Genetic algorithms for generalized hypertree decompositions

Nysret Musliu and
Werner Schafhauser*

Institute of Information Systems,
Database and Artificial Intelligence Group (DBAI),
Vienna University of Technology,
Favoritenstraße 9, 1040 Wien, Austria
Fax: +43 (1) 58801 18492
E-mail: musliu@dbai.tuwien.ac.at
E-mail: schafha@dbai.tuwien.ac.at
*Corresponding author

Abstract: Many practical problems in mathematics and computer science may be formulated as constraint satisfaction problems (CSPs). Although CSPs are $\mathcal{NP}$-hard in general, it has been proven that instances of CSPs may be solved efficiently, if they have generalized hypertree decompositions of small width. Unfortunately, finding a generalized hypertree decomposition of minimum width is an $\mathcal{NP}$-hard problem. Based on a genetic algorithm (GA) for tree decompositions we propose two extensions searching for small-width generalized hypertree decompositions. We carry out comprehensive experiments in order to obtain suitable operators and parameter settings and apply each genetic algorithm to numerous benchmark examples for tree and generalized hypertree decompositions. Compared to the best solutions known from literature our GAs were able to return results of equal quality for many benchmark instances and even for some benchmarks improved solutions were obtained.

Keywords: constraint satisfaction problems; structural decomposition methods; tree decompositions; generalized hypertree decompositions; genetic algorithms.

Reference to this paper should be made as follows: Nysret Musliu and Werner Schafhauser, 'Genetic algorithms for generalized hypertree decompositions', *European Journal of Industrial Engineering*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Nysret Musliu is an Assistant Professor in the Database and Artificial Intelligence Group, Vienna University of Technology, Austria. He received his Ph.D. in Computer Science from the Vienna University of Technology and his master's degree in Computer Science and Telecommunication from the University of Prishtina, Kosova. His research interests include problem solving and search in artificial intelligence, metaheuristic techniques, constraint satisfaction, hypertree and tree decompositions, scheduling, and other combinatorial optimization problems.

Werner Schafhauser obtained his master's degree in computer science from the Vienna University of Technology in 2006. He is currently a PhD student at the Database and Artificial Intelligence Group at the Vienna University of Technology. His research interests are constraint satisfaction problems, structural decompositions methods and meta-heuristic optimization.

1 Introduction

In mathematics and computer science, especially in the fields of operations research and artificial intelligence, many important real-world problems may be modeled as *constraint satisfaction problems* (CSPs). For instance, boolean satisfiability problems, scheduling problems, boolean conjunctive queries, the graph k-colorability problem and many other interesting tasks possess a representation as CSP. The main advantage of CSPs is that they represent a very general class, meaning that all methods for solving CSPs will automatically solve problems which possess a formulation as CSP. However, CSPs are $\mathcal{NP}$-hard in general, implying that all known algorithms which are able to solve CSPs require exponential running time in the worst case.

Formally speaking, a constraint satisfaction problem is a triple $\langle X, D, C \rangle$, where X is a set of *variables*, D is a collection of *domains* for each variable, and C is a set of *constraints*. Each constraint in C is defined over a subset of X, its *scope*, and specifies the value combinations which are allowed to be assigned to the variables in its scope. A solution for a CSP is an assignment of domain values to all variables which is consistent with all constraints.

The structure of a CSP instance is represented by its *constraint hypergraph*. Given a CSP instance the corresponding constraint hypergraph is obtained by introducing a vertex for each variable in X and for each constraint in C a hyperedge is introduced containing the vertices which correspond to the variables within the scope of the constraint.

Example 1. A CSP instance is given below. Each constraint C_i, is written as a pair consisting of the variables over which it is defined (its scope) and of the constraint relation R_i specifying the allowed value combinations for those variables. The constraint hypergraph obtained from that CSP instance is shown in Figure 1.

Variables:	$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$
Domains:	$D = \{D_{x_1}, D_{x_2}, D_{x_3}, D_{x_4}, D_{x_5}, D_{x_6}\}$
	$D_{x_1} = \{a, b\}$, $D_{x_2} = D_{x_3} = ... = D_{x_6} = \{b, c\}$
Constraints:	$C = \{C_1, C_2, C_3\}$
	$C_1 = \langle\{x_1, x_2, x_3\}, R_1\rangle$
	$C_2 = \langle\{x_1, x_5, x_6\}, R_2\rangle$
	$C_3 = \langle\{x_3, x_4, x_5\}, R_3\rangle$
	$R_1 = \{(a, b, c), (a, c, b), (b, b, c)\}$
	$R_2 = \{(a, b, c), (a, c, b)\}$
	$R_3 = \{(c, b, c), (c, c, b)\}$
Solution:	$x_1 = a, x_2 = b, x_3 = c, x_4 = b, x_5 = c, x_6 = b$

It is well known that acyclic CSPs may be recognized and solved efficiently. Decomposition methods may be used for identifying and solving tractable classes of CSPs by exploiting the structure of the constraint hypergraph. Given a CSP instance I, a decomposition method transforms I into a solution-equivalent and acyclic CSP instance I'. If I' can be found in time polynomial in the size of I and if also the size of the largest constraint relation in I' is polynomial in the size of I we may solve I' in polynomial time since I' is acyclic (Gottlob et al. (2000)). Many decomposition methods use a measure called *width* in order to bound the size of the largest constraint relation in I'. Informally speaking, the smaller the width of decomposition of the constraint hypergraph is, the faster the corresponding CSP may be solved.

The concept of *generalized hypertree decompositions* was proposed by Gottlob et al. (2001). Given a (constraint) hypergraph, the minimum width over all its possible generalized hypertree decompositions is defined as the *generalized hypertree-width* of the hypergraph. In order to solve a CSP efficiently, we tend to find a generalized hypertree decomposition of width near or equal to the generalized hypertree-width. Unfortunately, the problem of deciding whether there exists a generalized hypertree decomposition of a hypergraph of width at most k is known to be $\mathcal{NP}$-complete (Gottlob et al. (2007)).

McMahan (2004) combined well-known *tree decomposition* and minimum set covering heuristics in order to obtain generalized hypertree decompositions. Tree decompositions, which were introduced in Robertson and Seymour (1986), represent another decomposition method and have been subject to research during the last decades.

Larrañaga et al. (1997) proposed a genetic algorithm for decomposing the moral graph of Bayesian networks, a problem strongly related to tree decompositions. They applied their algorithm to two benchmark networks and observed that their genetic approach returned competitive results when compared to other existing heuristic methods for decomposing Bayesian networks. The genetic algorithm in Larrañaga et al. (1997) tried to minimize a weight associated with the decompositions of Bayesian networks which is not exactly the same as the width of tree decompositions.

In this paper we implemented a genetic algorithm, named *GA-tw*, searching for small-width tree decompositions based on the GA presented by Larrañaga et al. (1997). Unlike the algorithm proposed by Larrañaga et al. (1997) we used the width of tree decompositions as objective function. We did a series of experiments in order to obtain suitable operators and parameter settings for *GA-tw* and we applied *GA-tw* to benchmark instances from the Second DIMACS graph coloring challenge (Johnson and Trick (1993)). Furthermore we present two extensions of *GA-tw*, named *GA-tw+* and *GA-ghw*, which are used to minimize the width of generalized hypertree decompositions of hypergraphs. *GA-tw+* follows McMahan's approach in order to transform the tree decompositions in *GA-tw*'s final population into generalized hypertree decompositions. *GA-ghw* uses the width in terms of generalized hypertree decompositions as fitness functions and looks already during the genetic search process for small-width generalized hypertree decompositions. *GA-tw+* and *GA-ghw* were evaluated on benchmark hypergraphs from industry and literature described in Ganzow et al. (2005).

All three genetic algorithms presented in this paper were able return comparable results for tree and generalized hypertree decompositions for many benchmark instances when compared to the best results in literature. Moreover, for some benchmarks the genetic algorithms were able to return improved upper bounds on their treewidth and generalized hypertree-width respectively.

This paper is organized as follows: Section 2 gives the basic definitions used in this paper. Afterwards, in Section 3, we summarize already existing methods for generalized hypertree decompositions and related concepts. Next, in Section 4 we describe how tree decompositions and generalized hypertree decompositions may be obtained via a method called *bucket elimination* and vertex orderings. In Section 5 we explain the basics of genetic algorithms and in Section 6 we describe our genetic algorithms for tree and generalized hypertree decompositions. Section 7 presents our computational results. Finally, we conclude in Section 8.

2 Preliminaries

Definition 1 (Hypergraph). A *hypergraph* is a structure $\mathcal{H} = (V, H)$ that consists of vertices $V = \{v_1, ..., v_n\}$ and a set of subsets of these vertices $H = \{h_1, ..., h_m\}$, $h_i \subseteq V$, called hyperedges. W.l.o.g. we assume that each vertex is contained in at least one hyperedge. Hyperedges differ from edges of regular graphs in that they may be defined over more than two vertices. Note that every regular graph may be regarded as a hypergraph whose hyperedges connect two vertices.

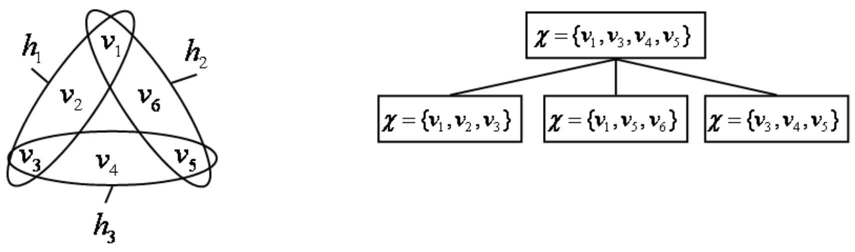
Definition 2 (Tree Decomposition). Let $\mathcal{H} = (V, H)$ be a hypergraph. A *tree for a hypergraph* $\mathcal{H}$ is a pair $\langle T, \chi \rangle$ where $T = (N, E)$ is a rooted tree. We define $vertices(T) = N$ and refer to the vertices of T as "nodes" in order to avoid confusion with the vertices in $\mathcal{H}$. χ is a labeling function which associates to each node $p \in vertices(T)$ the set $\chi(p) \subseteq V$. A *tree decomposition* of a hypergraph $\mathcal{H}$ is a tree $\langle T, \chi \rangle$ for $\mathcal{H}$ which satisfies the following two conditions:

1. for each hyperedge $h \in H$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$.

2. for each vertex $v \in V$ the set $\{p \in vertices(T) \mid v \in \chi(p)\}$ induces a (connected) subtree of T (connectedness condition).

The *width* of a tree decomposition $\langle T, \chi \rangle$ is $max_{p \in vertices(T)}|\chi(p) - 1|$. The *treewidth* of $\mathcal{H}$, abbreviated $tw(\mathcal{H})$, is the minimum width over all its tree decompositions.

Tree decompositions were originally defined by Robertson and Seymour (1986). Since every graph may be regarded as a hypergraph with two vertices in each of its hyperedges, Definition 2 covers the definition for tree decompositions of graphs and extends the concept of tree decompositions onto arbitrary hypergraphs.



h_1
h_2
h_3
v_1
v_2
v_3
v_4
v_5
v_6
$\chi = \{v_1, v_3, v_4, v_5\}$
$\chi = \{v_1, v_2, v_3\}$
$\chi = \{v_1, v_5, v_6\}$
$\chi = \{v_3, v_4, v_5\}$

Figure 1 Example of a hypergraph and a possible tree decomposition of *width=3*.

Definition 3 (Generalized Hypertree Decomposition). Let $\mathcal{H} = (V, H)$ be a hypergraph. A *generalized hypertree decomposition* of a hypergraph $\mathcal{H} = (V, H)$ is a triple $\langle T, \chi, \lambda \rangle$ where $\langle T, \chi \rangle$ is a tree decomposition of $\mathcal{H}$ and λ is a labeling function which associates to each node $p \in vertices(T)$ the set $\lambda(p) \subseteq H$. $\langle T, \chi, \lambda \rangle$ satisfies the following additional condition:

- for each $p \in vertices(T)$, $\chi(p) \subseteq \bigcup \lambda(p)$.

The *width* of a gen. hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)}|\lambda(p)|$. The *generalized hypertree-width* of $\mathcal{H}$, abbreviated $ghw(\mathcal{H})$, is the minimum width over all its generalized hypertree decompositions.

According to the previous definition a generalized hypertree decomposition of a hypergraph $\mathcal{H}$ is a tree decomposition of $\mathcal{H}$ at the same time. The additional condition says that in each node p of the generalized hypertree decomposition each vertex in the set $\chi(p)$ must be contained by at least one hyperedge in the set $\lambda(p)$.

3 Algorithms for generalized hypertree decompositions and related concepts

Recently, many methods have been proposed for the generation of (generalized) hypertree decompositions. Whereas the generation of generalized hypertree decompositions of optimal width is $\mathcal{NP}$-hard, for fixed k, deciding whether there exists a hypertree decomposition of width at most k, can be done in polynomial time. Note, that a hypertree decomposition is a generalized hypertree decomposition that includes an additional condition (fourth condition). That special condition forbids variables that disappear from the set χ in some node p of decomposition hypertree

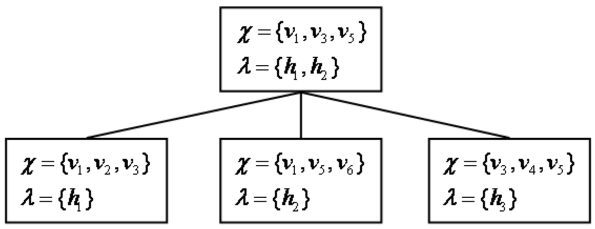


Figure 2 Generalized hypertree decomposition for the hypergraph in Figure 1 of *width=2.*

to appear again in the subtrees rooted at p. The exact algorithm *opt-k-decomp* for hypertree decompositions, has been developed by Gottlob et al. (1999). For fixed k this algorithm, decides in polynomial time whether a hypergraph has k-bounded hypertree-width and, in this case, computes an optimal hypertree decomposition in normal form. Two implementations of *opt-k-decomp* are used successfully for the generation of hypertree decompositions of small instances of CSPs. However, for larger and important practical cases, the exact algorithm is not practical and runs out of time and space. Gottlob and Samer (2007) proposed the backtracking-based algorithm *det-k-decomp* for generating hypertree decompositions. Compared to *opt-k-decomp* this algorithm could be used for larger instances and it gives also competitive results with other heuristic based approaches, for hypergraphs that are not too large. Another backtracking algorithm has been recently proposed by Subbarayan and Andersen (2007). Although both proposed backtracking algorithms improve significantly the previous exact algorithms, for larger problems the only alternative for generating hypertree decompositions are heuristic methods. Moreover, the hypertree-width of a hypergraph can be larger than the generalized hypertree-width, $hw \leq 3ghw + 1$ (Adler et al. (2005)), which also justifies the use of heuristic methods.

Different heuristic methods have been proposed for generation of generalized hypertree decompositions. In Korimort (2003) a heuristic method is proposed which is 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 by Samer (2005). 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. An extension of algorithms for tree decompositions to generate generalized hypertree decompositions is proposed in Dermaku et al. (2005). In this paper the authors propose also new methods for generating hypertree decompositions based on hypergraph partitioning algorithms. To our best knowledge the methods proposed in Dermaku et al. (2005) and Gottlob and Samer (2007) give the best results yet in the literature for CSP hypergraph library examples from Ganzow et al. (2005). In general the proposed methods in literature are able to generate good generalized hypertree decompositions for different instance sizes. As previously mentioned the generalized hypertree decompositions can be generated from tree decompositions, and thus methods for the generation of tree decompositions can also be used for obtaining generalized hypertree decompositions.

Several complete and heuristic algorithms have been proposed in literature in order to generate tree decompositions. Examples of complete algorithms for tree decompositions are Shoikhet (1997) and Gogate and Dechter (2004). Gogate and Dechter (2004) reported good results for tree decompositions by using the branch and bound algorithm. They showed that their algorithm is superior compared to the algorithm proposed in Shoikhet (1997). The branch and bound algorithm proposed in Gogate and Dechter (2004) applies different pruning techniques, and provides anytime solutions, which are good upper bounds for tree decompositions. Heuristic techniques for generation of tree decompositions with small width are mainly based on searching for a good ordering of graph vertices. Several heuristics that run in polynomial time have been proposed for finding good vertex orderings. These heuristics select the ordering of vertices based on different criteria, such as the degree of the vertices, the number of edges to be added to make the vertex simplicial etc. Maximum Cardinality Search (MCS) proposed by Tarjan and Yannakakis Tarjan and Yannakakis (1984) constructs the ordering of vertices iteratively by picking the next vertex which has the largest number of neighbors in the ordering (the ties are broken randomly). The min-fill heuristics picks iteratively the vertex which adds the smallest number of edges when eliminated. The min-degree heuristic picks the next vertex to be eliminated based on its degree. According to Gogate and Dechter (2004) the min-fill heuristic performs better than MCS and the min-degree heuristic. The min-degree heuristic has been improved Clautiaux et al. (2004) by adding a new criterion based on the lower bound of the treewidth for the graph obtained when the vertex is eliminated. For other types of vertex-ordering-based heuristics see Koster et al. (2001). Metaheuristic approaches have also been used for tree decompositions. Simulated annealing was used by Kjaerulff (1992). A genetic algorithm for decomposing Bayesian networks, a problem strongly related to tree decompositions, is presented in Larrañaga et al. (1997). A tabu search approach for tree decompositions has been proposed by Clautiaux et al. (2004). The authors reported good results for DIMACS vertex coloring instances (Johnson and Trick (1993)). Their approach improved the previous results in literature for 53% of instances. Some of the results in Clautiaux et al. (2004) have been further improved by Gogate and Dechter (2004). Recently, an iterated local search algorithm has been proposed for tree decompositions by Musliu (2007). This algorithms gives competitive results to previous proposed approaches and could improve many upper bounds for the DIMACS vertex coloring instances (Johnson and Trick (1993)). The reader is referred to Bodlaender (2005) for other approximation algorithms, and the information about lower bounds algorithms.

4 Bucket elimination

In McMahan (2004) it is shown how a method originating from constraint satisfaction named *bucket elimination* Dechter (2003) may be used for the creation of tree decompositions and generalized hypertree decompositions.

Given a hypergraph $\mathcal{H} = (V, H)$ and a vertex ordering $\sigma = (v_1, ..., v_n)$ of the hypergraph's vertices, algorithm *bucket elimination* (Figure 3) returns a tree decomposition for $\mathcal{H}$. Initially, the algorithm creates a bucket for each vertex of the hypergraph and puts the vertices of each hyperedge into the bucket of the maximum vertex within this hyperedge (the maximum vertex of $V' \subseteq V$ is the vertex with the highest index according to σ). Afterwards the buckets are processed in order given by σ. When processing bucket B_{v_i}, we compute the set $A := \chi(B_{v_i}) - \{v_i\}$. A is copied to bucket B_{v_j} of its maximum vertex v_j. Additionally B_{v_i} and B_{v_j} are connected by an edge. Finally we get a tree decomposition, where the buckets and the introduced edges act as a tree and the contents of the buckets represent the vertices within the χ-sets.

Algorithm: *bucket elimination*

```
Input:   a hypergraph $\mathcal{H} = (V, H)$
         a vertex ordering $\sigma = (v_1, ..., v_n)$ of the vertices in $V$
Output:  a tree decomposition $\langle T, \chi \rangle$ for $\mathcal{H}$

1. initially $B = \emptyset$, $E = \emptyset$
   **for each** vertex $v_i$ introduce an empty bucket $B_{v_i}$, $\chi(B_{v_i}) := \emptyset$
2. fill the buckets $B_{v_1}, ..., B_{v_n}$ as follows:
   **for each** hyperedge $h \in H$
     let $v \in h$ be the maximum vertex of $h$ according to $\sigma$
     $\chi(B_v) := \chi(B_v) \cup h$
3. **for** $i = n$ **to** 2 **do**
     let $A := \chi(B_{v_i}) - \{v_i\}$
     let $v_j \in A$ be the next vertex following $v_i$ in $\sigma$
     $\chi(B_{v_j}) := \chi(B_{v_j}) \cup A$
     $E := E \cup (B_{v_i}, B_{v_j})$
4. **return** $\langle (B, E), \chi \rangle$, where $B = \{B_{v_1}, ..., B_{v_n}\}$
```

Figure 3 Algorithm *bucket elimination* (McMahan (2004)).

In addition McMahan (2004) showed how *bucket elimination* can be extended in order to obtain generalized hypertree decompositions. The main idea behind his approach is that every generalized hypertree decomposition $\langle T, \chi, \lambda \rangle$ may be considered as a tree decomposition which satisfies an additional property:

- for each $p \in vertices(T), \chi(p) \subseteq \bigcup \lambda(p)$.

Thus, every tree decomposition may be transformed into a generalized hypertree decomposition by attaching hyperedges to decomposition nodes until the additional property is satisfied. In order to keep the width of the resulting generalized hypertree decomposition small we have to attach as few hyperedges as possible to each tree decomposition node. This task may be described as a series of minimum set cover problems (Karp (1972)). The minimum set cover problem may be formulated as minimization problem as follows:

Given	$T = \{t_1, ..., t_n\}$	a set
	$S = \{S_1, ..., S_m\}$, $\forall i : S_i \subseteq T$	a collection of subsets of T

Find a subcollection $C \subseteq S$, such that C covers all elements of T and $|C|$ is minimal. A minimum set cover problem instance is represented as a pair $\langle T, S \rangle$.

When transforming tree decompositions into generalized hypertree decompositions, for each decomposition node p we have to solve the minimum set cover problem $\langle \chi(p), H_{\chi(p)} \rangle$, where $H_{\chi(p)}$ is obtained from H by deleting all vertices from the hyperedges in H which are not in $\chi(p)$. The obtained cover acts as $\lambda(p)$.

Minimum set cover itself is an $\mathcal{NP}$-hard problem but it can be formulated as an IP-program (Schrijver (1996)) meaning that exact solutions for small and middle-size instances may be solved exactly by an IP-solver within a reasonable amount of time. Moreover there exists a greedy algorithm for the set cover problem (Chvatal (1979)) which in practice returns a close-to-optimal solution for many instances.

Fortunately, at least one vertex ordering will force algorithm *bucket elimination* to return a tree decomposition of minimal width (Kloks (1994)) and at least one vertex ordering will force algorithm *bucket elimination* to return a tree decomposition which may be transformed into a generalized hypertree decomposition of minimal width (Schafhauser (2006)), if the arising set cover problems are solved exactly. Therefore, the set of all vertex orderings for a hypergraph $\mathcal{H}$ may act as search space for both the treewidth and the generalized hypertree-width.

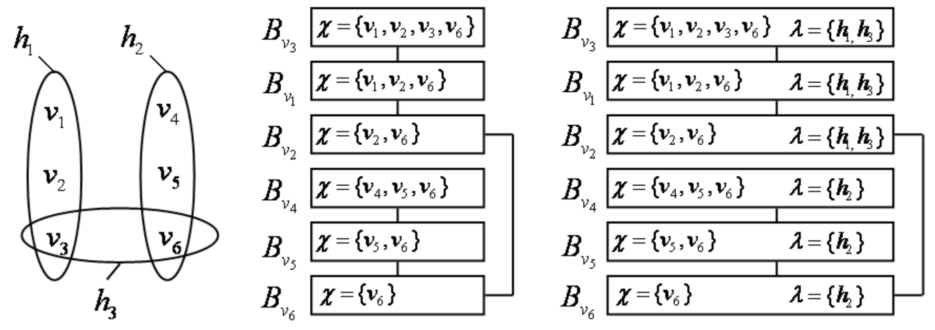

h_1
h_2
h_3
v_1
v_2
v_3
v_4
v_5
v_6
B_{v_3} $\chi = \{v_1, v_2, v_3, v_6\}$
B_{v_1} $\chi = \{v_1, v_2, v_6\}$
B_{v_2} $\chi = \{v_2, v_6\}$
B_{v_4} $\chi = \{v_4, v_5, v_6\}$
B_{v_5} $\chi = \{v_5, v_6\}$
B_{v_6} $\chi = \{v_6\}$
B_{v_3} $\chi = \{v_1, v_2, v_3, v_6\}$ $\lambda = \{h_1, h_3\}$
B_{v_1} $\chi = \{v_1, v_2, v_6\}$ $\lambda = \{h_1, h_3\}$
B_{v_2} $\chi = \{v_2, v_6\}$ $\lambda = \{h_1, h_3\}$
B_{v_4} $\chi = \{v_4, v_5, v_6\}$ $\lambda = \{h_2\}$
B_{v_5} $\chi = \{v_5, v_6\}$ $\lambda = \{h_2\}$
B_{v_6} $\chi = \{v_6\}$ $\lambda = \{h_2\}$

Figure 4 A hypergraph and the tree decomposition obtained via *bucket elimination* from the ordering $\sigma = (v_6, v_5, v_4, v_2, v_1, v_3)$. In the generalized hypertree decomposition (right) a minimum number of hyperedges was attached to each decomposition node.

5 Genetic algorithms

Genetic algorithms (GAs) were developed by Holland (1975). They try to find a good solution for an optimization problem by imitating the principle of evolution. Genetic algorithms alter and select individuals from a population of solutions for the optimization problem. In the following we describe frequently used terms within the field of genetic algorithms:

population	... set of candidate solutions
individual	... a single candidate solution
chromosome	... set of parameters determining the properties of a solution
gene	... single parameter

Figure 5 shows the structure of a genetic algorithm (Michalewicz and Fogel (2004)). A genetic algorithm tends to optimize the value of an objective function of an optimization problem, in terms of genetic algorithms also called *fitness function*. At the beginning a genetic algorithm creates an initial population containing randomly or heuristically created individuals. These individuals are evaluated and assigned a *fitness* value, which is the value of the fitness function for the solution represented by the individual. The population is evolved over a number of generations until a halting criterion is satisfied. In each generation the population undergoes *selection*, *recombination*, also denoted *crossover*, and *mutation*.

During the process of selection the genetic algorithm decides which individuals from the current population are allowed to enter the next population. This decision is based on the fitness value of the individuals and individuals of better fitness should enter the next population with higher probability than individuals of lower fitness. Not selected individuals are discarded and won't be evolved further.

The process of recombination or crossover combines different properties of several parent solutions within one or more children solutions, also denoted *offsprings*. Crossover exchanges properties between the individuals and should increase the average quality of the population.

During the process of mutation the individuals are slightly altered. Mutation is used to explore new regions of the search space and to avoid convergence to local optima.

In practice parameters are used in order to control the behavior of a genetic algorithm. Typical *control parameters* are mutation rate, crossover rate, population size and parameters for selection techniques. The choice of the control parameters has a crucial effect on the quality of the best solution found by a genetic algorithm.

```
Genetic Algorithm
$t = 0$
initialize population($t$)
evaluate population($t$)

while ¬terminated do
    $t = t + 1$
    select population($t$) from population($t - 1$)
    recombine population($t$)
    mutate population($t$)
    evaluate population($t$)
```

Figure 5 The structure of a genetic algorithm (Michalewicz and Fogel (2004)).

6 Genetic algorithms for tree decompositions and generalized hypertree decompositions

6.1 Algorithm GA-tw

First of all we implemented a genetic algorithm, named *GA-tw*, searching for the treewidth of hypergraphs. Figure 6 presents algorithm *GA-tw* in pseudo code notation.

The algorithm takes as input a hypergraph and several control parameters. Individual solutions are vertex orderings. Each individual is assigned the width of the tree decomposition returned by algorithm *bucket elimination* from the corresponding vertex ordering as its fitness value.

Initially *GA-tw* generates a population consisting of randomly created individuals. As selection technique we chose tournament selection. Tournament selection selects an individual by randomly choosing a group of several individuals from the former population. The individual of highest fitness (smallest width) within this group is selected to join the next population. This process is applied until enough individuals have entered the next population. Finally, after a certain number of generations, algorithm *GA-tw* will return the best fitness (smallest width) of an individual found during the search process.

Crossover and mutation operators

Within our genetic algorithms we implemented nearly all crossover operators and all mutation operators which were also applied in Larrañaga et al. (1997) for decomposing the moral graph of Bayesian networks.

```
Algorithm: GA-tw

  Input:    a hypergraph $\mathcal{H} = (H, V)$
            control parameters for the GA $n, p_m, p_c, s$ and $max\_gen$
  Output:   an upper bound on the treewidth of hypergraph $\mathcal{H}$

$t = 0$
initialize ($population(t), n$)
evaluate $population(t)$

while $t < max\_gen$ do
    $t = t + 1$
    $population(t)$ = tournament_selection($population(t-1), s$)
    recombine ($population(t), p_c$)
    mutate ($population(t), p_m$)
    evaluate $population(t)$

return the smallest width found during the search
```

Figure 6 Algorithm *GA-tw*.

Crossover operators:

- partially-mapped crossover (PMX), Goldberg and Lingle (1985)
- cycle crossover (CX), Oliver et al. (1987)

- order crossover (OX1), Davis (1985)
- order-based crossover (OX2), Syswerda (1991)
- position-based crossover (POS), Syswerda (1991)
- alternating-position crossover (AP), Larrañaga et al. (1994)

Mutation operators:

- displacement mutation operator (DM), e.g. Michalewicz (1992)
- exchange mutation operator (EM), e.g. Banzhaf (1990)
- insertion mutation operator (ISM), e.g. Michalewicz (1992)
- simple-inversion mutation operator (SIM), e.g Holland (1975)
- inversion mutation operator (IVM), e.g. Fogel (1990)
- scramble mutation operator (SM), e.g. Syswerda (1991)

In the following we describe the crossover and mutation operators which returned the best results in our computational evaluation of *GA-tw*.

Order Crossover (OX1)

The order crossover operator determines a crossover area within the parents by randomly selecting two positions within the ordering. The elements in the crossover area of the first parent are copied to the offspring. Starting at the end of the crossover area all elements outside the area are inserted in the same order in which they occur in the second parent.

Order-Based Crossover (OX2)

The order-based crossover operator selects at random several positions in the parent orderings by tossing a coin for each position. The elements of the first parent at these positions are deleted in the second parent. Afterwards they are reinserted in the order of the second parent.

Position-Based Crossover (POS)

The position-based crossover operator also starts with selecting a random set of positions in the parent strings by tossing a coin for each position. The elements at the selected positions are exchanged between the parents in order to create the offsprings. The elements missing after the exchange are reinserted in the order of the second parent.

Exchange Mutation Operator (EM)

The exchange mutation operator randomly selects two elements in the solution and exchanges them.

Insertion Mutation Operator (ISM)

The insertion mutation operator randomly chooses an element in the solution and moves it to a randomly selected position.

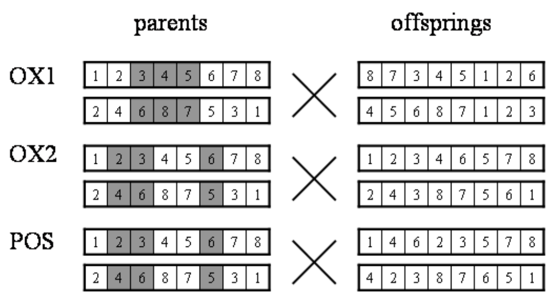


Figure 7 Selected crossover operators for vertex orderings.

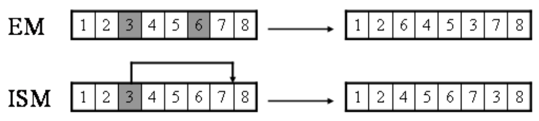


Figure 8 Selected mutation operators for vertex orderings.

6.2 Algorithm GA-tw+

In our second approach we extended algorithm *GA-tw* in order to obtain small-width generalized hypertree decompositions for a given hypergraph. The new algorithm is called *GA-tw+*. *GA-tw+* runs algorithm *GA-tw* as described above. In addition it creates generalized hypertree decompositions from the vertex orderings within the final population of *GA-tw*, as described in section 4. The arising set cover problems are solved exactly by the help of an IP-Solver. The smallest width of a computed generalized hypertree decomposition is returned as the result of algorithm *GA-tw+*.

6.3 Algorithm GA-ghw

Finally we implemented a genetic algorithm, named *GA-ghw*, which searches for small-width generalized hypertree decompositions already during the genetic search process. In contrast to algorithm *GA-tw*, algorithm *GA-ghw* assigns individuals the width of the generalized hypertree decomposition obtained from their corresponding vertex ordering as fitness value (see section 4).

Arising set cover problems are solved by the greedy set cover heuristic from Chvatal (1979). The heuristic successively takes a hyperedge containing most uncovered vertices until all vertices are covered. Ties are broken at random.

6.4 Implementation details

We implemented each genetic algorithm using C++ and STL. In algorithm *GA-tw+* we used the GNU Linear Programming Kit 4.9 (GLPK) as IP-solver to solve arising minimum set cover problems exactly.

6.5 Control parameters

All algorithms described above require the following control parameters: the population size n, the mutation rate p_m, the crossover rate p_c, the tournament selection group size s, and max_gen the number of generations over which the population will be evolved. The mutation rate p_m represents the probability that a single individual will be mutated within a generation of the GA whereas the crossover rate p_c specifies the fraction of the population which undergoes crossover within a single generation of the GA.

7 Computational results

7.1 Algorithm GA-tw

First of all we tried to find suitable values for the control parameters of algorithm *GA-tw*. Afterwards we applied algorithm *GA-tw* to 62 graphs of the Second DIMACS graph coloring challenge (Johnson and Trick (1993)), using the obtained parameter values. Each graph is considered as a hypergraph in which each hyperedge contains exactly two vertices.

Comparison of crossover and mutation operators

First of all we compared the crossover operators with each other by applying them to seven selected graphs of the Second DIMACS graph coloring challenge (Johnson and Trick (1993)).

For each crossover operator and each graph we ran algorithm *GA-tw* five times with the following parameter settings, n=50, p_m=0%, p_c=100%, s=2 and max_gen= 1000, meaning that we allowed solely the crossover operator to alter individual solutions. Table 1 shows the average width achieved during five runs with different crossover operators. Since position-based crossover (POS) achieved the best average width for all instances we chose it as the crossover operator for our further tests.

Instance/Operator	PMX	CX	OX1	OX2	POS	AP
games120	50.2	59.2	56.2	46.6	37.0	60.8
homer	72.8	98.0	118.4	53.8	42.2	143.8
inithx.i.3	331.8	368.0	321.6	204.4	129.8	370.2
le45025_d	391.8	394.2	396.2	375.8	370.0	401.6
myciel7	108.2	113.4	119.0	86.8	75.0	128.8
queen16_16	217.6	224.6	224.2	213.0	207.0	227.4
zeroin.i.3	98.0	99.4	93.0	51.4	40.2	101.4

Table 1 Comparison of crossover operators.

For a comparison of the mutation operators we ran algorithm *GA-tw* five times with the parameter settings, n=50, p_m=100%, p_c=0%, s=2 and max_gen=1000, allowing solely the mutation operator to alter individual solutions. Table 2 shows the average width achieved during five runs with different mutation operators. For a majority of instances the insertion mutation operator (ISM) returned the best average width, thus we used it as mutation operator in our further experiments.

Instance/Operator	DM	EM	ISM	SIM	IVM	SM
games120	54.0	38.2	37.4	49.8	56.4	48.8
homer	101.2	42.8	43.6	91.6	102.4	81.4
inithx.i.3	243.8	121.2	65.8	230.6	274.8	208.2
le450_25d	384.2	367.2	359.2	390.4	393.2	388.8
myciel7	110.8	78.4	70.4	106.2	113.4	99.6
queen16_16	217.6	209.0	202.4	222.6	220.2	220.2
zeroin.i.3	81.2	41.2	34.8	64.8	85.0	63.6

Table 2 Comparison of mutation operators.

Determining suitable crossover and mutation rates

We considered different combinations of the mutation rates, $p_m = 1\%, 10\%, 30\%$, and the crossover rates, $p_c = 80\%, 90\%, 100\%$, and applied algorithm *GA-tw* using those combinations to selected instances of the Second DIMACS graph coloring challenge (Johnson and Trick (1993)). For each combination and each graph we ran algorithm *GA-tw* five times with the parameter settings, n=200, s=2 and max_gen=1000. As crossover operator we used position-based crossover (POS), as mutation operator the insertion mutation operator (ISM). The average width achieved by each combination during the five runs are shown in Table 3. The combination of a crossover rate of 100% and a mutation rate of 30% achieved good average results for all instances and performed best for the large instances le450_25d and queen16_16, thus we chose this combination for our further experiments.

Instance/(p_c, p_m)	(80%, 1%)	(80%, 10%)	(80%, 30%)
games120	34.0	33.0	33.8
homer	32.0	32.0	31.6
inithx.i.3	35.4	35.0	35.0
le450_25d	344.2	341.8	340.8
myciel7	66.0	66.0	66.0
queen16_16	194.2	192.8	193.2
zeroin.i.3	33.0	32.8	33.0
Instance/(p_c, p_m)	(90%, 1%)	(90%, 10%)	(90%, 30%)
games120	33.0	33.6	33.2
homer	31.6	31.6	31.4
inithx.i.3	35.0	35.0	35.0
le450_25d	341.4	344.4	339.2
myciel7	66.0	66.0	66.0
queen16_16	191.4	192.0	191.8
zeroin.i.3	32.4	33.0	32.8
Instance/(p_c, p_m)	(100%, 1%)	(100%, 10%)	(100%, 30%)
games120	33.4	33.0	34.4
homer	31.2	31.6	31.2
inithx.i.3	35.0	35.0	35.0
le450_25d	342.2	339.8	335.6
myciel7	66.0	66.0	66.0
queen16_16	191.6	191.8	190.6
zeroin.i.3	33.0	32.8	33.0

Table 3 Comparison of different combinations of crossover rate and mutation rate.

Population size and tournament selection group size

We examined populations of 100, 200, 1000 and 2000 individuals. Table 4 shows the average width for selected instances returned by algorithm *GA-tw* after five runs with the parameter settings p_m=30%, p_c=100%, s=2 and max_gen=1000. A population of size $n = 2000$ achieved the best average results for three out of four instances. For such populations a tournament selection group size of $s = 3$ or $s = 4$ seems to be the best choice as it can be seen in Table 5.

Instance	$n = 100$	$n = 200$	$n = 1000$	$n = 2000$
le450_25d	342.4	339.66	335	334.8
le450_5b	273.6	266.33	266.2	264.6
queen16_16	194.4	191	190.8	189.2
zeroin.i.3	33	32.66	33	33

Table 4 Comparison of different population sizes n.

Instance	$s = 2$	$s = 3$	$s = 4$
le450_25d	334.8	332.2	331.8
le450_5b.col	264.6	257.4	264.4
queen16_16	189.2	188.2	187.6
zeroin.i.3	33	33	33

Table 5 Comparison of different group sizes s for tournament selection.

Final results for DIMACS benchmarks graphs

Finally we applied algorithm *GA-tw* on 62 graphs of the Second DIMACS graph coloring challenge (Johnson and Trick (1993)). We ran *GA-tw* with the parameter settings, $n = 2000$, $p_m = 30\%$, $p_c = 100\%$, $s = 3$ and $gen\_max = 2000$, obtained in the previous subsections.

As crossover and mutation operators we used position-based crossover (POS) and the insertion mutation operator (ISM). For each graph we performed ten runs on machine with an Intel(R) Pentium(R)- 4 3.40 GHz processor having 1GB RAM. Table 6 shows the results for the considered graphs. The columns *Graph*, *V* and *E* present the graph name and the number of vertices and edges of that graph. *ub* contains the value of the smallest upper bound for a graph reported in Bachoore and Bodlaender (2005), Bachoore and Bodlaender (2006), Clautiaux et al. (2003), Gogate and Dechter (2004) and Musliu (2007). *min*, *max* and *avg* present the best, worst and average width returned by algorithm *GA-tw* for an instance whereas *std. dev.* contains the standard deviation of the ten results returned by algorithm *GA-tw*. Column *min-time* presents the time which was needed by algorithm *GA-tw* for the run which returned the width in column *min*, *avg-time* the average running time of the ten runs. Instances for which *GA-tw* obtained a new upper bound on their treewidth are marked with a '+' in Table 6. Instances for which *GA-tw* returned worse results are marked with a '−'.

Compared with the best upper bounds known from literature for the considered instances algorithm *GA-tw* found an improved upper bound on the treewidth for 12 graphs, *GA-tw* was able to return the same upper bound for 35 graphs, and for 15 graphs the results delivered by *GA-tw* were worse. *GA-tw* achieved most improvements for the graphs of the class le450.

7.2 Algorithm GA-tw+

We tested algorithm *GA-tw+* on 25 hypergraphs of the CSP hypergraph library from Ganzow et al. (2005). *GA-tw+* was executed with the parameter settings obtained for algorithm *GA-tw*. As crossover and mutation operators we used position-based crossover (POS) and the insertion mutation operator (ISM). We tested algorithm *GA-tw+* on the following three machines:

1. Intel(R) Xeon(TM) 3.20 GHz processor having 4 GB RAM
2. Intel(R) Pentium(R)-4 3.40 GHz processor having 1 GB RAM
3. Intel(R) Pentium(R)-4 2.80 GHz processor having 512 MB RAM

We chose the first machine as reference for normalizing the execution times. Table 7 enlists the results of *GA-tw+* for the considered hypergraphs. The columns *Hypergraph*, *V* and *H* present the graph name and the number of vertices and hyperedges of that graph. Column *ub* contains the value of the smallest upper bound on the generalized hypertree-width for a hypergraph reported in Dermaku et al. (2005) or Gottlob and Samer (2007). *min*, *max* and *avg* present the best, worst and average width returned by algorithm *GA-tw+* for an instance whereas *std. dev.* contains the standard deviation of the ten results returned by algorithm *GA-tw+*. Within the column *min-time* we present the time which was needed by algorithm *GA-tw+* for the run which returned the width in column *min*, column *avg-time* presents the average time of the ten runs. Instances for which *GA-tw+* obtained a new upper bound on their treewidth are marked with a '+' in Table 7. Instances for which *GA-tw+* returned worse results are marked with a '−'.

Compared with the best upper bounds known for the considered instances from Dermaku et al. (2005) and Gottlob and Samer (2007) algorithm *GA-tw+* found an improved upper bound on the generalized hypertree-width for 10 hypergraphs, *GA-tw+* was able to return the same upper bound for 4 hypergraphs, and for 11 hypergraphs the width returned by *GA-tw+* was worse than the best upper bound known so far.

7.3 Algorithm GA-ghw

We tested algorithm *GA-ghw* on the same 25 hypergraphs from Ganzow et al. (2005) as algorithm *GA-tw+*.

In order to derive suitable genetic operators and control parameter settings for algorithm *GA-ghw* we applied the same policy as described in section 7.1. For determining applicable crossover and mutation operators and corresponding rates we tested algorithm *GA-ghw* with the seven instances *adder_*99, *b*10, *c*499, *grid*2*d_*20, *nasa*, *NewSystem*1 and *s*510 whereas for finding appropriate population and tournament selection group sizes we applied algorithm *GA-ghw* to the instances *c*499, *grid*2*d_*20, *nasa* and *s*510. Again position-based crossover (POS) and the insertion mutation operator (ISM) proved themselves to be adequate genetic operators. For the crossover rate, mutation rate, population size and tournament selection group size our experiments revealed that we may retain the settings for those parameters which have been obtained in section 7.1.

We tested algorithm *GA-ghw* on the following four machines and chose the first one as reference for normalizing the execution times:

1. Intel(R) Xeon(TM) 3.20 GHz processor having 4 GB RAM
2. Intel(R) Pentium(R)-4 3.40 GHz processor having 1 GB RAM
3. Intel(R) Pentium(R)-4 2.80 GHz processor having 512 MB RAM
4. Intel(R) Core(TM)2 Duo 2.40 GHz processor having 2 GB RAM.

Table 8 enlists the results of *GA-ghw* for the considered hypergraphs. The columns *Hypergraph*, *V* and *H* present the hypergraph name and the number of vertices and hyperedges of that graph. *ub* contains the value of the smallest upper bound on the generalized hypertree-width for a hypergraph reported in Dermaku et al. (2005) or Gottlob and Samer (2007), whereas *GA-tw+* gives the smallest width obtained for a hypergraph by algorithm *GA-tw+*. *min*, *max* and *avg* present the best, worst and average width returned by algorithm *GA-ghw* for an instance whereas *std. dev.* contains the standard deviation of the ten results returned by algorithm *GA-ghw*. Within the column *min-time* we present the time which was needed by algorithm *GA-ghw* for the run which returned the width in column *min*, column *avg-time* presents the average time of the ten runs. Instances for which *GA-ghw* obtained a new upper bound on their treewidth are marked with a '+' in Table 8. Instances for which *GA-ghw* returned worse results are marked with a '−'.

Compared with the best upper bounds known for the considered instances algorithm *GA-ghw* found an improved upper bound on the generalized hypertree-width for 11 hypergraphs, *GA-ghw* was able to return the same upper bound for 6 hypergraphs, and for 8 hypergraphs the width returned by *GA-ghw* was worse than the best upper bound known so far.

When comparing algorithm *GA-ghw* and algorithm *GA-tw+* with each other we observe that for most instances algorithm *GA-ghw* returned results improving (10 instances) or equalizing (11 instances) the results previously returned by algorithm *GA-tw+*. Worse results were returned only for 4 hypergraphs. In general the results show that using the width in terms of generalized hypertree decompositions as fitness function already within the genetic search leads to better results. However this approach takes significantly more time for most instances.

When comparing the proposed genetic algorithms with the heuristic methods presented in Dermaku et al. (2005), we have to bear in mind that the running times of the heuristic methods presented in Dermaku et al. (2005) were normalized to a different machine, namely to an Intel(R) Xeon(TM) 2.2 GHz (dual) processor with 2 GB RAM. Moreover, the algorithms in Dermaku et al. (2005) were executed five times for each instance, whereas our results were obtained from ten runs. Thus, the comparison considering the upper bounds should be taken only indicatory. Based on the running times reported in our experiments we conclude that the time performance of our algorithms is worse compared to the methods in Dermaku et al. (2005). However, setting no time limit to our genetic algorithms allowed us to find some new upper bounds on the generalized hypertree-width for some benchmark hypergraphs (e.g. nasa problem).

7.4 Comparison with exact methods for hypertree decompositions

In the experiments described in the previous subsections we obtained generalized hypertree decompositions of small width for many of the regarded benchmark hypergraphs. Since the hypertree-width of a hypergraph is a 3-approximation of its generalized hypertree-width, actually it holds that $hw \leq 3ghw + 1$ (Adler et al. (2005)), it would be interesting to assess the difference between the widths returned by algorithm *GA-ghw* and the hypertree-width of those instances. As mentioned in section 3, several algorithms deciding whether the hypertree-width of a hypergraph is at most k have been proposed and these algorithms may be used to compute the exact hypertree-width of hypergraphs, in particular for instances having small hypertree-width. Thus, for our final experiments we applied the backtracking based algorithm *IsDecomposable* for hypertree decompositions from Subbarayan and Andersen (2007) to the same benchmark hypergraphs as algorithm *GA-ghw* and in addition we considered the results returned by the backtracking based algorithm *det-k-decomp* in Gottlob and Samer (2007).

Algorithm *IsDecomposable* was executed on an Intel(R) Xeon(TM) 3.20 GHz processor having 4 GB RAM since the execution times of algorithm *GA-ghw* were normalized to that machine. For each instance we gave algorithm *IsDecomposable* the average running time of *GA-ghw* for the instance as time limit. Note that in Gottlob and Samer (2007) algorithm *det-k-decomp* was executed on a different machine, Intel(R) Xeon(TM) 2.2 GHz (dual) processor with 2 GB RAM, and *det-k-decomp* was executed five times with a one hour time limit, thus the reported times may be regarded only as rough reference values.

Table 9 enlists the results returned for the benchmark hypergraph. For each hypergraph we report the widths obtained by algorithm *GA-ghw*, *IsDecomposable* and *det-k-decomp*, (column *width*). Entries representing the hypertree-width of an instance are marked with a '*'. Algorithm *IsDecomposable* could not be applied to the instances c880, nasa and s641 because they consist of more than one connected component. Empty entries for algorithm *det-k-decomp* indicate that *det-k-decomp* was not applied to that instances in Gottlob and Samer (2007). Column $diff.$ gives the difference between the hypertree-width of a hypergraph and the smallest width of a generalized hypertree decomposition computed by algorithm *GA-ghw*. Finally, column *time* reports the running times for the regarded algorithms. For the exact methods running times represent the time for computing the hypertree-width of the hypergraph. For algorithm *IsDecomposable* a '-' entry indicates that the hypertree-width of an instance could not be computed within the given time limit, for algorithm *det-k-decomp* an empty entry indicates that *det-k-decomp* was not applied to an instance in Gottlob and Samer (2007).

We observe that for many instances with known hypertree-width algorithm *GA-ghw* is able to return a generalized hypertree decomposition whose width equals the hypertree-width or exceeds it by at most one. Only for the instances bridge_50, bridge_99 and grid2d_20 the width obtained by *GA-ghw* exceeds the hypertree-width by 3 and 4. For the instance s510 *GA-ghw* found a generalized hypertree decomposition whose width improves the hypertree-width by even 3.

A direct comparison of the generalized hypertree-width and hypertree-width cannot be done, because while for some of the instances we know the hypertree-width, this is not the case for their generalized hypertree-width. However, based on our results, we can conclude that the hypertree-width is very close to the best known upper bound on the generalized hypertree-width.

Considering the running times of the algorithms we examine that for instances having small hypertree-width the backtracking based algorithms take significantly less running time than algorithm *GA*-ghw. Thus, for instances which are expected to have small hypertree-width we suggest to apply algorithm *IsDecomposable* or algorithm *det-k-decomp* in order to obtain a good decomposition within short time. On the other hand, for hypergraphs whose hypertree-widths are supposed to be large algorithm *GA-ghw* is able to return better widths than the backtracking based methods. We conclude that for such instances algorithm *GA-ghw* is better suited.

	Graph	*V*	*E*	*ub*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
	anna	138	986	12	12	12	12	0,00	00:03:32	00:03:32
	david	87	812	13	13	13	13	0,00	00:02:34	00:02:32
	huck	74	602	10	10	10	10	0,00	00:02:00	00:01:59
	homer	561	3258	31	31	31	31	0,00	00:18:38	00:18:36
	jean.col	80	508	9	9	9	9	0,00	00:02:00	00:01:59
	games120	120	1276	32	32	32	32	0,00	00:07:42	00:07:32
	queen5_5	25	320	18	18	18	18	0,00	00:00:33	00:00:33
−	queen6_6	36	580	25	26	26	26	0,00	00:00:51	00:00:51
	queen7_7	49	952	35	35	36	35,2	0,42	00:01:32	00:01:34
	queen8_8	64	1456	45	45	47	46	0,47	00:02:47	00:02:30
	queen9_9	81	2112	58	58	60	58,5	0,71	00:03:50	00:03:50
	queen10_10	100	2940	72	72	73	72,4	0,52	00:05:39	00:05:35
+	queen11_11	121	3960	88	87	90	88,2	1,14	00:08:17	00:07:55
	queen12_12	144	5192	104	104	108	105,7	1,34	00:10:33	00:10:52
+	queen13_13	169	6656	122	121	125	123,1	1,29	00:15:06	00:14:50
	queen14_14	196	8372	141	141	148	144	2,16	00:19:41	00:19:24
+	queen15_15	225	10360	163	162	168	164,8	1,87	00:25:44	00:25:17
	queen16_16	256	12640	186	186	191	188,5	1,90	00:34:53	00:31:41
	fpsol2.i.1	496	11654	66	66	66	66	0,00	00:33:02	00:32:29
−	fpsol2.i.2	451	8691	31	32	33	32,6	0,52	00:24:05	00:23:45
	fpsol2.i.3	425	8688	31	31	33	32,3	0,67	00:24:22	00:22:49
	inithx.i.1	864	18707	56	56	56	56	0,00	00:56:18	00:55:42
−	inithx.i.2	645	13979	31	35	35	35	0,00	00:38:37	00:38:24
−	inithx.i.3	621	13969	31	35	35	35	0,00	00:37:41	00:37:17
−	miles1000	128	6432	49	50	50	50	0,00	00:09:19	00:09:24
	miles1500	128	10396	77	77	77	77	0,00	00:07:37	00:07:33
−	miles250	128	774	9	10	10	10	0,00	00:04:02	00:04:01
−	miles500	128	2340	22	24	25	24,1	0,32	00:07:22	00:07:16
−	miles750	128	4226	36	37	37	37	0,00	00:08:56	00:08:50
	mulsol.i.1	197	3925	50	50	50	50	0,00	00:11:11	00:11:05
	mulsol.i.2	188	3885	32	32	32	32	0,00	00:09:44	00:09:48
	mulsol.i.3	184	3916	32	32	32	32	0,00	00:09:39	00:09:32
	mulsol.i.4	185	3946	32	32	32	32	0,00	00:09:38	00:09:33
	mulsol.i.5	186	3973	31	31	31	31	0,00	00:09:44	00:09:31
	myciel3	11	20	5	5	5	5	0,00	00:00:14	00:00:14
	myciel4	23	71	10	10	10	10	0,00	00:00:34	00:00:34
	myciel5	47	236	19	19	19	19	0,00	00:01:20	00:01:18
	myciel6	95	755	35	35	35	35	0,00	00:03:52	00:03:48
−	myciel7	191	2360	54	66	66	66	0,00	00:12:37	00:12:24
−	school1	385	19095	184	185	199	192,5	5,66	01:18:04	01:21:35
−	school1_nsh	352	14612	155	157	170	163,1	5,40	01:10:39	01:09:05
	zeroin.i.1	211	4100	50	50	50	50	0,00	00:10:41	00:10:30
	zeroin.i.2	211	3541	32	32	33	32,7	0,48	00:09:54	00:09:46
	zeroin.i.3	206	3540	32	32	33	32,9	0,32	00:09:45	00:09:38
+	le450_5a	450	5714	253	243	263	248,3	7,12	01:47:13	01:51:24
	le450_5b	450	5734	248	248	253	249,9	1,60	01:52:12	01:49:50
+	le450_5c	450	9803	272	265	272	267,1	2,28	01:38:37	01:35:37
+	le450_5d	450	9757	267	265	268	265,6	1,07	01:30:02	01:25:08
−	le450_15a	450	8168	264	265	275	268,7	3,71	01:54:36	01:42:21
+	le450_15b	450	8169	270	265	271	269	1,63	01:47:03	01:39:08
+	le450_15c	450	16680	357	351	359	352,8	2,44	01:23:17	01:22:05
+	le450_15d	450	16750	354	353	361	356,9	2,56	01:21:04	01:17:57
−	le450_25a	450	8260	221	225	232	228,2	2,10	01:40:25	01:41:05
+	le450_25b	450	8263	228	227	239	234,5	3,47	01:40:45	01:46:06
+	le450_25c	450	17343	327	320	331	327,1	3,78	01:43:09	01:34:08
+	le450_25d	450	17425	330	327	335	330,1	2,33	01:51:52	01:35:06
−	DSJC125.1	125	736	60	61	63	61,9	0,74	00:08:21	00:07:47
−	DSJC125.5	125	3891	108	109	110	109,2	0,42	00:04:21	00:04:19
	DSJC125.9	125	6961	119	119	119	119	0,00	00:01:50	00:01:54
	DSJC250.1	250	3218	169	169	171	169,7	0,82	00:31:18	00:27:02
	DSJC250.5	250	15668	230	230	233	231,4	0,84	00:10:48	00:09:57
	DSJC250.9	250	27897	243	243	244	243,1	0,32	00:03:58	00:04:01

Table 6 Final results for DIMACS graphs.

	Hypergraph	*V*	*H*	*ub*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
−	adder_75	526	376	2	3	3	3	0	00:08:16	00:08:16
−	adder_99	694	496	2	3	4	3.4	0.52	00:11:29	00:11:27
−	atv_partial_system	124	88	3	4	4	4	0	00:01:51	00:01:51
+	b06	50	48	5	4	5	4.9	0.32	00:00:53	00:00:53
+	b08	179	170	10	9	10	9.1	0.32	00:04:22	00:04:22
+	b09	169	168	10	7	8	7.7	0.48	00:03:41	00:03:40
+	b10	200	189	14	13	14	13.3	0.48	00:05:40	00:06:19
−	bridge_50	452	452	2	6	7	6.9	0.32	00:10:36	00:10:32
−	bridge_99	893	893	2	7	7	7	0	00:25:19	00:25:18
	c1355	587	564	13	13	13	13	0	00:17:47	00:17:45
−	c1908	913	880	29	37	38	37.6	0.52	00:38:28	00:39:02
+	c499	243	202	13	11	12	11.1	0.32	00:07:47	00:07:53
+	c880	443	383	19	18	19	18.1	0.32	00:14:53	00:15:07
	clique_20	190	20	10	10	10	10	0	00:26:07	00:26:02
−	grid2d_20	200	200	7	10	10	10	0	00:10:43	00:10:42
+	grid3d_8	256	256	20	19	19	19	0	02:08:36	02:07:19
+	grid4d_4	128	128	17	13	14	13.3	0.48	01:03:50	01:07:58
+	grid5d_3	122	121	18	14	15	14.4	0.52	03:34:11	03:39:08
−	nasa	579	680	21	22	23	22.6	0.52	00:42:45	00:43:41
−	NewSystem1	142	84	3	4	5	4.4	0.52	00:02:45	00:02:46
−	NewSystem2	345	200	3	5	5	5	0	00:07:21	00:07:17
	s444	205	202	5	5	6	5.7	0.48	00:05:20	00:05:20
+	s510	236	217	20	19	20	19.3	0.48	00:10:37	00:10:33
	s526	217	214	7	7	8	7.1	0.32	00:07:00	00:07:04
−	s641	433	398	7	11	13	11.9	0.57	00:11:07	00:10:53

Table 7 *GA-tw+* results for selected benchmark hypergraphs.

	Hypergraph	*V*	*H*	*ub*	*GA-tw+*	*min*	*max*	*avg*	*std.dev.*	*min-time*	*avg-time*
−	adder_75	526	376	2	3	3	3	3	0	05:02:54	05:02:51
−	adder_99	694	496	2	3	3	3	3	0	08:33:10	08:33:53
	atv_partial.	124	88	3	4	3	3	3	0	00:31:09	00:31:10
+	b06	50	48	5	4	4	4	4	0	00:09:39	00:09:44
+	b08	179	170	10	9	9	9	9	0	01:04:39	01:04:44
+	b09	169	168	10	7	7	7	7	0	01:14:43	01:14:54
+	b10	200	189	14	13	11	12	11.8	0.42	01:51:47	01:51:39
−	bridge_50	452	452	2	6	6	6	6	0	06:33:56	06:33:25
−	bridge_99	893	893	2	7	6	6	6	0	24:18:57	24:18:06
	c1355	587	564	13	13	13	13	13	0	09:23:14	09:28:26
+	c1908	913	880	29	37	25	30	27.1	1.79	29:32:47	30:20:22
+	c499	243	202	13	11	11	12	11.7	0.48	02:13:10	02:13:13
+	c880	443	383	19	18	17	18	17.2	0.42	06:54:25	06:55:26
−	clique_20	190	20	10	10	11	12	11.2	0.42	01:30:29	01:30:43
−	grid2d_20	200	200	7	10	10	10	10	0	01:36:00	01:35:32
−	grid3d_8	256	256	20	19	21	22	21.3	0.48	04:53:40	04:49:49
+	grid4d_4	128	128	17	13	15	16	15.3	0.48	01:24:17	01:24:42
+	grid5d_3	122	121	18	14	16	18	16.7	0.82	01:25:32	01:24:51
+	nasa	579	680	21	22	19	22	19.9	0.74	17:13:13	17:19:44
	NewSystem1	142	84	3	4	3	4	3.1	0.32	00:36:45	00:36:59
−	NewSystem2	345	200	3	5	4	4	4	0	03:01:16	03:01:36
	s444	205	202	5	5	5	5	5	0	01:46:54	01:47:07
+	s510	236	217	20	19	17	17	17	0	02:40:09	02:41:52
	s526	217	214	7	7	7	7	7	0	02:23:17	2:23:04
	s641	433	398	7	11	7	7	7	0	05:30:43	05:30:27

Table 8 *GA-ghw* results for selected benchmark hypergraphs.

Hypergraph	*width*			*diff.*	*time*		
	GA-ghw	*IsDec.*	*det-k-dec.*		GA-ghw	*IsDec.*	*det-k-dec.*
adder_75	3	2*	2*	1	05:02:54	00:00:02	00:00:00
adder_99	3	2*	2*	1	08:33:10	00:00:03	00:00:00
atv_partial_system	3	3*	3*	0	00:31:09	00:00:05	00:00:00
b06	4	4*		0	00:09:39	00:00:30	
b08	9	25			01:04:39	-	
b09	7	17			01:14:43	-	
b10	11	27			01:51:47	-	
bridge_50	6	2*	2*	4	06:33:56	00:00:03	00:00:00
bridge_99	6	2*	2*	4	24:18:57	00:02:02	00:00:01
c1355	13	45			09:23:14	-	
c1908	25	64			29:32:47	-	
c499	11	39			02:13:10	-	
c880	17	-			06:54:25	-	
clique_20	11	10*		1	01:30:29	00:32:33	
grid2d_20	10	16	7*	3	01:36:00	-	00:52:20
grid3d_8	21	60			04:53:40	-	
grid4d_4	15	28			01:24:17	-	
grid5d_3	16	35			01:25:32	-	
nasa	19	-			17:13:13	-	
NewSystem1	3	3*	3*	0	00:36:45	00:00:05	00:00:00
NewSystem2	4	3*	3*	1	03:01:16	00:01:03	00:00:00
s444	5	5	5*	0	01:46:54	-	00:06:25
s510	17	43	20*	-3	02:40:09	-	00:34:42
s526	7	14	7*	0	02:23:17	-	00:28:35
s641	7	-	7*	0	05:30:43	-	00:26:51

Table 9 Comparison between *GA-ghw* and exact methods for hypertree decompositions.

8 Conclusion

Motivated by the results of McMahan (2004) and Larrañaga et al. (1997) we implemented a genetic algorithm, named *GA-tw*, searching for small-width tree decompositions and presented two extensions, *GA-tw+* and *GA-ghw*, for generalized hypertree decompositions of hypergraphs. We carried out a series of tests in order to estimate values for the control parameters of the genetic algorithms and it turned out that the position based crossover operator (POS) and the insertion mutation operator (ISM) are well-suited for finding tree decompositions and generalized hypertree decompositions of small width. We evaluated each of the three algorithms by a large number of benchmark instances and compared the obtained results to the currently best solutions for those benchmarks. For many benchmark instances the genetic algorithms were able to return comparable results and for some benchmark hypergraphs they found new upper bounds on the treewidth and generalized hypertree-width.

One interesting point for future research is the development of a self-adapting genetic algorithm for generalized hypertree decompositions which might be capable of adjusting the various control parameters during its execution. Another open task for further research is the combination of genetic algorithms with local optimization strategies for generalized hypertree decompositions. Furthermore, an interesting topic of future research is to extend the proposed genetic algorithm to deal with Weighted Hypertree Decompositions (Scarcello et al. (2004)).

Acknowledgments

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

References

Adler, I., Gottlob G. and Grohe M. (2005) ‘Hypertree-width and related hypergraph invariants’, *Proceedings of the 3rd European Conference on Combinatorics, Graph Theory and Applications (EUROCOMB’05), DMTCS Proceedings Series*, AE:5–10, 2005.

Bachoore, E. H. and Bodlaender, H. L. (2005) ‘New upper bound heuristics for treewidth’, In *WEA*, pages 216–227, 2005.

Bachoore, E. H. and Bodlaender, H. L. (2006) ‘A branch and bound algorithm for exact, upper, and lower bounds on treewidth’, In *AAIM*, pages 255–266, 2006.

Banzhaf, W. (1990) ‘The ’molecular’ traveling salesman problem’, *Biological Cybernetics*, 64:7–14, 1990.

Bodlaender,H. L. (2005) ‘Discovering treewidth’, *technical report UU-CS-2005-018, Utrecht University*, 2005.

Chvatal, V. (1979) ‘A greedy heuristic for the set covering problem’, *Math. of Operations Research*, 4:233–235, 1979.

Clautiaux, F., Carlier, J., Moukrim, A. and Nègre, S. (2003) 'New lower and upper bounds for graph treewidth', In *WEA*, pages 70–80, 2003.

Clautiaux, F., Moukrim, A., Négre, S. and Carlier, J. (2004) 'Heuristic and meta-heurisistic methods for computing graph treewidth', *RAIRO Oper. Res.*, 38:13–26, 2004.

Davis, L. (1985) 'Applying adaptive algorithms to epistatic domains', In *IJCAI*, pages 162–164, 1985.

Dechter, R. (2003) '*Constraint Processing*', Morgan Kaufmann, 2003.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N. and Samer, M. (2005) 'Heuristic methods for hypertree decompositions', Technical report, Technische Universität Wien, DBAI-TR-2005-53, 2005.

Fogel, D.B. (1990) 'A parallel processing approach to a multiple travelling salesman problem using evolutionary programming', In *Proceeding of the Fourth Anual Parallel Processing Symposium, Fullerton, CA*, pages 318–26, 1990.

Ganzow, T., Gottlob, G., Musliu, N. and Samer, M. (2005) 'A CSP hypergraph library', Technical report, Technische Universität Wien, DBAI-TR-2005-50, 2005.

'GLPK (GNU linear programming kit)', http://www.gnu.org/software/glpk/.

Gogate, V., and Dechter, R. (2004) 'A complete anytime algorithm for treewidth', In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence UAI-04*, pages 201–208, AUAI Press, Arlington, Virginia, USA, 2004.

Goldberg, D. E. and Lingle Jr., R. (1985) 'AllelesLociand the traveling salesman problem', In *ICGA*, pages 154–159, 1985.

Gottlob, G., Leone, N. and Scarcello, F. (1999) 'Tractable queries and constraints', *Proceedings of the Conference on Database and Expert Systems Applications (DEXA'99)*, LNCS 1677:1–15, 1999.

Gottlob, G., Leone, N. and Scarcello, F. (2000) 'A comparison of structural CSP decomposition methods', *Artif. Intell.*, 124(2):243–282, 2000.

Gottlob, G., Leone, N. and Scarcello, F. (2001) 'Hypertree decompositions: A survey', In *MFCS*, pages 37–57, 2001.

Gottlob, G., Miklos, Z. and Schwentick, T. (2007) 'Generalized hypertree decompositions: NP-hardness and tractable variants', In *Proc. of PODS*, 2007.

Gottlob, G. and Samer, M. (2007) 'A backtracking-based algorithm for computing hypertree-decompositions', Technical report, arXiv:cs.DS/0701083, 2007.

Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

Johnson, D. S. and Trick, M. A. (1993) 'The second dimacs implementation challenge: NP-hard problems: Maximum clique, graph coloring, and satisfiability', *Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society*, 1993.

Karp, R. M. (1972) 'Reducibility among combinatorial problems', In *Complexity of Computer Computations, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights*, pages 85–103, 1972.

Kjaerulff U. (1992) 'Optimal decomposition of probabilistic networks by simulated annealing', *Statistics and Computing*, 1:2–17, 1992.

Kloks, T. (1994) *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*, Springer, 1994.

Korimort, T. (2003) *Heuristic Hypertree Decomposition*, PhD thesis, Vienna University of Technology, 2003.

Koster, A. M. C. A., Bodlaender, H. and van Hoesel, S. (2001) 'Treewidth: Computational experiments', *Electronic Notes in Discrete Mathematics 8, Elsevier Science Publishers*, pages 54–57, 2001.

Larrañaga, P., Kuijpers, C. M. H., Murga, R. H. and Yurramedi, Y. (1994) 'Optimal decomposition of bayesian networks by genetic algorithms', Technical report, University of the Basque County Spain,I nternal Report EHU-KZAA-IKT-3-94, 1994.

Larrañaga, P., Kuijpers, C. M. H., Poza, M. and Murga, R. H. (1997) 'Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms', *Statistics and Computing (UK)*, 7(1):19–34, 1997.

McMahan, B. (2004) 'Bucket elimination and hypertree decompositions', Implementation Report, Institute of Information Systems (DBAI), TU Vienna, 2004.

Michalewicz, Z. (1992) *Genetic Algorithms + Datastructures = Evolution Programs*, Springer-Verlag, Berlin, 1992.

Michalewicz, Z. and Fogel, D. B. (2004) *How to Solve It: Modern Heuristics*, Springer, 2004.

Musliu, N. (2007) 'Generation of tree decompositions by iterated local search', In *EvoCOP 2007 - Seventh European Conference on Evolutionary Computation in Combinatorial Optimisation, LNCS*, volume 4446, pages 130–141. Springer, 2007.

Oliver, I. M., Smith, D. J. and Holland, J. R. C. (1987) 'A study of permutation crossover operators on the traveling salesman problem', In *ICGA*, pages 224–230, 1987.

Robertson, N. and Seymour, P. D. (1986) 'Graph minors. II. algorithmic aspects of tree-width', *J. Algorithms*, 7(3):309–322, 1986.

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

Scarcello, F., Greco, G. and Leone, N. (2004) 'Weighted hypertree decompositions and optimal query plans', *In Proceedings of the 23rd Symposium on Principles of Database Systems (PODS), ACM*, pages 210–221, 2004.

Schafhauser, W. (2006) *New heuristic methods for tree decompositions and generalized hypertree decompositions*, Master thesis, Vienna University of Technology, 2006.

Schrijver, A. (1996) *Theory of Linear and Integer Programming*, Wiley, 1996.

Shoikhet, K. and Geiger, D. (1997) 'A practical algorithm for finding optimal triangulations', *In Proc. of National Conference on Artificial Intelligence (AAAI'97)*, pages 185–190, 1997.

Subbarayan, S. and Andersen, H. R (2007) 'Backtracking procedures for hypertree, hyperspread and connected hypertree decomposition of CSPs', In *IJCAI'07*, pages 180–185, 2007.

Syswerda, G. (1991) 'Schedule optimization using genetic algorithms', In L.Davis, editor, *Handbook of Genetic Algorithms*, pages 332–40. Van Nostrand Teinhold, New York, 1991.

Tarjan, R.E. and Yannakakis, M. (1984) '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.