.

The community structure and modularity of graph representations of SAT formulas have also been investigated. Informally, communities are subgraphs with many edges inside the subgraph and few edges across the boundary of the subgraph; modularity is a measure of how well the graph can be decomposed into communities. Finding the best community decomposition is a tough problem, but fast heuristics exist that work well in practice [16]. Ansótegui et al. found that SAT instances encoding real-world problems typically have a highly modular structure [2, 3].

Newsham et al. showed that learned clauses with variables from few communities tend to have higher quality than those intersecting many communities; moreover, metrics of the community structure of a formula can be used to improve the accuracy of predicting solver runtime [45]. While there are obvious similarities between this line of research and our work, there are also two main differences. First, we work with balanced bipartitions of the formula (that is, cutting the formula into two parts of similar size), whereas a community decomposition may consist of many and highly imbalanced subgraphs, which would be impractical for our purposes. Second, our aim is not only to analyze formulas and predict solver runtime, but also to make solving more efficient.

A somewhat related concept is centrality, which also attempts to grasp the structure of SAT formulas. Katsirelos and Simon introduced a directed graph representation of formulas and approximated the frequency with which a random walk in the graph traverses each vertex corresponding to a literal or clause [34]. They showed that the measure computed this way, called the centrality of literals and clauses, correlates with various aspects of the solving process in a CDCL solver, like the variables selected for decisions, conflict clauses, and learned clauses. However, their work left it open how the gained insight could be used for improving the efficiency of SAT solving.

In some application domains, a modular structure arises in a natural way. An example is planning, which is often solved by transforming the problem to SAT and using a SAT solver. As shown by Huang et al., an encoding of planning as SAT results in a set of so-called transition variables, which link together the lower-level action plans of each time epoch [30]. Thus, removing the transition variables leads to a set of loosely coupled sub-formulas.

Besides SAT, the divide-and-conquer approach was also successfully applied in other problems related to Boolean formulas, such as #SAT [9] and MAX-2-SAT [22].

Earlier, we have also conducted a set of experiments with formula partitioning approaches [41], putting more focus on aspects of partitioning the formulas than on integrating partitioning into SAT solving. In contrast to this research, that early study only examined the hard partitioning approach for using partitioning to solve SAT, and it deduced that the results with this method are inconclusive.

In this research, we evaluate our partitioning approaches by introducing them into the SAT solver `glucose` [4, 5] (version 3.0). `glucose` is an open-source state-of-the-art solver (an highly improved version of the MiniSAT solver [20]), that achieved outstanding results in all of the SAT competitions of recent years. As the majority of modern SAT solvers, `glucose` is based on the CDCL algorithm [23], and implements most of the wide range of techniques invented over the last decades that make today's SAT solvers extremely efficient.

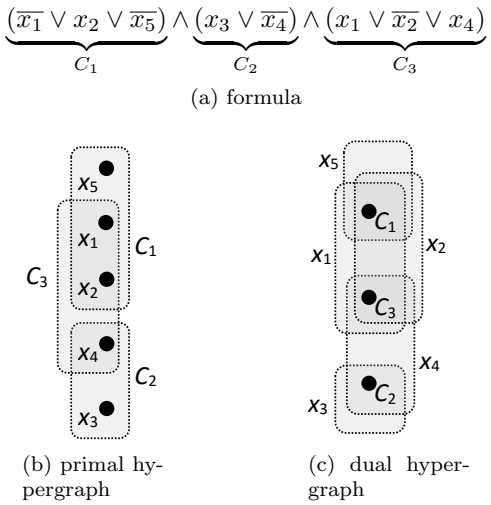
3 Preliminaries

A satisfiability problem can be described as a formula consisting of Boolean variables and clauses. A variable or its negation is called a *literal*. A disjunction of literals is called a *clause*. A *formula* (in conjunctive normal form) is obtained as a conjunction of clauses. An *assignment* (*partial assignment*) to a formula is a function that assigns a possible value (true or false) to all of (a subset of) the variables. We say that an assignment *satisfies* the formula if it makes the formula evaluate to true. A formula is *satisfiable* if there exists an assignment that satisfies it; otherwise, it is *unsatisfiable* [31].

In practice, beyond the number of variables and clauses, another important characteristic of the formula is its *length*, defined as the sum of the size of all clauses in the formula. The *LtV ratio* (Length to Variables ratio) or *average variable occurrence* of a formula is obtained by dividing the length of the formula by the number of its variables.

A *hypergraph* consists of a non-empty set V of *vertices*, and a set E of *hyperedges* (or simply: edges), where every element of E is a subset of V. The *size* of a hyperedge is defined as its cardinality. The *degree* of a vertex is the number of hyperedges it is contained in; a vertex of degree 0 is called *isolated*.

A *partitioning* of a hypergraph is the division of its vertices into two sets V_1, V_2 such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. A *partial partitioning* of a hypergraph is a pair $V_1, V_2 \subset V$ that only



(a) formula

(b) primal hypergraph

(c) dual hypergraph

Figure 2: Example formula and its hypergraph representations

requires $V_1 \cap V_2 = \emptyset$. A hypergraph (V, E) is *disconnected*, if it has a partitioning V_1, V_2 of V such that for all hyperedges $e \in E$ we have $e \subset V_1$ or $e \subset V_2$. A *cut set* is a set of hyperedges $F \subset E$ such that $(V, E \setminus F)$ is disconnected. A partitioning V_1, V_2 generates a cut set naturally through the edges e for which $V_1 \cap e \neq \emptyset$ and $V_2 \cap e \neq \emptyset$; such a generated cut set is referred to as a *cut*.

Given a Boolean formula, there are two natural ways to represent it as a hypergraph. In the *primal hypergraph*, each variable is represented by a vertex and each clause is represented by a hyperedge. The hyperedge corresponding to a clause c contains the vertices that represent variables occurring in c (with or without negation). In the *dual hypergraph*, each clause is represented by a vertex and each variable is represented by a hyperedge. The hyperedge corresponding to the variable x contains the vertices that represent clauses containing x (with or without negation). The primal and dual hypergraphs are demonstrated for an example formula in Fig. 2.

The paper also assumes familiarity with the basic concepts of state-of-the-art DPLL (Davis–Putnam–Logemann–Lovela and CDCL (conflict-driven clause learning) SAT solvers, including the notions of decision variables, conflicts, learned clauses and backjumps [25].

One of the most well-known methods for selecting decision variables in DPLL-based solvers is the VSIDS (Variable State Independent Decaying Sum) heuristic [44]. The VSIDS heuristic maintains an *activity* value for all variables in the formula, and always selects the variable with the highest activity as the next decision variable. The activity of each variable is initialized to 0, and is incremented every time the variable appears in a conflict clause. To prevent activity values growing too large, occasionally, the activity of all variables is multiplied by a constant smaller than 1. As a result of this, the activity of a variable is mostly determined by its presence in recent conflicts.

3.1 Evaluation methodology

The paper contains several different empirical studies, investigating either the performance of solver variants or the behavior of different partitioning algorithms. As input problems for these experiments, we used the set of input formulas from the SAT Challenge 2012 [6], the SAT Competitions of the years 2013 [7] and 2014 [10], and the SAT Race of 2015 [8]. These are formulas that pose a serious challenge for today's SAT solvers and can be seen as a diverse and representative set of benchmarks for solver evaluation.

In our experiments, instances of the 2012 SAT Challenge and 2013 SAT Competition were used as a *training* set of benchmarks to tune the parameters of our algorithms, while instances of the 2014 SAT Competition and the 2015 SAT Race were used to *validate* the performance of the tuned solvers.

Benchmarks for the 2012-2014 SAT events consist of three different sets of formulas: application formulas obtained from different application areas, hard combinatorial formulas designed especially for the competition through some combinatorial procedure, and random instances, generated according to some model of random SAT formulas.

The main metric that we used for comparing the performance of solvers is the number of problem instances they can solve within a given time limit. This is in line with the methodology used at the SAT competitions [39].

All measurements were carried out on a desktop computer with Intel Core i3-2100 CPU @ 3.1GHz and 4GB RAM, running Gentoo Linux 2.2.

4 Models of formula partitioning

When solving a SAT problem through partitioning, the main idea is to identify a part of the formula such that after removing it, the remaining formula falls into two independent parts. The part to remove may be a set of clauses or a set of variables in the formula.

In practice, removing a part of the formula means guiding the SAT solver such that it first focuses on this part of the formula in the solution process. Then, through evaluating this part, the solver somehow reduces the problem to deciding the satisfiability of two smaller subformulas.

Most of today's state-of-the-art SAT solvers are variants of the DPLL algorithm, and as such, they proceed by repeatedly branching on the value of chosen variables. A natural way to guide such an algorithm toward our aim is to find a set of variables D that divide the formula if removed, and assign values to these variables first. At the latest when all these variables are assigned values, the formula will be split. Such a set of variables can be found in different ways. In our research, we investigated two approaches (referred to as *partitioning models*) to obtain such a set, corresponding to the two hypergraph representations introduced earlier:

- **Dual:** We find a cut in the dual hypergraph of the formula. The edges of the cut directly correspond to a set of variables D that divides the formula if removed.
- **Primal:** We find a cut in the primal hypergraph of the formula, corresponding to a set of clauses, and take the set of variables D that appear in at least one of these clauses. If we remove all these variables, the formula will fall into two parts.

Note that in the dual case, a cut of the hypergraph directly represents a set of variables to remove. On the other hand, a cut in the primal hypergraph only yields such a set through taking the union of variables present in the clauses of the cut. Even for primal cuts containing a small portion of the clauses, this union might contain a large portion of the variables.

Let us revisit the example formula of Fig. 2. In the dual hypergraph, one example of a cut is the set $D = \{x_4\}$ which divides the formula after removal. In the primal case, the clause C_3 forms a cut in the hypergraph, so selecting $D = \{x_1, x_2, x_4\}$ gives a set of variables that, once removed, split the formula into two parts.

There is no general recipe to tell which of the two is the more promising approach to find a small set of cut variables. On the one hand, the advantage of partitioning the dual hypergraph is that we directly optimize for a set of variables, without possibly ending up with much more variables when taking the union of clauses. On the other hand, as shown in the next section, partitioning the dual hypergraph is a considerably more difficult task, since most partitioning algorithms perform much better on primal hypergraphs than on dual ones.

Both partitioning models were already examined to some extent in the literature [46] [19] [37] [1] [49]. In our preliminary experiments, we have also considered a third approach, in which we find a cut in the primal hypergraph, and then introduce new variables to split the clauses in the cut into two parts [41]. However, we have found that since this model often increases the number of variables significantly, using it has a strong negative effect on the runtime of the SAT solver in almost all cases; therefore, this option will not be considered here.

5 Partitioning algorithms

5.1 State-of-the-art techniques

In both partitioning approaches, we have to find a cut in a given hypergraph. On one hand, we aim to find a cut that has few hyperedges, to make the formula fall apart as soon as possible. On the other hand, we would like the partitioning to be balanced, that is, to ensure that both partitions contain at least a certain portion of all vertices; otherwise, we could end up cutting off an insignificantly small part of the hypergraph. Therefore, a limit parameter $L \in [0.5; 1)$ is introduced in partitioning problems. Given L, we say that a partitioning of a hypergraph is *balanced* if both of the partitions contain at most $L \cdot |V|$ vertices. A cut is said to be balanced if the underlying partitioning is balanced.

Finding a small balanced cut in a hypergraph is a well-known problem for its numerous applications in VLSI netlist partitioning and other areas [15] [32] [40]. Unfortunately, finding the minimal cut that satisfies a balance constraint is known to be NP-hard [38]. However, various sophisticated heuristics were developed to address the problem; the most notable among these is the Kernighan-Lin (KL) heuristic [35], and its improved version, the Fiduccia-Mattheyses (FM) heuristic [21].

The Fiduccia-Mattheyses heuristic is a linear-time algorithm, which (along with its further improved variants [18] [28] [32]) is known to be particularly successful in returning small balanced cuts in many application areas. The algorithm proceeds in passes. Each pass starts from a balanced partition of the vertices, with the first one usually being initialized by some randomized heuristic. Passes consist of steps, in which one vertex is moved from one partition to the other. In each step, the vertex with maximal *gain* is chosen, that is, the one which reduces the cut size the most, among those that can be moved without violating the balance constraint. After the step, the moved vertex remains *locked* in the new partition until the end of the pass. When no more vertex can be moved, the pass is over, and the best state (in terms of cut size) observed throughout the pass is chosen as the initial partition for the next pass.

The success of the heuristic is due to the fact that while the algorithm makes greedy choices in each step, it continues the pass even if the following move increases the size of the cut, thus allowing it to occasionally escape local minima. Also, a special data structure allows to access and update the gain values of all vertices efficiently, resulting in linear runtime.

However, one big disadvantage of the FM heuristic is that it only moves one vertex in every step. As the size of the input hypergraph grows, these steps cover smaller and smaller distances in the solution space (relative to the size of the space), thus making the search less effective. Therefore, an improved version of the algorithm was devised, known as the *multilevel Fiduccia-Mattheyses (MLFM)* heuristic, to deal with hypergraphs of larger size [32]. The MLFM algorithm works in three phases. In the *clustering phase*, iterative clustering steps are executed, each time merging some vertices of the hypergraph to obtain a smaller hypergraph on the next level, while trying to retain as much of the structure of the original hypergraph as possible in the process. In the *partitioning phase*, the basic FM algorithm is used to find a balanced cut in the resulting hypergraph, which is significantly smaller than the original one. Finally, a *separation phase* is executed to iteratively unmerge vertices that were clustered together in the first step, also running a few FM passes after separation in each level to further refine the partition.

In our experiments, we used `hMetis`, a state-of-the-art implementation of the MLFM algorithm that had achieved impressive results in various applications [32, 33].

In our previous study [41], we also investigated some other improved variants of the FM algorithm. The *multimove FM* algorithm allows each vertex to be moved k times in a pass (instead of only once), so as to traverse a bigger part of the search space. The *soft gain FM* uses a more sophisticated gain function, assigning an extra gain value to vertices that contribute to moving a given portion of a hyperedge into a partition, thereby rewarding small steps in the right direction. The *edgemoving FM* is a remarkably different variant, which proceeds by selecting an edge in each step, and moving all vertices of the edge in question to a given partition. While these algorithms gave good results on some instances, none of them showed consistently good

performance. Moreover, we have found that `hMetis` is superior to these methods, either in terms of runtime or quality of the cut returned.

5.2 The CFPH algorithm

5.2.1 The limit of FM-based techniques

The FM heuristic and its improved versions give strong results in the application areas they were designed for. In our experiments, we have also found that they are indeed able to find good cuts in the primal hypergraphs of formulas. These hypergraphs have the specialty that they mostly contain relatively small hyperedges.

When the FM algorithm chooses the next vertex to move, hyperedges that have at least two vertices in both partitions have no effect on the decision, since regardless of the vertex we move, these hyperedges will remain cut after the move. For hyperedges of larger size, this is almost always the case, as it is very unlikely for such edges to have all or all but one of their vertices falling into the same partition in a given state of the algorithm. Hence, if the majority of hyperedges in a hypergraph is large, then the FM algorithm ignores these in most cases, selecting vertices to move ineffectively or even close to arbitrarily. As a result, in hypergraphs where the average hyperedge size is too large, the FM algorithm usually returns cuts of very poor quality. While the MLFM algorithm does improve on this problem to some extent by reducing the size of hyperedges in the first phase, this only slightly extends the domain of problems with which it can effectively deal.

Unfortunately, this is a problem in the case of dual hypergraphs, where hyperedge size corresponds to the number of occurrences of the variable in the formula. This is usually a much higher number than the clause size (hyperedge size in the primal hypergraph) in formulas obtained from application areas. For example, in the application problems of the 2013 SAT Competition [7], the average clause size over all instances was 3.36, while the average variable occurrence was 59.92. Therefore, when applied to the dual hypergraphs of formulas, FM-like algorithms often return poor partitions, sometimes even ones where all hyperedges are cut.

5.2.2 Our new approach

Thus, we developed a heuristic with especially the aim of finding small balanced cuts in hypergraphs with large edges. The idea behind this heuristic, named Constructive Formula Partitioning Heuristic (CFPH), is to start with two empty sets P_1, P_2, and place vertices into these throughout the process, in every step inserting all vertices of a given hyperedge into one of the two sets, this way making sure that the hyperedge in question will not be cut.

The algorithm works in two phases. In the first phase, called the *clean phase*, CFPH iterates through the hyperedges repeatedly, searching for hyperedges such that (i) all already placed vertices of the edge are in the same set P_i and (ii) the remaining (not yet placed) vertices of the hyperedge can be added to this set without violating the balance constraint. If such a hyperedge is found, it is selected with a given probability p, in which case all its unplaced vertices are added to P_i. In the special case when both sets can be chosen for a hyperedge since none of its vertices are placed yet, it is added to the set with the smaller number of vertices. If no edge is selected in an entire iteration, then a finalization stage is reached, and all hyperedges satisfying the two conditions are selected afterwards (not just with probability p as before, but surely). The procedure stops when no more hyperedge satisfies the two criteria.

This first phase of the algorithm is repeated a certain number of times (denoted by T). Each generated partial partitioning is evaluated by the number of hyperedges that have all their vertices already placed into one of the sets. In the end, the partition that had the largest number of such edges is chosen as the initial state for the second phase.

In the second phase, named *conclusion phase*, the remaining unplaced vertices are assigned to one of the sets. For this, a limit parameter ℓ is used, which is initialized to 1 and doubled in every step, until it reaches or surpasses the maximum hyperedge size. In each step, the algorithm iterates over all the hyperedges that still contain an unplaced vertex. If the next edge has at most

ℓ vertices placed in one of the sets, then all unplaced vertices in the edge are added to the other set, but only if this can be done without violating the balance constraint. If both sets satisfy this condition for an edge in a given step, then one is chosen at random. Finally, at latest when ℓ is at least as large as the maximum hyperedge size, all unplaced vertices that appear in at least one hyperedge are placed in one of the sets; isolated vertices can be placed arbitrarily at the end.

The motivation behind the conclusion phase is to distribute the vertices of the remaining hyperedges in an asymmetric manner. Note that all hyperedges considered in this phase will certainly be cut in the final partition. However, this way, if some passes of the FM algorithm are run on the generated partition, it has a significantly higher chance for further improvement, as there are more hyperedges that only have a small number of vertices in one of the two sets.

5.2.3 Pseudocode of the algorithm

For the sake of completeness, we also provide a pseudocode for the CFPH algorithm, shown in Algorithm 1. In order to present a compact description, we develop some notation that simplifies the code significantly. We denote the input hypergraph by H, its vertex set by V (as before), the number of its hyperedges by m, its i^{th} hyperedge by e_i, and the size of the largest hyperedge in H by H_{max}. We denote a partial partition by a pair (P_1, P_2). By assigning $\emptyset$ to such a partition, we mean resetting both P_1 and P_2 to the empty set. For an index $j \in \{1, 2\}$, let $\overline{j}$ denote the index of the other set (that is, $\overline{1} = 2$ and $\overline{2} = 1$). Let $s_{i,j}$ denote the number of nodes in e_i that are already placed into P_j. Let $Val(P_1, P_2)$ denote the number of hyperedges that already have all their vertices placed into one of the sets. We assume that the values of $s_{i,j}$ and $Val(P_1, P_2)$ are always updated throughout the algorithm.

As before, L denotes the balance parameter for the partitioning. Calculated from L, LIMIT specifies the maximum number of nodes allowed in a partition. We also develop the notion of a '**for** loop run until termination', meaning a loop that iterates over a range of values again and again until it is terminated by a BREAK instruction. Finally, we assume the existence of a *rand()* function that returns a random number uniformly distributed on $[0, 1]$.

5.3 Parameter tuning for the partitioning methods

Before we can apply the described algorithms in practice, there are some parameters for which a value has to be chosen. While `hMetis` is an off-the-shelf partitioning algorithm that has a set of recommended parameters, our own implementations of the FM and CFPH algorithms both have a parameter that is yet to be defined. Hence we executed a process of parameter tuning on a set of formulas chosen randomly from the benchmarks of the SAT competition 2013 [7]. For tuning the minor parameters of the partitioning algorithms, we used a smaller set of 60 instances, and for selecting the time limit to use, we examined a larger set of 300 instances.

First of all, we need to specify a reasonable value as the balance constraint for our partitioning algorithms. We have chosen to accept cuts where both sets contain at least 30% of the vertices of the hypergraph (thus, $L = 0.7$).

In case of the FM algorithm, the parameter to tune is the maximum number of passes executed. This was set to 10, since we found that further passes decrease the size of the cut by less than 1% on average. In case of CFPH, we had to determine the number of times the clean phase is executed before the best candidate is chosen and the algorithm jumps to the conclusion phase. This was also set to 10; the improvement after the 10^{th} round was below 2% in average, and in 66% of the cases, no better candidate was ever found after this point. With that, the parameters of the partitioning algorithms are set.

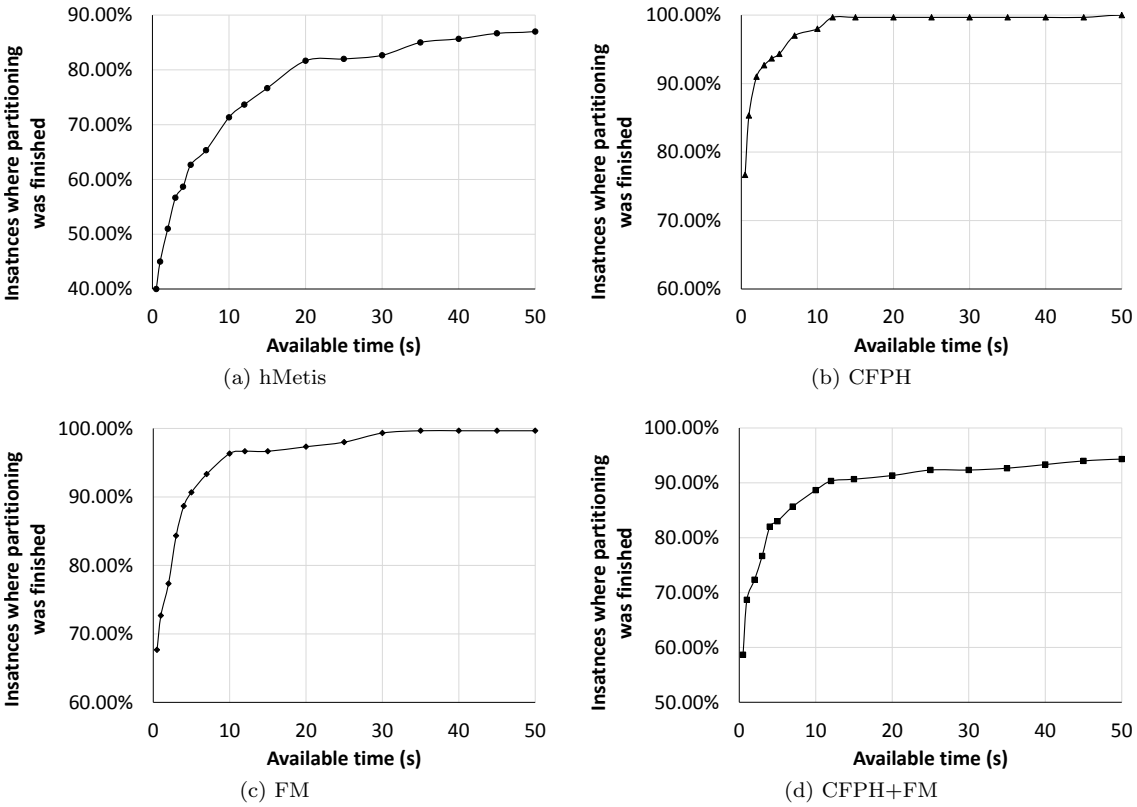
However, there is one more issue to address: our set of input formulas contains some instances of immense size, which require far too much time to partition with any of our algorithms. Since we do not want the partitioning process to significantly reduce the time available for deciding satisfiability, we need to limit the running time of the partitioning algorithm.

For this, we aimed to identify a 'knee point', over which increasing the time limit further only results in marginal growth in the number of finished runs. Fig. 3 shows the percentage of instances

Algorithm 1 Constructive formula partitioning heuristic (CFPH)

function CFPH(H, L, T, p)
  LIMIT $\leftarrow L \cdot |V|$
  $P_1^*, P_2^* \leftarrow \emptyset$ ▷ Clean phase
  for t from 1 to T **do**
    $P_1, P_2 \leftarrow \emptyset$
    $finalize \leftarrow false$
    $count \leftarrow 0$
    for i from 1 to m until termination **do**
      if $count = m$ **then**
        if $finalize = true$ **then**
          BREAK
        else
          $finalize \leftarrow true$
      if $s_{i,1} + s_{i,2} < |e_i|$ AND $\exists j$ such that $s_{i,\bar{j}} = 0$ AND $|P_j| + |e_i| - s_{i,j} \leq$ LIMIT **then**
        $j \leftarrow$ the index j for which the conditions are true
        (if true for both j, then the index of the set with smaller size)
        if $finalize = true$ OR $rand() < p$ **then**
          add the yet unplaced vertices of e_i to P_j
          $count \leftarrow 0$
      $count \leftarrow count + 1$
    if $Val(P_1, P_2) > Val(P_1^*, P_2^*)$ **then**
      $(P_1^*, P_2^*) = (P_1, P_2)$

  $(P_1, P_2) = (P_1^*, P_2^*)$ ▷ Conclusion phase
  for ℓ from 1 to H_{max} exponentially (doubled in every iteration) **do**
    for i from 1 to m **do**
      if $s_{i,1} + s_{i,2} < |e_i|$ AND $\exists j$ such that $s_{i,\bar{j}} \leq \ell$ **then**
        select such a value j (if both values fit, select one randomly)
        add the yet unplaced vertices of e_i to P_j
  return (P_1, P_2)



(a) hMetis (b) CFPH

(c) FM (d) CFPH+FM

Figure 3: Percentage of dual hypergraphs partitioned within a given time

on which the different partitioning algorithms (executed on the dual hypergraph of the formula) are finished within time, as a function of the time available. Based on the diagrams, we chose a time limit of 20 seconds, and used this value in all our further experiments.

5.4 Choosing the most appropriate partitioning algorithm

To sum up, we considered the following four algorithms for partitioning hypergraphs in our experiments:

- **FM**: Our implementation of the FM algorithm, initialized with a randomized balanced partition
- **hMetis**: The `Metis` implementation of the MLFM algorithm
- **CFPH**: The CFPH algorithm
- **CFPH+FM**: The FM algorithm, initialized with the partition returned by CFPH

We have chosen not to include the further FM versions from [41] in this set. The reason for this in case of the multimove FM and soft gain FM variants was that `hMetis` managed to find significantly smaller cuts than these algorithms in almost all cases. The running time of the edgemoving FM, on the other hand, scales poorly, and therefore it cannot be used in practice for bigger hypergraphs.

Given this set of four algorithms, our aim is to develop a heuristic which, based on simple properties of the hypergraphs (such as number of vertices, number of edges or average hyperedge size), chooses the algorithm in this set which will probably return the best cut. In fact, since our

Table 1: Number of times when each model-algorithm combination turned out to be the best

	hMetis	FM	CFPH	CFPH+FM
Primal	21	4	0	5
Dual	121	0	91	58

input is not a hypergraph but a Boolean formula, we need more than that: we have to choose a combination of partitioning model (primal or dual hypergraph) to represent the formula and partitioning algorithm (one of the above four) to apply on the chosen model. If we find that the properties of the formula have an important role in determining the combination that gives the best result on the formula, then we can indeed develop a heuristic which uses the most appropriate partitioning model and algorithm with high probability. This approach of heuristically choosing the most promising algorithm from a set of candidates, known as the *algorithm portfolio* approach, has been successfully applied to various problems before [24, 51].

5.4.1 Empiric results with the partitioning methods

To evaluate the different partitioning algorithms, we ran all combinations of partitioning models and algorithms on the training set of 300 instances that were previously used in the process of parameter tuning. For each instance, we chose the model and algorithm that lead to the smallest cut (ties were broken based on smaller runtime). We made an exception in the case of CFPH+FM, since running the FM heuristic after the CFPH algorithm often yielded only a minor improvement or no improvement at all; therefore, CFPH+FM was only considered better than CFPH in case it returned a cut that was at least 1% smaller than the cut returned by CFPH.

The results of the experiment are shown in Table 1. For each combination of model and algorithm, the table shows the number of instances on which the algorithm turned out to be the most effective. We can clearly see that in the vast majority of cases, one of four certain combinations yields the smallest cut.

Unfortunately, if we examine the 21 formulas for which Primal - hMetis was found to be the most successful combination, we see that they do not exhibit strong similarities in any of the basic parameters of the formula. The fact that there is no simple characterization of the cases when this combination finds the best cut means that there is no straightforward way to determine when our heuristic should select this model and algorithm for partitioning. Therefore, we decided to develop a heuristic that only chooses a partitioning algorithm from the other three candidates (Dual - hMetis, Dual - CFPH and Dual - CFPH+FM).

5.4.2 Connection to the properties of the formula

We found that the domain where these three methods are successful can be mostly characterized using two parameters of the input formula. One of these is the average variable occurrence (LtV ratio) in the formula, which corresponds to the average hyperedge size in the dual hypergraph. As mentioned in Section 5.2.1, the MLFM heuristic (implemented by `hMetis`) yields significantly worse results with the growth of average hyperedge size, which was precisely the motivation for designing the CFPH algorithm. Therefore, above a given LtV ratio, CFPH returns better cuts than `hMetis`.

Note that the performance of CFPH also gets weaker as average hyperedge size grows. However, it is significantly less sensitive to this parameter than FM-like algorithms, and therefore, above a certain LtV ratio, it begins to find smaller cuts than those. The sensitivity of the two algorithms to LtV ratio is easily seen in Fig. 4, which shows the average quality of the cut returned by the algorithms in different intervals of the LtV ratio.

The other notable difference between the three partitioning algorithms is a result of the time limit: since `hMetis` takes too much time on larger formulas, it is better to run CFPH or CFPH+FM

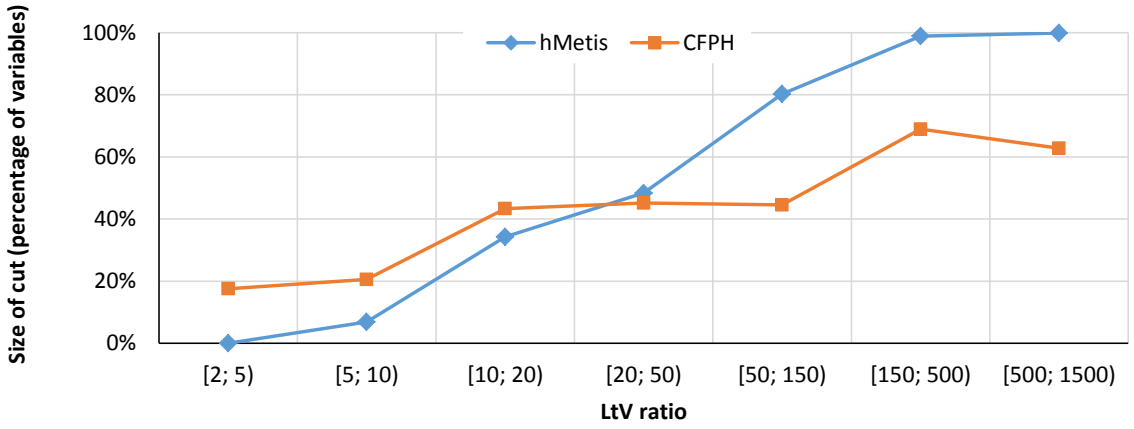


Figure 4: Average quality of cut returned for different LtV ratios

above a certain formula size. Our experiments suggest that over approximately 300,000 clauses, `hMetis` is almost always interrupted by the timeout.

These results are illustrated in Fig. 5. Each point in the figure corresponds to a formula in the training set, located according to its LtV ratio and number of clauses. Each point is labeled with the partitioning algorithm that finds the smallest cut in the dual hypergraph of the formula. Fig. 5a) shows the results when the algorithms are run without a time limit, while 5b) shows the case when a time limit of 20 seconds is imposed.

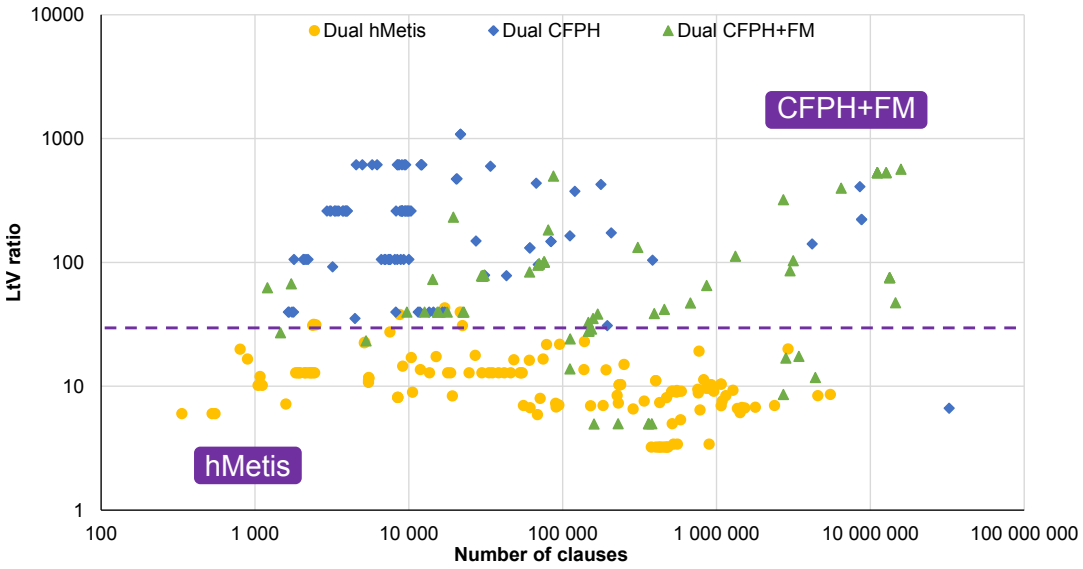
It is clearly visible from the figures that `hMetis` is indeed the best choice in case of formulas with a small LtV ratio. Furthermore, if a time limit of 20 seconds is introduced, then `hMetis` is only beneficial to use under a certain number of clauses. As for the other two algorithms, while there also tends to be a region where CFPH dominates the graph and one where CFPH+FM does, these are not separable from one another as clearly as `hMetis` is from the other two algorithms. Considering the fact that CFPH+FM often finds a better cut than CFPH but never a worse one, and that running both CFPH and FM still finishes within time in almost all cases, we decided to always run the FM algorithm after CFPH (that is, choose the CFPH+FM algorithm) in this region. Therefore, the final form of our heuristic is the following:

- if the input formula has at most 200,000 clauses and an LtV ratio at most 30, we call `hMetis` on the dual hypergraph
- otherwise, we call CFPH+FM on the dual hypergraph

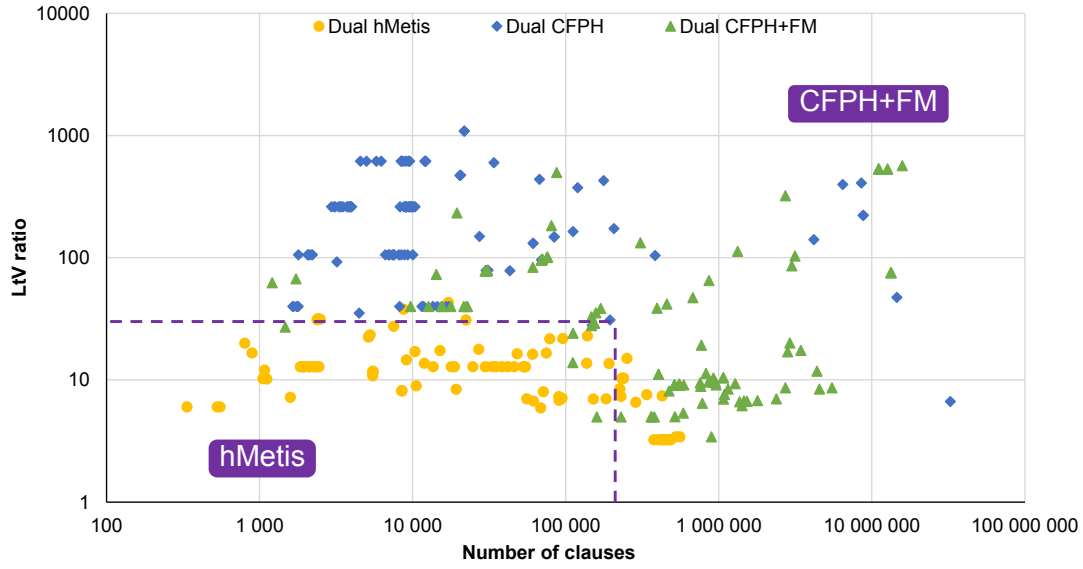
6 Partitioning-based solving techniques

As described in Section 4, formula partitioning can be introduced into a SAT solver by finding a set of variables that separates the formula, and assigning values to these variables first. This only requires us to modify the part of the solver that is responsible for choosing the decision literals; all other parts of the solver can be left intact.

Like numerous other state-of-the-art SAT solvers, `glucose` uses the VSIDS heuristic to select its decision variables. Despite its simplicity, VSIDS is known to be a very effective heuristic, and is thoroughly investigated in the corresponding literature [44, 48]. Its outstanding success is usually attributed to the fact that it increases the locality of the search, and that it only imposes a very small computational overhead.



(a) without time limit



(b) with a time limit of 20s

Figure 5: Best partitioning algorithm for each formula of the training set

6.1 Hard partitioning

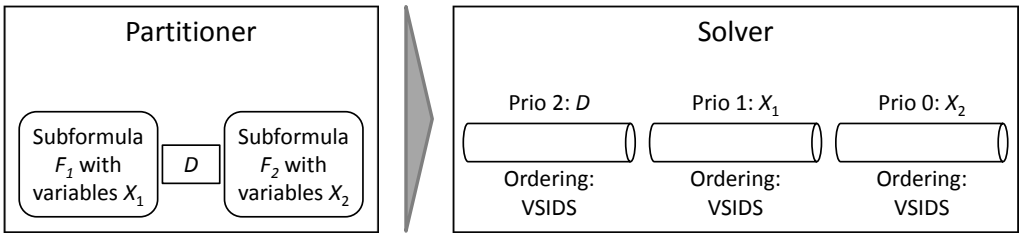
When introducing formula partitioning into `glucose`, our main goal is to make the formula fall apart as early as possible in the solution process, while not modifying the original VSIDS rule for decision variable selection to a larger extent than necessary. One possible method to implement this is the *hard partitioning* approach, depicted in Fig. 6a.

Before running the solver, we call a partitioning algorithm on the formula, obtaining a set of variables D that makes the formula fall apart into subformulas F_1, F_2 when removed, such that all other variables are only present in F_1 or in F_2. The set of variables appearing in F_i is denoted by X_i ($i \in \{1, 2\}$). In the solver, we modify the variable selection rule by introducing priority levels, that is, assigning an integer priority from $\{2, 1, 0\}$ to each variable. We assign the highest priority to the variables in D, second highest priority to the variables in X_1 and lowest priority to the variables in X_2.

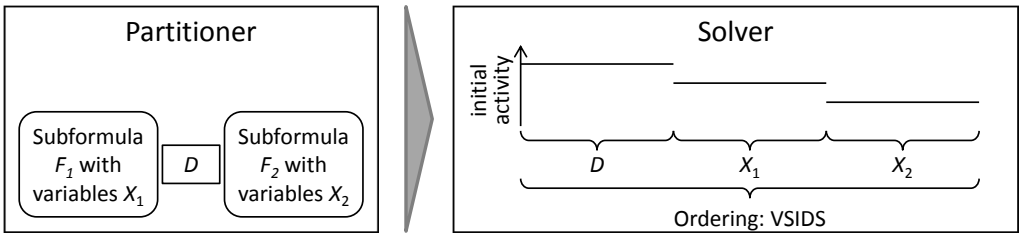
Given the priority levels, the decision variable selection is modified as follows: we always pick a decision variable from the highest priority level that still has an unassigned variable, and among those variables, we select the variable with highest activity value. This way, highest priority variables ensure that (if we found a relatively small cut) the formula will be split early on, and when choosing between variables with the same priority, we can still expect VSIDS to make relatively good decisions.

Note that it is not obvious that separating the two lower priority levels from each other is necessary. However, it was already shown by Biere and Sinz in [11] that solving independent parts of the formula separately can notably improve solver performance.

Also note that being a CDCL solver, `glucose` repeatedly adds learned clauses to the formula, by which F_1 and F_2 may cease to be independent at some point. That is, before the formula is split, it is possible that the solver adds some learned clauses that contain variables from both X_1 and X_2. We chose to ignore this phenomenon, since our experiments showed that the number of such clauses usually stays relatively small. It is important to note that even if F_1 and F_2 are not independent, this does not impair the correctness of the approach.



(a) hard partitioning



(b) soft partitioning

Figure 6: Conceptual overview of the hard partitioning and soft partitioning approaches

6.2 Soft partitioning

The hard partitioning solver described above is a possible implementation of a formula partitioning technique into a SAT solver. However, our empirical results with this approach were rather

disappointing: although it can solve some instances that the original `glucose` algorithm cannot, but it timed out on several other instances that `glucose` could solve (see Section 6.3 for details). This implies that although formula partitioning may be a beneficial approach, but by making it the primary aspect of decision variable selection, we gain less than what we lose by not allowing VSIDS to make decisions based on locality.

Therefore, we devised another solver version to overcome this problem, to which we will refer to as *soft partitioning* solver. The idea behind soft partitioning, depicted in Fig. 6b, is to initially guide the solver in the direction of splitting the formula, but allow VSIDS to override these decisions once it has a deeper understanding of the formula.

To do this, no priority levels are used, but instead, the initialization value of variable activities is modified. We introduce three new parameters I_2, I_1, I_0 of the solver, and when initializing activity values, we set the activity of all variables in D to I_2, all variables of F_1 to I_1 and all variables of F_2 to I_0 (we assume $I_2 \geq I_1 \geq I_0$). At the beginning of the solution process, activity values will be close to their initialization values, and the solver will first choose variables from D as decision variables. However, the values will repeatedly be updated, and if a variable from X_1 or X_2 appears in significantly more conflicts than one from D, its activity value will eventually grow larger and it will be selected over the other variable. This way, the advantages of partitioning and VSIDS can be combined: at the beginning, decisions are based on partitioning, but later, when VSIDS can already make more informed decisions, it becomes the dominant factor.

6.3 Empirical experience with the solver variants (on the training set)

As described, we have three different solver variants available for comparison: the original `glucose`, the hard partitioning solver and the soft partitioning version (the latter with different possible initialization values). Our next aim is to gather some experience with the practical behavior of these solver variants, and to tune their parameters. In order not to compromise the later validation study, we used the training set for this purpose (see Section 3.1).

For the soft partitioning solver, we evaluated three different parameter sets that assign different importance to the highest priority level, and thus, implement different compromises between the original `glucose` and the hard partitioning solver. Each such version is identified by the tuple (I_2, I_1, I_0) of initialization values; the three tested versions were (3, 1, 0), (25, 2, 0) and (200, 2, 0).

We first investigated the behavior of the solvers on all benchmarks of the 2012 SAT Challenge and the 2013 SAT Competition. Not surprisingly, all partitioning algorithms return cuts of very bad quality on randomly generated formulas, since these benchmarks do not exhibit a structure. This way, the partitioning solvers are unable to split the formula early in the solution process as intended, hence we cannot expect partitioning-based methods to yield good results in this case. Since we did not want these instances to influence our final choice of solver parameters, we only included the application (APP) and hard combinatorial (HC) benchmarks of the corresponding years in the training set. This yields a training set of 1770 formulas.

In our experiments, we found that the relationship between the performance of the different solvers shows a strong dependency on the time available for solving an instance. We decided to select two specific time limits that are considerably different, a larger one of 500 seconds and a smaller one of 50 seconds, and focus on training the partitioning solvers for these specific times. However, to get a wider perspective on the properties of the different versions, we also executed some experiments with further time limit values.

It is important to note that, when talking about a time limit in case of a partitioning solver, we also include the time used for partitioning in the runtime of the algorithm. So, for example, a time limit of 50 seconds means that 50 seconds are available altogether for partitioning and solving. We apply this principle throughout our experiments to ensure a fair comparison.

Table 2 shows the number of instances successfully solved within a time limit of 500 seconds by the different algorithms. It is clear that the hard partitioning solver is not competitive with the other approaches. On the other hand, if we consider the whole training set, all the soft partitioning

Table 2: Number of successfully solved instances by solver variants, with a time limit of 500s

	2012		2013		All
	APP	HC	APP	HC	
# of formulas	570	600	300	300	1770
glucose	378	301	100	106	885
Soft part. (3, 1, 0)	379	294	103	110	886
Soft part. (25, 2, 0)	385	299	104	107	895
Soft part. (200, 2, 0)	387	290	103	106	886
Hard part.	177	237	42	64	520

Table 3: Number of successfully solved instances, with a time limit of 50s

	2012		2013		All
	APP	HC	APP	HC	
# of formulas	570	600	300	300	1770
Glucose	157	209	27	37	430
Soft part. (3, 1, 0)	154	218	22	57	451
Soft part. (25, 2, 0)	160	220	26	60	466
Soft part. (200, 2, 0)	165	223	26	60	474
Hard part.	80	180	12	44	316

variants solve slightly more instances than `glucose`. The improvement compared to `glucose` is highest (approximately 1%) for the (25, 2, 0) version.

The improvement becomes much more significant if we choose a smaller time limit for solving the benchmarks. Table 3 shows the results for the time limit of 50 seconds. We can see that all three soft partitioning variants perform significantly better than the original `glucose`. In this case, the most successful version increases the number of solved instances by more than 10%. Note that in this case, the version yielding the best results is clearly the one with largest initialization value.

If we investigate solver performance for various different time limits, then we can see that there is a general tendency for soft partitioning solvers to show larger improvement as we reduce the time available to the solvers. Also, for larger time limits, soft partitioning versions with smaller initialization values usually outperform versions with larger initialization values, while for smaller ones, the better results are usually obtained by larger initialization values. These trends are visible in Table 4 (the table also includes the original `glucose`, which corresponds to the initialization values (0, 0, 0)).

These phenomena can be explained by the intuition that given more time, the VSIDS heuristic is able to make more intelligent decisions, since it has access to significantly more refined activity values. When we select a smaller time limit, VSIDS will not be able to gather enough information to make a sophisticated choice, thus selecting decision variables based on insight from partitioning will yield better results. However, as the amount of available time is increased, VSIDS will begin to outperform the decision strategy that aims to partition the formula. Too strong initialization is counterproductive in this case, since it may still override VSIDS at the point where it is already able to make more reasonable decisions. However, even for a larger time limit, initializing activities with small values can still be beneficial, since it guides the solver in the beginning of the decision process when VSIDS is not yet well-informed enough.

It is another natural idea to examine the performance of the new solver separately on satisfiable and unsatisfiable formulas. Table 5 compares `glucose` and the most successful soft partitioning solver for both time limits, dividing the successfully solved formulas into two groups according to their satisfiability. We see that in both cases, the vast majority of the improvement can be attributed to the satisfiable category. This implies that while the soft partitioning solver does not

Table 4: Number of solved instances for various time limits and initializations

Time limit	20 s	50 s	100 s	500 s
Glucose	310	430	540	885
Soft part. (3, 1, 0)	317	451	563	886
Soft part. (25, 2, 0)	323	466	569	895
Soft part. (200, 2, 0)	324	474	580	886
Soft part. (1000, 100, 0)	331	464	582	877

Table 5: Number of solved instances, separated according to satisfiability

(a) 500 s

	SAT	UNSAT	All
Glucose	519	366	885
Soft part.	528	367	895

(b) 50 s

	SAT	UNSAT	All
Glucose	301	129	430
Soft part.	344	130	474

seem to outperform `glucose` in deciding unsatisfiability, it is indeed remarkably better at finding a satisfying assignment if it exists.

We also experimented with parameter combinations where the two lower priority levels are separated further apart (that is, where there is a larger difference between the second and third parameter of the soft partitioning solver). However, we have found that these versions did not yield a notable improvement over the previously discussed parameter sets.

Altogether, according to the results on the training set, in the case of the 500 seconds time limit, the best results were returned by the (25, 2, 0) soft partitioning solver, while for 50 seconds, the (200, 2, 0) soft partitioning solver turned out to be the most successful one. Hence we chose these solvers to be the most promising ones to use in the corresponding cases, and evaluated their performance on a separate set of validation formulas (see Section 7).

We can summarize our findings on the training set as follows:

- the overall performance of the hard partitioning solver is not competitive with that of the original `glucose`
- the soft partitioning solver, on the other hand, is able to solve a higher number of problem instances than `glucose`
- as we reduce the amount of time available, the performance increase obtained by using the soft partitioning solver becomes more significant
- for smaller time limits, better results are usually obtained by a stronger initialization of VSIDS
- the soft partitioning solver clearly outperforms `glucose` on satisfiable formulas, while the efficiency of the two solvers is close to identical on unsatisfiable formulas

6.4 Heuristic for choice of solver

Even in the cases when `glucose` and the soft partitioning solver manage to solve a similar number of formulas in time, there is a number of formulas on which only one of them succeeds. Therefore, given the original `glucose` and a version of the soft partitioning solver, it is a natural question whether we can characterize the cases where one of the two outperforms the other. Using the previously mentioned algorithm portfolio approach, this could allow us to devise a combined solver that merges the advantages of the two solvers, solving more instances than either of the two.

Hence, we aimed at developing a heuristic rule based on which we can decide to use either `glucose` or the soft partitioning solver, both for the 50 seconds and the 500 seconds case. For both time limits, we settled for using a simple rule that chooses according to only one of the basic characteristics of the input formula, by selecting a single splitting point in the range of the given feature, and using one of the solvers for formulas where the feature is below this value and the other solver where it is above. We also experimented with possibilities that are based on more than one feature of the formula, but we found that these do not result in better characterization of the success of solvers; on the other hand, since they are more complex, they increase the risk of overfitting our decision rule to the properties of the training set.

For the case of 500 seconds, the LtV ratio turned out to be the feature that best describes the relationship between the two solvers. The optimal division of the training set is obtained by choosing a ratio of 16.5 as a splitting point, and running the soft partitioning solver for smaller LtV ratios and the original `glucose` for larger ones. This combination of the solvers succeeds on 912 instances of the training set; compared to the 885 solved by `glucose` and the 895 by the (25, 2, 0) soft partitioning solver, the combined solver clearly outperforms both underlying solvers.

For the 50 seconds time limit, the best division is achieved by splitting according to the size of the cut returned by the partitioning algorithm. We found that the combined solver would give the best results on the training set if the soft partitioning solver was applied below a cut size of around 42,000 variables, and the original `glucose` was run above this value. Note that this division means selecting the soft partitioning solver in almost all cases; however, this is not surprising since the soft partitioning solver usually outperforms the original `glucose` for this time limit. The division increases the number of solved instances to 484 (while only 430 are solved by `glucose` and 474 by the (200, 2, 0) soft partitioning solver). Hence, the combined solver is again considerably better than both underlying solvers.

The complete process of selecting a partitioning algorithm and a solver version to use is summarized on the flowcharts of Fig. 7. Note the slight structural difference between the two charts: in the 500 seconds case, the solver can be chosen before running the partitioning algorithm, whereas in the 50 seconds case, the choice depends on the output of the partitioning algorithm. Also, the condition for the second decision in Fig. 7 a) can be simplified, as it is already known at this point that the LtV ratio is below 16.5, so there is no need to check whether it is below 30 or not.

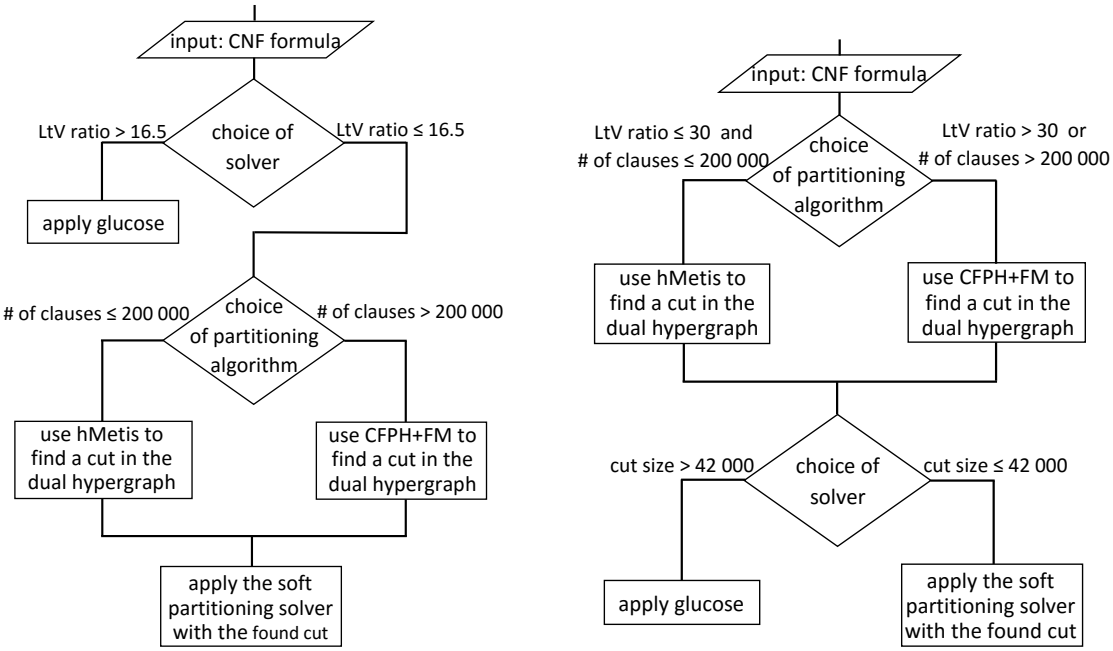
7 Validation

Given our different solver variants, it remains to evaluate their performance on the validation benchmark set. This set was chosen to include the application and hard combinatorial benchmarks of the 2014 SAT Competition and the benchmarks of the 2015 SAT Race. Thus the validation set is disjoint from the training set.

For the case of the 500 seconds time limit, we compared three different solver variants: the original `glucose`, the soft partitioning solver described in Section 6.2 with parameters (25, 2, 0), and the combined solver configured for this case as described in Section 6.4. Table 6 shows the number of instances solved within 500 seconds for these variants. We can see that the soft partitioning solver is indeed capable of successfully improving on the performance of `glucose`: it solves 19 instances more than `glucose`, which approximately means a 7% increase in the number of solved formulas. This is a significantly better result than the one obtained on the training set.

On the other hand, the table also shows that our combined solver gave weaker results than the soft partitioning solver on this set of formulas. A more careful analysis of the results showed that this is only due to the fact that the soft partitioning solver produced much better results on this set of formulas than on the training set. Of course, this does not change the fact that in this case, the combined solver did not manage to improve on the soft partitioning solver at all. However, it is worth noting that it still significantly outperformed `glucose`.

For the 50 seconds case, we evaluated the original `glucose`, the (200, 2, 0) soft partitioning solver, and the combined solver for 50 seconds. The results for this case are shown in Table 7. It is clear that for this case, the soft partitioning solver succeeds on 15 instances more than `glucose`,



(a) 500 seconds case

(b) 50 seconds case

Figure 7: Summary of the solving process for both time limits

Table 6: Number of successfully solved instances for the 500 s case

	2014		2015	All
	APP	HC		
# of formulas	300	300	300	900
Glucose	85	67	124	276
Soft part. (25, 2, 0)	92	72	131	295
Combined solver	93	70	128	291

Table 7: Number of successfully solved instances for the 50 s case

	2014		2015	All
	APP	HC		
# of formulas	300	300	300	900
Glucose	26	36	59	121
Soft part. (25, 2, 0)	26	45	65	136
Combined solver	29	44	66	139

which is an even more remarkable increase of about 12% in the number of solved formulas. The combined solver is able to outperform even this, solving about 15% more formulas than the original solver.

Note that we are not measuring the performance of solvers by their running time, but by the number of solved instances, where an increase of a few percents is indeed a remarkable improvement. For example, in the HC category of the 2014 SAT Competition, a solver with only 2% weaker performance than the winning algorithm was already not among the top three solvers, and a performance difference of 12% already excluded the solver from the top ten places.

Also, we can observe that for both time limits, the soft partitioning solver gave consistently better results than `glucose` in all subcategories of the validation set.

Therefore, as a conclusion of the tests, we can deduce that introducing soft partitioning is indeed a reliable way to further enhance solver performance. The performance improvement obtained with this method shows a strong dependence on the time available for solving: as the time limit is reduced, the improvement becomes more significant. Also, the 50 second case shows that a combined algorithm created from a soft partitioning solver and a regular one can surpass even the soft partitioning solver in performance.

8 Further analysis

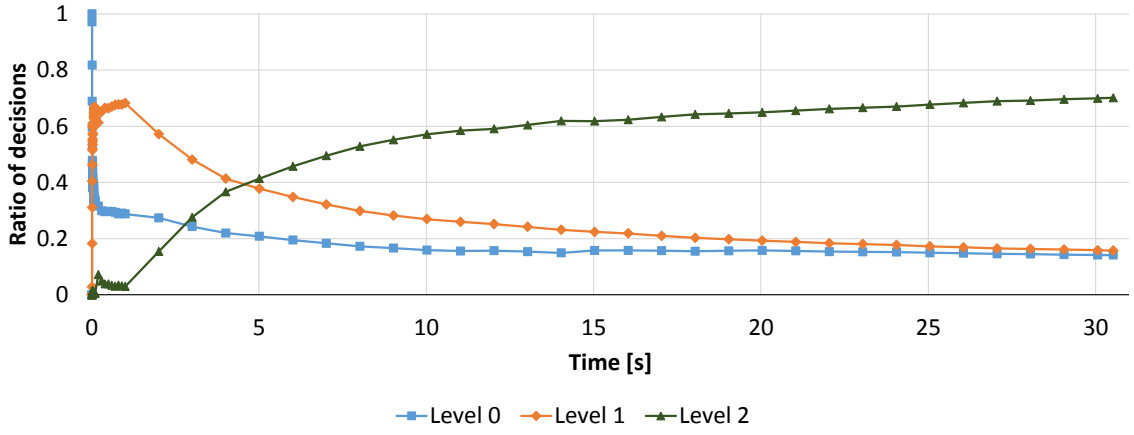
The aim of this section is to present some further insight into the operation of the soft partitioning solver, including the reasons and limits of its effectivity. Specifically, we investigate how the solver's focus changes among the different priority levels along the solution process, the influence of different activity initialization methods on the solver's progress, the impact of the used partitioning method on overall effectiveness, the types of problem instances on which the new approach excels, and the effect of time limits above 500 seconds.

8.1 Example run

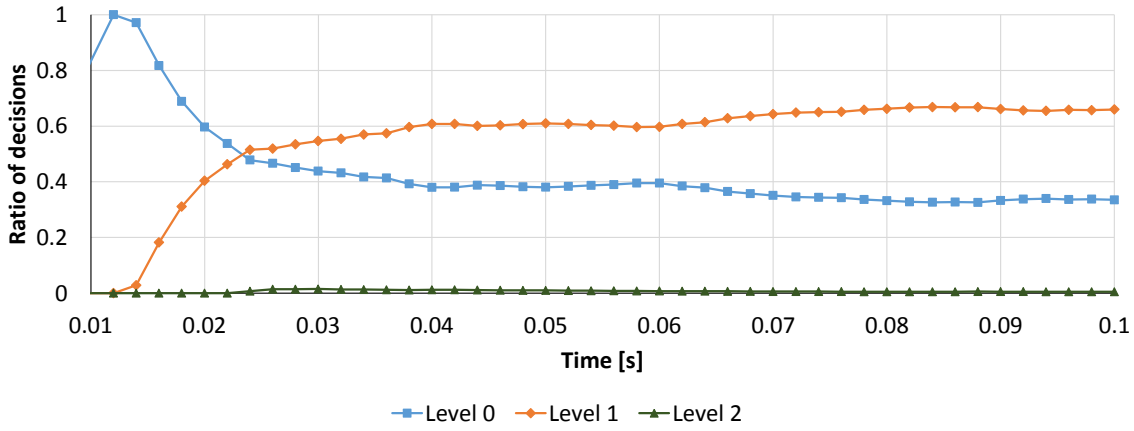
To gain deeper insight into how partitioning and partition-based SAT solving work, we present detailed information on the operation of the algorithms on the `38bits_10.dimacs.cnf` problem instance of the 2015 SAT Race. This formula has 448 variables and 12,700 clauses, and is known to be satisfiable. The formula length (total number of literals) is 91,605, so that the LtV ratio is roughly 204.5.

We apply our program optimized for the 50sec time limit, as presented in Figure 7b. Since the LtV ratio is greater than 30, the program decides to use the CFPH+FM heuristic for partitioning the dual hypergraph. The partitioning heuristic finishes in about 1 second and returns a cut with 50 variables. Since the cut is small, the next step is to use the soft partitioning solver with the variables grouped into three priority levels (level 0: cut variables, levels 1 and 2: the variables in the two disjoint subformulas that would arise after removing the variables of level 0). The initial activity of the variables on level 0 is set to 200, on level 1 to 2, and on level 2 to 0.

The solver finds a solution after roughly 31 seconds. It is interesting to analyze how the solver makes use of the partitioning information it received in the form of the three priority levels during those 31 seconds. To this end, Figure 8a shows how the decisions of the solver are spread across



(a) Full time range



(b) First 0.1 sec

Figure 8: Temporal development of the ratio of decisions on the different priority levels

the priority levels, as a metric of how much time the solver spends on each priority level, and how this changes as time passes. Since the beginning of the plot seems somewhat chaotic, we zoom in on the first 0.1 second in Figure 8b. (0.1 second might seem like a negligibly short period of time, but the solver actually makes over 21,000 assignments during that period and in particular, it makes decisions with far-reaching consequences.)

Based on the two plots, the solution process consists of the following phases:

1. At the very beginning (until about 0.025 sec), the solver mainly focuses on level 0, i.e., the cut variables. For example,
   - The first 89 decisions are made on level 0.
   - From the first 200 decisions, 150 are made on level 0.
   - After 0.01 second, already 44 of the 50 variables on level 0 are assigned values.
2. Between 0.025 and 1 second, the solver spends most of its time on level 1. Level 2 is hardly touched, but the solver often jumps back to level 0.
3. Between 1 and 5 seconds, the time share of level 0 further decreases. The solver still spends most of its time on level 1, but more and more often, the solver goes on to work with level 2.

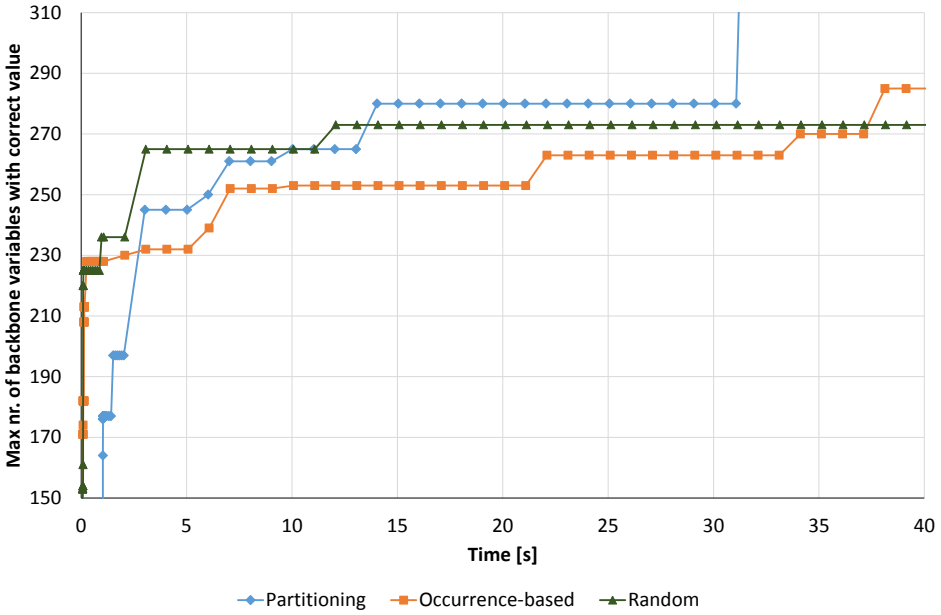


Figure 9: Comparison of different VSIDS activity initialization methods

4. From 5 seconds onwards, the focus of the solver shifts to level 2. Although the solver spends less and less time on the other two levels, their time share remains non-negligible.

This analysis reinforces that the solver indeed makes use of the partitioning information: it first focuses on level 0, then on level 1, and finally on level 2. But since partitioning information is taken into account only indirectly through the initial activity values of the variables, the pattern is not so crisp. There are no strict phase boundaries, and in particular, beside the dominant priority level of each phase, the share of the other priority levels remains non-negligible.

An interesting consequence is that for the above behavior to happen, the formula does not have to be completely cut into disjoint parts. When most of the cut variables have values, the ties between the two other priority levels are weak, so that the solver can already start focusing on one of the emerging subformulas.

8.2 Impact of VSIDS activity initialization

In order to better highlight the effect of the partition-based initialization of activity values, we also experimented with two other initialization methods (beyond the default initialization of `glucose`). In the *random* method, the activity of each variable is initialized with a random value from the same interval as used by our soft partitioning method ([0,200]). In the *occurrence-based* method, the initial activity of each variable is proportional to the number of its occurrences in the formula, again taking the numbers from the same interval.

We use again the problem instance of Section 8.1. For the purpose of comparing the effect of the three activity initialization methods, we wanted to measure the progress that each solver variant makes with time. Measuring the real progress of a CDCL solver is a non-trivial undertaking. Here, we used the following methodology: first, we determined the backbone of the SAT instance, i.e., the set of variables whose values are the same in all solutions [36]. In our case, we found out through a sequence of solver runs with different assumptions that the formula has a backbone with 398 variables. Then, we measure the progress by the best solution found so far in terms of the number of backbone variables having their correct values.

Figure 9 compares the effect of the three activity initialization methods on the sample problem instance. The figure gives rise to the following findings:

Table 8: Number of instances solved within 50 seconds, separated according to which partitioning heuristic is used

(a) Region where CFPH+FM is used

	2014		2015	All
	APP	HC		
Glucose	25	8	11	44
Soft part.	25	17	13	55
Combined	28	17	13	58

(b) Region where `hMetis` is used

	2014		2015	All
	APP	HC		
Glucose	1	28	48	77
Soft part.	1	28	52	81
Combined	1	27	53	81

- The partitioning-based solver variant starts approximately 1 second later than the others. This is due to the time it takes to partition the formula.
- In the beginning, the partitioning-based solver variant makes steady progress so that, despite the initial disadvantage, it gets ahead of the occurrence-based solver variant already at about 3 seconds.
- Although its pace decreases, the partition-based solver variant makes still clearly faster progress than the other solver variants, so that at about 14 seconds, it also passes by the third solver variant.
- At about 32 seconds (1 second for partitioning plus 31 seconds for solving), the partition-based solver finds a solution. The other two solver variants are lagging behind. (The plot does not show it, but the other solver variants finish at around 136 and 142 seconds.)

To sum up our findings: for the partition-based method, initialization takes longer and thus the actual solving starts later, but the initial disadvantage is compensated by faster progress in the solving phase, which in the end leads to finding the solution significantly faster than with the two other methods.

8.3 Impact of the partitioning heuristic

As shown in Section 7, the soft partitioning solver achieves a significant overall improvement over `glucose`. In the partitioning phase, sometimes the CFPH+FM heuristic, sometimes `hMetis` is used, as decided automatically based on the characteristics of the formula. Therefore, it is not clear how much the two heuristics contribute each to the overall success of the approach.

In order to clarify this, Table 8 shows the data of Table 7 separated according to the partitioning heuristic used. From this, we can draw multiple conclusions: (i) both partitioning heuristics contribute positively to the success of the proposed method, (ii) `hMetis` is used more often than CFPH+FM, (iii) the contribution of CFPH+FM to the overall improvement over `glucose` is higher than that of `hMetis`.

8.4 Impact of the problem instance

It is also interesting to analyze whether the improvement of the presented method over `glucose` arises only on a special family of problem instances. To this end, Table 9 lists the problem instances of the validation set that the soft partitioning technique managed to solve within 50 seconds but `glucose` did not, along with their number of variables and clauses. Table 10 shows a similar list for a time limit of 500 seconds.

As can be seen, both lists are quite heterogeneous, containing instances from the application and hard combinatorial categories from 2014 and the 2015 competition, several different types of applications / constructions, and spanning several orders of magnitudes in their sizes. This suggests that the advantage of soft partitioning over `glucose` is not only due to a specific family of instances, but is rather a more general phenomenon.

Table 9: Problem instances of the validation set solved by the soft partitioning solver that were not solved by `glucose` (time limit: 50 seconds)

Benchmark	#Vars	#Clauses
2014/application/9vliw_m_9stages_iq3_C1_bug8.cnf	521,179	13,378,580
2014/application/E02F20.cnf	10,420	393,557
2014/application/openstacks-p30_3.085-SAT.cnf	324,116	1,376,650
2014/application/openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf	324,116	1,376,650
2014/hard_combinatorial/jkkk-one-one-10-34-sat.cnf	11,607	65,748
2014/hard_combinatorial/ndist.b.20499.cnf	266,479	368,961
2014/hard_combinatorial/ndist.b.20998.cnf	272,966	377,943
2014/hard_combinatorial/ndist.b.22495.cnf	292,427	404,889
2014/hard_combinatorial/ndist.b.22994.cnf	298,914	413,871
2014/hard_combinatorial/ndist.b.23992.cnf	311,888	431,835
2014/hard_combinatorial/ndist.b.24491.cnf	318,375	440,817
2014/hard_combinatorial/ndist.b.25489.cnf	331,349	458,781
2014/hard_combinatorial/ndist.b.26487.cnf	344,323	476,745
2014/hard_combinatorial/ndist.b.27485.cnf	357,297	494,709
2014/hard_combinatorial/ndist.b.29481.cnf	383,245	530,637
2015/main/42bits_12.dimacs.cnf	538	14,509
2015/main/E02F20.cnf	10,420	393,557
2015/main/gss-18-s100.cnf	31,364	93,785
2015/main/manthey_DimacsSorterHalf_29_8.cnf	4,495	14,793
2015/main/manthey_DimacsSorter_28_7.cnf	3,507	11,386
2015/main/manthey_DimacsSorter_31_0.cnf	4,358	14,320
2015/main/manthey_single-ordered-initialized-w48-b8.cnf	11,520	87,592
2015/main/manthey_single-ordered-initialized-w54-b9.cnf	16,281	126,603
2015/main/mrpp_4x4_12_12.cnf	2,672	14,307
2015/main/mrpp_8x8_24_11.cnf	9,854	70,443
2015/main/openstacks-p30_3.085-SAT.cnf	324,116	1,376,650

Table 10: Problem instances of the validation set solved by the soft partitioning solver that were not solved by `glucose` (time limit: 500 seconds)

Benchmark	#Vars	#Clauses
2014/application/002-80-8.cnf	13,408	308,225
2014/application/UCG-15-10p0.cnf	199,304	743,886
2014/application/aaai10-planning-ipc5-pathways-17-step21.cnf	53,919	304,454
2014/application/atco_enc1_opt1_04_32.cnf	57,220	550,700
2014/application/beempgsol2b1.cnf	26,455	76,531
2014/application/gss-18-s100.cnf	31,364	93,785
2014/application/hwmcc10-timeframe-expansion-k45-pdtpmsgoodbakery-tseitin.cnf	98,935	296,401
2014/application/korf-17.cnf	6,664	89,966
2014/application/openstacks-p30_3.085-SAT.cnf	324,116	1,376,650
2014/application/openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf	324,116	1,376,650
2014/application/smtlib-qfbv-aigs-lfsr_004_127_112-tseitin.cnf	350,506	874,435
2014/application/vmpc_29.cnf	841	120,147
2014/hard_combinatorial/Hidoku_enu_7.cnf	4,711	22,674
2014/hard_combinatorial/ccp-s9-facto1.cnf	234	3,510
2014/hard_combinatorial/jkkk-random-132906006148277-10-10-34-SUM-sat.cnf	39,677	203,349
2014/hard_combinatorial/jkkk-random-132906006427000-10-10-35-OR-sat.cnf	41,391	192,150
2014/hard_combinatorial/jkkk-random-132906006493326-10-10-35-OR-sat.cnf	41,366	191,911
2014/hard_combinatorial/rook-40-0-1.cnf	67,117	257,392
2014/hard_combinatorial/rook-48-0-1.cnf	115,101	444,424
2014/hard_combinatorial/sgen4-sat-160-8.cnf	160	384
2014/hard_combinatorial/toughsat_factoring_426s.cnf	2,009	10,395
2015/main/002-80-8.cnf	13,408	308,225
2015/main/48bits_13.dimacs.cnf	688	18,146
2015/main/6s167-opt.cnf	4,640	13,077
2015/main/aes_64_1_keyfind_1.cnf	596	2,376
2015/main/atco_enc1_opt1_04_32.cnf	57,220	550,700
2015/main/jgiraldezlevy.2200.9086.08.40.22.cnf	2,200	9,086
2015/main/jgiraldezlevy.2200.9086.08.40.81.cnf	2,200	9,086
2015/main/jgiraldezlevy.2200.9086.08.40.85.cnf	2,200	9,086
2015/main/manthey_DimacsSorterHalf_31_2.cnf	5,408	18,446
2015/main/manthey_DimacsSorterHalf_36_0.cnf	4,551	14,683
2015/main/openstacks-p30_3.085-SAT.cnf	324,116	1,376,650
2015/main/vmpc_29.cnf	841	120,147

Table 11: Number of instances solved, from the instances of 2013, for different time limits

(a) 1000 sec

	APP	HC	All
Glucose	142	130	272
Soft part.	147	131	278

(b) 2000 sec

	APP	HC	All
Glucose	169	164	333
Soft part.	161	166	327

8.5 Higher time limits

As already mentioned, the suggested method excels for relatively short time limits like 50 seconds, and its effect is less pronounced for higher time limits like 500 seconds. Nevertheless, it can be interesting to check what happens for even higher time limits.

Table 11 shows the results of some experiments with higher time limits. As can be seen, for a time limit of 1000 seconds, the soft partitioning solver has some slight advantage over `glucose` for both application and hard combinatorial instances. For 2000 seconds, the small advantage on hard combinatorial instances can still be observed, but on application instances, `glucose` has become better.

9 Conclusion

In the paper, we examined the concept of solving SAT problems through dividing the input formula into two smaller parts. Our aim was to investigate how such an approach can be best implemented in the context of modern SAT solvers.

We investigated different models for reducing the problem to hypergraph partitioning, and compared multiple methods to find a small balanced cut in a hypergraph. We examined previously developed high-performance partitioning algorithms (FM, MLFM) and their appropriateness for this application. For hypergraphs where hyperedge size is beyond the reach of these methods, we developed the CFPH algorithm. We also showed that the domains where these algorithms outperform each other can be roughly characterized by some simple features of the hypergraph, allowing us to develop an effective heuristic that combines these approaches to always choose the most promising partitioning method.

We presented two different techniques (hard partitioning and soft partitioning) to implement partitioning-based formula solving in a CDCL SAT solver. Both approaches necessitate only modest changes within the SAT solver. We introduced these modifications into a state-of-the-art SAT solver, and evaluated the performance of the resulting algorithms on a wide variety of formulas obtained from the SAT competitions of recent years. One of the two techniques, the soft partitioning solver turned out to be a very successful approach that yields a notably better success rate than the original solver if the time available for solving is relatively small. Also, for the case of small time limit, we managed to merge the original and the soft partitioning solver in a combined algorithm that gives even better results than the soft partitioning solver. On the other hand, we found that the partitioning approach does not yield a significant performance increase for large time limits. Since the time available for the solvers in SAT competitions is usually in a larger magnitude, this may explain why partitioning-based techniques are rarely applied in SAT solvers optimized for the competitions. On the other hand, there are a number of real-world use cases where SAT solvers must work with much lower time limits, e.g. in the context of rapid design space exploration [47] or interactive verification tools [13]. In such cases, a well-tuned partitioning-based approach can yield substantial benefits.

There are various opportunities for further improvement. The most important open question is whether further changes to the solver or to the approach can allow us to significantly outperform the original solver for larger time limits, too. One such possibility is to experiment with techniques that partition the formula into more than two parts, either by an algorithm designed for this

purpose (for example, for the minimum multicut or the minimum multiway cut problem [38, Chapter 5]) or by applying the presented partitioning methods recursively. Also, developing more sophisticated partitioning algorithms or further partitioning models might allow us to split the formula even earlier in the solution process. Another question worth investigating is whether further refining the underlying SAT solver to our setting of partitioning, for example, by prohibiting learned clauses to connect the two subformulas, has notable positive effect on the performance of partitioning-based solvers.

Acknowledgments

The work of Z. Á. Mann was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947).

References

[1] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162(1):49–88, 2005.

[2] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, pages 410–423, 2012.

[3] Carlos Ansótegui and Jordi Levy. On the modularity of industrial SAT instances. In *14th International Conference of the Catalan Association for Artificial Intelligence*, pages 11–20, 2011.

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *21st International Joint Conference on Artificial Intelligence*, volume 9, pages 399–404, 2009.

[5] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming – Proceedings of CP 2012*, pages 118–126. Springer, 2012.

[6] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors. *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2. University of Helsinki, 2012. Department of Computer Science Series of Publications B.

[7] Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1. University of Helsinki, 2013. Department of Computer Science Series of Publications B.

[8] Tomás Balyo, Markus Iser, and Carsten Sinz. SAT Race 2015. Available at: `http://baldur.iti.kit.edu/sat-race-2015/sr15.pdf`.

[9] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 157–162, 2000.

[10] Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2. University of Helsinki, 2014. Department of Computer Science Series of Publications B.

[11] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:201–208, 2006.

[12] Per Bjesse, James Kukula, Robert Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In *Theory and Applications of Satisfiability Testing – Selected Revised Papers of SAT 2003*, pages 315–329. Springer, 2004.

[13] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 454–464, 2001.

[14] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[15] Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov. Design and implementation of the Fiduccia-Mattheyses heuristic for VLSI netlist partitioning. In *Algorithm Engineering and Experimentation – Selected Papers of ALENEX'99*, pages 182–198. Springer, 1999.

[16] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, 2004.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[18] Ali Dasdan and Cevdet Aykanat. Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(2):169–178, 1997.

[19] Vijay Durairaj and Priyank Kalla. Exploiting hypergraph partitioning for efficient Boolean satisfiability. In *Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop*, pages 141–146. IEEE, 2004.

[20] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing – Selected Revised Papers of SAT 2003*, pages 502–518. Springer, 2004.

[21] Charles M. Fiduccia and Robert M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[22] Martin Fürer and Shiva Prasad Kasiviswanathan. Exact MAX 2-SAT: Easier and faster. In *SOFSEM 2007: Theory and Practice of Computer Science – Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science*, pages 272–283. Springer, 2007.

[23] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

[24] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[25] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 379–572. Springer, 1999.

[26] Marijn Heule and Oliver Kullmann. Decomposing clause-sets: Integrating DLL algorithms, tree decompositions and hypergraph cuts for variable- and clause-based graph representations of CNFs. Technical report, University of Wales Swansea, 2006. CSR 2-2006.

[27] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing*, pages 50–65. Springer, 2011.

[28] Achim G. Hoffmann. The dynamic locking heuristic – a new graph partitioning algorithm. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'94)*, volume 1, pages 173–176. IEEE, 1994.

[29] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, volume 3, pages 1167–1172, 2003.

[30] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. SAS+ planning as satisfiability. *Journal of Artificial Intelligence Research*, 43(1):293–328, 2012.

[31] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[32] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[33] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI Design*, 11(3):285–300, 2000.

[34] George Katsirelos and Laurent Simon. Eigenvector centrality in industrial SAT instances. In *Principles and Practice of Constraint Programming*, pages 348–356, 2012.

[35] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[36] Philip Kilby, John Slaney, Sylvie Thiebaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 1368–1373, 2005.

[37] Wei Li and Peter Van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 542–548. IEEE, 2004.

[38] Zoltán Ádám Mann. *Optimization in computer engineering–Theory and applications*. Scientific Research Publishing, Inc. USA, 2011.

[39] Zoltán Ádám Mann. Typical-case complexity and the SAT competitions. In *Proceedings of the 5th Pragmatics of SAT Workshop (POS-14)*, volume 27 of *EasyChair Proceedings in Computing*, pages 72–87, 2014.

[40] Zoltán Ádám Mann, András Orbán, and Viktor Farkas. Evaluating the Kernighan-Lin heuristic for hardware/software partitioning. *International Journal of Applied Mathematics and Computer Science*, 17(2):249–267, 2007.

[41] Zoltán Ádám Mann and Pál András Papp. Formula partitioning revisited. In *Proceedings of the 5th Pragmatics of SAT Workshop (POS-14)*, volume 27, pages 41–56. EasyChair Proceedings in Computing, 2014.

[42] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.

[43] Anthony Monnet and Roger Villemaire. Scalable formula decomposition for propositional satisfiability. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 43–52. ACM, 2010.

[44] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, pages 530–535. ACM, 2001.

[45] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on SAT solver performance. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2014)*, pages 252–268, 2014.

[46] Tai Joon Park and Allen Van Gelder. Partitioning methods for satisfiability testing on large formulas. In *Automated Deduction – Cade-13*, pages 748–762. Springer, 1996.

[47] Thomas Schlichter, Martin Lukasiewycz, Christian Haubelt, and Jurgen Teich. Improving system level design space exploration by incorporating SAT-solvers into multi-objective evolutionary algorithms. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, pages 309–316, 2006.

[48] Ohad Shacham and Emmanuel Zarpas. Tuning the VSIDS decision heuristic for bounded model checking. In *Proceedings of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, pages 75–79. IEEE, 2003.

[49] Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Parallel Processing and Applied Mathematics – Revised Selected Papers of PPAM 2007*, pages 249–258. Springer, 2008.

[50] Jose Torres-Jimenez, Luis Vega-Garcia, Cesar A. Coutino-Gomez, and Francisco J. Cartujano-Escobar. SSTP: An approach to solve SAT instances through partition. In *4th WSEAS International Conference on Information Science, Communications and Applications (ISA 2004)*, pages 484–403, 2004.

[51] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.