DBAI

TECHNICAL REPORT

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Heuristic Methods for Hypertree Decompositions

DBAI-TR-2005-53

Artan Dermaku Tobias Ganzow Georg Gottlob
Benjamin McMahan Nysret Musliu Marko Samer

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2005, (LAST MODIFIED: 2006)

TU
TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2005-53, 2005, (LAST MODIFIED: 2006)

Heuristic Methods for Hypertree Decompositions

Artan Dermaku[1] **Tobias Ganzow**[2] **Georg Gottlob**[3]
Ben McMahan[4] **Nysret Musliu**[5] **Marko Samer**[6]

Abstract. In this paper we propose new algorithms for generating generalized hypertree decompositions. The well known heuristics for generating tree decompositions based on vertex ordering have been extended to produce hypertree decompositions. We investigate the generation of hypertree decompositions based on the tree decompositions of the primal and the dual graph of the hypergraph. Further, we propose a method for generating hypertree decompositions using hypergraph partitioning. We use different algorithms for partitioning hypergraphs. The proposed algorithms are experimentally evaluated in benchmark problems from the literature and industry. Using the proposed algorithms we improve the best existing upper bounds for hypertree width for many problems.

[1]Institut für Informationssysteme (DBAI), Technische Universität Wien, Favoritenstr. 9-11, A-1040 Wien, Austria. E-mail: dermaku@dbai.tuwien.ac.at
[2]Mathematische Grundlagen der Informatik, RWTH Aachen, D-52056 Aachen. E-mail: ganzowinformatik.rwth-aachen.de
[3]Institut für Informationssysteme (DBAI), Technische Universität Wien, Favoritenstr. 9-11, A-1040 Wien, Austria. E-mail: gottlob@dbai.tuwien.ac.at
[4]Department of Computer Science, Rice University, Houston, TX 77005-1892, U.S.A. Email: mcmahanb@rice.edu
[5]Institut für Informationssysteme (DBAI), Technische Universität Wien, Favoritenstr. 9-11, A-1040 Wien, Austria. E-mail: musliu@dbai.tuwien.ac.at
[6]Institut für Informationssysteme (DBAI), Technische Universität Wien, Favoritenstr. 9-11, A-1040 Wien, Austria. E-mail: samer@dbai.tuwien.ac.at

Acknowledgements: This work was supported by the Austrian Science Fund (FWF) project: *Nr. P17222-N04, Complementary Approaches to Constraint Satisfaction*

1 Introduction

Many important problems in artificial intelligence, database systems, and operations research can be formulated as constraint satisfaction problems (CS problems, or CSPs, for short). Such problems include problems in scheduling, planning, configuration, diagnosis, machine vision, spatial and temporal reasoning, truth maintenance, theory of graphs and networks, etc. Although solving a CS problem is known to be NP complete in general, many of the problems that arise in practice have special properties that allow them to be solved efficiently. The question of identifying restrictions to the general problem that are sufficient to ensure tractability is important from both a practical and a theoretical point of view, and has been extensively studied.

A CS problem consists of a finite set of variables each with a finite domain of possible values, and a set of constraints (relations) on the allowed values for a specified subsets of variables. A typical textbook example for a CS problem is the famous *graph 3-colorability* problem, i.e., the problem of deciding whether the vertices of a graph G can be colored by three colors such that vertices joined by an edge receive different colors. This problem can be formulated in a natural way as a CSP: we consider each vertex of G as a variable with domain $Red, Green, Blue$, and for each edge (v, w) of G we take the binary constraint $C_{v,w}$ that just excludes the case where both variables v and w receive the same color, i.e., $C_{v,w}$ =(Red, Green), (Green, Red), (Red, Blue), (Blue, Red), (Green, Blue), (Blue, Green).

Restrictions for obtaining tractable classes of CS problems may either involve the *nature* of the constraints (i.e., which combinations of values are allowed for variables that are mutually constrained) or they may involve the *structure* of the constraints (i.e., which variables may be constrained by which variables). In this paper we concentrate in the second aproach, which investigates classes of CS problems which can be solved efficiently by exploiting their structure. The structure of a CS problem can be modeled by its *constraint hypergraph*, a hypergraph whose vertices are the variables of the problem, and whose hyperedges correspond to the constraints of the problem (more precisely, each constraint gives rise to a hyperedge containing exactly the variables which are in the scope of the constraint).

The evaluation of *boolean queries* on databases, an important and excessively studied task in database theory, is known to be equivalent to finding solutions for a CS problem. This equivalence allows us to apply techniques and results obtained in database theory directly to solving CS problems, and vice versa. A prominent tractable class of CSPs, the *acyclic* CSPs, originates from database theory. A CSP is acyclic if its constraint hypergraph is acyclic (there have been several definitions of hypergraph acyclicity considered in literature; fortunately, in our context the most general concept of acyclicity ("$alpha$ acyclicity") suffices, see [9, 16]). If the hypergraph associated to a CSP is acyclic, then the problem can be solved efficiently by Yannakakis' classical algorithm [33]. Yannakakis' algorithm, formulated in terms of conjunctive query evaluation, processes the "join tree" corresponding to an acyclic query in a bottom-up fashion. This algorithm is also highly parallelizable [16]. Yannakakis' algorithm was later used to compute the solution of acyclic CS problems. Incidentally, for graph 3-colorability, as considered above, the constraint hypergraph is nothing but the given graph G itself; if G is a tree, then the coloring problem is trivial.

The favorable results about acyclic CS problems extend to classes of "nearly acyclic" CS problems. Several decomposition methods have been suggested in the literature to transform an arbitrary CSP into an acyclic one, making the vague notion of "nearly acyclic" precise. The most prominent methods include: *tree clustering* [8], *hinge decompositions* [19, 20], *cycle cutset* and *cycle hypercutset* [7, 16], *hinge-tree clustering* [16], *bounded query-width* [4], and *bounded hypertree-width* [17]. The latter method has been shown to be the most general one [15].

The parameter *hypertree width* corresponds to a new decomposition method called *hypertree decomposition*. The hypertree width of a CSP is the width of an optimal hypertree decomposition of its constraint hypergraph. Acyclic CSPs have hypertree width 1. Bounded hypertree width yields the largest class of tractable queries, compared with all other acyclicity-based classes; Moreover, it could be shown that for each k, the class of CSPs with query width k is properly contained in the class of CSPs whose hypertree width is bounded by k [17]. This is remarkable, since bounded query width allowed the hitherto largest class of tractable CS problems, but in contrast to the NP hardness of finding query decompositions of fixed width (see [17]), hypertree decompositions of fixed width can be found in polynomial time. Thus the notion of bounded hypertree width not only shares the desirable properties of bounded query width, it also does not share the bad properties of the latter, and, in addition is a more general concept. A further in-depth comparison of hypertree width, clique-width and other decomposition methods in the more general context of model checking, can be found in Gottlob and Pichler [18].

For each constant k, it is possible to check in polynomial time whether a hypergraph is of hypertree-width k, and, in the positive case, to produce a hypertree decomposition of width k of the given hypergraph. By means of the hypertree decomposition, the corresponding CS problem can then be solved in polynomial time. (Furthermore, by results of [17] the tasks of finding a hypertree decomposition of width k (if any) and of solving the corresponding CSP problem belong to the complexity class $LOGCFL$, a complexity class of *highly parallelizable* problems which is located very low in the polynomial hierarchy, $LOGCFL \subseteq AC^1$.) Therefore, the efficient generation of hypertree decompositions of small width is of high relevance for constraint solving. Preliminary results on practical applications of hypertree decomposition for constraint solving and for diagnosis algorithms are reported in [13] and [30], respectively.

Formal definition of hypertree and tree decompostions is given below.

Definition 1 (Gottlob, Leone, and Scarcello [17]) Let $H = (V(H), E(H))$ be a hypergraph, consisting of a nonempty set $V(H)$ of *vertices*, and a set $E(H)$ of subsets of $V(H)$, the *hyperedges* of H. A *hypertree decomposition* of a hypergraph H is a hypertree $< T, \chi, \lambda >$ for H which satisfies all the following conditions:

1. for each hyperedge $h \in E(H)$, there exists $p \in vertices(T)$ such that $vertices(h) \subseteq \chi_p$;
2. for each vertex $Y \in V(H)$, the set $\{p \in vertices(T) \mid Y \in \chi_p\}$ induces a (connected) subtree of T;
3. for each vertex $p \in vertices(T), \chi_p \subseteq var(\lambda_p)$;
4. for each vertex $p \in vertices(T)$, $var(\lambda_p) \cap \chi_{T_p} \subseteq \chi_p$

The *width* of the hypertree decomposition $< T, \chi, \lambda >$ is $max_{p \in vertices(T)}|\lambda_p|$. The *hypertree width*, $hw(H)$, of H is the minimum width over all its hypertree decompositions.

If the fourth condition in definiton of hypertree decompostions is ignored, the corrosponding decomposition is called generalized hypertree decompostion. Note that the first three condions of hypertree decompositons are sufficient to solve the the corresponding CS problem in polynomial time. The fourth condition was added to aid the proof that, for a fixed k, determining if a hypergraph H has hypertree width k can be solved in polynomial time. In this paper we investigate the genaration of generalized hypertree decompositions, i.e. the proposed methods in this paper guarantee to produce the hypertree decompositions which fulfills first three conditions of hypertree decompositions.

In this paper we also use tree decompostions to generate hypertree decompositions. The concept of tree decompositions was introduced by Robertson and Seymour: [28].

Definition 2 (see [28], [2]) Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair (T, χ), where $T = (I, F)$ is a tree with node set I and edge set F, and $\chi = \{\chi_i : i \in I\}$ is a family of subsets of V, one for each node of T, such that

1. $\bigcup_{i \in I} \chi_i = V$,
2. for every edge $(v, w) \in E$, there is an $i \in I$ with $v \in \chi_i$ and $w \in \chi_i$, and
3. for all $i, j, k \in I$, if j is on the path from i to k in T, then $\chi_i \cap \chi_k \subseteq \chi_j$.

The *width* of a tree decomposition is $\max_{i \in I}|\chi_i| - 1$. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all possible tree decompositions of G.

1.0.1 Current Algorithms for Hypertree Decompositions

The exact algorithm OPT-K-DECOMP for the generation of optimal hypertree decompositions, developed by Gottlob, et al. [14], has been implemented in Smalltalk. For details about the implementation of this algorithm, see [21]. Additionally, this algorithm has also been implemented in C++ by the research group at the University of Calabria, Italy[1]. Both implementations are used successfully for the generation of hypertree decompositions of small instances of CS problems. However, for larger and important practical cases, the exact algorithm is not practical and runs out of time and space. To overcome the limitations of the exact algorithm heuristic methods were proposed. In [26] a heuristic method is proposed based on the vertex connectivity of the given hypergraph (in terms of its primal and incidence graphs). The application of branch decomposition heuristics for hypertree decomposition was investigated in [29]. These heuristic methods were used to find hypertree decompositions of small width for problem instances where the exact algorithm OPT-K-DECOMP did not yield results within a reasonable amount of time. However, the preliminary heuristics were still not useful to give good results for larger problem instances;

[1] see http://si.deis.unical.it/∼frank/Hypertrees/html/body.htm

in particular, we have identified several important practical cases in which the current heuristics cannot give satisfying solutions.

To obtain better results for hypertree decompostions for different range of problems, in this paper we propose new heuristic algorithms for generation of hypertree decompositions. In particularly we investigate the application of algorithms which were used very succesfully for generation of tree decompostions based on vertex ordering. Further, we investigate the use of hypergraph partitioning algorithms to obtain hypertree decompositions. This paper is organized as follows. In Section 2 we propose an algorithm for generation of generalized hypertree decompositions based on tree decomposition of primal graph (obtained from the hypergraph). Section 3 investigates the use of tree decompostion of dual graph for generalized hypertree decompostions. Generation of hypertree decompositions through hypergraph partitioning is proposed in Section 4. Further, in Section 5 the proposed algorithms are evaluated in benchmark examples, and Section 6 gives the conclusion remarks.

2 Bucket Elimination for Hypertree Decomposition

Bucket elimination (BE) is used in Constraint Satisfaction. The method uses the topological structure of the problem to help find a solution efficiently. In particular, the method approximates the induced width of its primal graph, which has shown to be identical to the treewidth of its primal graph [11]. The method has the property that given an optimal variable order, BE will produce a tree decomposition of optimal width [6, 5]. The algorithm works as follows: Assume we are given an order $x_1, \ldots, x_n$ of the variables of a CS problem (or vertices of hypergraph which represents the CS problem). BE starts by creating n "buckets", one for each variable x_i. For an constraint $r_i(x_{i_1}, \ldots, x_{i_k})$ of the problem, we place the relation r_i with variables $x_{i_1}, \ldots, x_{i_k}$ in bucket max$\{i_1, \ldots, i_k\}$. We now iterate on i from n to 1, eliminating one bucket at a time. In iteration i, we find in bucket i several relations, where x_i is a variable in all these relations. We compute their join, and project out x_i. Let the result be r'_i. If r'_i is empty, then the result of the CSP is empty. Otherwise, let j be the largest index smaller than i such that x_j is a variable of r'_i; we move r'_i to bucket j. The answer to the original CSP is 'yes' if none of the joins returns an empty result.

Note that for the given CSP the corresponding hypergraph's hyperedges represent the scope of constraints of the CSP, and the vertices of the hypergraph represents the variables of the CSP problem.

This method can easily be extended to create a tree decomposition $< T = (I, F), \chi >$ of CSP. First, a node $i \in I$ is created for each bucket i in the algorithm. Then each node's label χ_i contains the variables that appear in the corresponding bucket i. Edges $(i, j) \in F$ are created when, in the algorithm, the result of bucket i is placed in bucket j.

2.1 Variable Orders

Bucket elimination requires an underlying variable order. Since choosing an optimal order for BE is NP-hard [1], we choose the order heuristically, using the join graph (primal graph) of the hypergraph H which represents the CSP. The join graph $G_H = (V_H, E_H)$ of a hypergraph H contains a vertex for every variable in the relations of CSP and an edge between two vertices iff there is a relation in CSP that contains both variables.

An order that is often used in constraint satisfaction [5] is the Maximum-Cardinality Search (MCS) order of [31]. The order is computed iteratively. At each iteration MCS picks a vertex that has the highest connectivity with the vertices already chosen, breaking ties arbitrarily. In other words, MCS picks the vertex with the greatest number of neighbors in the set of vertices already picked.

Other vertex-ordering heuristics have been explored in the context of treewidth approximation and constraint satisfaction [5, 27]. One variable-ordering heuristic method is based on using lexicographic breadth-first search to triangulate a graph. Two variants were developed [27]. Two greedy heuristics can also be found in the literature. The first one, called *min-induced-width* ordering, computes the order iteratively. At each iteration it adds the vertex v with the smallest degree. Next it adds to the graph v's induced edges, i.e. with edges that connect the neighbors of v, and deletes v from the graph. A variation of the min-induced-width ordering is the *min-fill* heuristic. At each iteration we pick the vertex that has the smallest *fill set*, which is the edge set needed to be filled to make the parent set fully connected. The *parent set* of a node v are all the nodes adjacent to v that precede it in the ordering. Once this vertex has been picked we update the graph by added its induced edges and deleting the vertex as in the min-induced-width ordering. In [5], min-fill is said to have been shown empirically to produce orders with lower width than min-induced-width.

We use in this paper three vertex-ordering heuristics: *MCS*, *min-fill* hueristic, and *min-induced-width* heuristic.

2.2 Two simple extentions

Hypertree decompositions, in satisfying their own properties, must also satisfy the properties of tree decompositions. In particular, the first property of hypertree decompositions satisfy the first two properties of tree decompositions and the second property of hypertree decompositions is the same as the third property of tree decompositions. The intuition behind the extensions is then that the edge label λ is made up of atoms needed to "cover" the variables found in χ. The extension also makes the greedy assumption that a low treewidth will allow for a lower hypertree width.

The first approach we attempted , called hyperBE, basically builds the hypertree decomposition as the algorithm proceeds. Given an order $x_1, \ldots, x_n$ of the variables of a CSP. It starts by creating n "buckets", one for each variable x_i. For an constraint (relation) $r_i(x_{i_1}, \ldots, x_{i_k})$ of the CSP, we place the relation r_i with variables $x_{i_1}, \ldots, x_{i_k}$ in bucket $\max\{i_1, \ldots, i_k\}$. We now iterate on i from n to 1, eliminating one bucket at a time. In iteration i, we find in bucket i several relations, where x_i is a variable in all these relations. We compute their join, and project out x_i. Let the result be r'_i. If r'_i is empty, then the result of the CSP is empty. Otherwise, let j be the largest index smaller than i such that x_j is a variable of r'_i; we move r'_i to bucket j. Now, this is where

the extention appears. We add relations from the CSP to bucket j to cover any new variables that might have appeared from placing r'_i in the bucket. We do this greedily, just picking atoms from the previous bucket that cover any uncovered variable.

A second approach, called coverBE, first creates a tree decomposition using BE, then creates the λ labels for the nodes of the tree greedily by attempting to cover the variables in the χ label. So, for each node of the tree, the algorithm covers its variables by iteratively picking the relation (hyperedge of hypergraph which represents the scope of relation) with the highest cost function (defined later) until all the variables are covered. As a cost function, we have found two that worked well. The first one picks the hyperedge that covers the most number of uncovered variables. Ties are broken by taking the hyperedge that on average contains the least occuring variables of all the hyperedges in hypergraph which reperesents the CSP. The other cost function does not include a tie breaker. A weight is assigned to each variable as 0 if the variable is covered and as $(1 - \frac{occurance}{|Constraints(CSP)|})$ if it is not covered. $Occurance$ is the number of constraints in CSP the variable appears in. The algorithm then picks the atom that has the highest weighted sum until all the variables are covered.

There are some things to be noted with these extensions. The first and most important is that both of them use early projection and thus violate the fourth condition of hypertree decompositions. This, though, is not as bad as it sounds because the fourth condition was added to aid the proof that, for a fixed k, determining if a hypergraph which represents CSP has hypertree width k can be solved in polynomial time. Since we are approximating hypertree decompositions, this restriction isn't necessary.

Another major item to note is that the first extension creates a *complete decomposition*, meaning that for every constraint of the CSP, $A \in constraints(CSP)$, there is some node $p \in vertices(T)$ such that $var(A) \subseteq \chi(p)$ and $A \in \lambda(p)$. But for the second extension, coverBE, it does not necessarily produce a complete decomposition as not every hyperedge (relation) may be used. This is not a problem, since a complete decomposition can easily be created by adding nodes to the hypertree decomposition containing the missing atoms and connecting them to the nodes in the tree where the χ label is a superset of the variables in the atom. In general, a hypertree decomposition can be transformed to a complete decomposition in logspace [17].

3 Dual Bucket Elimination

Another approach we investigated is called *dual bucket elimination*. Dual bucket elimination is very similar to bucket elimination, but instead of constructing a tree decomposition of the original hypergraph in the first step, we construct a tree decomposition of the *dual* hypergraph. The dual hypergraph of a hypergraph, as exemplified in Fig. 1, is simply obtained by swapping the roles of hyperedges and vertices. It is easy to see that there is always a one-to-one mapping between a hypergraph and its dual hypergraph, i.e., we do not loose any information by this transformation.

Our intuition for using the dual hypergraph instead of the original hypergraph is that bucket elimination tries to minimize the size of the labeling sets, which are the χ-labels of the hypertree in the case of using the original hypergraph. However, the width of a hypertree decomposition

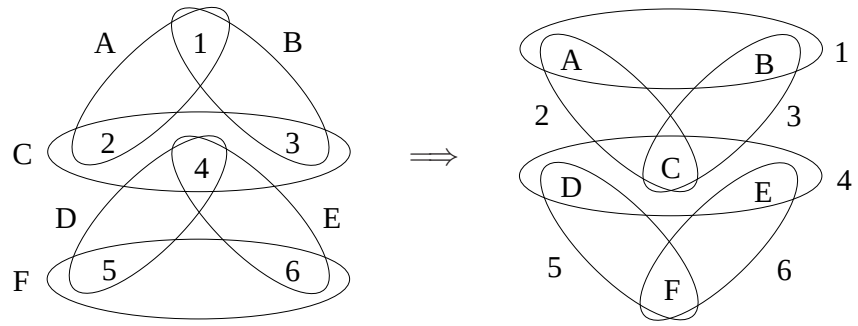

Figure 1: A hypergraph and its dual hypergraph

is determined by the size of the λ-labels and not of the χ-labels. So our aim is to apply bucket elimination in order to minimize the λ-labels, which is exactly what is done when applying bucket elimination to the dual hypergraph. Hence, our procedure is the following: (i) build the dual hypergraph, (ii) apply bucket elimination to construct a tree decomposition, (iii) interpret the labeling sets as λ-labels of a hypertree, and (iv) set the χ-labels appropriately in a straight-forward way. The resulting hypertree is then a hypertree decomposition of the original hypergraph.

The attentive reader may have noticed that there are two problems with this approach: First, the hypertree-width is at least the cardinality of the largest edge in the dual hypergraph which is equal to the maximum number of hyperedges having a common vertex in the original hypergraph. Second, the λ-labels satisfy the connectedness condition of a tree decomposition by construction. Hence, the hypertree-width may be larger than necessary. We can overcome the first problem by slightly modifying the bucket elimination algorithm such that the second condition of a tree decomposition is violated. To overcome the second problem, we reset the λ-labels by set cover heuristics after the χ-labels have been set. Although the results of dual bucket elimination are not outperforming other hypertree decomposition heuristics, for some examples we obtain in this way hypertree decompositions of smallest width.

4 Hypertree Decomposition through Hypergraph Partitioning

In this section we consider the use of hypergraph partitioning algorithms for generating hypertree decompositions. Let $H(V, E)$ be a hypergraph where V is the set of vertices, and E is the set of hyperedges (each hyperedge is a subset of the vertex set V). Vertices and hyperedges can have different weights. In Hypergraph Partitioning the aim is to find partitions of set V in two (or k) disjoint subsets such that the number of vertices in each set V_i is bounded, and some objective defined over hyperedges is optimized. Most commonly used objective is to minimize the sum of the weights of hyperedges connecting two or more subsets.

An example of partitioning a hypergraph in two parts is given in Figure 2. The hypergraph contains 10 hyperedges and 14 vertices. The cut divides the hypergraph in two parts. The first part contains 8 vertices, whereas the second part contains 6 vertices. These two partitions are connected

by two hyperedges $h1$ and $h7$. If all hyperedges in the hypergraph weighed 1, the cost of the cut (weighted sum of the cut) will be 2.

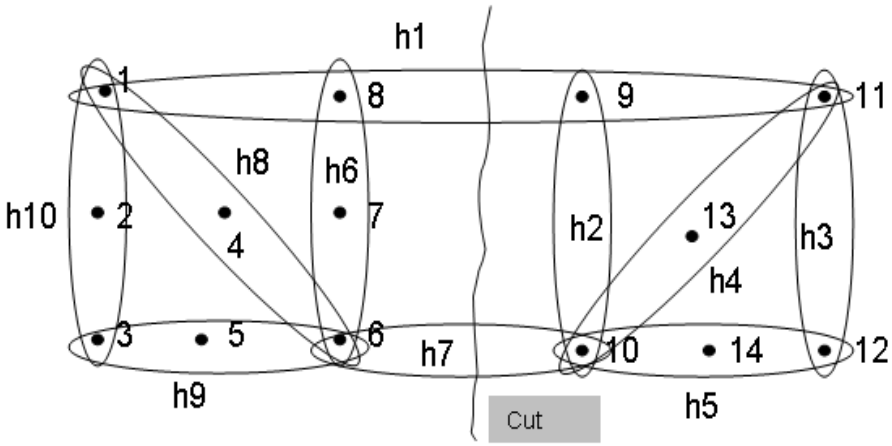

Figure 2: Example of partitioning a hypergraph in two parts

Hypergraph partitioning constrained on the number of vertices in each partition is NP-Complete problem. Thus, different heuristics methods have been used in the literature to produce a good hypergraph partitioning for large hypergraphs. In this paper we experiment with two heuristics which has been used very succesfully in the literature, and additionally propose and apply a new method for hypergraph partitioning. In this section, we first show how the hypertree decompostions can be constructed using hypergraph partitioning and then we describe in detail the heuristics used for hypergraph partitioning.

The basic idea of using hypergraph partitioning for hypertree decomposition is due to Korimort [26]: The given hypergraph is partitioned into subhypergraphs, the subhypergraphs are partitioned into subsubhypergraphs, etc. For each such partitioning step, a new hypertree node is constructed which is labeled with the corresponding set of vertices respectively hyperedges used to separate the hypergraph. If vertices are chosen as separators, we have the χ-labels given and use set cover heuristics to compute the λ-labels. Otherwise, if hyperedges are chosen as separators, we have the λ-labels given and compute the χ-labels in a straight forward way as described in [17]. Note, however, that it is not enough to construct a hypertree by connecting the hypertree nodes according to the partitioning tree. The problem is that the partitionings of a hypergraph and its subhypergraphs are not independent of each other since the connectedness condition has to be satisfied by the resulting hypertree decomposition. Therefore, Korimort [26] suggested to add a *special hyperedge* to each subhypergraph which contains the vertices in the intersection between the hypergraph and its subhypergraph, i.e., the vertices which must occur in the χ-labels of the child hypertree node in order to satisfy the connectedness condition. By the hypertree conditions, we know that for each hyperedge there must be a hypertree node which contains all vertices in the

hyperedge in its χ-labels. Thus, when constructing a hypertree top-down, we know that there must be a hypertree node in the subtree which contains all vertices in the corresponding special hyperedge in its χ-labels. Hence, we choose this hypertree node as child of the actual hypertree node. It is then easy to check that a hypertree decompostion obtained in this way satisfies all hypertree conditions. A detailed description of this procedure can be found in [26].

4.1 Partitioning with Fiduccia-Mattheyses algorithm

The hypergraph partitioning algorithm proposed by Fiduccia and Mattheyses in [10] is based on an iterative refinement heuristic. In a first step, the hypergraph is arbitrarily partitioned into two parts, followed by a sequence of passes during which the partitioning is optimized by successively moving vertices to the opposite partition. The selection criteria for choosing the next move are based on the so-called *gain* which is associated to every vertex and a balancing constraint that prevents the application of moves which would lead to an imbalanced partitioning. The gain is a measure for the impact of the move on the size of the hyperedge-cut; positive values indicate that the size of the cut decreases, i.e. that the solution is improved. Furthermore, there is a locking mechanism to prevent situtations where sets of vertices would be moved back and forth between the partitions again and again. Towards this end, at the beginning of each pass all vertices are unlocked (i.e., free to move), and once a vertex is moved to the opposite partition it is locked. The next move is determined by choosing one of the vertices with the highest gain among the remaining unlocked vertices whose movement does not violate the balancing constraint. A pass is finished after all vertices have been moved, such that the partitioning corresponds to the initial partitioning except that the partitions are swapped. Note, that since the "best" next move does not necessarily have a positive gain, the solution might get worse during a pass allowing the algorithm the chance to climb out of local minima. However, the best solution seen so far is memorized and, after the pass is finished, this solution is taken as the initial solution for the next pass. The whole algorithm terminates if the initial solution could not be improved during a pass. Fiduccia's and Mattheyses' main contribution was to show that these gains can be calculated efficiently at the beginning of the pass, and, even more important, that the gains can also be updated efficiently after a move has been made.

4.2 Implementation

The implementation of the Fiduccia-Mattheyses algorithm (FM) is oriented at the architecture proposed in [3] that identifies and defines several components needed for the implementation of an arbitrary move-based partitioning heuristic. The main advantage of this "decentralized" approach is the possibility to easily replace a component by a more efficient implementation or by a modified version without having to change anything in the other parts.

There are several possibilities of how to handle the special hyperedges which are introduced after each partitioning step to ensure the connectedness condition of the hypertree decomposition. As these hyperedges are not contained in the original hypergraph, they have to be replaced in the final decomposition by possibly more than one hyperedge of the original graph. So the question is

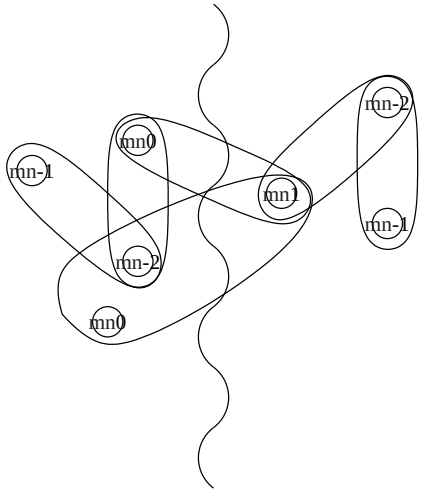

Figure 3: Gains

how to evaluate the cost of a cut that contains such special hyperedges.

To determine which evaluation function yields the best results, we implemented four different variants of the FM algorithm. First, the original algorithm using efficient gain updates that does not differentiate between “normal” and special hyperedges, and second, a minor variant that handles hyperedges with associated weights. The first variant can be seen as a special case of the latter where the weight of each hyperedge is set to one. During the construction of the hypertree, for each arising special hyperedge it is determined how many edges of the original hypergraph are needed to cover all of its nodes, and this value is taken as the weight of the special hyperedge. However, the apparent problem with this approach is that the number of hyperedges needed to cover the vertices of the special hyperedge is not always an accurate measure for the contribution of the hyperedge to the separator size. This problem appears in those cases where a hyperedge needed for the covering is also part of the cut and thus would be counted twice, or in case that there are two (or even more) special hyperedges in the cut whose sets of covering edges have a non-empty intersection. Thus, by simpy adding all weights of the cut hyperedges, a solution might be valuated worse than it actually is.

This problem is addressed in a third variant that evaluates the cost of the set of hyperedges being cut more accurately by determining the number of hyperedges of the original hypergraph that are needed to cover the vertices of all hyperedges in the cut. Note that this modification eliminates the possibility of efficient gain updates since the amount that a single hyperedge contributes to the total cost of a cut highly depends on the set of hyperedges being cut.

The fourth variant completely avoids the problem by moving all nodes contained in a special hyperedge at once. Thus, there is never a special hyperedge contained in the separator, and hence the valuation of the separator is straightforward. According to Korimort [26], considering separators containing special hyperedges should make the problem of finding a good decomposition harder and should not lead to better results.

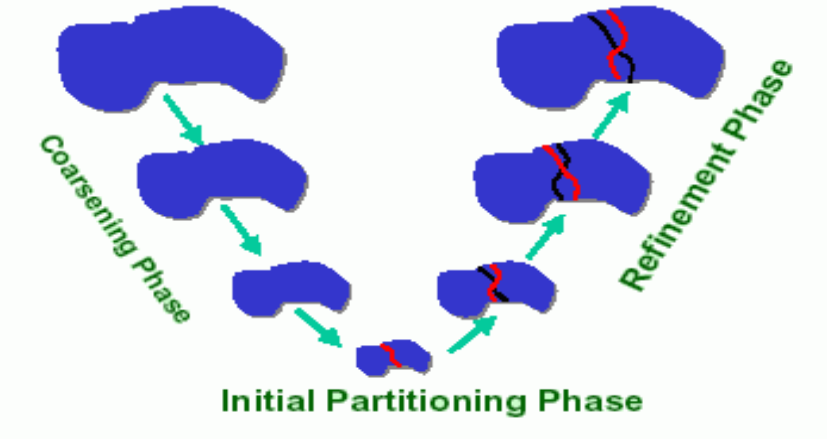

Figure 4: Multilevel partitioning algorithms [22]

4.3 Partitioning through HMETIS

The next approach which we use in order to achieve a good hypergraph partitioning is based on hMETIS algorithms. hMETIS is a software package for partitioning hypergraphs, developed at the University of Michigan. According to the literature, hMETIS is one of the best available packages for hypergraph partitioning [22, 23, 25].

We will give a brief description of the hMETIS algorithms. More information about hMETIS can be found in [22, 23, 25, 24]. In general the algorithm, as illustrated in Figure 2 [22], comprise of three phases.

In the *coarsening* phase the group of vertices of hypergraph will be merged together in order to create the single vertices and smaller hypergraph. In this way the size of large hyperedges will be reduced, and it is very helpful because of the fact that FM algorithm is better than other algorithms when refining smaller hyperedges [23]. There are three possibilities to merge the vertices during the coarsening phase: the finding of a maximal set of vertices which have the common hyperedges (edge coarsening), the merging of vertices within the same hyperedge (hyperedge coarsening), and finally the modified hyperedge coarsening which also merges the vertices within hyperedges that have not yet been contracted [23].

After the coarser hypergraph is created, the next phase called the *initial partitioning* phase computes a bisection of those hypergraphs tending a small cut and a specified balanced constraint. The coarser hypergraph has a small number of verticies, usually less than 200 vertices [23], therefore the partitioning time tends to be small. In order to compute the initial partitioning hMETIS uses two different algorithms followed by the Fiduccia-Mattheyses (FM) refinement algorithm [23]. Because the algorithms are randomised, different runs result in different solutions, and the best initial partitioning will be selected for the next phase.

During the *uncoarsening* phase the partitioning will be successively projected to the next level finer hypergraph and a partitioning refinement algorithm will be used to reduce the cut-set in order to improve the quality of partitioning. hMETIS implements a variety of algorithms that are based

on the FM algorithm which repeatedly moves vertices between partitions in order to improve the cut [23, 25].

The hMETIS package offers a stand-alone library which provides the HMETIS_PartRecursive() and HMETIS_PartKway() routines. HMETIS_PartRecursive() routine computes a *k-way* partitioning and is based on recursive partitioning of hypergraph in two partitions (multilevel recursive bisection) [23, 25]. HMETIS_PartKway() routine also computes *k-way* partitioning and is based on recursive partitioning of hypergraph in more than two partitions (multilevel *k-way* partitioning) [25]. We use both routines in order to achieve appropriate partitions that lead to a hypertree decomposition of small width. hMETIS package offers the possibility to change different parameters which have an impact on the quality of partitioning. Therefore we make a series of tests with parameters of different values, and we come to the conclusion that the parameters which mostly impact the quality are the number of desired partitions *nparts*, and the imbalance factor between partitions *ubfactor*. For a complete description of parameters see [22]. The test results show that for *nparts* less than 3 the hypertree decomposition was not necessarily better, and usually higher *ubfactors* lead to smaller hypertree-widths.

4.4 Partitioning with the algorithm that includes tabu search mechanism

In this section we present a new hypergraph partitioning algorithm based in the ideas of tabu search. Tabu search [12] is a powerful modern meta-heuristic technique, which has been used successfully for many practical problems. The basic idea of tabu search is to avoid cycles (visiting the same solution) during the search by using the tabu list. In the tabu list specific information about the search history for a fixed number of past iterations are stored. The acceptance of the solutions for the next iterations in this technique depends not only on its quality, but also on the information about the history of the search. In the tabu list one stores, for instance, the moves or inverse moves that have been used during a specific number of past iterations. The stored moves are made tabu for several iterations. A solution is classified as a tabu solution if it is generated from a move that is in the tabu list. In this technique, a complete neighborhood (with defined moves) of the current solution is generated during each iteration. In the basic variant of TS the best solution (not tabu) from the neighborhood is accepted for the next generation. However, it is also possible to accept the tabu solution if it fulfills some conditions, which are determined by the so called aspiration criteria (i.e., the solution is tabu but has the best objective function value so far).

In this paper we propose a simple iterative improvement algorithm which applies the ideas of tabu search for hypergraph partitioning. In the iterative improvement phase, which includes moving of vertices from one partition to another partition, the information about the moves of vertices between partitions are stored in the memory (tabu list). This information about the history of moves is then used in the process of selecting the solution for the next iteration. For example, if the solution accepted for the next iteration is obtained by moving of vertex 3 from first partition to the second partition, then vertex 3 will be added in the tabu list. The vertices added in the tabu list are kept in the list only for a determined number of iterations. When selecting a solution from the neighborhood for the next iteration, the solutions obtained by moving vertices that are in the tabu list are not taken into consideration for selection. An exception is made if the solution has the best

objective function value so far (aspiration criteria [12]).

Note that tabu search has been used for hypergraph partitioning previously in the literature. Our algorithm includes several changes compared to basic tabu search algorithm for hypergraph partitioning. First, to reduce the size of the neighborhood in our algorithm only part of the neighborhood is generated during each iteration. The neighborhood is obtained only by moving vertices which are contained in the hyperedges that are in the cut of the current solution. This heuristic is similar with min-conflicts heuristic, as we take into considerations only the vertices which appear in cut (seperator) hyperedges. Additionally, we include in the algorithm some randomness during the search. Not all vertices of seperators are moved during each iteration, but with some probability in some iterations the vertices of only one seperator (which is selected randomly) are moved to create the neighborhood of the solution. The pseudo code of the procedure which includes the tabu mechanism and generates the restricted neighborhood is given in Algorithm 1.

Algorithm 1 Partitioning with algorithm which inlcudes tabu meachnism and restricted neighborhood

Generate initial solution

Initialize tabu list

while termination-condition not true **do**

With probability p:
pick randomly one of seperators and generate the whole neighborhood of current solutions by moving nodes of selected seperator

With probability $1 - p$: Generate the whole neighborhood of current solution by moving nodes of all seperators

Evaluate neighborhood solutions

Select the solution for the next iteration based on selection criteria which includes tabu mechanism

Update tabu list

end while

The algorithms starts with very simple initial solution. In the initial solution one partition contains only one node and the second partition the rest of the nodes. The neighborhood of the current solution is generated by moving nodes from the partition in which they are located to the other partition. With some probability p only the nodes of one seperator, which is selected randomly, are moved, whereas with some probability $1 - p$ all nodes which are vertices of seperators will be moved. Clearly, the size of the neighborhood in the case when only one seperator is selected

is much smaller compared to case when all seperators are considered. After the generation of the neighborhood the solutions are evaluated according to the fitness function. The fitness function is the sum of weights of all seperators (hyperedges that connect two partitions). We experimented with different weights of seperators. In the first variant all seperators have weight 1 whether they are special hyperedges or not. In the second variant the real weight of seperators is used, and in the third variant the maximal weight of 2 is set to seperators which have weight larger than 1 (as a consequence of being special hyperedges). The following criteria is applied to determine the solution that will be accepted for the next iteration. The best solution from the neighborhood, if it is not tabu, becomes the current solution in the next iteration. If the best solution from the neighborhood is tabu, then the aspiration criteria is applied. For the aspiration criterion, we use a standard version [12] according to which the tabu status of a move is ignored if the move has a cost better than the current best solution. For finding the most appropriate tabu length for tabu search approach we experimented with different lengths of tabu list and different probability p. The length of the tabu list was selected to be dependent on the size of the problem (number of nodes in the hypergraph). We experimented with these lengths of tabu list: $\frac{|V|}{2}$, $\frac{|V|}{3}$, $\frac{|V|}{5}$, $\frac{|V|}{10}$.

4.5 Combination of Partitioning Algorithms with Bucket Elimination

We additionally experimented with the combination of the hypergraph partitioning algorithms with the bucket elimination algorithm. The algorithm first applies both bucket elimination and hypergraph partitioning algorithm to find the upper bound for the hypertree width. HMETIS package is used for hypergraph partitioning. In HMETIS different combination of parameters for balancing and number of partitions are used. The parameters which produced the best results are selected for further runs of the partitioning algorithm. Using the obtained upper bound for the hypertree decompositions the algorithm that combines hypergraph partitioning and bucket elimination runs as follows: It applies recursively the hypergraph partitioning algorithm in the given hypergraph. The partitioning of a particular subgraph SHx is stopped if the separator obtained by the partitioning of that subgraph is larger than the upper bound for the hypertree width or if the hypertree width of the subgraph obtained by applying bucket elimination in that subgraph is smaller or equal to the size of the separator produced by hypergraph partitioning. Although some results can be improved by this algorithm, the disadvantage of this algorithm is the time performance, because the bucket elimination algorithm is executed in every subgraph obtained from the partitoning algorithm.

5 Evaluation of the heuristics

In this section we report the computational results obtained with the current implementation of the methods described in this paper. The results for problems from CSP hypergraph library [32] are given. This collection of problems contains hypergraph representation of of several classes of CSP instances. These instances include industrial examples from DaimlerChrysler, NASA, and the ISCAS circuits as well as synthetically generated examples like Grids and Cliques. The Library contains problems of different size. For detailed description of these instances the reader

is refered to [32]. These instances can be downloaded from DBAI Hypertree Project web site [2]. Additionally, the executables of the current implementation of the algorithms described in this paper can be downloaded from this web site . All experiments for the methods presented in this paper were perfomed in a machine with a Intel Xenon (2x) processor, 2.2Mhz, 2GB memory. For each problem 5 independent runs with each algorithm are executed. The maximal time for each run is set to be 1 hour.

5.1 Comparison of algorithms based on vertex ordering

Tables 1, 2, 3 present the results obtained by applying Bucket Elimination (BE) and Dual Bucket Elimination (DBE) algorithms in 112 examples from DaimlerChrysler, NASA, Grids, Cliques, and ISCAS circuits. The first column represents the name of instance and its characteristics (number of hyperedges and variables). The second and third column show the results obtained by OPT-K-DECOMP algorithm. Column W represents the width of the generated hypertree and column T the time in which the hypertree decomposition is generated. Furthermore, the fourth and fifth columns represent the results obtained by algorithms proposed in [26] (these results were obtained using a Intel Pentum III, 1 GHZ, 25MB RAM). The last four columns represent the results obtained by the Bucket Elimination (BE) and the Dual Bucket Elmination (DBE) algorithms. For BE and DBE the best width found over 5 runs and the everage time of 5 runs is presented. BE algorithm applies the coverBE approach (see Section 2, according to our experiments this approach gives better results than hyperBE).

The exact algorithm OPT-K-DECOMP can be used only for the small examples and for larger and important practical cases, the exact algorithm is not practical and runs out of time and space. The algorithms developed in [26] are tested using the DaimlerChrysler instances and could be used for larger problem instances than OPT-K-DECOMP algorithm. For these instances the algorithm gives very good results with respect to the hypertree width, however the time performance is much worse compared to BE and DBE. In [26] results are not given for other problems, and for larger instances this algorithm is very time consuming and, for example, for the NASA problem the algorithm proposed in [26] could not give satisfiable results for the width of hypertree decomposition. Comparing BE and DBE, the tables 1, 2, and 3 show that BE outperforms DBE with regards to the width of the hypertree generated. BE gives better results for 59 problems, whereas DBE elimination gives better results than BE for 15 problems. The total sum of widths for all examples for BE is 2881, whereas for DBE 3386. Overall, the results show that BE and DBE can give very good upper bounds for the width of hypertree decompositions for different sized problems in a reasonable amount of time.

5.2 Comparison of partitioning algorithms

In this section we compared results for three partitioning techniques described in section 4. The results obtained based on Fiduccia-Mattheyses algorithm (FM), algorithm which includes tabu search (TS) and HMETIS (HM) are presented.

[2] http://www.dbai.tuwien.ac.at/proj/hypertree/downloads/

We tested several variants for assigning weights to special hyperedges. The first possibility is to not consider weights at all which is the same as assigning a weight of 1 to all hyperedges. Second, we set the weight of a special hyperedge equal to the number of edges of the original hypergraph needed to cover all of its vertices. As described above, this may lead to an unfair valuation of seperators containing more than one hyperedge. In a third variant we tried to set the weight of special hyperedges to 2 and of all other edges to 1. This restricts the use of special hyperedges in separators, but does not penalize it too much. The results for the HM technique presented in Table 4 indicate that the first and the latter heuristic both yield similar results and clearly outperform the second one. Note that the class of examples Misc represent the NASA problem and two other random problems which are not presented in this paper.

Tables 5, 6, and 7 are the results obtained by appplying FM, TS, and HM in 140 examples from DaimlerChrysler, NASA, Grids, Cliques, and ISCAS circuits. Results for FM are obtained by the original algorithm using efficient gain updates that does not differentiate between "normal" and special hyperedges. Slightly better hypertree widths can be obtained by using the third variant for special hyperedges. This variant evaluates the cost of the set of hyperedges being cut more accurately by determining the number of hyperedges of the original hypergraph that are needed to cover the vertices of all hyperedges in the cut. However this variant has much worse time performace. The results presented for HM2 and TS are obtained by setting the weight of special hyperedges to 2 (other hyperedges have weight 1). The column HM best represents the best result obtained by HM techniques by using all variants considering the weights of special hyperedges. For each technique the best width found over 5 runs and the average time (for 5 runs) needed to find hypertree decomposition is presented.

From the results we can conclude that the best hypertree widths for the most of problems are obtained by using HMETIS partitioning algorithm. FM gives better results only for 11 instances and TS only for 6 instances. An explanation why HMETIS gives much better results could be that both FM and TS produce only a bi-partitioning of the hypergraph, whereas HMETIS divides the hypergraph into more components. Regarding the time performace the algorithms give comparable results and in general all the partitioning algorithms generated solutions quickly. Note that the combination of hypergraph partitioning algorithm with BE give marginal improvements to results obtained by hypergraph partitioning algorithm and BE individually. Furthermore, this algorithm has the disadvantage that it is very time consuming and because the time perfomance of this algorithm is much worse, results of this algorithm are not given in this paper.

5.3 Comparison of hypergraph partitioning algorithm and node ordering based heuristics

Based on the results given in Tables 1, 2, 3, 5, 6, 7 we can conclude that the best results are obtained by BE and HM algorithms. Comparing these two algorithms, the BE algorithm gives better results than HM for 52 instances. HM performs better for 29 instances. HM's time performance is overall better than the time performance of the BE algorithm. As the decomposition methods based on hypergraph partitioning are fast, it seems to be a good idea to run both heuristics and select the best result as an output.

Additional results for SAT problems are given in Tables 8, 9, 10, 11. For these instances the BE algorithm gives in general better results than HM. HM performs better for some instances.

6 Conclusions

In this paper we presented two classes of heuristic algorithms for generation of generalized hypertree decompositions with small width. We proposed the generation of hypertree decompositions based on the tree decompostions of the primal and dual graph. To generate tree decompostions we used Bucket Elimination and three heuristics for finding of vertex orderings. Further, we proposed a method for generating hypertree decompositions based on recursive partitioning of the hypergraph in weakly connected subgraphs. We used a state of the art hypergraph partitioning library, a well known hypergraph bi-partitioning heuristic in the literature, and proposed a new hypergraph partitioning heuristic based on ideas of tabu search. Additionally, we investigated the hybridization of the proposed heuristic algorithms. The proposed methods were evaluated in more than 200 problems from industry, literature and random generated problems.

Overall, we found that good tree decompositions can make good hypertree decompositions, even when just using greedy set-covering techniques. The tree decompositions of hypergraphs obtained from their primal graph gave in general better results compared to tree decompositions constructed from their dual graph. The experimental results have shown that hypergraph partitioning algorithms can be used successfully for generating hypertree decompositions in a short amount of time. Results show that better hypergraph partitioning algorithms give better hypertree decompositions.

Comparison of heuristics based on tree decompostions and hypergraph partitiong has shown that methods based on tree decompositions give slightly better results than methods based on hypergraph partitioning. However, for many problems hypergraph partitioning based algorithms give better upper bounds for hypertree decompositions. The experiments have shown that the time performance of hypergraph partitioning based algorithms is better compared to tree decomposition based algorithms. In general the results indicate that running of these two techniques in very large instances gives very good upper bounds for hypertree decompositions. By applying the algorithms proposed in this paper we could obtain new upper bounds for the width of hypertree decomposition for many large hypergraphs, whose width were not known previously in the literature.

References

[1] Arnborg, Corneil, and Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic and Discrete Methods*, 8(2):277–284, 1987.

[2] H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[3] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Design and implementation of move-based heuristics for VLSI hypergraph partitioning. *ACM Journal on Experimental Algorithms*, 5, 2000.

[4] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000. Database theory (Delphi, 1997).

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[6] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34, 1987.

[7] Rina Dechter. Constraint networks. In *Encyclopedia of Artificial Intelligence*, volume 1, pages 276–285. 2nd edition, 1992.

[8] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *AI*, 38(3):353–366, 1989.

[9] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

[10] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC '82: Proceedings of the 19th conference on Design automation*, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[11] Eugene C. Freuder. Complexity of k-tree structured constraint satisfaction problems. In *AAAI*, pages 4–9, 1990.

[12] Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[13] Georg Gottlob, Martin Hutle, and Franz Wotawa. Combining hypertree, bicomp, and hinge decomposition. In *15th European Conference on Artificial Intelligence (ECAI 02)*, Lyon, France, July 2002.

[14] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Tractable queries and constraints. *Proceedings of the Conference on Database and Expert Systems Applications (DEXA'99)*, LNCS 1677:1–15, 1999.

[15] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. 124(2):243–282, 2000.

[16] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[17] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decomposition and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[18] Georg Gottlob and Reinhard Pichler. Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *28th International Colloquium on Automata, Languages and Programming (ICALP01)*, volume 2076, pages 708–719, 2001.

[19] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. 66(1):57–89, 1994.

[20] Marc Gyssens and Jan Paredaens. A decomposition methodology for cyclic databases. In Hervé Gallaire, Jean-Marie Nicolas, and Jack Minker, editors, *Advances in Data Base Theory*, volume 2, pages 85–122. Plemum Press, New York, 1984.

[21] M. Hutle. Constraint satisfaction problems - hybrid decomposition and evaluation. Master's thesis, Technical University of Vienna, 2002.

[22] G. Karypis and V. Kumar. hmetis: A hypergraph partitioning package version 1.5.3, 1998.

[23] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(1):69–79, 1999.

[24] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[25] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 343–348, New York, NY, USA, 1999. ACM Press.

[26] Thomas Korimort. *Heuristic Hypertree Decomposition*. PhD thesis, Vienna University of Technology, 2003.

[27] A. Koster, H. Bodlaender, and S. van Hoesel. Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics 8, Elsevier Science Publishers*, 2001.

[28] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal Algorithms*, 7:309–322, 1986.

[29] Marko Samer. Hypertree-decomposition via branch-decomposition. In *19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 1535–1536, 2005.

[30] Markus Stumptner and Franz Wotawa. Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems. In *18th International Joint Conference on Artificial Intelligence (IJCAI 03)*.

[31] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13:566–579, 1984.

[32] Nysret Musliu Marko Samer Tobias Ganzow, Georg Gottlob. A csp hypergraph library. Technical Report, DBAI-TR-2005-50, Technische Universitt Wien, 2005.

[33] Mihalis Yannakakis. Algorithms for acyclic database schemes. In C. Zaniolo and C. Delobel, editors, *Very Large Data Bases, 7th International Conference, Sep. 9–11, 1981, Cannes, France, Proceedings*, pages 81–94. IEEE Computer Society, 1981.

Table 1: Results for Bucket Elimination and Dual Bucket Elimination

Instance (Atoms / Variables)	Min	opt-k-decomp		[26]		BE		DBE	
		W	T	W	T	W	T	W	T
adder_15 (76 / 106)	2	2	5	2		2	0	2	0
adder_25 (126 / 176)	2	2	40	2		2	0	2	0
adder_50 (251 / 351)	2			2		2	0	2	0
adder_75 (376 / 526)	2			2		2	0	2	1
adder_99 (496 / 694)	2			2		2	1	2	1
bridge_15 (137 / 137)	2	2	19	2	40	3	0	3	0
bridge_25 (227 / 227)	2	2	138	3	65	3	0	3	0
bridge_50 (452 / 452)	2	2	2211	3	174	3	1	3	1
bridge_75 (677 / 677)	2			2	726	3	1	3	2
bridge_99 (893 / 893)	2			2	1190	3	2	3	3
NewSystem1 (84 / 142)	3			3	31	3	0	3	0
NewSystem2 (200 / 345)	3			4	88	4	0	4	0
NewSystem3 (278 / 474)				4	271	5	1	5	0
NewSystem4 (418 / 718)				4	741	5	2	5	1
atv_partial_system (88 / 125)	3			3	47	3	0	4	0
NASA (680 / 579)						22	25	56	876
grid2d_10 (50 / 50)	4			?		5	0	6	0
grid2d_15 (112 / 113)	6			?		8	0	9	0
grid2d_20 (200 / 200)	7			?		12	0	11	0
grid2d_25 (312 / 313)	9			?		15	3	15	3
grid2d_30 (450 / 450)	11			?		19	7	20	8
grid2d_35 (612 / 613)	12			?		23	15	23	17
grid2d_40 (800 / 800)	14			?		26	28	25	31
grid2d_45 (1012 / 1013)	16			?		31	51	30	56
grid2d_50 (1250 / 1250)	17			?		33	86	32	94
grid2d_60 (1800 / 1800)	21			?		41	204	39	233
grid2d_70 (2450 / 2450)	24			?		48	474	47	503
grid2d_75 (2812 / 2813)	26			?		48	631	50	692
grid3d_4 (32 / 32)	5			?		6	0	6	0
grid3d_5 (62 / 63)	[6,8]			?		9	0	10	0
grid3d_6 (108 / 108)	[9,11]			?		14	1	14	1
grid3d_7 (171 / 172)	[11,14]			?		18	5	20	5
grid3d_8 (256 / 256)	[14,17]			?		25	17	27	17
grid3d_9 (364 / 365)	[18,22]			?		33	68	26	63
grid3d_10 (500 / 500)	[21,27]			?		41	164	40	202
grid3d_11 (665 / 666)	[26,32]			?		52	466	53	514
grid3d_12 (864 / 864)	[30,37]			?		63	1036	62	1139

Table 2: Results for Bucket Elimination and Dual Bucket Elimination

Instance (Atoms / Variables)	Min	opt-k-decomp		[26]		BE		DBE	
		W	T	W	T	W	T	W	T
grid3d_13 (1098 / 1099)	[35,42]			?		73	2357	68	2667
grid3d_14 (1372 / 1372)	[41,49]			?		78	3600	93	3600
grid3d_15 (1687 / 1688)	[46,56]			?		104	3600	103	3600
grid3d_16 (2048 / 2048)	[53,63]			?		114	3600	131	3600
grid4d_3 (40 / 41)				?		6	0	8	0
grid4d_4 (128 / 128)				?		17	4	18	5
grid4d_5 (312 / 313)				?		39	148	37	138
grid4d_6 (648 / 648)				?		68	2153	71	2115
grid4d_7 (1200 / 1201)				?		109	3600	110	3600
grid4d_8 (2048 / 2048)				?		148	3600	166	3600
grid5d_3 (121 / 122)				?		18	5	20	6
grid5d_4 (512 / 512)				?		62	2039	68	2058
grid5d_5 (1562 / 1563)				?		137	3600	159	3600
clique_10 (10 / 45)	5	5	0	?		5	0	5	0
clique_15 (15 / 105)	8			?		8	4	8	0
clique_20 (20 / 190)	10			?		10	47	10	0
clique_25 (25 / 300)	13			?		13	351	13	0
clique_30 (30 / 435)	15			?		15	1656	15	0
clique_35 (35 / 595)	18			?		18	3600	18	0
clique_40 (40 / 780)	20			?		20	3600	20	0
clique_45 (45 / 990)	23			?		23	3600	23	0
clique_50 (50 / 1225)	25			?		25	3600	25	1
clique_60 (60 / 1770)	30			?		30	3600	30	2
clique_70 (70 / 2415)	35			?		35	3600	35	3
clique_75 (75 / 2775)	38			?		38	3600	38	4
clique_80 (80 / 3160)	40			?		40	3600	40	5
clique_90 (90 / 4005)	45			?		45	3600	45	8
clique_99 (99 / 4851)	50			?		50	3600	50	12
c432 (160 / 196)	>3			?		9	1	9	1
c499 (202 / 243)	>3			?		13	1	20	2
c880 (383 / 443)				?		19	2	25	4
c1355 (546 / 587)				?		13	2	22	7
c1908 (880 / 913)				?		34	7	33	19
c2670 (1193 / 1350)				?		31	9	35	26
c3540 (1669 / 1719)				?		65	56	73	413
c5315 (2307 / 2485)				?		44	64	61	574
c6288 (2416 / 2448)				?		41	102	45	828
c7552 (3512 / 3718)				?		38	85	35	191

Table 3: Results for Bucket Elimination and Dual Bucket Elimination

		opt-k-decomp		[26]		BE		DBE	
Instance (Atoms / Variables)	Min	W	T	W	T	W	T	W	T
s27 (13 / 17)	2	2	0	?		2	0	2	0
s208 (104 / 115)	>3			?		7	0	7	0
s298 (133 / 139)	>3			?		5	0	8	1
s344 (175 / 184)	>3			?		7	0	8	0
s349 (176 / 185)	>3			?		7	0	9	0
s382 (179 / 182)	>3			?		5	0	8	1
s386 (165 / 172)				?		8	1	15	3
s400 (183 / 186)	>3			?		6	0	8	1
s420 (212 / 231)	>3			?		9	0	9	1
s444 (202 / 205)	>3			?		6	0	8	1
s510 (217 / 236)	>3			?		23	1	31	0
s526 (214 / 217)	>3			?		8	1	13	3
s641 (398 / 433)				?		7	1	14	2
s713 (412 / 447)				?		7	1	13	2
s820 (294 / 312)	>3			?		13	3	27	79
s832 (292 / 310)	>3			?		12	3	28	87
s838 (422 / 457)	>3			?		16	1	15	2
s953 (424 / 440)	>3			?		40	8	53	29
s1196 (547 / 561)				?		35	11	53	80
s1238 (526 / 540)				?		34	13	56	107
s1423 (731 / 748)				?		18	3	22	27
s1488 (659 / 667)				?		23	18	77	1035
s1494 (653 / 661)				?		24	19	78	1098
s5378 (2958 / 2993)				?		85	141	108	504
b01 (45 / 47)	>4			?		6	0	6	0
b02 (26 / 27)	3	3	2	?		3	0	5	0
b03 (152 / 156)	>3			?		7	0	11	1
b04 (718 / 729)				?		24	6	39	63
b05 (961 / 962)				?		18	10	29	48
b06 (48 / 50)	4			?		5	0	6	0
b07 (432 / 433)	>3			?		19	2	29	7
b08 (170 / 179)	>3			?		10	0	13	1
b09 (168 / 169)	>3			?		10	0	12	1
b10 (189 / 200)	>3			?		14	1	18	2
b11 (757 / 764)				?		30	8	47	67
b12 (1065 / 1070)				?		27	19	39	137
b13 (342 / 352)	>3			?		9	1	10	2

Examples	aggregate htw				
	w/o weights	w/ weights	she-weight 2	"best of"	bestof2
DaimlerChrysler	**49**	50	50	47	48
Misc	**99**	107	102	97	98
Grid2D	278	349	**275**	271	273
Grid3D	**503**	639	507	482	489
Grid4D5D	**458**	534	462	455	455
Clique	593	**577**	598	570	591
ISCAS85	383	421	**379**	370	373
ISCAS89	564	567	**534**	514	532
ISCAS99	265	269	**242**	238	239
Total	3192	3513	**3149**	3044	3098
Total (w/o cliques)	2599	2936	**2551**	2474	2507

Table 4: Comparison between different hyperede weighting schemes for hMETIS

Table 5: Results for Partitioning algorithms

Instance (Atoms / Variables)	Min	FM		TS		HM2		HM best	
		W	T	W	T	W	T	W	T
adder_15 (76 / 106)	2	2	0	4	0.2	2	3	2	3
adder_25 (126 / 176)	2	2	1	4	0.2	2	7	2	6
adder_50 (251 / 351)	2	2	6	4	1.2	2	13	2	12
adder_75 (376 / 526)	2	2	21	5	2	2	21	2	19
adder_99 (496 / 694)	2	2	53	5	3.2	2	28	2	25
bridge_15 (137 / 137)	2	8	1	8	0.8	4	7	3	6
bridge_25 (227 / 227)	2	13	1	6	1.4	4	11	3	11
bridge_50 (452 / 452)	2	29	5	10	3.2	4	24	4	22
bridge_75 (677 / 677)	2	44	10	10	5.4	4	39	3	35
bridge_99 (893 / 893)	2	64	18	10	6.8	4	48	4	45
NewSystem1 (84 / 142)	3	4	1	6	0.8	4	5	3	5
NewSystem2 (200 / 345)	3	9	2	6	2.2	4	14	4	13
NewSystem3 (278 / 474)		17	4	11	4	5	19	5	18
NewSystem4 (418 / 718)		22	8	12	6.8	5	31	5	29
atv_partial_system (88 / 125)	3	4	0	5	0.6	4	6	4	6
NASA (680 / 579)		56	20	98	33.6	33	90	32	84
grid2d_10 (50 / 50)	4	5	0	8	0.2	5	3	5	3
grid2d_15 (112 / 113)	6	10	1	12	1.2	10	11	10	10
grid2d_20 (200 / 200)	7	15	2	18	2.2	14	29	12	28
grid2d_25 (312 / 313)	9	18	5	26	4.6	15	50	15	43
grid2d_30 (450 / 450)	11	21	11	29	8	16	70	16	58
grid2d_35 (612 / 613)	12	30	20	41	12.6	19	87	19	73
grid2d_40 (800 / 800)	14	28	38	41	19.8	22	108	22	91
grid2d_45 (1012 / 1013)	16	40	58	47	31.2	25	130	25	109
grid2d_50 (1250 / 1250)	17	44	88	52	40.8	28	154	28	130
grid2d_60 (1800 / 1800)	21	55	203	75	74.6	34	209	34	178
grid2d_70 (2450 / 2450)	24	65	347	65	119	41	283	41	239
grid2d_75 (2812 / 2813)	26	70	504	99	157.8	44	324	44	274
grid3d_4 (32 / 32)	5	6	0	12	0.2	6	1	6	1
grid3d_5 (62 / 63)	[6,8]	8	1	18	0.8	11	4	10	3
grid3d_6 (108 / 108)	[9,11]	12	1	25	1.8	15	9	14	9
grid3d_7 (171 / 172)	[11,14]	18	2	33	4.8	19	27	16	24
grid3d_8 (256 / 256)	[14,17]	25	5	44	8.6	21	48	20	40
grid3d_9 (364 / 365)	[18,22]	34	9	56	14.4	24	67	24	56
grid3d_10 (500 / 500)	[21,27]	41	20	67	26.4	31	93	31	77
grid3d_11 (665 / 666)	[26,32]	40	36	83	42.6	37	119	37	99
grid3d_12 (864 / 864)	[30,37]	53	61	98	66.4	45	150	44	127

Table 6: Results for Partitioning algorithms

Instance (Atoms / Variables)	Min	FM		TS		HM2		HM best	
		W	T	W	T	W	T	W	T
grid3d_13 (1098 / 1099)	[35,42]	60	107	122	100.8	53	186	53	158
grid3d_14 (1372 / 1372)	[41,49]	86	161	176	162.2	69	230	69	196
grid3d_15 (1687 / 1688)	[46,56]	93	253	151	245.4	76	278	76	244
grid3d_16 (2048 / 2048)	[53,63]	100	400	174	328	87	339	82	303
grid4d_3 (40 / 41)		8	0	20	0.4	9	2	8	2
grid4d_4 (128 / 128)		17	1	40	3.4	19	13	18	12
grid4d_5 (312 / 313)		32	8	78	16.8	28	58	28	48
grid4d_6 (648 / 648)		58	40	140	66.8	47	123	47	106
grid4d_7 (1200 / 1201)		89	134	182	193.8	74	229	71	208
grid4d_8 (2048 / 2048)		120	441	310	580.8	107	408	107	393
grid5d_3 (121 / 122)		18	1	49	3.6	20	11	19	10
grid5d_4 (512 / 512)		49	25	137	56.2	46	92	46	78
grid5d_5 (1562 / 1563)		118	280	362	474	111	328	111	319
clique_10 (10 / 45)	5	5	0	6	0.2	5	0	5	0
clique_15 (15 / 105)	8	12	0	8	1.4	8	1	8	1
clique_20 (20 / 190)	10	20	0	11	3.4	10	1	10	1
clique_25 (25 / 300)	13	25	0	14	8.2	13	2	13	2
clique_30 (30 / 435)	15	30	0	16	16.2	15	3	15	3
clique_35 (35 / 595)	18	35	0	19	28.8	18	5	18	5
clique_40 (40 / 780)	20	40	0	22	50.8	20	6	20	6
clique_45 (45 / 990)	23	45	1	24	80.2	23	10	23	9
clique_50 (50 / 1225)	25	50	1	28	144.6	25	15	25	13
clique_60 (60 / 1770)	30	60	2	34	340.2	59	1	50	1
clique_70 (70 / 2415)	35	70	4	39	601	68	2	67	1
clique_75 (75 / 2775)	38	75	6	41	920	71	3	71	2
clique_80 (80 / 3160)	40	80	8	43	1248	76	4	72	3
clique_90 (90 / 4005)	45	90	12	50	2045	89	8	78	5
clique_99 (99 / 4851)	50	99	19	54	2844.8	99	14	97	8
c432 (160 / 196)	>3	15	3	24	3.6	13	20	12	19
c499 (202 / 243)	>3	18	3	27	4.6	18	30	17	28
c880 (383 / 443)		31	8	41	7.4	29	50	25	46
c1355 (546 / 587)		32	10	55	13.8	22	66	22	61
c1908 (880 / 913)		65	23	70	25.6	29	86	29	77
c2670 (1193 / 1350)		66	56	78	45.8	38	119	38	106
c3540 (1669 / 1719)		97	133	129	103.8	73	166	73	149
c5315 (2307 / 2485)		120	250	157	156.6	72	242	68	214
c6288 (2416 / 2448)		148	478	329	245	45	210	45	186
c7552 (3512 / 3718)		161	514	188	351	37	365	37	309

Table 7: Results for Partitioning algorithms

Instance (Atoms / Variables)	Min	FM		TS		HM2		HM best	
		W	T	W	T	W	T	W	T
s27 (13 / 17)	2	2	0	3	0	2	0	2	0
s208 (104 / 115)	>3	7	1	11	0.8	7	10	7	9
s298 (133 / 139)	>3	7	1	17	1.8	7	11	6	10
s344 (175 / 184)	>3	8	2	12	1.6	8	21	7	19
s349 (176 / 185)	>3	8	1	12	1.6	9	21	7	19
s382 (179 / 182)	>3	7	2	17	2	8	16	7	15
s386 (165 / 172)		13	2	26	2.8	11	16	11	15
s400 (183 / 186)	>3	8	2	18	2.2	8	18	7	17
s420 (212 / 231)	>3	10	2	14	1.6	10	29	10	24
s444 (202 / 205)	>3	8	2	25	2.8	8	21	8	20
s510 (217 / 236)	>3	23	4	41	4.2	27	27	27	25
s526 (214 / 217)	>3	13	2	32	3	11	29	11	27
s641 (398 / 433)		19	5	21	5.4	14	31	14	28
s713 (412 / 447)		21	5	25	5.4	14	33	14	31
s820 (294 / 312)	>3	23	8	77	9.4	24	42	19	38
s832 (292 / 310)	>3	22	8	71	9.8	26	42	20	39
s838 (422 / 457)	>3	19	6	24	6.4	15	55	15	46
s953 (424 / 440)	>3	50	18	70	11.6	45	52	45	47
s1196 (547 / 561)		50	22	73	15.2	43	69	43	62
s1238 (526 / 540)		56	20	80	18.4	43	66	43	59
s1423 (731 / 748)		29	25	54	19	27	78	26	71
s1488 (659 / 667)		45	46	148	36	39	85	39	77
s1494 (653 / 661)		49	45	150	38.4	38	85	36	77
s5378 (2958 / 2993)		178	308	169	271	89	279	89	246
b01 (45 / 47)	>4	5	0	10	0.2	5	2	5	2
b02 (26 / 27)	3	4	0	7	0.2	4	1	4	1
b03 (152 / 156)	>3	11	1	16	1.6	9	15	8	14
b04 (718 / 729)		44	26	69	26.2	38	82	35	73
b05 (961 / 962)		42	33	70	29.8	32	114	32	99
b06 (48 / 50)	4	5	0	12	0.2	5	3	5	2
b07 (432 / 433)	>3	38	7	59	9.8	33	57	31	54
b08 (170 / 179)	>3	14	2	20	2	12	21	12	19
b09 (168 / 169)	>3	13	2	20	1.8	12	20	12	19
b10 (189 / 200)	>3	17	3	33	3	16	24	16	21
b11 (757 / 764)		65	28	98	27.2	38	89	38	79
b12 (1065 / 1070)		38	55	83	38.6	34	111	34	102
b13 (342 / 352)	>3	10	4	17	3.6	8	36	8	33

Table 8: Results for SAT problems

Instance (Atoms / Variables)	BE (width)	DBE (width)	HM best (width)
uf20-01 (91 / 20)	6	7	8
uf20-050 (91 / 20)	6	8	9
uf20-099 (91 / 20)	6	8	8
uf75-01 (325 / 75)	20	29	23
uf75-050 (325 / 75)	19	30	23
uf75-099 (325 / 75)	20	30	23
uuf75-01 (325 / 75)	20	29	23
uuf75-050 (325 / 75)	19	29	23
uuf75-099 (325 / 75)	20	29	22
uf150-01 (645 / 150)	40	61	39
uf150-050 (645 / 150)	38	60	38
uf150-099 (645 / 150)	38	59	37
uuf150-01 (645 / 150)	38	61	37
uuf150-050 (645 / 150)	38	60	38
uuf150-099 (645 / 150)	37	58	37
uf200-01 (860 / 200)	49	78	50
uf200-050 (860 / 200)	50	81	51
uf200-099 (860 / 200)	51	78	51
uuf200-01 (860 / 200)	52	81	50
uuf200-050 (860 / 200)	51	78	50
uuf200-099 (860 / 200)	50	77	50
ais6 (581 / 61)	10	11	14
ais8 (1520 / 113)	14	15	21
ais10 (3151 / 181)	19	19	23
2bitcomp_5 (310 / 95)	11	11	11
2bitmax_6 (766 / 192)	15	23	27
2bitadd_10 (1422 / 330)	25	24	28
2bitadd_11 (1562 / 363)	28	24	34
2bitadd_12 (1702 / 396)	25	24	34
flat30-1 (300 / 90)	10	17	15
flat30-50 (300 / 90)	11	22	14
flat30-99 (300 / 90)	11	18	14
flat75-1 (840 / 225)	27	49	32
flat75-50 (840 / 225)	30	56	30
flat75-99 (840 / 225)	28	44	32
flat150-1 (1680 / 450)	52	44	54
flat150-50 (1680 / 450)	55	93	49
flat150-99 (1680 / 450)	53	84	53
flat200-1 (2237 / 600)	65	115	66
flat200-50 (2237 / 600)	68	110	65
flat200-99 (2237 / 600)	71	109	70

Table 9: Results for SAT problems

Instance (Atoms / Variables)	BE (width)	DBE (width)	HM best (width)
sw180-1 (3100 / 500)	50	100	55
sw180-99 (3100 / 500)	48	100	55
sw181-1 (3100 / 500)	51	100	52
sw181-99 (3100 / 500)	44	100	54
sw182-1 (3100 / 500)	32	100	40
sw182-99 (3100 / 500)	42	100	41
sw18p0-1 (3100 / 500)	17	100	24
aim-50-1_6-no-3 (80 / 50)	9	11	9
aim-50-1_6-yes1-3 (80 / 50)	10	10	11
aim-50-2_0-no-3 (100 / 50)	12	14	13
aim-50-2_0-yes1-3 (100 / 50)	11	12	13
aim-50-3_4-yes1-3 (170 / 50)	13	16	15
aim-50-6_0-yes1-3 (300 / 50)	14	19	17
aim-100-1_6-no-3 (160 / 100)	19	22	21
aim-100-1_6-yes1-3 (160 / 100)	17	21	20
aim-100-2_0-no-3 (200 / 100)	23	25	26
aim-100-2_0-yes1-3 (200 / 100)	20	23	24
aim-100-3_4-yes1-3 (340 / 100)	26	31	28
aim-100-6_0-yes1-3 (600 / 100)	28	37	31
aim-200-1_6-no-3 (320 / 200)	37	42	39
aim-200-1_6-yes1-3 (320 / 200)	38	40	37
aim-200-2_0-no-3 (400 / 200)	47	49	47
aim-200-2_0-yes1-3 (400 / 200)	41	45	45
aim-200-3_4-yes1-3 (680 / 200)	52	60	48
aim-200-6_0-yes1-3 (1200 / 200)	58	79	57
dubois20 (160 / 60)	2	2	2
dubois21 (168 / 63)	2	2	2
dubois22 (176 / 66)	2	2	2
dubois23 (184 / 69)	2	2	2
dubois24 (192 / 72)	2	2	2
dubois25 (200 / 75)	2	2	2
dubois26 (208 / 78)	2	2	2
dubois27 (216 / 81)	2	2	2
dubois28 (224 / 84)	2	2	2
dubois29 (232 / 87)	2	2	2
dubois30 (240 / 90)	2	2	2
dubois50 (400 / 150)	2	2	2
dubois100 (800 / 300)	2	2	2

Table 10: Results for SAT problems

Instance (Atoms / Variables)	BE (width)	DBE (width)	HM best (width)
ii8a1 (186 / 66)	7	8	8
ii8a2 (800 / 180)	15	16	17
ii8a3 (1552 / 264)	18	30	21
ii8a4 (2798 / 396)	18	54	25
ii8b1 (2068 / 336)	30	42	30
ii8c1 (3065 / 510)	40	37	42
ii8d1 (3207 / 530)	42	39	44
ii8e1 (3136 / 520)	41	39	43
ii32b1 (1374 / 228)	14	16	27
ii32b2 (2558 / 261)	12	27	51
ii32c1 (1280 / 225)	13	15	24
ii32c2 (2182 / 249)	13	23	51
ii32c3 (3272 / 279)	11	34	73
ii32d1 (2703 / 332)	16	23	29
ii32e1 (1186 / 222)	12	14	24
ii32e2 (2746 / 267)	12	29	47
jnh201 (800 / 100)	15	20	35
jnh205 (800 / 100)	16	23	28
jnh210 (800 / 100)	16	20	30
jnh215 (800 / 100)	16	23	30
jnh220 (800 / 100)	16	24	32
jnh1 (850 / 100)	15	20	33
jnh5 (850 / 100)	16	19	28
jnh10 (850 / 100)	16	20	31
jnh15 (850 / 100)	16	21	32
jnh20 (850 / 100)	16	22	28
jnh301 (900 / 100)	15	21	33
jnh305 (900 / 100)	15	20	32
jnh310 (900 / 100)	15	22	31

Table 11: Results for SAT problems

Instance (Atoms / Variables)	BE (width)	DBE (width)	HM best (width)
par8-1-c (254 / 64)	7	11	7
par8-2-c (270 / 68)	6	12	7
par8-3-c (298 / 75)	8	11	8
par8-4-c (266 / 67)	7	11	7
par8-5-c (298 / 75)	7	11	8
par8-1 (1149 / 350)	18	21	23
par8-2 (1157 / 350)	19	21	22
par8-3 (1171 / 350)	19	21	19
par8-4 (1155 / 350)	18	21	23
par8-5 (1171 / 350)	19	21	22
par16-1-c (1264 / 317)	16	32	24
par16-2-c (1392 / 349)	16	33	26
par16-3-c (1332 / 334)	16	33	26
par16-4-c (1292 / 324)	17	33	29
par16-5-c (1360 / 341)	17	32	28
par16-1 (3310 / 1015)	31	39	43
par16-2 (3374 / 1015)	33	38	43
par16-3 (3344 / 1015)	33	38	43
par16-4 (3324 / 1015)	33	38	42
par16-5 (3358 / 1015)	32	38	42
hole6 (133 / 42)	7	7	7
hole7 (204 / 56)	8	8	8
hole8 (297 / 72)	9	9	9
hole9 (415 / 90)	10	10	10
hole10 (561 / 110)	11	11	11
pret60_25 (160 / 60)	5	5	5
pret60_40 (160 / 60)	5	5	5
pret60_60 (160 / 60)	5	5	5
pret60_75 (160 / 60)	5	5	5
pret150_25 (400 / 150)	5	5	5
pret150_40 (400 / 150)	5	5	5
pret150_60 (400 / 150)	5	5	5
pret150_75 (400 / 150)	5	5	5