htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond

Michael Abseher, Nysret Musliu, and Stefan Woltran

TU Wien, Institute of Information Systems 184/2,
Favoritenstraße 9–11, 1040 Vienna, Austria,
[abseher,musliu,woltran]@dbai.tuwien.ac.at

Abstract. Decompositions of graphs play a central role in the field of parameterized complexity and are the basis for many fixed-parameter tractable algorithms for problems that are NP-hard in general. Tree decompositions are the most prominent concept in this context and several tools for computing tree decompositions recently competed in the 1st Parameterized Algorithms and Computational Experiments Challenge. However, in practice the quality of a tree decomposition cannot be judged without taking concrete algorithms that make use of tree decompositions into account. In fact, practical experience has shown that generating decompositions of small width is not the only crucial ingredient towards efficiency. To this end, we present *htd*, a free and open-source software library, which includes efficient implementations of several heuristic approaches for tree decomposition and offers various features for normalization and customization of decompositions. The aim of this article is to present the specifics of *htd* together with an experimental evaluation underlining the effectiveness and efficiency of the implementation.

Keywords: Tree decompositions, Dynamic programming, Software library

1 Introduction

Graph decompositions are an important concept in the field of parameterized complexity and a wide variety of such approaches can be found in the literature including tree decompositions [11, 23, 37], branch decompositions [38], and hypertree decompositions [22] (of hypergraphs), to mention just a few. The concept of tree decompositions gained special attention since many NP-hard search problems become tractable when the parameter *treewidth* is bounded by some constant k [8, 12, 36]. A problem that exhibits tractability by bounding some problem-inherent constant is also called fixed-parameter tractable (FPT) [18].

The standard technique for solving a given problem using this concept is the computation of a tree decomposition followed by a dynamic programming (DP) algorithm that traverses the nodes of the decomposition and consecutively solves the respective sub-problems [36]. For problems that are FPT w.r.t. treewidth, the general run-time of such algorithms for an instance of size n is $f(k) \cdot n^{\mathcal{O}(1)}$, where f is an arbitrary function over width k of the used tree decomposition. In fact, this approach has been used for several applications including the solving of inference problems in probabilistic networks [32], frequency assignment [30], computational biology [42], logic programming [33], routing problems [17], and solving of quantified boolean formulae [14].

From a theoretical point of view the actual width k is the crucial parameter towards efficiency for FPT algorithms that use tree decompositions. In the literature various approaches for optimizing width when computing decompositions have been proposed (see Section 2), but to the best of our knowledge no software frameworks exist which offer the feature to customize tree decompositions by other criteria than just minimizing the plain width. However, experience shows that even decompositions of the same width can lead to significant differences in the run-time of DP algorithms and recent results confirm that the width is indeed not the only important parameter that has a significant influence on the performance [27, 33]. In particular, [4, 6] has underlined that considering such additional criteria is highly beneficial. Nevertheless, a post-processing phase based on machine learning was needed to determine "good" tree decompositions. Therefore we see a strong need to offer a specialized decomposition framework that allows for directly constructing customized decompositions, i.e., decompositions which reflect certain preferences of the developer, in order to optimally fit to the DP algorithm in which they are used. In this paper we present a free, open-source framework (*htd*) which supports a vast amount of input graph types and different types of decompositions. *htd* includes efficient implementations of several heuristic approaches for computing tree decompositions. Furthermore, *htd* provides various built-in customization and manipulation operations in order to fit the needs of developers of DP algorithms. These include normalizations, optimization of tree decompositions, computation of induced edges and labeling operations (see Section 3).

Just recently, *htd* participated in the "First Parameterized Algorithms and Computational Experiments Challenge" ("PACE16")[1] where it was ranked at the third place in the heuristics track. Although *htd* provides lots of additional convenience functions, the results of *htd* with respect to the optimization of width are very close to those of the heuristic approaches ranked at the first two places. In Section 4, we will present some complementing experimental evaluation that also sheds light on the effect of customization when decompositions are used in a specific DP algorithm.

htd has been already used successfully in different projects, like D-FLAT [2], a framework for rapid-prototyping of dynamic programming algorithms on tree decompositions, or dynQBF [14], a DP-based solver for quantified boolean formulae. Our framework is available for download as free, open-source software at `https://github.com/mabseher/htd`. We consider *htd* as a potential starting point for researchers to contribute their algorithms in order to provide a new framework for all different types of graph decompositions. A detailed report on *htd* and all its features can be found in [5].

[1] See `https://pacechallenge.wordpress.com/track-a-treewidth/` for more details.

2 Background

The intuition underlying tree decompositions is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices in one node and thereby isolating the parts responsible for the cyclicity. Formally, the notions of tree decomposition and treewidth are defined as follows [37, 13].

Definition 1. *Given a graph $G = (V, E)$, a* tree decomposition *of G is a pair (T, χ) where $T = (N, F)$ is a tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions hold:*

1. *For each vertex $v \in V$, there exists a node $i \in N$ such that $v \in \chi_i$.*
2. *For each edge $(v, w) \in E$, there exists an $i \in N$ with $v \in \chi_i$ and $w \in \chi_i$.*
3. *For each $i, j, k \in N$: If j lies on the path between i and k then $\chi_i \cap \chi_k \subseteq \chi_j$.*

The width *of a given tree decomposition is defined as $max_{i \in N}|\chi_i| - 1$ and the* treewidth *of a graph is the minimum width over all its tree decompositions.*

Note that the tree decomposition of a graph is in general not unique. In the following we consider rooted tree decompositions.

Definition 2. *A* normalized *(or* nice*) tree decomposition of a graph G is a rooted tree decomposition (T, χ) where each node $i \in N$ is of one of the following types: (1)* Leaf*: i has no child nodes; (2)* Introduce Node*: i has one child j with $\chi_j \subset \chi_i$ and $|\chi_i| = |\chi_j| + 1$; (3)* Forget Node*: i has one child j with $\chi_j \supset \chi_i$ and $|\chi_i| = |\chi_j| - 1$; (4)* Join Node*: i has two children j, k with $\chi_i = \chi_j = \chi_k$.*

Each tree decomposition can be transformed into a normalized one in linear time without increasing the width [29]. Apart from nice tree decompositions, one can find numerous other normalizations in the literature [3].

While the problem of constructing an optimal tree decomposition, i.e. a decomposition with minimal width, is intractable [7], researchers have proposed several exact methods for small graphs and efficient heuristic approaches that usually construct tree decompositions of almost optimal width for larger graphs. Examples of exact algorithms for tree decompositions are [39, 21, 9]; greedy heuristic algorithms include Minimum Degree heuristic [10], Maximum Cardinality Search (MCS) [40], and Min-Fill heuristic [16]. Metaheuristic techniques have been provided in terms of genetic algorithms [31, 35], ant colony optimization [25], and local search based techniques [28, 15, 34]. Recent surveys [13, 26] provide further details.

Several software frameworks that implement some of the tree decomposition algorithms mentioned above are publicly available. These libraries include QuickBB [21][2], htdecomp [19][3] and Jdrasil[4]. Very recently, also an open database for computation, storage and retrieval of tree decompositions was initiated [41].

[2] Available at `http://www.hlt.utdallas.edu/~vgogate/quickbb.html`
[3] Available at `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`
[4] Available at `https://github.com/maxbannach/Jdrasil`

3 A Framework for (Customized) Tree Decompositions

The aforementioned tree decomposition software frameworks focus mainly on computing tree decompositions of small width; optimizing the resulting decomposition with respect to a concrete DP algorithm to be applied is therefore left to the user. Our system, *htd*, provides an efficient implementation of several algorithms that compute tree decompositions of small width and aims for much richer customizations of the resulting tree decomposition: this includes normalizations, further optimization criteria (for instance, minimizing the total number of children of join nodes), or augmenting the information in the bags (for instance, including the subgraph induced by the vertices of the bag) of the provided decomposition. In what follows we highlight the main features.

Computing Tree Decompositions: The framework provides efficient implementations of several heuristic approaches for tree decompositions including MCS, Minimum Degree heuristic, Min-Fill heuristic and their combination. The default tree decomposition strategy in *htd* is Min-Fill.

Manipulation Operations and Normalizations: Manipulations in the context of *htd* refer to all operations which alter a given decompositions, e.g., by ensuring that the maximum number of children of decomposition nodes is limited. Normalizations are manipulations which combine multiple base manipulation operations, e.g., one can use them to request only nice tree decompositions.

Optimization: In the context of *htd*, the term "optimization" is used to refer to operations which improve certain properties of a decomposition. Via optimization the obtained decomposition adheres to certain preferences. *htd* offers the ability to optimize by searching for the optimal root node and also by choosing the best tree decomposition based on a custom (potentially multi-level) criterion. Combining these two strategies might yield even better results.

Computation of Induced Edges: Any DP algorithm needs precise knowledge about the (hyper-)edges of the problem instance's graph representation induced by a bag because this is the information which actually matters when the problem at hand has to be solved. Computing this information inside the DP step is not only time-consuming, it also potentially destroys the property of fixed-parameter tractability in practice because in each bag one might need to scan the whole edge set of the input. *htd* computes and delivers this information very efficiently by extending decomposition algorithms appropriately.

Labels and Labeling Operations: Labels generalize the concept that is used for the induced edges to arbitrary information. They can be used to take care of supplemental information of the input instance and they can enhance the knowledge base represented by a decomposition. In both cases, one can define labeling functions and operations which automate the process of assigning labels.

All these customization features can be applied directly in the context of the decomposition procedure and no further code changes are necessary. The basic work-flow of how to obtain a customized decomposition of a problem instance using *htd* is the following: At first, the input specification which represents an instance of a given problem must be transformed to its corresponding instance graph. Depending on the actual need, a developer can choose from several built-in graph types, like directed and undirected ones as well as hypergraphs. In order to be able to manage additional information about the input instance, *htd* also offers labeled and so-called named counterparts for each graph type. A labeled graph type allows to add arbitrary labels to the vertices and edges of the graph.

Named graph types extend this helpful feature by providing a bi-directional one-to-one mapping between labels and the vertices or edges, respectively. This can be useful in cases where one wants to access vertices and edges not only by their identifiers but also by a custom "name" which can be of arbitrary data type. Importers for graph formats like DIMACS and GraphML are provided by *htd*, but one can also construct the graph manually using a custom importer.

After all information of the input instance is parsed, the next step for a developer is to decide for the decomposition algorithm to use and (optionally) choose from the wide range of built-in manipulation operations or to implement its own, custom manipulation. By running the decomposition algorithm with the instance graph as its input, we obtain a customized decomposition which then can be used in the dynamic programming algorithm. Alternatively, the decomposition can be exported or printed using built-in functionality.

4 Experimental Evaluation

In this section we give first results regarding the performance characteristics of our framework. The experiments in this paper are based on two investigations. At first, we have a look at the actual efficiency of *htd* in terms of the minimum width achieved within a fixed time period. Afterwards we highlight the usefulness of *htd*'s ability to optimize tree decompositions based on custom criteria.

All our experiments were performed on a single core of an Intel Xeon E5-2637@3.5GHz processor running Debian GNU/Linux 8.3. For repeatability of the experiments we provide the complete testbed and the results under the following link: `www.dbai.tuwien.ac.at/proj/dflat/evaluation_htd100b1.zip`

4.1 Efficient Computation of Minimum-Width Decompositions

In order to have an indication for *htd*'s actual competitiveness, we compare *htd* 1.0.0-beta1 [1] to the participants of Track A ("Treewidth") of the Parameterized Algorithms and Computational Experiments Challenge 2016 ("PACE16").

The following list of algorithms contains *htd* and all further participants of the sequential heuristics track of the PACE treewidth challenge in the variant in which they were submitted. For each of the algorithms we provide the name of the binary, the ID in the PACE challenge (if applicable), the location of its source code and the exact identifier of the program version in the GitHub repository. Most of the algorithms use Min-Fill as basic decomposition strategy.

- htd 0.9.9: "htd_gr2td_minfill_exhaustive.sh" (PACE-ID: 5)
  Available at `https://github.com/mabseher/htd`
  GitHub-Commit-ID: f4f9b8907da2025c4c0c6f24a47ff4dd0bde1626

- htd 1.0.0-beta1: "htd_gr2td_exhaustive.sh" (No participant of PACE)
  Available at `https://github.com/mabseher/htd`
  GitHub-Commit-ID: f04bfd256e0be0eb536ef04410b541b15206255d
- "tw-heuristic" (PACE-ID: 1)
  Available at `https://github.com/mrprajesh/pacechallenge`.
  GitHub-Commit-ID: 6c29c143d72856f649de99846e91de185f78c15f
- "tw-heuristic" (PACE-ID: 6)
  Available at `https://github.com/maxbannach/Jdrasil`
  GitHub-Commit-ID: fa7855e4c9f33163606a0677485a9e51d26d7b0a
- "tw-heuristic" (PACE-ID: 9)
  Available at `https://github.com/elitheeli/2016-pace-challenge`
  GitHub-Commit-ID: 2f4acb30b5c48608859ff27b5f4e217ee8346ca5
- "tw-heuristic" (PACE-ID: 10) [20]
  Available at `https://github.com/mfjones/pace2016`
  GitHub-Commit-ID: 2b7f289e4d182799803a014d0ee1d76a4de70c1f
- "flow_cutter_pace16" (PACE-ID: 12) [24]
  Available at `https://github.com/ben-strasser/flow-cutter-pace16`
  GitHub-Commit-ID: 73df7b545f694922dcb873609ae2759568b36f9f

htd already proved its efficiency on the instances of the PACE challenge by achieving the third place in the competition[5]. Because these instances are relatively small – *htd* is able to decompose any of the instances in less than two seconds – we want to present here also results for larger instances.

For this purpose we consider in our experiments instances which are used in competitions for quantified boolean formulas (QBFs). In fact, we decompose here the Gaifman graph underlying the CNF matrix of the QBFs thus ignoring the actual quantifier prefix (DP-based solvers for QBFs like dynQBF [14] handle the quantifier information internally and only require a decomposition of the matrix as input). We used DataSet 1 from the QBFEVAL'16 competition[6]. The data set contains 825 instances, most of them being significantly larger than the instances of the PACE challenge. In the experiments, each test run was limited to a run-time of at most 100 seconds and 32 GB of main memory. For the actual evaluation we use the testbed of the PACE challenge[7].

In Figure 1 we present the outcome of our experiments in a plot of the cumulative frequency of the obtained widths for each of the algorithms (IDs are those of the PACE challenge). A point (x, y) in the diagram indicates that y decompositions have a width of x or less. This means, an algorithm is better in terms of decomposition width when its line chart reaches the top with minimal width. We can see that already the "old" version of *htd* used in the PACE challenge outperforms its competitors in the region between width 250 and 1000. The recent version of *htd* improves upon these results and shows that only in regions with very high width, two of its competitors are able to decompose more instances.

[5] See `https://pacechallenge.wordpress.com/2016/09/12/here-are-the-results-of-the-1st-pace-challenge/`
[6] Available at `http://www.qbflib.org/TS2016/Dataset_1.tar.gz`
[7] Available at `https://github.com/holgerdell/PACE-treewidth-testbed`

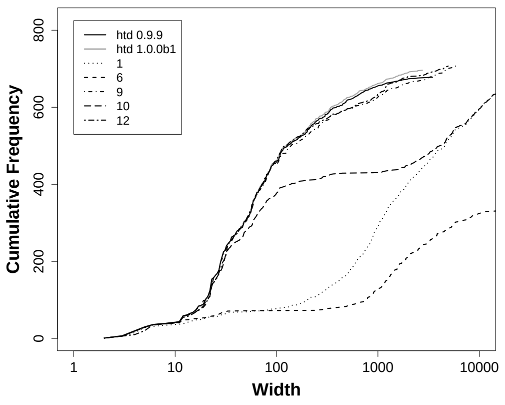

Fig. 1. Decomposition quality for instances of QBFEVAL2016 challenge

Iterations	Solved Instances (Total)	Solved Instances (Mean)	Total User-Time
1	491	98.2	5904.54
5	512	102.4	6375.08
10	512	102.4	5908.78

Table 1. Results for dynQBF using tree decompositions with low join node complexity

4.2 Using Customized Tree Decompositions to Increase Efficiency

Next, we highlight the usefulness of *htd*'s ability to optimize tree decompositions via an application scenario using the QBF solver dynQBF [14][8]. dynQBF makes extensive use of *htd*'s features (especially concerning induced edges) and benefits from customized tree decompositions as provided by *htd*.

For the following experiments we consider the 200 instances of the 2QBF track of the QBFEVAL2014 competition[9]. These instances differ from those used in Section 4.1 due to the fact that after decomposing we still need to run the dynamic programming algorithm. For each of the test runs we allow a single thread, 10 minutes execution time and 16 GB of main memory. This time, we only consider *htd* as decomposition library as, to the best of our knowledge, currently no other framework considers custom preferences for optimization.

Table 1 shows the results of the experiments running dynQBF (using *htd* 1.0.0-beta1). For each instance, five different tree decompositions were generated using the Min-Fill heuristics with different seeds. The first column shows the number of optimization iterations, i.e., the number of iterations after which the best known decomposition is returned. The second and third column show the total and average number of instances over the five repetitions which were solved successfully using the best decomposition found and the last column shows the total amount of solving time, restricted to the instances successfully solved.

As fitness function for the optimization we aim at minimizing the complexity of join nodes given by the formula $\sum_{j \in J} \prod_{c \in C_j} |\chi_c|$ where J is the set of join

[8] Available at `https://github.com/gcharwat/dynqbf/releases/tag/v0.3-beta`
[9] Available at `http://www.qbflib.org/TS2010/2QBF.tar.gz`

nodes and C_j is the set of children of node j. That is, we want the total sum of the products of the join node children's bag sizes to be as small as possible. The intuition is that when the given measure is small, the dynamic programming algorithm is more efficient in join nodes.

We can see that with a single iteration, i.e., without optimization, dynQBF solves about 98 out of 200 instances in average. The table illustrates that the number of solved instances in our example scenario increases when we use five iterations for the optimization phase. Therefore the total solving time increases. An interesting observation is the fact that using ten optimization iterations we need almost the same amount of time as without optimization, but we still can solve more instances.[10] That means that the (on average) four additional instances come for free in our scenario.

Note that with a statistical significance of over 99.95%, the width of the obtained decompositions does not change with the number of iterations, i.e., the customized tree decompositions indeed increase the efficiency of the dynQBF algorithm. Hence, by using an optimization function of not more than ten lines of code, one can already achieve improvements using *htd*.

5 Conclusion

In this paper we presented a new open-source framework for tree decompositions called *htd*. To the best of our knowledge, *htd* is the first software framework which aims for optimizing tree decompositions by other criteria than just the plain width. We gave an overview over its features and provided an introduction on how to use the library (for more details, see [5]). Moreover, we evaluated our approach by comparing the performance of *htd* and other participants of the Parameterized Algorithms and Computational Experiments Challenge 2016. The outcome of the evaluation indicates that the performance characteristics of the new framework are indeed encouraging. Furthermore, we showed that customizing tree decompositions is a powerful feature which can improve efficiency of dynamic programming algorithms using those decompositions.

For future work we want to further improve the built-in heuristics and algorithms in order to enhance the capabilities for generation of customized decompositions of small width. Furthermore we are currently working on making some exact algorithms for tree decompositions amenable to customization. Last, but not least, we invite researchers and software developers to contribute to the library as we try to initiate a joint collaboration on a powerful framework for graph decompositions and any input is highly appreciated.

Acknowledgments This work has been supported by the Austrian Science Fund (FWF): P25607-N23, P24814-N23, Y698-N23.

[10] When we use a pool of ten decompositions to choose from, the chance for obtaining an even better decomposition increases. However, no additional instance is solved when we change from five to ten iterations, but the run-time for the solved instances further decreases (compensating the time required for computing more decompositions).

References

1. Abseher, M.: htd 1.0.0-beta1 (2016), github.com/mabseher/htd/tree/v1.0.0-beta1
2. Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., Woltran, S.: The D-FLAT System for Dynamic Programming on Tree Decompositions. In: Proc. JELIA. LNCS, vol. 8761, pp. 558–572. Springer (2014)
3. Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., Woltran, S.: D-FLAT: Progress Report. Tech. Rep. DBAI-TR-2014-86, TU Wien (2014), available at http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2014-86.pdf
4. Abseher, M., Dusberger, F., Musliu, N., Woltran, S.: Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. In: Proc. IJCAI. pp. 275–282. AAAI Press (2015)
5. Abseher, M., Musliu, N., Woltran, S.: htd – A Free, Open-Source Framework for Tree Decompositions and Beyond. Tech. Rep. DBAI-TR-2016-96, TU Wien (2016), available at http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-96.pdf
6. Abseher, M., Musliu, N., Woltran, S.: Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. Tech. Rep. DBAI-TR-2016-94, TU Wien (2016), available at http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-94.pdf
7. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. Journal on Algebraic Discrete Methods 8(2), 277–284 (1987)
8. Arnborg, S., Proskurowski, A.: Linear time algorithms for NP-hard problems restricted to partial k-trees. Discrete Applied Mathematics 23(1), 11–24 (1989)
9. Bachoore, E.H., Bodlaender, H.L.: A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth. In: Proc. AAIM. LNCS, vol. 4041, pp. 255–266. Springer (2006)
10. Berry, A., Heggernes, P., Simonet, G.: The minimum degree heuristic and the minimal triangulation process. In: Proc. WG. pp. 58–70. Springer (2003)
11. Bertelè, U., Brioschi, F.: On non-serial dynamic programming. Journal of Combinatorial Theory, Series A 14(2), 137–148 (1973)
12. Bodlaender, H.L., Koster, A.M.C.A.: Combinatorial Optimization on Graphs of Bounded Treewidth. The Computer Journal 51(3), 255–269 (2008)
13. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Information and Computation 208(3), 259–275 (2010)
14. Charwat, G., Woltran, S.: Dynamic Programming-based QBF Solving. In: Proceedings of the 4th International Workshop on Quantified Boolean Formulas. CEUR Workshop Proceedings, vol. 1719, pp. 27–40 (2016)
15. Clautiaux, F., Moukrim, A., Négre, S., Carlier, J.: Heuristic and meta-heuristic methods for computing graph treewidth. RAIRO Operations Research 38, 13–26 (2004)
16. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
17. Dourisboure, Y.: Compact routing schemes for generalised chordal graphs. Journal of Graph Algorithms and Applications 9(2), 277–297 (2005)
18. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science, Springer (1999)
19. Ganzow, T., Gottlob, G., Musliu, N., Samer, M.: A CSP Hypergraph Library. Tech. Rep. DBAI-TR-2005-50, TU Wien (2005), available at http://www.dbai.tuwien.ac.at/proj/hypertree/csphgl.pdf
20. Gaspers, S., Gudmundsson, J., Jones, M., Mestre, J., Rümmele, S.: Turbocharging Treewidth Heuristics. In: Proc. IPEC (2016), to appear.

21. Gogate, V., Dechter, R.: A Complete Anytime Algorithm for Treewidth. In: Proc. UAI. pp. 201–208. AUAI Press (2004)
22. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions and Tractable Queries. Journal of Computer and System Sciences 64(3), 579–627 (2002)
23. Halin, R.: S-functions for graphs. Journal of Geometry 8, 171–186 (1976)
24. Hamann, M., Strasser, B.: Graph bisection with pareto-optimization. In: Proc. ALENEX. pp. 90–102. SIAM (2016)
25. Hammerl, T., Musliu, N.: Ant colony optimization for tree decompositions. In: Proc. EvoCOP. LNCS, vol. 6022, pp. 95–106. Springer (2010)
26. Hammerl, T., Musliu, N., Schafhauser, W.: Metaheuristic Algorithms and Tree Decomposition. In: Handbook of Computational Intelligence, pp. 1255–1270. Springer (2015)
27. Jégou, P., Terrioux, C.: Bag-Connected Tree-Width: A New Parameter for Graph Decomposition. In: Proc. ISAIM. pp. 12–28 (2014)
28. Kjaerulff, U.: Optimal decomposition of probabilistic networks by simulated annealing. Statistics and Computing 2(1), 2–17 (1992)
29. Kloks, T.: Treewidth, Computations and Approximations, LNCS, vol. 842. Springer (1994)
30. Koster, A.M.C.A., van Hoesel, S.P.M., Kolen, A.W.J.: Solving Frequency Assignment Problems via Tree-Decomposition 1. Electronic Notes in Discrete Mathematics 3, 102–105 (1999)
31. Larranaga, P., Kujipers, C.M., Poza, M., Murga, R.H.: Decomposing bayesian networks: Triangulation of the moral graph with genetic algorithms. Statistics and Computing 7 (1), 19–34 (1997)
32. Lauritzen, S.L., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems. Journal of the Royal Statistical Society, Series B 50, 157–224 (1988)
33. Morak, M., Musliu, N., Pichler, R., Rümmele, S., Woltran, S.: Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In: Proc. LION. LNCS, vol. 7219, pp. 130–144. Springer (2012)
34. Musliu, N.: An iterative heuristic algorithm for tree decomposition. Studies in Computational Intelligence, Recent Advances in Evolutionary Computation for Combinatorial Optimization 153, 133–150 (2008)
35. Musliu, N., Schafhauser, W.: Genetic algorithms for generalized hypertree decompositions. European Journal of Industrial Engineering 1(3), 317–340 (2007)
36. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Mathematics And Its Applications, Oxford University Press (2006)
37. Robertson, N., Seymour, P.: Graph minors. III. Planar tree-width. Journal of Combinatorial Theory, Series B 36(1), 49–64 (1984)
38. Robertson, N., Seymour, P.: Graph minors. X. Obstructions to tree-decomposition. Journal of Combinatorial Theory, Series B 52(2), 153 – 190 (1991)
39. Shoikhet, K., Geiger, D.: A Practical Algorithm for Finding Optimal Triangulations. In: Proc. AAAI/IAAI. pp. 185–190. AAAI Press / The MIT Press (1997)
40. Tarjan, R.E., Yannakakis, M.: Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM Journal on Computing 13, 566–579 (1984)
41. van Wersch, R., Kelk, S.: Toto: An open database for computation, storage and retrieval of tree decompositions. Discrete Applied Mathematics 217, 389–393 (2017)
42. Xu, J., Jiao, F., Berger, B.: A tree-decomposition approach to protein structure prediction. In: Proc. CSB. pp. 247–256 (2005)