Heuristic Methods for Hypertree Decomposition

Artan Dermaku[1], Tobias Ganzow[2], Georg Gottlob[3,1,⋆],
Ben McMahan[4], Nysret Musliu[1], and Marko Samer[1]

[1] Institut für Informationssysteme (DBAI), TU Wien, Austria
[2] Mathematische Grundlagen der Informatik, RWTH Aachen, Germany
[3] Computing Laboratory, University of Oxford, UK
[4] Department of Computer Science, Rice University, USA

Abstract. The literature provides several structural decomposition methods for identifying tractable subclasses of the constraint satisfaction problem. *Generalized hypertree decomposition* is the most general of such decomposition methods. Although the relationship to other structural decomposition methods has been thoroughly investigated, only little research has been done on efficient algorithms for computing generalized hypertree decompositions. In this paper we propose new heuristic algorithms for the construction of generalized hypertree decompositions. We evaluate and compare our approaches experimentally on both industrial and academic benchmark instances. Our experiments show that our algorithms improve previous heuristic approaches for this problem significantly.

1 Introduction

Many important problems in artificial intelligence, database systems, and operations research can be formulated as *constraint satisfaction problems (CSPs)*. Such problems include scheduling, planning, configuration, diagnosis, machine vision, spatial and temporal reasoning, etc. [5]. A *CSP instance* consists of a finite set V of variables, a set D of domain elements, and a finite set of constraints. A constraint defines for its scope $V' \subseteq V$ the allowed instantiations of the variables in V' by values in D. The question is if there exists an instantiation of all variables such that no constraint of the instance is violated.

Although solving CSPs is known to be NP-hard in general, many problems that arise in practice have particular 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 theoretical and a practical point of view. Such restrictions may either involve the *nature* of the constraints (i.e., which instantiations of the variables are allowed) or they may involve the *structure* of the constraints. In this paper we consider the second approach. The structure of a CSP instance can be modeled by its *constraint hypergraph*. Hypergraphs are a generalization of graphs where each edge (called *hyperedge*) connects an arbitrary subset of vertices. Let V be the set of variables and E be the set of constraint scopes of some CSP instance. Then the constraint hypergraph of this instance is given by $H = (V, E)$.

⋆ Corresponding author: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, England, UK, E-Mail: georg.gottlob@comlab.ox.ac.uk

A. Gelbukh and E.F. Morales (Eds.): MICAI 2008, LNAI 5317, pp. 1–11, 2008.

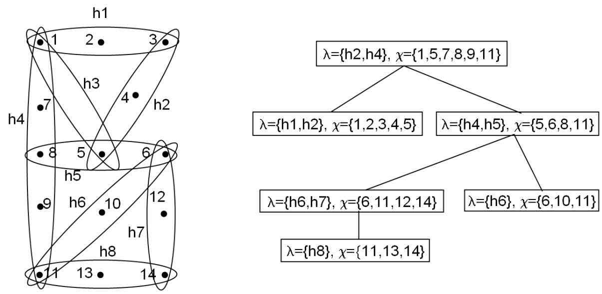

Fig. 1. A hypergraph and its generalized hypertree decomposition

A prominent tractable class of CSPs are *acyclic* CSPs. A CSP instance is acyclic if and only if its constraint hypergraph is acyclic. If a CSP instance is acyclic, then it can be solved efficiently by Yannakakis's classical algorithm [28]. The favorable results on acyclic CSPs extend to classes of "nearly acyclic" CSPs. Several methods have been suggested in the literature to transform an arbitrary CSP instance into an acyclic one, making the vague notion of "nearly acyclic" precise. The most prominent methods include: *tree clustering* [6], *hinge decomposition* [16,17], *cycle cutset* and *cycle hypercutset* [4,12], *hinge-tree clustering* [12], *query decomposition* [2], and *(generalized) hypertree decomposition* [13]. *Generalized hypertree decomposition* [13] is the most general one [11]. Each of these methods has an associated measure for the degree of acyclicity. In the case of generalized hypertree decomposition, this measure is called *generalized hypertree-width*. The smaller the generalized hypertree-width, the "more acyclic" the CSP is; acyclic CSPs have generalized hypertree-width one.

The formal definition of a generalized hypertree decomposition is given in the following: Let $H = (V(H), E(H))$ be a hypergraph. A *tree decomposition* [25] of H is a tree $T = (V(T), E(T))$ together with a mapping $\chi : V(T) \rightarrow 2^{V(H)}$ labeling each node of the tree with a set of vertices of H such that (i) for each $e \in E(H)$ there is a node $t \in V(T)$ with $e \subseteq \chi(t)$ and (ii) for each $v \in V(H)$ the set $\{t \in V(T) \mid v \in \chi(t)\}$ induces a connected subtree. The *width* of a tree decomposition is the maximum $|\chi(t)| - 1$ over all nodes $t \in V(T)$. The *treewidth* of a hypergraph is the minimum width over all its tree decompositions. A *generalized hypertree decomposition* [13] of H is a tree decomposition (T, χ) of H together with a mapping $\lambda : V(T) \rightarrow 2^{E(H)}$ labeling each node of the tree with a set of hyperedges of H such that for each $t \in V(T)$ it holds that $\chi(t) \subseteq \bigcup \lambda(t)$. The *width* of a generalized hypertree decomposition is the maximum $|\lambda(t)|$ over all nodes $t \in V(T)$. The *generalized hypertree-width* of a hypergraph is the minimum width over all its generalized hypertree decompositions. We say that a tree decomposition and a generalized hypertree decomposition is *optimal* if it has minimal width. Figure 1 shows a hypergraph H and one of its generalized hypertree decompositions. The width of the underlying tree decomposition is 5, and the width of the generalized hypertree decomposition is 2.

It has been an open question whether optimal generalized hypertree decompositions can be computed in polynomial time if the width is bounded by some constant. This question was recently answered negatively, i.e., computing optimal generalized hypertree decompositions is NP-hard even for bounded width [14]. Thus, in order to enforce a polynomial runtime in the case of bounded width, an additional condition has been added to the definition of generalized hypertree decompositions: A *hypertree decomposition* [13] of H is a rooted generalized hypertree decomposition of H such that for each $t \in V(T)$ it holds that $(\bigcup \lambda(t)) \cap (\bigcup_{t' \in V(T_t)} \chi(t')) \subseteq \chi(t)$, where T_t denotes the subtree of T rooted at t. Although optimal hypertree decompositions can be computed in polynomial time for bounded width, the degree of the polynomial in the runtime estimation depends on the width. Thus, known algorithms for computing optimal hypertree decompositions are only practical for small bounds on the width [15]. For this reason we consider heuristic approaches for computing (not necessarily optimal) hypertree decompositions of small width. Since these heuristic approaches do not need the additional condition of a hypertree decomposition, we actually construct *generalized* hypertree decompositions that potentially allow smaller widths. Note that a generalized hypertree decomposition suffices to solve the corresponding CSP instance in polynomial time. For simplicity, we will write hypertree decomposition instead of generalized hypertree decomposition in the remainder of this paper.

Heuristically constructed hypertree decompositions are not necessarily optimal. However, the smaller the width of the obtained hypertree decomposition, the faster the corresponding CSP instance can be solved. In fact, a CSP instance can be solved based on its hypertree decomposition as follows: For each node t of the hypertree, all constraints in $\lambda(t)$ are "joined" into a new constraint over the variables in $\chi(t)$. For bounded width, i.e., for bounded cardinality of $\lambda(t)$, this yields a polynomial time reduction to an equivalent acyclic CSP instance which can be solved by Yannakakis's algorithm [28].

Several heuristic approaches for the construction of hypertree decompositions have been proposed in the literature: In [22] an approach based on the vertex connectivity of the given hypergraph (in terms of its primal graph and incidence graph) has been presented, and in [26] the application of branch decomposition heuristics has been considered. Moreover, in [24] the application of genetic algorithms has been investigated.

The problem of these approaches, however, is that they work only well on a restricted class of benchmark examples or they show a very poor runtime performance even on rather small examples. In this paper we present new heuristic approaches that run very fast on a large variety of benchmark examples from both industry and academics; moreover, the results of our algorithms are in most cases superior to those of previous approaches, and even if they are not better than previous results, they are very close to them.

This paper is organized as follows: In Section 2 we present the application of tree decomposition and set cover heuristics, and in Section 3 we present the application of various hypergraph partitioning heuristics. Finally, we evaluate our algorithms experimentally in Section 4, and we conclude in Section 5. Note that due to space limitations we can only succinctly describe our approaches.

2 Tree Decomposition and Set Cover

In this section we combine heuristics for constructing tree decompositions based on linear vertex orderings with set cover heuristics. Recall that the definition of a hypertree decomposition can be divided into two parts: (i) the definition of a tree decomposition (T, χ) and (ii) the introduction of λ such that $\chi(t) \subseteq \bigcup \lambda(t)$ for every node t. Since the χ-labels contain vertices of the underlying hypergraph and the λ-labels contain hyperedges, i.e., sets of vertices, of the underlying hypergraph, it is clear that the condition in (ii) is nothing else but covering the vertices in $\chi(t)$ by hyperedges in $\lambda(t)$ for every node t. In other words, if we are given a tree decomposition, we can simply construct a hypertree decomposition by putting for each node t hyperedges into $\lambda(t)$ that cover the vertices in $\chi(t)$. This is the basic idea of a heuristic hypertree decomposition approach [23] originally (but somehow misleadingly) named after the CSP solving technique *bucket elimination (BE)* [5]. The heuristic assumption behind this approach is that (heuristically obtained) tree decompositions of small width allow hypertree decompositions of small width by applying set cover heuristics.

The literature provides several powerful tree decomposition heuristics. An important class of such heuristics is based on finding an appropriate linear ordering of the vertices from which a tree decomposition can be constructed [1]. We use the following three well-known vertex ordering heuristics [1,5,23]: *Maximum cardinality*, *minimum induced-width* (also known as *minimum degree*), and *minimum fill-in*. We construct a tree decomposition based on vertex orderings for each of these ordering heuristics and compute the λ-labels by applying the following two set cover heuristics: The first one iteratively picks a hyperedge that covers the largest number of uncovered vertices. The second one assigns the weight 0 to each covered vertex and the weight $1 - \frac{m}{n}$ to each uncovered vertex, where m is the number of hyperedges containing the vertex and n is the number of hyperedges in the hypergraph. The heuristic iteratively picks the hyperedge that has the highest weighted sum over all its vertices. In both heuristics, ties are broken randomly. At each node t we apply both heuristics and define $\lambda(t)$ as the smaller set.

We also considered an approach dual to the above one in the sense that we apply the above steps to the dual hypergraph. The dual hypergraph of a hypergraph is simply obtained by swapping the roles of hyperedges and vertices. For symmetry reasons, let us call this approach *dual bucket elimination (DBE)*. Our intuition for using the dual hypergraph instead of the original hypergraph is that the ordering heuristics aim at minimizing the labeling sets, which are the χ-labels in the case of the original hypergraph. However, the width of a hypertree decomposition is determined by the size of the λ-labels. So our aim was to apply the ordering heuristics in order to minimize the λ-labels, which is exactly what is done when applying BE to the dual hypergraph. Our procedure is the following: (i) build the dual hypergraph, (ii) apply BE 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 straightforward way (see e.g. [13]). The resulting hypertree is then a hypertree decomposition of the original hypergraph.

3 Hypergraph Partitioning

In this section we consider the use of hypergraph partitioning heuristics that aim at finding a partitioning of the vertices of the hypergraph that is optimal w.r.t. a valuation function on the set of hyperedges connecting the partitions while, at the same time, being subject to restrictions on the relative sizes of the partitions. As hypergraph partitioning with restrictions on the sizes of the partitions is NP-complete [9], various successful heuristic methods have been proposed in the literature. In the following we consider the applicability of an algorithm due to Fiduccia and Mattheyses [7] as well as the hMETIS library [18], and we propose a new heuristic for hypergraph partitioning based on tabu search. Note that our construction of hypertree decompositions using hypergraph partitioning heuristics is itself heuristic since the computation of cuts needed for such a construction is NP-hard and not even fixed-parameter tractable [27].

Korimort [22] was the first who applied hypergraph partitioning heuristics for hypertree decomposition by recursively separating the hypergraph into smaller and smaller subgraphs. In each partitioning step, a new hypertree node is created and labeled with a set of hyperedges (called *separator*) disconnecting the subgraphs; the χ-labels can be computed in a straightforward way as described in [13]. Note, however, that a hypertree decomposition cannot be constructed by simply connecting the obtained nodes according to the partitioning tree since such a decomposition does not necessarily satisfy the connectedness condition. For this reason Korimort suggested to add a *special hyperedge* to each subgraph containing the vertices in the intersection between the subgraphs in order to enforce their joint appearance in the χ-label of a later generated node. This node can then be used as the root of the subtree. See [22] for more details.

Unfortunately, the introduction of special hyperedges raises the problem of how to evaluate a cut whose separator contains such hyperedges. Being not contained in the original hypergraph, they have to be replaced by possibly more than one hyperedge in the final decomposition which might increase the size of the λ-label of the corresponding node, and thus the width of the decomposition.

To address this problem, we considered hyperedges with associated integral weights and three different weighting schemes; the cut can be uniformly evaluated as the sum of the weights of all hyperedges in the separator. All weighting schemes assign the weight one to all ordinary hyperedges contained in the original hypergraph. The weighting scheme (W1) also assigns the weight one to all special hyperedges, thus treating them like ordinary hyperedges. In the weighting scheme (W+) the weight assigned to a special hyperedge equals the number of ordinary hyperedges needed to cover the vertices of the special hyperedge. Obviously, this might lead to an inaccurate evaluation of separators containing more than a single special hyperedge. As a compromise, we considered the scheme (W2) assigning the weight two to all special hyperedges.

3.1 Partitioning Using Fiduccia-Mattheyses

The hypergraph partitioning algorithm proposed by Fiduccia and Mattheyses [7] is based on an iterative refinement heuristic. First, the hypergraph is arbitrarily partitioned into two parts. In the following, during a sequence of passes, the partitioning is optimized by successively moving vertices to the opposite partition. The decision which vertex is to be moved next is based on a balancing constraint and the *gain* associated to each vertex. The gain is a measure for the impact of the move on the size of the separator. Moreover, a locking mechanism prevents vertices from being moved twice during a pass. Although the algorithm chooses the best possible move in each step, it is nevertheless capable of climbing out of local minima since even the best possible move might lead to a worse solution. A pass is finished after all vertices have been moved once, and the best solution seen during the pass is taken as the initial solution for the next pass. The algorithm terminates if the initial solution could not be improved during a pass.

The significance of the Fiduccia-Mattheyses algorithm (FM) is due to the possibility of efficient gain updates both at the beginning of a pass and, even more important, during a pass, after a move has been made. Obviously, the gain updates are still efficient if instead of considering the size of the cut we use the sum of the weights of the hyperedges of the cut as a cost function.

Following a note by Korimort [22] stating that considering separators containing special hyperedges should make the problem of finding a good decomposition harder and should not lead to better results, we implemented a variant where all vertices contained in a special hyperedge are moved at once which avoids the valuation problem caused by such hyperedges. However, our results do not support Korimort's assumption.

3.2 Partitioning Using hMETIS

Another approach is based on hMETIS [18,19,20,21], a software package for partitioning hypergraphs developed at the University of Michigan, which is claimed to be one of the best available packages for hypergraph partitioning.

The algorithm follows a multilevel-approach and is organized in three phases. During a first *coarsening phase*, the algorithm constructs a sequence of smaller and smaller hypergraphs by merging vertices and hyperedges. The manual makes no definite statement about the stop criterion, but it mentions that the number of vertices of the coarsest hypergraph is usually between 100 and 200. In the second phase, a variant of the FM algorithm is used to create several bipartitions of this coarsest hypergraph starting from random bipartitions. In the following *uncoarsening phase*, the bipartitions are successively projected onto the next level's finer hypergraph and optimized around the cut. At the end, after having reached the original hypergraph, the best bipartition is chosen.

Unlike the original algorithm of Fiduccia and Mattheyses we implemented, the two routines offered by the hMETIS package are capable of computing k-way partitionings. The routines allow for control over several parameters influencing the phases of the partitioning, however, the interdependencies are non-trivial and the exact impact is hard to estimate. Several tests showed that the parameters with most influence on the width of the resulting hypertree decomposition are the number of partitions (*nparts*), which corresponds to the number of children of the current node in the hypertree decomposition, and the imbalance factor (*ubfactor*) controlling the relative size of the partitions and thus directly influencing the balance of the resulting hypertree. For a complete description of all parameters see [19]. Our test results show that the hypertree decompositions tend to get better for *nparts* > 2, and usually higher *ubfactors* lead to decompositions of smaller width.

3.3 Partitioning Using Tabu Search

In this section we present a new hypergraph partitioning algorithm based on the ideas of tabu search [10]. The algorithm starts with a simple initial solution where one partition contains only one vertex and the second partition contains all other vertices. In each iteration the hyperedges connecting the partitions are considered and the neighborhood of the current solution is generated by moving vertices from one partition into the other partition. This is done by choosing one of two strategies in each iteration: The first one is chosen with probability p; in this case we move the vertices of a single randomly selected connecting hyperedge. The second one is chosen with probability $1 - p$; in this case we move the vertices of all connecting hyperedges. After the neighborhood is generated, the solutions are evaluated according to a fitness function. The fitness function is the sum of weights over all connecting hyperedges; we tried several weights for these hyperedges. 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 criterion* is applied. For the aspiration criterion, we use a standard version [10] according to which the tabu status of a move is ignored if the move yields a solution that is better than the currently best solution. For finding the most appropriate length of the tabu list, we tried several lengths and probabilities p. In particular, we tried the following lengths for the tabu list depending on the number $|V|$ of vertices in the hypergraph: $\frac{|V|}{2}$, $\frac{|V|}{3}$, $\frac{|V|}{5}$, $\frac{|V|}{10}$.

4 Experimental Results

In this section we report the experimental results obtained by our implementations of the heuristics described in this paper. We use hypergraphs from the CSP hypergraph library [8] as benchmarks. This collection contains various classes of constraint hypergraphs from industry (DaimlerChrysler, NASA, and ISCAS) as well as synthetically generated ones (Grids and Cliques). The CSP hypergraph library and the executables of the current implementations can be downloaded from the DBAI Hypertree Project website [3]. All experimental results were obtained on a machine with Intel Xeon (dual) processor, 2.2 GHz, 2 GB main memory. Since ties are broken randomly in our algorithms, we applied each method five times to all instances. The results are presented in Table 1. There are two columns for each method: the minimal width obtained during five runs and the average runtime in seconds over five runs. We emphasized the minimal widths in each row by bold numbers, which makes it easy to identify the algorithms that perform best on each class of benchmarks.

The first two columns (BE and DBE) are based on tree decomposition and set cover heuristics. Our experiments show that BE clearly outperforms DBE. Only on the Clique hypergraphs, DBE is faster than BE; this however is not surprising since the number of hyperedges (i.e., the number of vertices of the dual hypergraph) is much smaller than the number of vertices for all instances in this class.

The last four columns (FM-W1, TS-W2, HM-W2, HM-best) are based on hypergraph partitioning heuristics. The results for FM are obtained using weighting scheme (W1), i.e., we do not distinguish between ordinary and special hyperedges. The results for TS and HM are obtained using weighting scheme (W2). We found out that, in most cases, the weighting schemes (W1) and (W2) yield similar results which are significantly better than those achieved when using scheme (W+). The column HM-best displays the minimal width obtained by the HM algorithm over all three weighting schemes together with the runtime for the selected weighting scheme. Our experiments show that HM clearly outperforms the other partitioning based approaches. An explanation for this is that hMETIS is a highly optimized library that combines several hypergraph partitioning techniques. Seemingly this also influences the results when using hMETIS for computing hypertree decompositions.

Comparing all approaches proposed in this paper, we conclude that the best results are obtained by BE and HM. BE outperforms HM on the classes DaimlerChrysler, NASA, ISCAS, and Cliques, whereas HM outperforms BE on the class Grids. In particular, BE is superior on all industrial benchmark examples. Concerning the runtime, Table 1 shows that all our algorithms are fast. Note, moreover, that the runtime of BE is especially low on almost all benchmark classes.

Compared to previous heuristic hypertree decomposition algorithms [22,26,24], our new methods demonstrate a significant improvement. Two previous approaches [22,26] are clearly outperformed by BE; they have been evaluated on the DaimlerChrysler benchmark class where BE returns the same or better results in less than one second. The third approach [24] builds up on an unpublished preliminary version [23] of our algorithm described in Section 2. In particular, the authors apply genetic algorithms in order to find linear vertex orderings that lead to hypertree decompositions of smaller width. In this way they are able to slightly improve our results for ten of our hypergraph examples; for other examples they get the same or even worse results. However,

Table 1. Experimental results obtained by Bucket Elimination (BE, DBE), Fiduccia-Mattheyses (FM), Tabu Search (TS), and hMETIS (HM)

	BE		DBE		FM (W1)		TS (W2)		HM (W2)		HM (best)	
Instance (Vertices/Edges)	width	time	width	time	width	time	width	time	width	time	width	time
adder_15 (106/76)	**2**	0s	**2**	0s	**2**	0s	4	0s	**2**	3s	**2**	3s
adder_25 (176/126)	**2**	0s	**2**	0s	**2**	1s	4	0s	**2**	7s	**2**	6s
adder_50 (351/251)	**2**	0s	**2**	0s	**2**	6s	4	1s	**2**	13s	**2**	12s
adder_75 (526/376)	**2**	0s	**2**	0s	**2**	21s	5	2s	**2**	21s	**2**	19s
adder_99 (694/496)	**2**	0s	**2**	0s	**2**	53s	5	3s	**2**	28s	**2**	25s
bridge_15 (137/137)	**3**	0s	**3**	0s	8	1s	8	1s	4	7s	**3**	6s
bridge_25 (227/227)	**3**	0s	**3**	0s	13	1s	6	1s	4	11s	**3**	11s
bridge_50 (452/452)	**3**	0s	**3**	0s	29	5s	10	3s	4	24s	4	22s
bridge_75 (677/677)	**3**	0s	**3**	0s	44	10s	10	5s	4	39s	**3**	35s
bridge_99 (893/893)	**3**	0s	**3**	0s	64	18s	10	7s	4	48s	4	45s
NewSystem1 (142/84)	**3**	0s	**3**	0s	4	1s	6	1s	4	5s	**3**	5s
NewSystem2 (345/200)	**4**	0s	**4**	0s	9	2s	6	2s	**4**	14s	**4**	13s
NewSystem3 (474/278)	**5**	0s	**5**	0s	17	4s	11	4s	**5**	19s	**5**	18s
NewSystem4 (718/418)	**5**	0s	**5**	0s	22	8s	12	7s	**5**	31s	**5**	29s
atv_partial_system (125/88)	**3**	0s	4	0s	4	0s	5	1s	4	6s	4	6s
NASA (579/680)	**21**	0s	56	13s	56	20s	98	34s	33	90s	32	84s
c432 (196/160)	**9**	0s	**9**	0s	15	3s	24	4s	13	20s	12	19s
c499 (243/202)	**13**	0s	20	0s	18	3s	27	5s	18	30s	17	28s
c880 (443/383)	**19**	0s	25	0s	31	8s	41	7s	29	50s	25	46s
c1355 (587/546)	**13**	0s	22	0s	32	10s	55	14s	22	66s	22	61s
c1908 (913/880)	32	0s	33	1s	65	23s	70	26s	**29**	86s	**29**	77s
c2670 (1350/1193)	**33**	0s	35	1s	66	56s	78	46s	38	119s	38	106s
c3540 (1719/1669)	**63**	2s	73	11s	97	133s	129	104s	73	166s	73	149s
c5315 (2485/2307)	**44**	3s	61	24s	120	250s	157	157s	72	242s	68	214s
c6288 (2448/2416)	**41**	10s	45	77s	148	478s	329	245s	45	210s	45	186s
c7552 (3718/3512)	37	4s	**35**	8s	161	514s	188	351s	37	365s	37	309s
s27 (17/13)	**2**	0s	**2**	0s	**2**	0s	3	0s	**2**	0s	**2**	0s
s208 (115/104)	**7**	0s	**7**	0s	**7**	1s	11	1s	**7**	10s	**7**	9s
s298 (139/133)	**5**	0s	8	0s	7	1s	17	2s	7	11s	6	10s
s344 (184/175)	**7**	0s	8	0s	8	2s	12	2s	8	21s	**7**	19s

Table 1. (*continued*)

Instance (Vertices/Edges)	BE		DBE		FM (W1)		TS (W2)		HM (W2)		HM (best)	
	width	time	width	time	width	time	width	time	width	time	width	time
s349 (185/176)	**7**	0s	9	0s	8	1s	12	2s	9	21s	**7**	19s
s382 (182/179)	**5**	0s	8	0s	7	2s	17	2s	8	16s	7	15s
s386 (172/165)	**8**	0s	15	0s	13	2s	26	3s	11	16s	11	15s
s400 (186/183)	**6**	0s	8	0s	8	2s	18	2s	8	18s	7	17s
s420 (231/212)	**9**	0s	**9**	0s	10	2s	14	2s	10	29s	10	24s
s444 (205/202)	**6**	0s	8	0s	8	2s	25	3s	8	21s	8	20s
s510 (236/217)	**23**	0s	31	0s	**23**	4s	41	4s	27	27s	27	25s
s526 (217/214)	**8**	0s	13	0s	13	2s	32	3s	11	29s	11	27s
s641 (433/398)	**7**	0s	14	0s	19	5s	21	5s	14	31s	14	28s
s713 (447/412)	**7**	0s	13	0s	21	5s	25	5s	14	33s	14	31s
s820 (312/294)	**12**	0s	27	2s	23	8s	77	9s	24	42s	19	38s
s832 (310/292)	**12**	0s	28	2s	22	8s	71	10s	26	42s	20	39s
s838 (457/422)	**15**	0s	**15**	0s	19	6s	24	6s	**15**	55s	**15**	46s
s953 (440/424)	**39**	0s	53	1s	50	18s	70	12s	45	52s	45	47s
s1196 (561/547)	**34**	0s	53	2s	50	22s	73	15s	43	69s	43	62s
s1238 (540/526)	**34**	0s	56	3s	56	20s	80	18s	43	66s	43	59s
s1423 (748/731)	**18**	0s	22	1s	29	25s	54	19s	27	78s	26	71s
s1488 (667/659)	**23**	0s	77	26s	45	46s	148	36s	39	85s	39	77s
s1494 (661/653)	**23**	0s	78	27s	49	45s	150	38s	38	85s	36	77s
s5378 (2993/2958)	**87**	7s	108	23s	178	308s	169	271s	89	279s	89	246s
b01 (47/45)	**5**	0s	6	0s	**5**	0s	10	0s	**5**	2s	**5**	2s
b02 (27/26)	**3**	0s	5	0s	4	0s	7	0s	4	1s	4	1s
b03 (156/152)	**7**	0s	11	0s	11	1s	16	2s	9	15s	8	14s
b04 (729/718)	**26**	0s	39	2s	44	26s	69	26s	38	82s	35	73s
b05 (962/961)	**18**	0s	29	1s	42	33s	70	30s	32	114s	32	99s
b06 (50/48)	**5**	0s	6	0s	**5**	0s	12	0s	**5**	3s	**5**	2s
b07 (433/432)	**19**	0s	29	0s	38	7s	59	10s	33	57s	31	54s
b08 (179/170)	**10**	0s	13	0s	14	2s	20	2s	12	21s	12	19s
b09 (169/168)	**10**	0s	12	0s	13	2s	20	2s	12	20s	12	19s
b10 (200/189)	**14**	0s	18	0s	17	3s	33	3s	16	24s	16	21s
b11 (764/757)	**31**	0s	47	3s	65	28s	98	27s	38	89s	38	79s
b12 (1070/1065)	**27**	0s	39	3s	38	55s	83	39s	34	111s	34	102s
b13 (352/342)	9	0s	10	0s	10	4s	17	4s	**8**	36s	**8**	33s
grid2d_10 (50/50)	**5**	0s	6	0s	**5**	0s	8	0s	**5**	3s	**5**	3s
grid2d_15 (113/112)	**9**	0s	**9**	0s	10	1s	12	1s	10	11s	10	10s
grid2d_20 (200/200)	12	0s	**11**	0s	15	2s	18	2s	14	29s	12	28s
grid2d_25 (313/312)	**15**	0s	**15**	0s	18	5s	26	5s	**15**	50s	**15**	43s
grid2d_30 (450/450)	19	0s	20	0s	21	11s	29	8s	**16**	70s	**16**	58s
grid2d_35 (613/612)	23	0s	23	0s	30	20s	41	13s	**19**	87s	**19**	73s
grid2d_40 (800/800)	26	0s	25	0s	28	38s	41	20s	**22**	108s	**22**	91s
grid2d_45 (1013/1012)	31	1s	30	1s	40	58s	47	31s	**25**	130s	**25**	109s
grid2d_50 (1250/1250)	33	1s	32	1s	44	88s	52	41s	**28**	154s	**28**	130s
grid2d_60 (1800/1800)	42	2s	39	3s	55	203s	75	75s	**34**	209s	**34**	178s
grid2d_70 (2450/2450)	49	4s	47	4s	65	347s	65	119s	**41**	283s	**41**	239s
grid2d_75 (2813/2812)	52	6s	50	7s	70	504s	99	158s	**44**	324s	**44**	274s
grid3d_4 (32/32)	**6**	0s	**6**	0s	**6**	0s	12	0s	**6**	1s	**6**	1s
grid3d_5 (63/62)	9	0s	10	0s	**8**	1s	18	1s	11	4s	10	3s
grid3d_6 (108/108)	14	0s	14	0s	**12**	1s	25	2s	15	9s	14	9s
grid3d_7 (172/171)	20	0s	20	0s	18	2s	33	5s	19	27s	**16**	24s
grid3d_8 (256/256)	25	0s	27	0s	25	5s	44	9s	21	48s	**20**	40s
grid3d_9 (365/364)	34	0s	26	0s	34	9s	56	14s	**24**	67s	**24**	56s
grid3d_10 (500/500)	42	1s	40	1s	41	20s	67	26s	**31**	93s	**31**	77s
grid3d_11 (666/665)	52	1s	53	2s	40	36s	83	43s	**37**	119s	**37**	99s
grid3d_12 (864/864)	63	3s	62	3s	53	61s	98	66s	45	150s	**44**	127s
grid3d_13 (1099/1098)	78	5s	68	6s	60	107s	122	101s	**53**	186s	**53**	158s
grid3d_14 (1372/1372)	88	10s	93	10s	86	161s	176	162s	**69**	230s	**69**	196s
grid3d_15 (1688/1687)	104	15s	103	15s	93	253s	151	245s	**76**	278s	**76**	244s
grid3d_16 (2048/2048)	120	24s	131	24s	100	400s	174	328s	87	339s	**82**	303s
grid4d_3 (41/40)	**8**	0s	**8**	0s	**8**	0s	20	0s	9	2s	**8**	2s
grid4d_4 (128/128)	**17**	0s	18	0s	**17**	1s	40	3s	19	13s	18	12s
grid4d_5 (313/312)	35	0s	37	0s	32	8s	78	17s	**28**	58s	**28**	48s
grid4d_6 (648/648)	64	3s	71	2s	58	40s	140	67s	**47**	123s	**47**	106s
grid4d_7 (1201/1200)	109	14s	110	14s	89	134s	182	194s	74	229s	**71**	208s
grid4d_8 (2048/2048)	164	62s	166	62s	120	441s	310	581s	**107**	408s	**107**	393s
grid5d_3 (122/121)	**18**	0s	20	0s	**18**	1s	49	4s	20	11s	19	10s

Table 1. (*continued*)

	BE		DBE		FM (W1)		TS (W2)		HM (W2)		HM (best)	
Instance (Vertices/Edges)	width	time	width	time	width	time	width	time	width	time	width	time
grid5d_4 (512/512)	63	2s	68	2s	49	25s	137	56s	**46**	92s	**46**	78s
grid5d_5 (1563/1562)	161	42s	159	42s	118	280s	362	474s	**111**	328s	**111**	319s
clique_10 (45/10)	**5**	0s	**5**	0s	**5**	0s	6	0s	**5**	0s	**5**	0s
clique_15 (105/15)	**8**	0s	**8**	0s	12	0s	**8**	1s	**8**	1s	**8**	1s
clique_20 (190/20)	**10**	0s	**10**	0s	20	0s	11	3s	**10**	1s	**10**	1s
clique_25 (300/25)	**13**	0s	**13**	0s	25	0s	14	8s	**13**	2s	**13**	2s
clique_30 (435/30)	**15**	1s	**15**	0s	30	0s	16	16s	**15**	3s	**15**	3s
clique_35 (595/35)	**18**	2s	**18**	0s	35	0s	19	29s	**18**	5s	**18**	5s
clique_40 (780/40)	**20**	5s	**20**	0s	40	0s	22	51s	**20**	6s	**20**	6s
clique_45 (990/45)	**23**	10s	**23**	0s	45	1s	24	80s	**23**	10s	**23**	9s
clique_50 (1225/50)	**25**	19s	**25**	0s	50	1s	28	145s	**25**	15s	**25**	13s
clique_60 (1770/60)	**30**	60s	**30**	0s	60	2s	34	340s	59	1s	50	1s
clique_70 (2415/70)	**35**	155s	**35**	0s	70	4s	39	601s	68	2s	67	1s
clique_75 (2775/75)	**38**	239s	**38**	0s	75	6s	41	920s	71	3s	71	2s
clique_80 (3160/80)	**40**	350s	**40**	0s	80	8s	43	1248s	76	4s	72	3s
clique_90 (4005/90)	**45**	724s	**45**	2s	90	12s	50	2045s	89	8s	78	5s
clique_99 (4851/99)	**50**	1280s	**50**	4s	99	19s	54	2845s	99	14s	97	8s

the runtime in their approach increases drastically. For instance, the authors obtained a hypertree decomposition of width 19 (-2) for the NASA example but the runtime to obtain this result was more than 17 hours.

5 Conclusion

In this paper we presented two generic heuristic approaches for the construction of generalized hypertree decompositions. The first one is based on tree decomposition and set cover heuristics, and the second one is based on hypergraph partitioning heuristics. We evaluated our algorithms empirically on a variety of benchmark examples and could show that our approaches clearly improve previous approaches for this problem. Future research is to enhance our techniques in order to obtain hypertree decompositions of smaller width but without increasing the runtime of the algorithms excessively.

Acknowledgments. This research was supported by the Austrian Science Fund (FWF), project P17222-N04. The work of Tobias Ganzow and Ben McMahan has been carried out while visiting the Technische Universität Wien. The visit of Tobias Ganzow was funded by the Wolfgang Pauli Institute (WPI).

References

1. Bodlaender, H.L.: Discovering treewidth. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 1–16. Springer, Heidelberg (2005)
2. Chekuri, C., Rajaraman, A.: Conjunctive query containment revisited. Theoretical Computer Science 239(2), 211–229 (2000)
3. DBAI Hypertree Project website, Vienna University of Technology, `http://www.dbai.tuwien.ac.at/proj/hypertree/`
4. Dechter, R.: Constraint networks. In: Encyclopedia of Artificial Intelligence, 2nd edn., vol. 1, pp. 276–285. Wiley and Sons, Chichester (1992)

5. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
6. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence 38(3), 353–366 (1989)
7. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proc. of the 19th Conference on Design Automation (DAC 1982), pp. 175–181. IEEE Press, Los Alamitos (1982)
8. Ganzow, T., Gottlob, G., Musliu, N., Samer, M.: A CSP hypergraph library. Technical Report, DBAI-TR-2005-50, Vienna University of Technology (2005)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Co., New York (1979)
10. Glover, F., Laguna, M.: Tabu search. Kluwer Academic Publishers, Dordrecht (1997)
11. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence 124(2), 243–282 (2000)
12. Gottlob, G., Leone, N., Scarcello, F.: The complexity of acyclic conjunctive queries. Journal of the ACM 48(3), 431–498 (2001)
13. Gottlob, G., Leone, N., Scarcello, F.: Hypertree decomposition and tractable queries. Journal of Computer and System Sciences 64(3), 579–627 (2002)
14. Gottlob, G., Miklós, Z., Schwentick, T.: Generalized hypertree decompositions: NP-hardness and tractable variants. In: Proc. of the 26th ACM Symposium on Principles of Database Systems (PODS 2007), pp. 13–22. ACM Press, New York (2007)
15. Gottlob, G., Samer, M.: A backtracking-based algorithm for hypertree decomposition. ACM Journal of Experimental Algorithmics (to appear)
16. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. Artificial Intelligence 66(1), 57–89 (1994)
17. Gyssens, M., Paredaens, J.: A decomposition methodology for cyclic databases. In: Gallaire, H., Nicolas, J.-M., Minker, J. (eds.) Advances in Data Base Theory, vol. 2, pp. 85–122. Plemum Press, New York (1984)
18. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multilevel hypergraph partitioning: Applications in VLSI domain. IEEE Transactions on Very Large Scale Integration Systems 7(1), 69–79 (1999)
19. Karypis, G., Kumar, V.: hMetis: A hypergraph partitioning package, version 1.5.3 (1998)
20. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing 48(1), 96–129 (1998)
21. Karypis, G., Kumar, V.: Multilevel k-way hypergraph partitioning. In: Proc. of the 36th ACM/IEEE Conference on Design Automation (DAC 1999), pp. 343–348. ACM Press, New York (1999)
22. Korimort, T.: Heuristic Hypertree Decomposition. PhD thesis, Vienna University of Technology (2003)
23. McMahan, B.: Bucket eliminiation and hypertree decompositions. Implementation Report, Database and AI Group, Vienna University of Technology (2004)
24. Musliu, N., Schafhauser, W.: Genetic algorithms for generalized hypertree decompositions. European Journal of Industrial Engineering 1(3), 317–340 (2007)
25. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. Journal of Algorithms 7, 309–322 (1986)
26. Samer, M.: Hypertree-decomposition via branch-decomposition. In: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 1535–1536. Professional Book Center (2005)
27. Samer, M., Szeider, S.: Complexity and applications of edge-induced vertex-cuts. Technical Report, arXiv:cs.DM/0607109 (2006)
28. Yannakakis, M.: Algorithms for acyclic database schemes. In: Proc. of the 7th International Conference on Very Large Data Bases (VLDB 1981), pp. 81–94. IEEE Press, Los Alamitos (1981)