Formula partitioning revisited

Zoltán Ádám Mann and Pál András Papp

Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Magyar tudósok körútja 2., 1117 Budapest, Hungary

Abstract

Dividing a Boolean formula into smaller independent sub-formulae can be a useful technique for accelerating the solution of Boolean problems, including SAT and #SAT. Nevertheless, and despite promising early results, formula partitioning is hardly used in state-of-the-art solvers. In this paper, we show that this is rooted in a lack of consistency of the usefulness of formula partitioning techniques. In particular, we evaluate two existing and a novel partitioning model, coupled with two existing and two novel partitioning algorithms, on a wide range of benchmark instances. Our results show that there is no one-size-fits-all solution: for different formula types, different partitioning models and algorithms are the most suitable. While these results might seem negative, they help to improve our understanding about formula partitioning; moreover, the findings also give guidance as to which method to use for what kinds of formulae.

1 Introduction

Dividing a problem into smaller sub-problems that can be solved independently is a technique that works well for many hard problems [7]. Accordingly, it has been suggested to use this idea to accelerate solving of the Boolean satisfiability (SAT) problem [5, 24, 26, 9, 17, 20, 30, 27, 28, 13] as well as some other related problems like #SAT [4] and MAX-2-SAT [11].

Despite several promising preliminary results, formula partitioning has not become a mainstream technique. We attribute this to the following. Most previous attempts were focusing on a specific family of Boolean formulae and managed to achieve good results on those formulae. However, there are huge differences between different formula types and therefore there is no guarantee that the same method will work well on other formulae as well. For example, a random formula with 200 variables and 1,000 clauses may be similarly difficult for a state-of-the-art SAT solver as a formula encoding a verification problem with 10,000 variables and 1,000,000 clauses, but a partitioning algorithm that can partition one of them well may not at all be appropriate for the other. Hence, the aim of this paper is to develop a better understanding of the success factors of formula partitioning: when is it useful at all and which partitioning method works best? As we will see, there are several ways to partition a Boolean formula into independent sub-formulae: the *partitioning model* determines what kinds of cuts will be considered (e.g., partitioning the primal or dual hypergraph representation of the formula) and how a cut can be used to let the formula fall into pieces; and the *partitioning algorithm* selects the actual variables or clauses that form a cut.

In contrast to most previous work, we evaluate several different partitioning models and partitioning algorithms on a wide range of benchmark instances, with the aim of an objective and unbiased comparison. We consider two existing and a novel partitioning model, combined with two existing and two novel partitioning algorithms (altogether 12 combinations). Ideally, we would like to find a method that works well on all investigated benchmarks. However, according to our empirical results, none of the investigated methods yields consistently good results across the whole range of benchmarks. This is in contrast to the success stories reported

previously in the literature; we believe that this is because using a more heterogeneous set of benchmarks, we get a more accurate and more realistic picture than previous studies. This is also probably the reason why formula partitioning has not found wide-spread application in state-of-the-art solvers. On the positive side, our findings give an indication about which methods work well on what kinds of benchmarks; this insight might pave the way for more successful application of partitioning-based methods in SAT in the future.

The contributions of the paper are as follows: (i) a new partitioning model for using cuts of the primal hypergraph representation of the formula; (ii) two new partitioning algorithms that clearly outperform previous algorithms for partitioning the dual hypergraph representation of large problem instances; (iii) an empirical comparison of 8 partitioning methods on a wide range of different instances, leading to a much better understanding of the capabilities of formula partitioning than any previous study; and (iv) recommendations resulting from the empirical results on which method to use for what kind of instances.

2 Previous work

Biere and Sinz showed that, if the formula consists of multiple disconnected parts, this can be taken advantage of with a small change in a modern SAT solver [5]. They also showed that, even without explicitly aiming for this, it occurs during the run of a SAT solver that the formula becomes disconnected; if the solver recognizes this, it can make the search more efficient.

Several researchers devised algorithms for explicitly partitioning the input formula. One of the first such approaches is due to Park and Gelder: they use partitioning heuristics borrowed from VLSI design (the Fiduccia-Mattheyses algorithm), applied to the so-called dual hypergraph representation of the formula, meaning that the set of clauses is partitioned by the removal of variables [26]. Their empirical results confirm the usefulness of this approach; however, the investigated problem instances that were hard to solve at that time can be meanwhile considered very easy. Similar methods were suggested later by Durairaj and Kalla [9] and Torres-Jimenez et al. [30]. Other researchers also used similar techniques to define a variable ordering for a SAT solver that will lead to a divide-and-conquer behaviour [17, 20]. The primal hypergraph representation of the problem, in which the set of variables is partitioned by the removal of clauses, has also been considered [28, 24]. Amir and McIlraith used a graph representation similar to the primal hypergraph, and searched for a vertex separator in it [1]. The community structure (i.e., a decomposition into highly connected parts with few connections among them) of SAT formulae has also been investigated [2].

If the idea of partitioning the input formula is recursively applied to the received sub-formulae as well, and so on, then we get a *tree decomposition*. Some researchers considered to generate the whole tree decomposition, and then use this information for variable ordering during the search [17, 24, 6]. The difficulty of this approach is that finding an optimal tree decomposition is in itself a tough problem.

All the above works focused on one specific decomposition method. Heule and Kullmann presented a comprehensive review and comparison of multiple methods [13]. Their goal was similar to ours; however, while their work is mostly of theoretical nature, our aim is to find out what works well in practice.

It is also worth mentioning that several parallel SAT solvers are based on the idea of partitioning the search space between the search instances [15, 23]. However, search space partitioning is not the same as formula partitioning: e.g., for partitioning the search space into 8 parts, it is sufficient to choose 3 variables and assign values to them in all possible ways, leading to 8 disjoint parts of the search space. Formula partitioning is much more complex.

3 Preliminaries

We are given n Boolean variables. A *literal* is a variable or its negation. A *clause* is the disjunction of literals. A *formula* (in conjunctive normal form) is the conjunction of clauses. An *interpretation* of the variables is a function that assigns to each variable one of its two possible values $\{true, false\}$. A *solution* is an interpretation which makes the formula evaluate to *true*.

A *hypergraph* is a pair $(V, \mathcal{F})$, where V is a finite, non-empty set and the elements of $\mathcal{F}$ are subsets of V. The elements of V are called *vertices*, the elements of $\mathcal{F}$ are called *hyperedges*.

A hypergraph $(V, \mathcal{F})$ is *disconnected*, if there is a partition $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$, such that there is no hyperedge $E \in \mathcal{F}$ with $E \cap V_1 \neq \emptyset$ and $E \cap V_2 \neq \emptyset$. A *cut* in a hypergraph $(V, \mathcal{F})$ is a set of hyperedges $\mathcal{F}' \subseteq \mathcal{F}$, such that $(V, \mathcal{F} \setminus \mathcal{F}')$ is disconnected.

In the *primal hypergraph* of a formula, each variable is represented by a vertex and each clause is represented by a hyperedge. The hyperedge representing a clause c contains the vertices that represent variables occurring (with or without negation) in c.

In the *dual hypergraph* of a formula, each clause is represented by a vertex and each variable is represented by a hyperedge. The hyperedge representing a variable x contains the vertices that represent clauses containing x (with or without negation).

Assume that the set of variables X is partitioned: $X = X_1 \cup X_2$, $X_1 \cap X_2 = \emptyset$. Let c be a clause that contains variables from both X_1 and X_2. Let c_1 denote the part of c with variables from X_1 and let c_2 denote the part of c with variables from X_2. Then, *splitting* c means introducing a new auxiliary variable z_c and replacing c with the two new clauses $c_1 \vee z_c$ and $c_2 \vee \overline{z}_c$. It can be seen easily that this transformation does not change the satisfiability of the problem instance.

4 Partitioning methods

4.1 Partitioning model

The underlying idea of all partitioning-based methods is to identify a part f of the input formula F such that removing f would cause the formula to fall into independent pieces. Of course, f cannot be simply removed, but the solver can be guided in such a way that it first assigns values to (some of) the variables in f, so that afterwards, the formula will indeed fall into pieces that can be solved independently. There are multiple options for choosing f and for its "removal." Specifically, we will investigate the following *partitioning models*:

- **Dual**: Find a cut of the dual hypergraph, corresponding to a set of variables, such that removing these variables would cause the formula to fall into pieces. The SAT solver is guided such that it assigns values to these variables first.
- **Primal / model generation**: Find a cut of the primal hypergraph, corresponding to a set of clauses, such that removing these clauses would cause the formula to fall into pieces. Take all variables appearing in these clauses. The SAT solver is guided such that it assigns values to these variables first.
- **Primal / auxiliary variables**: Find a cut of the primal hypergraph, corresponding to a set of clauses C_0, such that removing these clauses would cause the formula to fall into pieces with variable sets X_1 and X_2, respectively. Split all clauses in C_0 with respect to the partition (X_1, X_2) by introducing $|C_0|$ new auxiliary variables. The SAT solver is guided such that it assigns values to these variables first.

It can be seen easily that all three methods will indeed lead to a partitioned formula. The first two methods have already been used by other researchers to partition Boolean formulae. The third method is, to our knowledge, new.

4.2 Partitioning algorithms

In all three partitioning models, we need to find a cut in a hypergraph. In order for this cut to be useful, it should fulfil two requirements: (i) its size (i.e., the number of hyperedges in the cut) should be as small as possible and (ii) it should be a *balanced* cut, i.e., the two sub-hypergraphs resulting after removal of the cut hyperedges should be of similar size. More specifically, each of the two resulting sub-hypergraphs should contain at most βN vertices, where N is the number of vertices of the original hypergraph and $1/2 < \beta < 1$ is a given constant.

Unfortunately, finding a balanced cut of minimal size is an NP-hard problem [21]. On the other hand, there are well-known heuristics for this problem that have proven to be very successful in VLSI netlist partitioning and other related applications, including the Kernighan-Lin (KL) heuristic [19], its improved version, the Fiduccia-Mattheyses (FM) heuristic [10] and further improvements to FM [22]. In this work, we used the FM heuristic and some of its variants because it is a well-established method in netlist partitioning and its use in Boolean formula partitioning has also been advocated in the literature (e.g. in [26]).

The FM heuristic works in passes. In each pass, the algorithm starts with a balanced partition of the vertices, and makes changes to this partition in a series of steps. In each step, a vertex is moved from one part to the other. A vertex can be moved if (i) it has not yet been moved in the current pass and (ii) moving it to the other part will not violate the balance criterion. The algorithm is greedy in the sense that, in each step, it selects from the vertices that can be moved the one whose moving will most decrease the cut size. However, it makes this move even if it is a worsening move, i.e. when even the best move will actually increase the cut size. This way, the algorithm can escape local optima. At the end of the pass, the partition with the smallest cut that was observed during the pass is selected and used as starting partition for the next pass. The algorithm terminates if either a pre-defined number of passes have been made or the last pass failed to decrease the cut size. By using a sophisticated data structure called gain bucket array, one pass of the FM algorithm can be performed in linear time.

In this work, we used the following partitioning algorithms:

- **FM**: The original FM algorithm.
- **Multi-move**: An extended version of the FM algorithm, in which each vertex is allowed to be moved k times per pass, instead of just once as in the original algorithm. Approaches of this kind have been suggested in the literature to improve the effectiveness of FM [14, 8].
- **Soft gain**: In each step, the FM algorithm chooses the vertex with the highest gain from the vertices that can be moved. In the original algorithm, the gain of a vertex is defined as the decrease in cut size achieved by moving the given vertex to the other part. In the soft-gain version, the gain of a vertex can be positive if, although moving it to the other part will not decrease the cut size, but it is a move "in the right direction." We define some thresholds $0.5 < t_1 < t_2 < \ldots < t_\ell = 1$ and corresponding soft gains $0 < g_1 < g_2 < \ldots < g_\ell = 1$. For a given hyperedge E with r vertices, if less than $t_i r$ of its vertices are in the same part but moving vertex v to the other part results in at least $t_i r$ of the vertices of E being in the same part, then v gets a gain of g_i.
- **Hyperedge moving**: In this version of the algorithm, each step consists of moving all vertices of a hyperedge to the same part. A hyperedge can be moved if it has not been

moved in the current pass and moving it does not violate the balance criterion. From the hyperedges that can be moved, we select randomly, where the selection probability of a hyperedge is proportional to the number of its vertices in the target part.

The first two algorithms are well-known and widely used hypergraph partitioning methods [22]. The third and fourth have been developed by us in the course of this work, because our empirical results suggested that the first two algorithms are not appropriate for partitioning the dual hypergraphs of large formulae. In such hypergraphs, the number of vertices is huge (it can easily be in the range of $10^5 \dots 10^7$) compared to the number of hyperedges (usually in the range of $10^2 \dots 10^4$), and the average size of hyperedges is also quite big ($10^2 \dots 10^3$). In such cases, the gain of most vertices is zero, and thus the FM heuristic becomes meaningless. The third and fourth algorithms above are ideas to overcome this problem. Specifically, the soft gain approach helps by differentiating between the vertices whose "hard" gain would be all zero by encouraging moves that will likely be useful together with other moves; whereas hyperedge moving enforces this in a more drastic way by moving, in a single step, all vertices that are necessary for the given hyperedge not to be cut.

Finally, it should be noted that the primal partitioning combined with model generation leads actually to a dual cut: the variables that occur in the cut clauses build a cut in the dual hypergraph. Thus, we can either search for a dual cut directly or find a primal cut and then use model generation to identify the corresponding dual cut. One could argue that it is more effective to search for a good dual cut directly in the dual hypergraph; however, because of the above-mentioned difficulties of finding good cuts in the dual hypergraphs, it may be better to make the detour through the primal hypergraph. We will come back to this question later on.

5 Empirical results

We implemented the partitioning methods described above in C++. Although our implementation does not contain any low-level optimization, we did use the gain bucket array data structure for FM and also adapted it to the other methods to ensure a reasonable level of efficiency. Furthermore, instead of explicitly creating the hypergraph representations of the formulae, our methods work directly on the formulae themselves, thus avoiding the conversion overhead.

Our aim is to compare the efficiency and effectiveness of the different partitioning methods on a wide range of benchmark instances. The measurements that we describe next were carried out on a PC with Intel Core i7 CPU Q720, 2x1.60GHz, 4 GB RAM, running Windows 7. In order to ensure reproducibility of the results, we made available the source code of our implementations as well as the detailed results under `http://www.cs.bme.hu/~manusz/data/SAT_partitioning`.

5.1 Comparison using different families of benchmarks

For the first experiments, we used several different families of benchmark instances: inductive inference problem instances [18]; board-level routing problem instances [29]; SAT-encodings of graph coloring problem instances, random 3-SAT instances with controlled backbone size, and logistics and blocks world instances, from SATLIB [16] (see Table 1).

Our experience is that the partitioning methods give similar results on instances of the same problem family, but may behave quite differently on different problem families; therefore, in the following, we give the results as averages over the investigated problem families.

First, we analyze the effectiveness of the primal methods. Table 2 shows the results of the four investigated algorithms in terms of the obtained cut sizes, and Table 3 shows the

Table 1: Characteristics of the used benchmarks

Category	Number of instances	Average number of variables	Average number of clauses
Controlled backbone size	10	100	403
Inductive inference / big	10	1,240	16,479
Inductive inference / small	14	412	7,372
Graph coloring	12	600	2,237
Board-level routing / small	9	237	16,103
Board-level routing / big	6	418	133,802
Logistics	4	1,881	11,682
Blocks world	3	668	8,500

Table 2: Average cut sizes resulting from partitioning the primal hypergraph, as percentage of the number of all clauses

	FM	Multi-move	Soft gain	Hyperedge moving
Controlled backbone size	36.23%	36.38%	36.38%	65.38%
Inductive inference / big	0.90%	0.90%	0.90%	–
Inductive inference / small	0.87%	0.87%	1.07%	24.81%
Graph coloring	8.80%	8.80%	8.80%	36.58%
Board-level routing / small	2.87%	3.00%	2.86%	–
Board-level routing / big	1.40%	1.27%	1.87%	–
Logistics	7.00%	6.99%	6.57%	45.05%
Blocks world	5.48%	5.71%	5.34%	43.56%

corresponding runtimes. We used a timeout of 1 minute; "–" in the table means that the algorithm did not terminate during this time. As can be seen, the hyperedge moving algorithm is useless on the primal hypergraphs. The other three algorithms achieve very similar results, but the plain FM algorithm is clearly better than the other two in terms of speed.

The primal hypergraphs of the investigated formulae have between 100 and 5,000 vertices

Table 3: Average runtimes of partitioning the primal hypergraph (in milliseconds)

	FM	Multi-move	Soft gain	Hyperedge moving
Controlled backbone size	10	256	108	94
Inductive inference / big	722	2,581	2,084	–
Inductive inference / small	282	2,007	390	4,853
Graph coloring	36	2,187	725	1,514
Board-level routing / small	353	3,698	470	–
Board-level routing / big	3,984	21,521	3,496	–
Logistics	1,491	3,380	1,842	23,411
Blocks world	799	2,453	1,337	25,711

and between 400 and 140,000 hyperedges; the average size of a hyperedge is between 2 and 6. Such hypergraphs are similar to the ones that occur in netlist partitioning for which the FM algorithm was originally devised. This explains why the plain FM algorithm performs well on these hypergraphs, and the other variants add only overhead without significant gain. In particular, even for the biggest problems (over 130,000 clauses), FM finds a relatively small cut; probably, the structured nature of these application instances contributes to this good result.

Table 4: Model generation: number of variables in the cut clauses, as percentage of the number of all variables

Benchmarks	Variables in primal cut
Controlled backbone size	97.10%
Inductive inference / big	77.39%
Inductive inference / small	43.39%
Graph coloring	48.14%
Board-level routing / small	99.09%
Board-level routing / big	99.73%
Logistics	28.55%
Blocks world	39.03%

However, if the primal partition is used with the model generation approach, then the total number of variables in the cut clauses is actually more important than the number of cut clauses. Table 4 shows these numbers for the FM method (the results of the Multi-move and Soft gain algorithms are very similar). As can be seen, even a small number of cut clauses can lead to a large set of variables that need to be assigned values before the formula falls into pieces. This is the case most notably for the board-level routing problems and the random instances with controlled backbone size. On other benchmarks, e.g. the logistics instances, the model generation method seems to work quite well.

It may be surprising that a small number of cut clauses can lead to such a high number of variables in the cut clauses, given that the clauses are usually not big. This is explained by the fact that the number of cut clauses is only small relative to the total number of clauses, which can be quite high. For example, in the big board-level routing instances, the average number of clauses is 133,802, so that the cut of size 1.40% delivered by FM means that the number of cut clauses is 1,873 on average. Since such instances have 418 variables on average, it is no surprise that the 1,873 cut clauses contain practically all variables.

When the method of auxiliary variables is used, then the issue of too many variables in the cut clauses does not apply. However, this method suffers from a growth of the formula, which can be substantial in some cases, as shown in Table 5. In particular, the number of variables increases by the number of auxiliary variables, which is equal to the number of cut clauses; thus, the resulting percentage increase is high if the number of cut clauses is high relative to the number of variables. Consequently, the relative increase in the number of variables is high in the same cases where the model generation approach resulted in a high number of variables in the cut clauses: the board-level routing and the controlled backbone instances. On the other hand, the increase in the number of clauses (which is due to the fact that the cut clauses are split in two in this method), is relatively small in most cases. Also, it is important to mention that an increase in the size of the formula does not necessarily make it harder.

Having assessed the two approaches working with the primal hypergraph, let us now turn

Table 5: Auxiliary variables: growth of the formulae

Benchmarks	Increase of the nr. of variables	Increase of the nr. of clauses
Controlled backbone size	146.00%	36.23%
Inductive inference / big	12.82%	0.90%
Inductive inference / small	13.70%	0.87%
Graph coloring	32.81%	8.80%
Board-level routing / small	147.92%	2.87%
Board-level routing / big	521.75%	1.40%
Logistics	53.75%	7.00%
Blocks world	68.07%	5.48%

to the dual methods.

Table 6: Average cut sizes resulting from partitioning the dual hypergraph, as percentage of the number of all variables

	FM	Multi-move	Soft gain	Hyperedge moving
Controlled backbone size	60.70%	61.30%	52.50%	71.40%
Inductive inference / big	99.79%	99.79%	64.46%	34.97%
Inductive inference / small	100.00%	100.00%	96.65%	15.14%
Graph coloring	23.96%	23.85%	22.65%	37.88%
Board-level routing / small	100.00%	100.00%	100.00%	44.31%
Board-level routing / big	100.00%	100.00%	100.00%	69.09%
Logistics	76.43%	76.65%	19.44%	37.28%
Blocks world	98.63%	98.62%	52.66%	50.52%

Table 7: Average runtimes when partitioning the dual hypergraph (in milliseconds)

	FM	Multi-move	Soft gain	Hyperedge moving
Controlled backbone size	160	191	30	8
Inductive inference / big	1,650	1,802	3,230	9,441
Inductive inference / small	180	164	172	211
Graph coloring	2,594	2,232	105	844
Board-level routing / small	687	2,123	736	152
Board-level routing / big	21,292	21,467	22,451	1,723
Logistics	2,321	1,828	4,481	3,621
Blocks world	2,523	1,508	2,797	2,173

Tables 6 and 7 show the results and runtime of the dual partitioning methods, respectively. As can be seen, the FM and Multi-move algorithms have again very similar performance. However, this time, the Soft gain and Hyperedge moving algorithms perform significantly better:

for each problem family, the best result is achieved by either of these two. As mentioned earlier, the dual hypergraphs of typical CNF formulae have a huge number of vertices and relatively few but big hyperedges. As a result, the FM algorithm does not find non-trivial cuts in most cases. The Soft gain and Hyperedge moving algorithms were designed specifically for coping with such hypergraphs, and they can indeed find significantly better cuts. The Hyperedge moving algorithm finds non-trivial cuts in all cases, even when none of the others do.

It is difficult to make a meaningful comparison between primal and dual methods. Comparing the primal results in Table 2 and the dual results in Table 6, one could state that the primal methods yield smaller relative cuts. However, a primal cut is not used directly, but either through model generation or with auxiliary variables, and – as we have seen before – both of these methods incur a penalty; on the other hand, dual cuts can be used directly. It is a better idea to compare the results of primal model generation with dual partitioning results because, as mentioned earlier, the set of variables that appear in the clauses of the primal cut is actually a dual cut. Hence, the results in Table 4 should be compared to the ones in Table 6. According to this comparison, searching directly for a good dual cut is in most cases better than the dual cut induced by primal model generation: for 4 benchmark families, Hyperedge moving yields the best dual cut, for 3 benchmark families, Soft gain is the winner, and for only one benchmark family is the dual cut induced by primal model generation superior.

5.2 Impact on SAT solver performance

In a next set of experiments, we investigated how a SAT solver can make use of partitioning. We used our own experimental solver, which is a simple implementation of the CDCL (constraint-driven clause learning) approach [12], underlying most of the state-of-the-art solvers. The advantage of using this solver is that it is very easy to modify and since it uses the same principles as today's best solvers, the results obtained with this solver are likely to transfer to other CDCL solvers as well, making it ideal for experimentation. On the other hand, the choice of the solver is also a limitation: lacking the low-level optimization techniques of top solvers, its runtime on big instances is prohibitively high. In the future, we definitely want to try the investigated partitioning techniques also in conjunction with a state-of-the-art solver.

We extended the solver to handle partitioned formulae. For this purpose, the solver receives as input – beside the CNF formula – three sets of variables: the cut variables (X_1) and the sets of variables of the two sub-formulae remaining after removal of the cut variables (X_2, X_3). The solver sets the priorities of the variables so that first the variables in X_1, then the variables in X_2, and finally the variables in X_3 are assigned values. This is done using three distinct queues for the three sets of variables. Within X_i, the VSIDS heuristic is used [25].

This way of using the partitioning information in the solver is similar to the method used by Huang and Darwiche [17]. There is an important difference though: while Huang and Darwiche use decompositions recursively to derive the whole ordering of the variable assignments, we use only one decomposition on the top level, and leave everything else to the VSIDS heuristic. Since VSIDS is a very successful heuristic [12], we prefer to minimize the interference on it.

We compared the performance of the solver on the investigated benchmarks with the presented partitioning methods as preprocessor and also without partitioning. The results are summarized in Table 8. As can be seen, in most cases, one of the partitioning methods helps to reduce solver runtime, sometimes even quite significantly. In particular, the primal model generation and the dual soft gain methods were quite effective. Looking also at the results on each individual instance, not only the instance families, reveals that also several other partitioning methods (primal – model generation – soft gain, primal – auxiliary variables – FM, dual

Table 8: Improvement of SAT solver performance by means of partitioning

	Best partitioning method	Improvement w.r.t. base solver
Controlled backbone size	Dual – FM	28%
Inductive inference / big	Primal – model generation – FM	92%
Inductive inference / small	Dual – Soft gain	35%
Graph coloring	Dual – Soft gain	55%
Board-level routing / small	*Base solver*	–
Board-level routing / big	Primal – model generation – FM	36%
Logistics	Primal – model generation – FM	16%
Blocks world	*Base solver*	–

– multi-move, dual – hyperedge moving) proved useful in some cases. Altogether, there is no clear winner among the partitioning methods in terms of their impact on SAT solving.

An important assumption behind formula partitioning is that we aim at small cuts. Although our results are limited by the solver and benchmarks, they show that in most cases there is indeed a correlation between cut size and the resulting speedup. However, there are some benchmark families where smaller cuts do not seem to lead to higher speedup; in particular, this is the case for the small board-level routing problems and the blocks world problems, where partitioning was not helpful at all. This is an important area for further research.

Another important detail is that clauses learned while processing the variables in X_1 may connect variables in X_2 and X_3, thus corrupting the cut. Our experience is that this phenomenon does sometimes happen, but its impact is marginal: such clauses are used for propagation in less than 1% of the cases.

5.3 Comparison on SAT competition instances

In order to further diversify the set of investigated problem instances, we ran another series of experiments with a subset of the problem instances from the 2013 SAT competition (`http://www.satcompetition.org/2013/`)[1].

The competition benchmark instances are grouped into three categories: Application, Hard combinatorial, and Random. Some of the competition instances (especially in the Application category) are really huge, containing hundreds of thousands of variables and tens of millions of clauses, and are still relatively easy to solve because of their intrinsic structure and the substantial optimizations in state-of-the-art CDCL solvers. We did not include such huge instances in our experiments because (i) our partitioning algorithms are not optimized for handling such large amounts of data efficiently and (ii) the basic idea of formula partitioning is to help exponential-time solvers by reducing the problem size, but in such cases, solution time is certainly not exponential in the size of the instance, making partitioning superfluous. Hence, we chose the instances from the Application and Hard combinatorial series using the following methodology. We first ordered the instances in ascending order according to their size, where size is measured as the total number of literals in all clauses. We grouped the instances in groups of five, and included these groups of instances until we encountered a group in which at

[1]Because of the above-mentioned limitations of our solver, we tested only the partitioning algorithms, not the solver. Testing with a state-of-the-art solver on competition benchmarks is a topic for future research.

least half of the investigated partitioning methods timed out (using a timeout of 60 seconds) on at least 4 of the 5 instances. This way, we included 70 Application instances and 60 Hard combinatorial instances. (There would have been more Hard combinatorial benchmarks in the desired range; we reduced their number to 60 in order to avoid a bias of the results.)

Table 9: Characteristics of the used competition benchmarks

Category	Number of instances	Average number of variables	Average number of clauses	Average size
Application	70	11,462	50,275	159,565
Hard combinatorial	60	8,704	40,114	116,291
Random	40	801	7,848	37,732

There are two kinds of Random instances among the SAT competition benchmarks: (i) instances that are relatively small but still difficult to solve because their clauses-to-variables ratio is near the satisfiability threshold and (ii) instances that are much bigger but their clauses-to-variables ratio is far from the satisfiability threshold [3]. For reasons similar to the ones described above, we included only instances of the first kind in our experiments. These are k-SAT instances with $k = 3, 4, 5, 6, 7$; we sampled 4 satisfiable and 4 unsatisfiable instances from each family, altogether 40 instances.

The characteristics of the used competition benchmarks are summarized in Table 9.

Table 10: Results on Application instances

	Size of primal cut	Size of dual cut	Runtime of partitioning (ms)
Primal FM	5.07%	19.47%	5,881
Primal Multi-move	4.83%	18.79%	9,417
Primal Soft gain	5.08%	19.57%	6,741
Primal Hyperedge moving	84.30%	88.34%	47,489
Dual FM		63.40%	23,912
Dual Multi-move		61.87%	22,477
Dual Soft gain		40.26%	27,602
Dual Hyperedge moving		53.22%	30,808

The results on the competition benchmarks are presented in Tables 10-12. In these tables, "Size of primal cut" refers to the number of clauses in the cut delivered by the primal partitioning methods, relative to the number of clauses of the formula; "Size of dual cut" refers to the number of variables in the primal cut in the case of primal methods, or the number of variables forming the cut found by the dual methods, both relative to the number of variables of the formula. As mentioned before, the number of variables in a primal cut is of high importance if model generation is used; furthermore, it allows a direct comparison between primal and dual methods.

Based on Tables 10-12, the following observations can be made:

- Primal Hyperedge moving is not a competitive method. (This is no wonder, since Hyperedge moving was developed with specifically dual hypergraphs in mind.)
- The other three primal methods yield very similar results on all benchmark categories.

Table 11: Results on Hard combinatorial instances

	Size of primal cut	Size of dual cut	Runtime of partitioning (ms)
Primal FM	16.48%	60.73%	4,744
Primal Multi-move	16.66%	61.11%	6,188
Primal Soft gain	18.05%	61.60%	5,691
Primal Hyperedge moving	71.93%	92.90%	34,591
Dual FM		73.32%	12,779
Dual Multi-move		73.32%	12,542
Dual Soft gain		43.84%	13,724
Dual Hyperedge moving		61.50%	21,907

Table 12: Results on Random instances

	Size of primal cut	Size of dual cut	Runtime of partitioning (ms)
Primal FM	64.71%	99.51%	1,427
Primal Multi-move	64.75%	99.50%	3,024
Primal Soft gain	64.75%	99.50%	1,479
Primal Hyperedge moving	89.11%	97.84%	31,991
Dual FM		91.72%	4,904
Dual Multi-move		91.72%	4,588
Dual Soft gain		90.21%	10,715
Dual Hyperedge moving		87.61%	8,246

- Among the dual methods, Soft gain and Hyperedge moving usually outperform the other two. Dual FM and dual Multi-move yield almost identical results.
- Similarly to our earlier experiments, the number of variables in a small primal cut can be relatively high. This phenomenon is most apparent for Hard combinatorial benchmarks.
- Although Random benchmarks do not exhibit structure, the best dual methods can still find non-trivial cuts in them.
- For Application benchmarks, primal methods work significantly better than dual methods, probably due to the large size of these benchmarks. For Hard combinatorial benchmarks, primal and dual methods yield comparable results, but the winner seems to be the dual Soft gain method. On the Random benchmarks, dual methods are clearly superior.

5.4 Summary of the empirical results

Our experiments show that there is no clear winner among the investigated methods. Fortunately, the results give some indication about which method works well for what kind of formulae. Figure 1 contains the results for all 68 instances of the first set of experiments as well as the 170 used competition instances, except for 4 where none of the investigated partitioning methods found a cut smaller than 100%. For each instance, the best partitioning method is defined as the one yielding the smallest dual cut (for primal methods, this is the number of

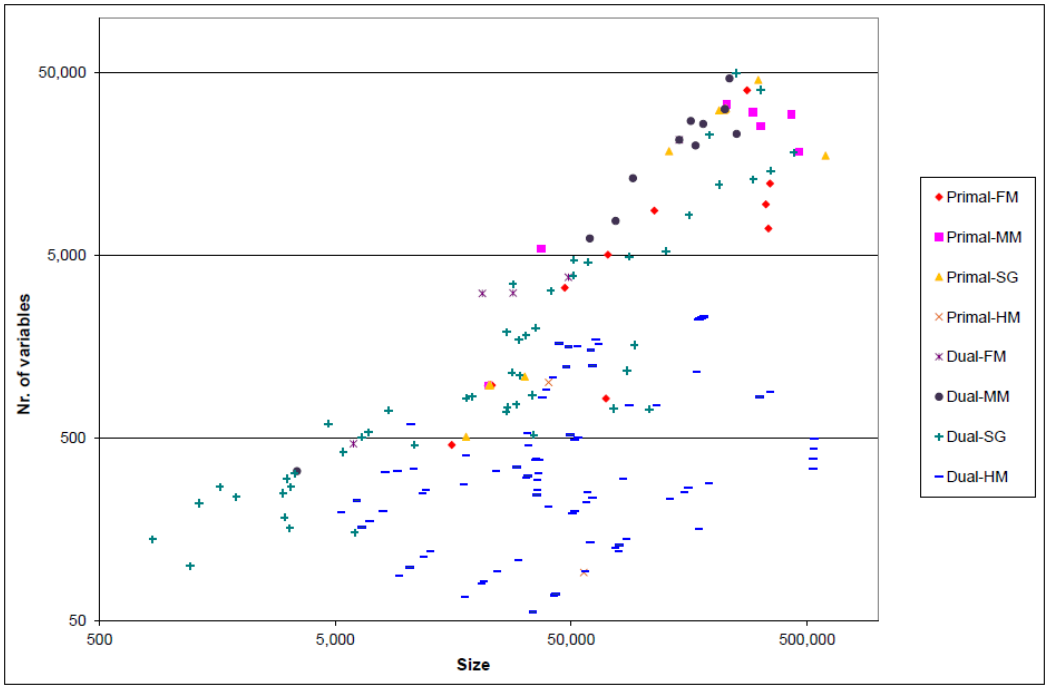

Figure 1: The best partitioning method, as a function of the size of the instance and the number of variables. The abbreviations are: MM: Multi-move, SG: Soft gain, HM: Hyperedge moving

variables that appear in the clauses of the primal cut); in case of ties, the method with lower runtime is the winner. The figure shows the winning partitioning method for each instance, as a function of the size of the instance and the number of variables. Table 13 shows an aggregated view: it contains for each method the number of instances on which it was best.

Table 13: Number of instances for which a given partitioning method was best

	Nr. of wins
Primal FM	14
Primal Multi-move	8
Primal Soft gain	15
Primal Hyperedge moving	2
Dual FM	5
Dual Multi-move	12
Dual Soft gain	78
Dual Hyperedge moving	100

For most instances, the best method is either dual Hyperedge moving or dual Soft gain. Which of the two works better depends on L/n, where L is the size of the formula and n is the number of variables. This is because L/n is the average number of clauses that a variable appears in, which is exactly the average hyperedge size in the dual hypergraph. If L/n is high

(lower-right part of the diagram), then the hyperedges of the dual hypergraph are big; for such hypergraphs, the Hyperedge moving algorithm is indeed the most efficient because it moves complete hyperedges in a single step. On the other hand, if L/n is low (upper-left part of the diagram), then the hyperedges in the dual hypergraph are smaller, so that the more precise moves and gain calculation of the Soft gain algorithm become better. Furthermore, if L/n is extremely low, then the dual Multi-move method becomes even better than dual Soft gain.

There is a single region, in which primal methods clearly outperform the dual methods: when both n and L are high ($n > 5,000$, $L > 200,000$). In this region, the dual methods time out because of the extreme size of the dual hypergraph. Hence, in this region, the primal FM, primal Multi-move and primal Soft gain methods are best.

The auxiliary variables approach is missing from this plot because it cannot be compared directly with the other methods. However, our experience shows that this approach is only useful if both n and L are relatively small.

6 Conclusions and future work

We presented a study of different methods to partition Boolean formulae. We investigated three partitioning models (primal – model generation, primal – auxiliary variables, dual) and four partitioning algorithms (FM, Multi-move, Soft gain, Hyperedge moving). We evaluated them on a wide range of different formulae. The empirical results reinforced our conjecture that none of the methods is a clear winner. For primal partitioning, FM, Multi-move and Soft gain perform similarly well. However, model generation may lead to a large set of variables that need to be assigned a value, even if the cut clause set is small; and auxiliary variables can considerably increase formula size. For dual partitioning, the Soft gain and Hyperedge moving algorithms proved best, but even these methods result sometimes in quite big sets of cut variables. We also found that for six of the eight benchmark families, partitioning could improve the performance of a CDCL solver, but in those six cases, three different partitioning methods were best. Using SAT competition benchmarks, we saw a clear difference between the effectiveness of primal vs. dual methods on the different benchmark categories. We also presented a summary of the strengths and weaknesses of the investigated methods, giving insight to predict which one(s) of the presented methods would work best on instances of given size and number of variables.

We see several opportunities for extending this work in the future. Further algorithmic ideas can be used to improve the presented partitioning methods or to come up with new ones. We also plan to extend our experiments to further benchmarks, with the aim of refining the rules for deciding automatically which partitioning method to use, from a portfolio of available partitioning methods, for a given problem instance. Alternatively, multiple partitioning methods can also be run in parallel. Also, when the formula falls into independent pieces, those pieces can be solved in parallel. Moreover, we will investigate how a state-of-the-art SAT solver can be accelerated by means of partitioning.

Acknowledgements

This work was partially supported by the Hungarian Scientific Research Fund (Grant OTKA 108947) and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162:49–98, 2005.

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

[3] Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo. Generating the uniform random benchmarks for SAT Competition 2013. In 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 of *Department of Computer Science Series of Publications B*, pages 97–98. University of Helsinki, 2013.

[4] Roberto J. Bayardo and Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 157–162, 2000.

[5] Armin Biere and Carsten Sinz. Decomposing SAT problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):201–208, 2006.

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

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

[8] 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.

[9] 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.

[10] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, 1982.

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

[12] 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.

[13] 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.

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

[15] Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Julian Stecklina, and Peter Steinke. A short overview on modern parallel SAT-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011.

[16] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the Year 2000*, pages 283–292. IOS Press, 2000.

[17] 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)*, pages 1167–1172, 2003.

[18] Anil P Kamath, Narendra K Karmarkar, KG Ramakrishnan, and Mauricio GC Resende. A continuous approach to inductive inference. *Mathematical programming*, 57(1-3):215–238, 1992.

[19] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, 1970.

[20] We 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.

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

[22] 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.

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

[24] 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.

[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[26] 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.

[27] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299. Springer, 2007.

[28] Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics*, pages 249–258. Springer, 2008.

[29] Xiaoyu Song, William NN Hung, Alan Mishchenko, Malgorzata Chrzanowska-Jeske, Andrew Kennings, and Alan Coppola. Board-level multiterminal net assignment for the partial cross-bar architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):511–514, 2003.

[30] 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 Int. Conf. on Information Science, Communications and Applications (ISA 2004)*, 2004. Paper 484-403.