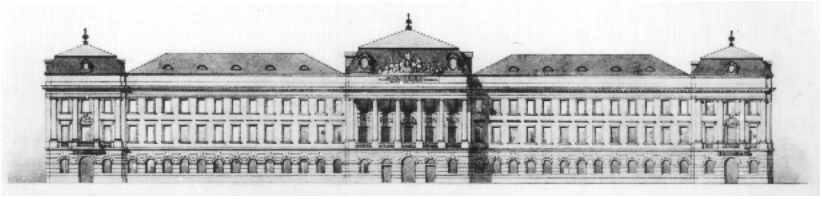
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

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

DBAI-TR-2016-96

Michael Abseher, Nysret Musliu, Stefan Woltran

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sek@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2016

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2016-96, 2016

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

Abseher Michael [1] **Nysret Musliu** [1] **Stefan Woltran** [1]

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. Practical experience showed that generating decompositions of small width is not the only crucial ingredient towards efficiency. In fact, additional features of tree decompositions are very important for good performance in practice. To turn the theoretical potential of structural decomposition into successful applications, we thus require implementations of decomposition methods which allow for a smooth integration and moreover are easily extendible and adaptable to the domain-specific needs. To this end, we present *htd*, a free and open-source library for graph decomposition. The current version of *htd* includes efficient implementations of several heuristic approaches for tree decomposition and offers various features for normalization and customization of the decomposition. The aim of this report is to present the main features of *htd* together with an experimental evaluation underlining the effectiveness and efficiency of the implementation.

[1]TU Wien. E-mail: {abseher, musliu, woltran}@dbai.tuwien.ac.at

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

Contents

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 [Bertelè and Brioschi, 1973; Halin, 1976; Robertson and Seymour, 1984], hypertree decompositions [Gottlob *et al.*, 2002] (of hypergraphs) and branch decompositions [Robertson and Seymour, 1991] to mention just a few. The concept of tree decompositions gained special attention as it can be shown that many NP-hard search problems become tractable when the parameter *treewidth*, i.e., the minimum width (maximum bag size - 1) over all tree decompositions of the problem instance, is bounded by some constant k [Arnborg and Proskurowski, 1989; Niedermeier, 2006; Bodlaender and Koster, 2008]. A problem exhibiting tractability by bounding some problem-inherent constant is also called fixed-parameter tractable (FPT) [Downey and Fellows, 1999].

The standard technique for solving problems 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 [Niedermeier, 2006]. 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 inference problems in probabilistic networks [Lauritzen and Spiegelhalter, 1988], frequency assignment [Koster *et al.*, 1999], computational biology [Xu *et al.*, 2005], and logic programming [Morak *et al.*, 2012].

From a theoretical point of view the actual width k is the crucial parameter towards efficiency for FPT algorithms that use tree decompositions. However, as recently stressed by Gutin [2015], to turn the concept of FPT to practical success, more empirical work is required.

In terms of FPT algorithms for treewidth, experience shows that even decompositions of the same width 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 run-time [Abseher *et al.*, 2015; Abseher *et al.*, 2016b; Jégou and Terrioux, 2014; Morak *et al.*, 2012]. Therefore we see a need to offer a specialized framework, allowing to obtain customized decompositions, i.e., decompositions which reflect certain preferences of the developer, in order to optimally fit to the dynamic programming algorithm in which they are used.

To cover the aforementioned points, in this report we present a free, open-source solution which supports a vast amount of graphs and different types of decompositions. Our framework can be easily extended as we provide programming interfaces for (almost) all classes and so one does not need to re-invent the wheel at any place. In detail, the design goals for our framework are as follows:

- Clean, easy-to-use interfaces
- Run-time and memory efficiency
- Utmost flexibility and extensibility
- Support for a variety of graph and decomposition types

- Support for a wide range of convenience features like various normalization strategies and automated modifications of decompositions directly in the context of the library in order to keep the code clean and structured

The software library is called *htd* and is availabe under `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.

In the remainder of this work, we provide a detailed description of the features of *htd*, shed some light at crucial algorithm decisions, illustrate its usage in some example scenarios, and we also give an experimental evaluation comparing the tree decomposition heuristics currently offered by *htd* to other implementations.

In order to find out how *htd* compares to other approaches for computing tree decompositions, *htd* is one of the participants of the "First Parameterized Algorithms and Computational Experiments Challenge" ("PACE16")[1] and it was ranked at the third place in the heuristics track. The results of *htd* are very close to those of the heuristic approaches ranked at the first two places. This underlines not only that *htd* is rich in features helping to make the development of dynamic programming algorithms more comfortable, but also that it is very competitive and efficient when compared to other approaches.

2 Background

Tree decomposition is a technique often applied for solving NP-hard problems. The underlying intuition 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 [Robertson and Seymour, 1984; Bodlaender and Koster, 2010].

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 $\chi : N \rightarrow 2^V$ 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, for which a root $r \in N$ is explicitly defined.

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

Definition 2. *Given a graph $G = (V, E)$, a* normalized *(or* nice*) tree decomposition of G is a rooted tree decomposition T 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 [Kloks, 1994]. While constructing an optimal tree decomposition, i.e. a decomposition with minimal width, is intractable [Arnborg *et al.*, 1987], researchers 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 [Shoikhet and Geiger, 1997; Gogate and Dechter, 2004; Bachoore and Bodlaender, 2006]; greedy heuristic algorithms include Maximum Cardinality Search (MCS) [Tarjan and Yannakakis, 1984], Min-Fill heuristic [Dechter, 2003], and Minimum Degree heuristic [Berry *et al.*, 2003], to mention just a few. Metaheuristic techniques have been provided in terms of genetic algorithms [Larranaga *et al.*, 1997; Musliu and Schafhauser, 2007], ant colony optimization [Hammerl and Musliu, 2010], and local search based techniques [Kjaerulff, 1992; Clautiaux *et al.*, 2004; Musliu, 2008]. A more detailed description of tree decomposition techniques is given in the recent surveys [Bodlaender and Koster, 2010; Hammerl *et al.*, 2015].

For a given graph G the treewidth can be found from its triangulation. Further we will give basic definitions, explain how the triangulation of graph can be constructed and show the relation between the treewidth and the triangulated graph.

Two vertices u and v of graph $G = (V, E)$ are neighbors, if they are connected with an edge $e \in E$. The neighborhood of vertex v is defined as: $N(v) := \{w | w \in V, (v, w) \in E\}$. A set of vertices is clique if there is an edge between each pair of vertices. An edge connecting two non-adjacent vertices which are part of a cycle is called chord. The graph is triangulated if there exists a chord between any pair of non-adjacent vertices in every cycle of length larger than 3.

A vertex of a graph is simplicial if its neighbors form a clique. An ordering of nodes $\sigma(1, 2, \ldots, n)$ of V is called a perfect elimination ordering for G if for any $i \in \{1, 2, \ldots, n\}$, $\sigma(i)$ is a simplicial vertex in $G[\sigma(i), \ldots, \sigma(n)]$ [Clautiaux *et al.*, 2004]. In [Fulkerson and Gross, 1965] it is shown that the graph G is triangulated if and only if it has a perfect elimination ordering. Given an elimination ordering of nodes the triangulation H of graph G can be constructed as following. Initially $H = G$, then in the process of elimination of vertices, the next vertex in order to be eliminated is made a simplicial vertex by adding new edges in order to connect all its neighbors in current G and H. The vertex is then eliminated from G. This process is repeated for all vertices in the ordering. A more detailed description of the algorithm for constructing a graph's triangulation for a given elimination ordering is found in [Koster *et al.*, 2001].

The treewidth tw of a triangulated graph can be calculated based on its cliques. For a given triangulated graph the treewidth is equal to its largest clique minus 1 [Gavril, 1972]. Moreover, the largest clique of triangulated graph can be calculated in polynomial time. The complexity of calculation of the largest clique for the triangulated graphs is $O(|V|+|E|)$ [Gavril, 1972]. For every graph $G = (V, E)$, there exists a triangulation of G, $\overline{G} = (V, E \bigcup E_t)$, with $tw(\overline{G}) = tw(G)$. Thus, finding the treewidth of a graph G is equivalent to finding a triangulation $\overline{G}$ of G with minimum clique size (for more information see [Koster *et al.*, 2001]).

3 A General Framework for Tree Decompositions and Beyond

In this section we want to have an in-depth look at some important properties of the new framework. During our work with D-FLAT [Abseher *et al.*, 2014], a framework for easy prototyping of dynamic programming on tree decompositions, we faced the problem that existing implementations for graph decomposition algorithms often only minimize the width. That is, they deliver a tree decomposition without possibility to transparently customize the result. Hence, post-processing outside the library is needed in order to obtain a decomposition which reflects certain preferences of a developer and which fits well to given dynamic programming algorithms. Furthermore, existing algorithms are often hard to adapt because at design time certain capabilities and mechanisms (like assigning arbitrary labels to the resulting decompositions automatically) were not considered and so extensions of functionality often requires the rewriting of huge portions of the old code.

To circumvent all these problems, the proposed library called *htd* is free, open-source software and it is available under `https://github.com/mabseher/htd`. The software was developed with the goal to serve the needs of virtually any algorithm related to graph decompositions. In the following we will highlight the library's main characteristics.

3.1 Support for a Variety of Input Graph Types

Since the library shall be able to decompose any given graph type, *htd* supports by default a variety of them to fit like a glove to the actual application domain where our library is applied. Indeed all input graphs can be stored in a data structure which is able to handle multi-hypergraphs, i.e., hypergraphs with potentially duplicated hyperedges. E.g., we could also store a directed graph in a data structure for multi-hypergraphs, but multi-hypergraphs are, for instance, not aware of the concept of incoming or outgoing neighbors and also reachability is defined differently than in the case of directed graphs. In order to enhance functionality, to enforce semantic coherence and to shift programming effort from the developer using the library to our framework, *htd* offers separate data types and programming interfaces for storing (multi-)hypergraphs, directed and undirected (multi-)graphs, trees and paths.

For each graph type, *htd* also offers an implementation which is able to deal with custom names so that instead of working with plain vertex and edge identifiers in terms of numeric integers one can additionally address a specific vertex or edge by its name. To fit the needs of dynamic programming algorithms in a very general and convenient way, one can use any data type which

is equality-comparable and which provides functionality for returning its hash code (like character strings) as an alias for the name of a vertex or an edge.

Furthermore, *htd* allows to add custom labels of any data type to the vertices and (hyper)edges of a given graph by providing appropriate wrappers for each graph type. These labels can, for instance, be used to store truth assignments for the endpoints of an hyperedge in case that hyperedges represent clauses for the problem of boolean satisfiability.

One big advantage of having a built-in support for labeled graph types is the fact that with the functionality one can keep the productive code clean and simple because one only needs a single graph representation in memory and no conversion or mapping of the graph structure between internal library code and the developed algorithm is necessary.

3.2 Support for a Variety of Decomposition Types

Clearly, from a graph decomposition library we expect the ability to decompose graphs. To serve this purpose, *htd* offers several decomposition methods by default and a wide range of interfaces allows to extend *htd*'s functionality easily and without even having to re-compile the library. Each decomposition algorithm in the context of *htd* takes an (potentially disconnected) input graph in any supported representation and constructs a decomposition of the requested type. The library distinguishes between four types of decomposition algorithms:

- **Graph Decomposition Algorithms**

  ... return a new, labeled multi-hypergraph GD where the bag of each vertex in GD is a subset of the vertices in G. This is the most general decomposition type currently supported by *htd*.

- **Tree Decomposition Algorithms**

  ... return a new, labeled tree TD where the bag of each vertex in TD is a subset of the vertices in G such that all criteria for tree decompositions are satisfied. In the first step, the default implementation of a tree decomposition algorithm in *htd* uses bucket-elimination [Dechter, 1999; Dermaku *et al.*, 2008] based on a generated eliminating ordering to obtain a tree decomposition for each connected component of the input graph. Afterwards, it connects the trees in the forest to a single tree by adding additional edges where appropriate. The vertex ordering required for bucket-elimination is obtained via the library's default ordering algorithm which the developer can select before the decomposition algorithm is invoked.[2]

- **Path Decomposition Algorithms**

  ... return a new, labeled path PD where the bag of each vertex in PD is a subset of the vertices in G such that all criteria for tree decompositions are satisfied. The default implementation constructs a tree decomposition of the input graph and then manipulates it by rearranging the join node's children in order to obtain a path structure.

[2] Apart from employing a custom algorithm given by the developer, *htd* currently provides default implementations for Min-Fill, Minimum Degree and Maximum Cardinality Search vertex elimination orderings. If the developer does not specify the ordering algorithm to use, the default setting is Min-Fill.

- **Hypertree Decomposition Algorithms**

  ... return a new, labeled tree TD where the bag of each vertex in TD is a subset of the vertices in G. Additionally, each vertex of TD has assigned a second label, consisting of a subset of hyperedges of the input graph such that the corresponding bag content is a subset of the set union of vertices contained in these hyperedges. That is, *htd*'s default implementation computes a generalized hypertree decomposition [Gottlob *et al.*, 2009].

  The current implementation starts by first generating a tree decomposition and then we solve the SET COVER problem for each of its bags. That is, we compute for each bag the minimum-cardinality set of all hyperedges such that each vertex in the bag has at least one hyperedge in which it is contained.

We can see that basically every algorithm is constructed in a very modular way, i.e., we only require the input graph which shall be decomposed and afterwards everything is up to the concrete implementations. This leads to light-weight interfaces and high flexibility for both developers "just" using the library and those who want to contribute to the library. For instance, although the default implementations of the algorithms rely on the simple bucket-elimination procedure, there is absolutely no need for a developer contributing to the library to use bucket-elimination or vertex elimination orderings at all.

Finally, before we have a deeper look at flexibility and extensibility of *htd*, we should have a quick look at an additional feature concerning decomposition algorithms which can be very helpful in practice: For reduced post-processing effort on the developer-side, *htd* offers the concept of manipulations which can be applied to a computed decomposition. The term "manipulation" in the context of *htd* refers to operations which manipulate the structure of the decomposition, like making a tree decomposition nice, or add/remove/change certain labels, like adding additional information to a decomposition node. Note that also the built-in path decomposition algorithm of *htd* is such a manipulation.

Manipulations can always be applied to a given decomposition but one can also specify the list of desired manipulations when calling the decomposition algorithm. In the latter case, the computed decomposition will be returned with the desired operations automatically applied. This helps to keep the code at developer-side clean because one does not have to do any post-processing in order to get the manipulations applied.

3.3 Automated Optimization of Decompositions

One of the main novelties of *htd* compared to existing implementations of decomposition algorithms is the support for automated optimization. In order to support the special needs of certain dynamic programming algorithms, like minimizing the number of join nodes, or even to allow complex optimizations, like the prioritization of selected vertices with respect to their average position in the decomposition, *htd* supports two optimization strategies.

The first strategy implemented by the library is an iterative approach which computes a (user-definable) number of decompositions and finally returns the decomposition for which the fitness evaluation (the result of the provided fitness function), is maximal. This built-in strategy allows to

mimic the approach which proved successfully in [Abseher *et al.*, 2015; Abseher *et al.*, 2016b], namely the computation of a pool of tree decompositions and afterwards selecting the optimal one via machine learning, by only writing a few lines of code.

The second strategy relies on the fact that one can select any vertex of a tree as its root and the outcome of this reallocation will still be a tree. Based on this fact, the second optimization strategy allows to automatically select the node as root of the tree decomposition for which the fitness evaluation for the tree decomposition rooted at the respective node is maximal. For utmost flexibility and highest performance one can select from a wide range of (custom) criteria which aim at narrowing down the subset of vertices of a decomposition which shall be considered as new root node. This is important especially for large graphs as exhaustively trying out all possible choices might be expensive.

Note that the two approaches can indeed be combined in order to improve the result of the optimization step. Additionally, one can assign also priorities to each level of a fitness evaluation if multi-criteria optimization is needed.

3.4 High Level of Flexibility and Extensibility

One of the main limitations of many software libraries is the fact that they are often developed as a by-product of some application and so the complete functionality is, in many cases, tailored towards a specialized application domain and extensions or adaptations are hard to implement. *htd* is different in that sense as it is developed with the goal of utmost flexibility and extensibility and it is developed independently from a concrete application domain[3]. Currently, the library provides around 80 interfaces for almost all parts of the library, allowing to easily replace, improve and extend functionality. Furthermore, *htd* provides explicit factory classes for most interfaces so that one can set new default implementations without even having to re-compile the library.

3.5 Working with the Library

After sketching some important characteristics of *htd*, in this section we now want to give an overview of the general workflow of how to compute decompositions via *htd* and we provide an example of an application scenario to illustrate how the library can be used in practice.

Figure 1 depicts the concept behind *htd*. At first, one clearly needs to parse the input and maybe do some conversion or pre-processing in order to obtain a graph representation of the input instance which we can directly feed into the decomposition algorithm. The decomposition algorithm may be either one of the built-ins of *htd* or a custom one provided by the developer. Optionally, the developer can provide additional information to the decomposition algorithm, like a custom vertex ordering as needed by bucket-elimination. Furthermore, it is possible to request different manipulations of the resulting decomposition, like computing additional labels or making a tree decomposition nice.

[3]Note that *htd* also provides a command-line application, named *htd_main* which is a small, light-weight front-end for the library that allows to trigger the main functions of *htd*. The front-end is fully configurable in order to easily investigate the effects of modifications to *htd*.

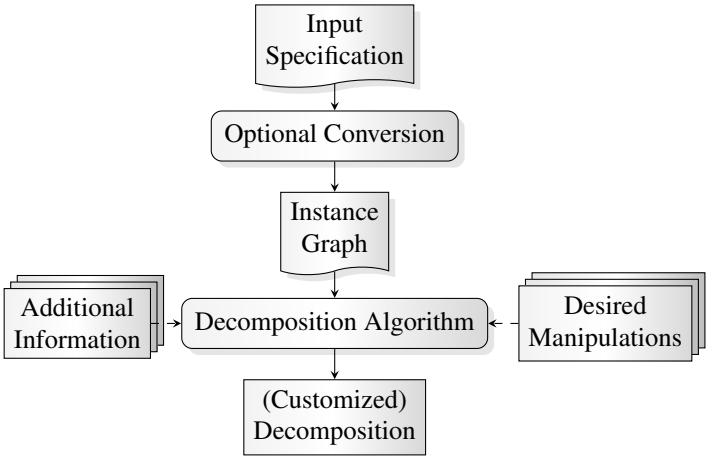


Figure 1 Workflow for computing customized decompositions using *htd*.

Note that the library supports a developer in any of the depicted steps, that is, it defines interfaces allowing the parsing of input files and it provides implementations for various graph types, decomposition algorithms and manipulation operations. Furthermore, the large collection of algorithms in *htd* contains lots of convenience functions.

Currently, *htd* is used in several projects, like D-FLAT [Abseher *et al.*, 2014], a framework for rapid-prototyping of dynamic programming algorithms on tree decompositions, dynASP [Morak *et al.*, 2010; Fichte *et al.*, 2016], an answer-set programming solver which is based on dynamic programming on tree decompositions, or dynQBF [Charwat and Woltran, 2016], a solver for quantified boolean formulae based on dynamic programming and binary decision diagrams.

In these projects it turned out that a very helpful "bonus" functionality of our library is the following: All built-in decomposition algorithms of *htd* automatically provide for each bag the set of induced edges, i.e., the set of all edges which form a subset of the bag content. This means in a dynamic programming algorithm there is no need any more to filter the full set of edges of the input graph for each bag in order to find out which of them are affected by the respective bag. This can be a very time-consuming task for large graphs. *htd* provides this important information (almost) for free, significantly accelerating the given applications.

```
...
// Create a new management instance (with ID 1) for the current thread.
htd::LibraryInstance * manager = htd::createManagementInstance(1);

// Import some graph in the format of the PACE challenge from stdin.
htd::GrFormatImporter importer(manager);
htd::IMultiGraph * graph = importer.import(std::cin);

// Get default decomposition algorithm.
htd::ITreeDecompositionAlgorithm * algorithm =
    manager->treeDecompositionAlgorithmFactory()
        .getTreeDecompositionAlgorithm();
```

```
// Set desired manipulations for algorithm.
algorithm.addManipulationOperation
    (new htd::NormalizationOperation(manager));

// Decompose the provided graph.
htd::ITreeDecomposition * td =
    algorithm->computeDecomposition(*graph);

// Output the width of the obtained tree decomposition.
std::cout << "Width: " << (td->maximumBagSize() - 1) << std::endl;
...
```

Listing 1 Example source code (C++) how to use *htd* in practice.

To underline how easy *htd* can be applied in practice, in Listing 1 we give a short working example in terms of the only five lines long C++ source code sufficient to compute a nice tree decomposition of a given input instance in the format of the "PACE16" challenge[4]. The first two lines of the source code take care of importing the input graph. The third line then gets the default tree decomposition algorithm of *htd*. All what remains is to set the manipulation operation for normalized tree decompositions and to decompose the graph. The last line of our example then outputs the width of the decomposition, but one may also proceed with a dynamic programming algorithm.

Note that the above source code is more or less a minimal working example, but one is free to implement any algorithm one can think of and it will work with the library as long as it implements the interfaces of *htd* properly. In the following list we give some examples for important interfaces of *htd*:

- `htd::IMultiHypergraph`

  All (custom) graph classes must implement the interface for hypergraphs with potentially duplicated edges. It provides the functionality which is common to all graph types, like accessing its vertices and edges or, for instance, to get the neighbors of a vertex.

- `htd::ITreeDecomposition`

  A tree decomposition in *htd* is a special case of a `htd::IMultiHypergraph` in which each vertex has a bag label and where the underlying graph is a tree. Like mentioned before, apart from the other functionality one would expect from a tree structure, the `htd::ITreeDecomposition` interface additionally defines a function to retrieve the hyperedges of the input graph which are induced by a bag.

- `htd::ITreeDecompositionAlgorithm`

  This interface must be implemented by all tree decomposition algorithms in the context of *htd*. It defines functionality to compute a `htd::ITreeDecomposition` of some

[4]The format specification is given under the following link:
`https://pacechallenge.wordpress.com/track-a-treewidth/`

`htd::IMultiHypergraph`. Apart from decomposing a given input graph, *htd*'s tree decomposition algorithms automatically apply provided manipulation operations.

- `htd::IDecompositionManipulationOperation`

  As mentioned in Section 3.2, manipulation operations are an important part of the library and cover all operations which are made to a decomposition's structure or its labels. For each type of decomposition, *htd* provides a separate interface in order to ensure that only compatible operations are performed during the computation of a decomposition.

4 Developer Documentation

htd is a relatively small piece of software for efficiently computing decomposition of large graphs and hypergraphs. The library is developed with the goal to optimally fit the needs of a wide range of dynamic programming algorithms. Although its code base is small, consisting of only some hundred thousand lines of source code, the library is able to efficiently decompose graphs containing millions of vertices and it offers various interfaces to provide the developers of dynamic programming algorithms exactly with those features they need without having to pay the price for functionality they don't need.

To employ *htd* for one's purposes and to fully exploit its potential it is important to know about its interface structure, the functionality each of the interfaces offers and how the underlying algorithms interact. In this section we therefore want to give a detailed introduction to *htd*'s application programming interface.

4.1 Introduction

Before we dig into the details of the *htd* software library, in this section we want to give an introduction to its macrostructure in order to make the remainder of the documentation easier to follow. *htd* is structured roughly as follows:

- Input graphs
- Graph decompositions
- Decomposition algorithms
- Manipulation operations
- Normalization operations
- Utility functions

Before using any of these features, the first thing to start of when developing a new application based on *htd* is creating a so-called *library instance* of *htd*. A library instance acts as a central point

of management allowing to configure the default settings for any algorithm within the library. Therefore we will use the term "manager" as a synonym for the term "library instance" for the remainder of this work.

An example of how to properly initialize the manager is given in Listing 2. Note that one can use more than one manager per application, for example, in situations where it is desired to have different configurations per thread. After creating the new manager, it may be used by algorithms to incorporate the developer's preferences. This is ensured by the fact that the manager contains a collection of factory classes (one for each interface available in *htd*) and so each algorithm can use exactly those settings which were defined by the developer.

```
// Create a new management instance (with ID 1) for the current thread.
htd::LibraryInstance * manager = htd::createManagementInstance(1);
```

Listing 2 Example source code (C++) how to initialize a library instance of *htd*.

For a minimal working example, the code shown in Listing 2 suffices due to the fact that *htd* provides efficient default implementations for each of the interface classes. Nevertheless, one may decide to use a different algorithm provided by *htd* or one can even use its own implementation. A toy example how to set and retrieve the default implementation of the graph class is given in Listing 3.

```
// Create a new management instance (with ID 1) for the current thread.
htd::LibraryInstance * manager = htd::createManagementInstance(1);

// Change the default implementation for graphs.
manager->graphFactory().setConstructionTemplate(new MyFancyGraphClass());

// Get a new instance of the graph class.
htd::IMutableGraph * g = manager->graphFactory().getGraph();
```

Listing 3 Example source code (C++) how to change the default graph type.

Note that in *htd*, factory classes always take control over the memory region pointed to by the argument of the function "setConstructionTemplate" in order to avoid copying the instance. Therefore, one must not free the pointed-to memory manually because this is done automatically by the factory class.

In the remainder of this chapter we present the functionality for each group of data structures and each algorithm category. All subsequent sections will rely on the fact that a properly initialized and configured library instance named "manager" already exists.

4.2 Graph Types

htd distinguishes five graph types, namely hypergraphs (graphs with hyperedges), (undirected) graphs, directed graphs, trees and paths. According to this distinction, *htd* provides the following interface classes:

- `htd::IHypergraph`, `htd::IMultiHypergraph`

Hypergraphs are the most general graph type in *htd* as they allow using hyperedges, i.e., edges with an arbitrary number of endpoints. Self-loops, induced by hyperedges containing the same vertex multiple times, are allowed.

Inheritance:

Each `htd::IHypergraph` is a `htd::IMultiHypergraph`.

- `htd::IGraph`, `htd::IMultiGraph`

  Graphs in the context of *htd* are hypergraphs where each hyperedge has exactly two endpoints. Again, self-loops are allowed.

  Inheritance:

  Each `htd::IGraph` is a `htd::IHypergraph` and each `htd::IMultiGraph` is a `htd::IMultiHypergraph`.

- `htd::IDirectedGraph`, `htd::IDirectedMultiGraph`

  Directed graphs in the context of *htd* are graphs where the order of endpoints matters. In addition to the functionality of graphs, the corresponding interface classes for directed graphs allow to easily retrieve the incoming and outgoing neighbors of a vertex. Self-loops are allowed.

  Inheritance:

  Each `htd::IDirectedGraph` is a `htd::IGraph` and each `htd::IDirectedMultiGraph` is a `htd::IMultiGraph`.

- `htd::ITree`

  This interface is implemented by all tree classes and it allows to access the tree.

  Inheritance:

  Each `htd::ITree` is a `htd::IGraph`.

- `htd::IPath`

  This interface is implemented by all path classes and it allows to access the path.

  Inheritance:

  Each `htd::IPath` is a `htd::ITree`.

The graph classes above only provide read-only methods in order to maintain a proper inheritance hierarchy. To illustrate the problem, think about the common statement that any tree is a graph. On the one hand, when looking at the aforementioned sentence from the read-only perspective there is no doubt that it is true. On the other hand, when looking at it from the write-perspective, i.e. when we want to modify the graph, the statement is no longer valid as we cannot add arbitrary edges to a tree without potentially violating the acyclicity requirement. To circumvent this problem, each read-only graph class has its mutable counterpart which is denoted by adding

the character sequence “Mutable” between the capital letter ‘I’ and the remainder of the graph class name. For instance, the mutable interface for the interface class `htd::IHypergraph` is `htd::IMutableHypergraph`.

The read-only graph classes support the aforementioned inheritance hierarchy (each path is a tree, each tree is a graph ...) while the mutable graph classes, which extend the immutable ones by adding the specialized functionality to modify the underlying graph, are not subject to inheritance.

The graph types containing the character sequence “Multi” in their names, that are `htd::IMultiHypergraph`, `htd::IMultiGraph` and `htd::IDirectedMultiGraph`, allow edge duplicates while all other graph types assume that edges with the same endpoints (in identical order) refer to the very same edge instance.

4.2.1 Labeled Graph Types

Furthermore, for each graph type there is also a labeled counterpart which allows to assign custom labels to the vertices and edges of the graph. One example for a labeled graph type is for instance the interface class htd::ILabeledTree (with its mutable version htd::IMutableLabeledTree). Using these two interfaces one can, in addition to the basic functionality provided by the tree classes, assign arbitrary, customizable labels to the vertices and edges of the tree. The labeled versions of the other graph types provide analogous functionality for the respective basic type.

4.2.2 Named Graph Types

While standard graph types implemented in *htd* take integers as identifiers for vertices and edges in order to save resources, input graphs often use different data types for referencing vertices and edges. To provide the developer with enough flexibility to use arbitrary data types as identifiers, *htd* offers for each graph type a template class which automatically takes care of the efficient one-to-one mapping between the identifier used in the context of the input graph and the integer identifiers used by *htd*. We will subsequently refer to those template wrappers for graph as “named graph types”.

One example for such a C++ template class representing a named graph type is the class `htd::NamedGraph<std::string,std::string>` which wraps an instance of `htd::MutableLabeledGraph` in such a way that instead of using an automatically assigned integer to reference vertices and edges, one can now use a (unique) std::string. Similarly, the class `htd::NamedGraph<int, std::string>` allows to use (custom) integer values as vertex identifiers and strings as edge identifiers. Generally, one can use any data type as identifier which provides a hash code and which is equality-comparable. The named versions of the other graph types work in an identical manner for the respective basic type.

4.3 Decomposition Types

Initially, the focus of *htd* was the efficient computation of tree decompositions only. Nevertheless, in order to perfectly fit the distinct needs of developers of dynamic programming algorithms, *htd* at the current stage of development offers support for four basic types of decompositions:

- `htd::IGraphDecomposition`

  Graph decompositions are the most general type of decompositions in the context of *htd*. The interface offers all possibilities of `htd::IMultiHypergraph` and extends it by providing functionality to add a bag information as well as assigning vertex and edge labels. The bag information in the context of *htd* is always a sorted vector of vertices from the original graph from which the decomposition was computed. Note that graph decompositions allow for multiple edges and disconnected graphs.

  Inheritance:

  Each `htd::IGraphDecomposition` is a `htd::IMultiHypergraph`.

- `htd::ITreeDecomposition`

  While plain graph decompositions represented by the decomposition interface `htd::IGraphDecomposition` are basically nothing more than arbitrary graphs which can take the information about the bag content assigned to a vertex as well as custom additional labels for all vertices and edges, tree decompositions represented by the interface `htd::ITreeDecomposition` offer much more functionality.

  For instance, each tree decomposition offers functions to efficiently access its join, introduce and forget nodes. Furthermore, one can use the write-capable extension interface `htd::IMutableTreeDecomposition` to manipulate the tree not only by adding children to nodes but also by adding parents (useful for creating intermediate nodes), deleting whole subtrees and the re-attachment of nodes to totally different parents, thus allowing almost any tree manipulation one can think of.

  Inheritance:

  Each `htd::ITreeDecomposition` is a `htd::IGraphDecomposition`.

- `htd::IPathDecomposition`

  Path decompositions are tree decompositions without join nodes. In order to support this type of decompositions optimally, *htd* provides a special interface for path decompositions. By checking inheritance from this interface, algorithms sometimes can take shortcuts as they can rely on the fact that each vertex has at most one child.

  Inheritance:

  Each `htd::IPathDecomposition` is a `htd::ITreeDecomposition`.

- `htd::IHypertreeDecomposition`

  The most involved type of graph decompositions currently supported by *htd* are hypertree decompositions [Gottlob *et al.*, 2002]. Hypertree decompositions extend the properties of tree decompositions by additionally adding functionality to retrieve the subsets of hyperedges from the input graph which are needed to cover the bag content of the node under focus.

Note that, generally, one could also implement this functionality by adding a custom label to each vertex of a tree decomposition, but by implementing this interface, efficiency can be increased.

Inheritance:

Each `htd::IHypertreeDecomposition` is a `htd::ITreeDecomposition`.

Analogous to the interfaces specific to the graph types, again we distinguish between read-only and write-capable decomposition interfaces. In the same way as before, the name of the write-capable interface for the four read-only decomposition interfaces is constructed by adding the character sequence "Mutable" between the letter 'I' and the remainder of the graph class name. As one example, `htd::IMutableTreeDecomposition` is the mutable counterpart of the interface class `htd::ITreeDecomposition`.

4.4 Decomposition Algorithms

In order to support the aforementioned decomposition types it is necessary for a good code design to distinguish different types of decomposition algorithms. The following algorithms take an input graph of type `htd::IMultiHypergraph`, that is, they can be fed with any graph type *htd* supports, and they return a pointer to a decomposition of the corresponding type.

- `htd::IGraphDecompositionAlgorithm`

  This type of algorithm provides the base interface for all decomposition algorithms in the context of *htd*. Algorithms implementing this interface partition the input graph and return a labeled multi-hypergraph of type `htd::IGraphDecomposition`.

  Note that there is generally no guarantee or need that the vertices in the decomposition, representing the partitions of the input graph, form a single connected component. Therefore one can implement any decomposition algorithm one can think of using this basic interface class.

- `htd::ITreeDecompositionAlgorithm`

  The purpose of algorithms implementing this specialized interface is to optimally serve the needs of dynamic programming algorithms which rely on tree decompositions. In contrast to the basic decomposition algorithm type `htd::IGraphDecompositionAlgorithm`, decomposition algorithms of type `htd::ITreeDecompositionAlgorithm` are somewhat more involved. This is caused by the fact that they return labeled trees of type `htd::ITreeDecomposition`, thus requiring that the output graph to consist of a single connected, but cycle-free, component.

  Extends:

  `htd::IGraphDecompositionAlgorithm`.

- `htd::IPathDecompositionAlgorithm`

  When we request that the resulting decomposition is cycle-free and that it must not contain vertices with more than two neighbors, one is perfectly served using decomposition algorithms of type `htd::IPathDecompositionAlgorithm` as they return labeled paths of type `htd::IPathDecomposition`.

  Extends:

  `htd::ITreeDecompositionAlgorithm`.

- `htd::IHypertreeDecompositionAlgorithm`

  In cases where the output of the decomposition algorithm shall be a labeled tree of type `htd::IHypertreeDecomposition`, one can use the designated algorithms of type `htd::IHypertreeDecompositionAlgorithm`.

  Extends:

  `htd::ITreeDecompositionAlgorithm`

Note that each of the interfaces mentioned above can also be called with additional parameters dedicated to desired manipulations. In cases where these additional parameters are provided, the decomposition algorithm is required to return a decomposition which fulfills all criteria requested via the provided manipulation operations. This feature often dramatically reduces implementation effort and it significantly improves readability and maintainability of the code to be written by the developer. More details about manipulation operations follow subsequently and examples how to use (custom) manipulation operations can be found in Section 6.

4.5 Decomposition Manipulation Algorithms

During the design stage of *htd*, one of the main goals was the ability to customize the resulting decomposition without requiring tedious post-processing work at developer side. For this reason, *htd* provides various, built-in manipulation operations and also allows to extend their functionality easily by implementing the corresponding interface classes which are of type `htd::IDecompositionManipulationOperation`.

For each of the decomposition types which *htd* distinguishes there exists a corresponding, specialized interface for tailored manipulation operations. The distinction into these specialized interface classes allows to take advantage of shortcuts potentially originating from the features of the graph types on which they can be applied, e.g. there are no join nodes in path decompositions, hence we do not have to handle them.

The manipulation operations can be applied directly by the decomposition algorithms or also in a post-processing step. In the latter case it is required to cast the read-only decomposition returned by the decomposition algorithm to the corresponding write-capable one as the manipulation clearly involves updating information of the decomposition. Supporting the development process and maintainability of the code, *htd* currently provides the following built-in manipulation operations:

- `htd::AddEmptyLeavesOperation`

  It often reduces the effort needed to implement dynamic programming algorithms when it is guaranteed that leaf nodes always have an empty bag. This is primarily because one does not need to treat leaves as a special case.

  To ensure that a tree or path decomposition has only leaves with empty bags, *htd* provides the manipulation operation htd::AddEmptyLeavesOperation which simply adds a new child with empty bag to each leaf node which does not already have an empty bag.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::AddEmptyRootOperation`

  This operation ensures that the root node of a tree or path decomposition has an empty bag. Similar to the operation `htd::AddEmptyLeavesOperation`, the manipulation operation `htd::AddEmptyRootOperation` often helps to reduce the amount of special cases which one has to think about during development of dynamic programming algorithms.

  This becomes apparent when we look at the fact that without empty root one still has to check correctness of partial solutions by taking into account that the vertices in the root still have to be forgotten. With empty root, this final check can often be done exactly by the same procedure as for all other forget nodes.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::AddIdenticalJoinNodeParentOperation`

  Sometimes it is needed to do some complex post-processing of join nodes which cannot be done directly in the dynamic programming step belonging to the respective join node, like it was needed in [Abseher *et al.*, 2016a]. For this purpose, *htd* offers the class `htd::AddIdenticalJoinNodeParentOperation` which ensures that each join node has a parent node with identical bag content. With this guarantee, one can easily implement the desired post-processing functionality without having to handle various special cases.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::CompressionOperation`

  All built-in implementations of decomposition algorithm in *htd* guarantee that the resulting decomposition is minimal in the sense that all bags in the decomposition are subset-maximal, hence no subsumed bags are contained. Clearly, this guarantee only holds as long as one does not apply manipulation operations which add nodes with bags subsumed by other nodes' bags or which manipulate existing bag contents.

  Especially for undoing manipulations applied beforehand or in order to compress tree and path decompositions computed by custom decomposition algorithms, *htd* provides the class `htd::CompressionOperation` which efficiently removes all vertices from the given decomposition which are not subset-maximal.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::ExchangeNodeReplacementOperation`

  Exchange nodes are inner nodes of decompositions where some vertices are forgotten and, at the same time, some vertices are introduced. Sometimes one wants to avoid these situations and wants to deal with introduce and forget nodes separately in the dynamic programming algorithm. For this purpose, one can employ the manipulation operation `htd::ExchangeNodeReplacementOperation` which replaces each exchange node with one forget node and one introduce node. In order to keep the width of the decomposition unchanged, the forget node will be the child node of the introduce node.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::InducedSubgraphLabelingOperation`

  Although the bag content of a vertex is very important, dynamic programming algorithms also need the information about the subgraph of the original graph which is induced by the vertices contained in a bag as this information is the actual part of input data on which the algorithm operates. Especially for large graphs finding the (hyper)edges which are induced by the bag content can be extremely expensive.

  As we will observe later (see Section 4.7), all built-in decomposition algorithms of *htd* already compute this information very efficiently and store the outcome directly in the resulting decomposition from which it can be accessed easily. Nevertheless, the class `htd::InducedSubgraphLabelingOperation` allows to automatically add an additional label to each decomposition node containing the set of induced edges. This is especially useful to efficiently compute the set of induced edges when a custom decomposition algorithm does not expose this information directly.

Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::IntroducedSubgraphLabelingOperation`

  An (hyper)edge of the input graph is relevant for many dynamic programming algorithm especially at the time when it is introduced, i.e., when all its endpoints occur together in a bag for the first time in a branch of a given tree decomposition.

  To assign a vertex label containing information about the set of introduced (hyper)edges to each node of the tree decomposition, one can use the manipulation operation `htd::IntroducedSubgraphLabelingOperation`.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::JoinNodeNormalizationOperation`

  Join nodes can be very complex to handle in dynamic programming algorithms when the children's bags differ from the respective parent node's bag. Especially in the early stages of the development of dynamic programming algorithms but also later on it can be very useful to have the guarantee that join nodes and their children share the same bag content. For this purpose, *htd* offers the class `htd::JoinNodeNormalizationOperation`.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::JoinNodeReplacementOperation`

  It is assumed that path decompositions are to be preferred over tree decompositions with join nodes when the width is equal, but also when the width is only slightly worse it can pay off to forgo join nodes. For this purpose, *htd* offers `htd::JoinNodeReplacementOperation`. This manipulation operation takes a tree decomposition as input and replaces join nodes by re-arranging and combining their children in an intelligent manner, so that the increase of the width is reduced.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::LimitChildCountOperation`

  Due to the fact that joins, in many cases, can be carried out more efficiently when the number of children per join node is bounded, *htd* offers the manipulation operation `htd::LimitChildCountOperation` which limits the number of children per node to a pre-definable upper bound. This is achieved by adding intermediate nodes with bag content identical to the join node under focus and distributing the supernumerous children properly among these additional intermediate nodes.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::LimitMaximumForgottenVertexCountOperation`

  Depending on the actual implementation of the dynamic programming algorithm which operates on the basis of the computed decomposition, it sometimes can be beneficial to have a known upper bound for forgotten vertices in order to carry out some steps of the algorithm more efficiently. Also for debugging purposes it is often very useful to have only small changes between neighboring nodes of the decomposition and hence it would be great to have a built-in manipulation to actually achieve this goal efficiently. The class `htd::LimitMaximumForgottenVertexCountOperation` serves exactly this purpose.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::LimitMaximumIntroducedVertexCountOperation`

  Analogous to the aforementioned manipulation operation which allows to limit the maximum number of forgotten vertices for each decomposition node, the built-in class `htd::LimitMaximumIntroducedVertexCountOperation` implements functionality to limit the maximum number of introduced vertices for each node of the decomposition.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::TreeDecompositionOptimizationOperation`

  This is probably the most involved manipulation operation currently implemented in *htd*. It allows to automatically change the root of a given tree so that the outcome of a provided

fitness function is maximized. Using this operation, one can easily get a customized decomposition in a few lines of code without having to touch other portions of the code.

Although the operation at hand is a rather complex and powerful one, it is geared towards performance and it is fully compatible with other manipulations, that is, the algorithms behind `htd::TreeDecompositionOptimizationOperation` will optimize the tree decomposition considering all desired manipulation operations.

For even better controllability of the optimization operation one can use different vertex selection strategies to filter the vertices which shall be considered as new root node. This allows to improve performance especially for large decompositions as there is no need to exhaustively check for all vertices in the tree decomposition if they are the optimal root node. Clearly, the built-in collection of vertex selection strategies can be easily extended by developers by implementing the corresponding, simple interface.

A detailed explanation of this operation is given in Section 6.

Implements the following interfaces:

- `htd::ITreeDecompositionManipulationOperation`

- `htd::ILabelingFunction`

  This interface is used by decomposition manipulation operations which generate labels for a corresponding bag. Labeling functions take an input graph of type htd::IMultiHypergraph and a sorted set of vertices, representing the bag content of a decomposition node, and they return a new label of type htd::ILabel. The actual data type of the value of the returned label is dependent on the implementation of the respective labeling function class. Generally, one can use any data type supported in C++.

  For convenience, one can access the concrete label value via the template function htd::accessLabel which takes a template argument representing the data type of the label value as well as a reference to the label and it returns the concrete label value in the given data type.

  Note that when a labeling function is provided to a decomposition algorithm, the labeling function will be applied automatically to each vertex of the resulting decomposition. This makes using custom labels on the one hand very easy and on the other hand it ensures highest efficiency and maintainability of the code.

  Implements the following interfaces:

  - `htd::IDecompositionManipulationOperation`

Also here, the list can be extended easily with own algorithms by implementing the desired interface(s) mentioned above. One only has to take care that the manipulation operation does not violate the properties of the decomposition on which it is applied, i.e., the decomposition must stay valid after the manipulation is applied, otherwise it will break the functionality of the algorithms which use the modified decomposition.

4.6 Decomposition Normalization Algorithms

While the manipulation operations described in Section 4.5 are dedicated to exactly one task per operation class, one sometimes requires more complex manipulations. One example is for instance making a given tree decomposition nice. This would involve the combination of many of the aforementioned basic manipulation operations. We refer to such complex manipulations which are generated by combining simpler manipulation operations by the term "normalizations". For convenience, *htd* offers the following built-in normalizations:

- `htd::WeakNormalizationOperation`

  When each join node of a tree decomposition only has children with a bag content identical to the bag content of the respective join node under focus, we call such decompositions weakly normalized.

  When we want to obtain a decomposition which fulfills this criterion, one can use the manipulation operation of type `htd::WeakNormalizationOperation`. Additionally, one can specify that the root node and/or all leaf nodes of the decomposition shall be empty. The manipulation operation at hand efficiently combines the manipulation operations `htd::JoinNodeNormalizationOperation`, `htd::AddEmptyRootOperation` and `htd::AddEmptyLeavesOperation`, the last two of them being optional.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::SemiNormalizationOperation`

  Semi-normalized tree decompositions in the context of *htd* are tree decomposition where each of its join nodes has exactly two children with the same bag content as the respective join node. Hence, the manipulation operation `htd::SemiNormalizationOperation` extends the class `htd::WeakNormalizationOperation` by combining it with the manipulation operation `htd::LimitChildCountOperation` with a child limit of 2.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::NormalizationOperation`

  Sometimes one wants to work in the dynamic programming algorithm with a fully normalized (nice) decomposition, that is a tree (or path) decomposition where join nodes and their children have the same bag content and the bag contents between adjacent non-join nodes differ in at most one element.

Specifically for this purpose, *htd* offers the class `htd::NormalizationOperation` which extends the class `htd::SemiNormalizationOperation` by combining it with the manipulation operations `htd::ExchangeNodeReplacementOperation`, `htd::LimitMaximumForgottenVertexCountOperation` (with a vertex limit of 1) as well as `htd::LimitMaximumIntroducedVertexCountOperation` (with a vertex limit of 1).

Implements the following interfaces:

- `htd::ITreeDecompositionManipulationOperation`
- `htd::IPathDecompositionManipulationOperation`

4.7 Useful Additional Functionality

An efficient way to compute tree decompositions is a very important ingredient of an application based on dynamic programming. While this is a widely accepted statement, a tree decomposition alone often is rather worthless as long as there is no way to efficiently retrieve the information stored in it. For this reason, *htd* offers a wide range of utility functions. A small selection of them is given in the following list:

- Easy Retrieval of Induced Subgraph Information

  Maybe one of the most unique features of *htd* in contrast to other tree decomposition frameworks is the fact that all implementations of graph decomposition algorithms which are implemented in *htd* automatically compute for each bag of the decomposition the subgraph of the input graph which is induced by the respective bag content. By using the function called "inducedHyperedges" one can easily and with almost no cost obtain the hyperedges which are induced by the bag with the given ID. This can lead to a significant boost in terms of performance as input graphs can have millions of edges and so doing a subset check for each of them in each bag can be very expensive.

- Tree and Graph Traversal Algorithms

  Due to the fact that they are in many cases easier to develop, describe and to implement, graph traversal algorithms often use recursion. A big issue with recursion is the fact that as soon as the graphs reach a certain size, the proper operation of practical implementations is no longer guaranteed as the stack is no longer capable of holding the information needed by the recursive function calls. Therefore and also because we think, a developer shall not have to re-invent the wheel, we provide the following interfaces and algorithms in *htd*:

  Built-In Implementations of `htd::IGraphTraversal`:

  The following two algorithms, the first one implementing breadth-first and the second one depth-first traversal, traverse a graph beginning from a custom starting vertex and take a lambda expression which is called for each visited vertex with the information about the vertex at hand, its predecessor during the traversal and its distance from the starting vertex.

In this way, one can easily determine even complex characteristic numbers (like the diameter) of the graph which is traversed.

  - `htd::BreadthFirstGraphTraversal`
  - `htd::DepthFirstGraphTraversal`

  Built-In Implementations of `htd::ITreeTraversal`:

  The interface `htd::ITreeTraversal` extends `htd::IGraphTraversal` and is dedicated to trees. Here, the predecessor of a vertex is always identical to its parent. Hence, there is no need to spend time for looking up the parent of the vertex currently visited as it is provided for free to the lambda expression. Note at this point that the parent of the root node in the context of *htd* is the "undefined vertex" referenced by the ID 0. The following three tree traversal algorithms are currently implemented in *htd*:

  - `htd::InOrderTreeTraversal`
  - `htd::PreOrderTreeTraversal`
  - `htd::PostOrderTreeTraversal`

- Connected Component Algorithms

  Sometimes it can be beneficial to pre-process a given input graph before decomposing it. Depending on the actual application scenario, one possibility of pre-processing a graph is by investigation of its (strongly) connected components. The following interfaces and their implementations help to implement such a pre-processing step effectively and efficiently:

  Built-In Implementations of `htd::IConnectedComponentAlgorithm`:

  - `htd::DepthFirstConnectedComponentAlgorithm`
    This algorithm for determining connected components is based on depth-first search and internally uses the class htd::DepthFirstGraphTraversal. Again, the use of recursion is avoided which allows to handle graphs of (almost) arbitrary size.

  Built-In Implementations of `htd::IStronglyConnectedComponentAlgorithm`:

  - `htd::TarjanStronglyConnectedComponentAlgorithm`
    This class implements Tarjan's algorithm for determining the strongly connected components of direct graphs which is described in [Tarjan, 1972]. The implementation of the algorithm is fully iterative, allowing to handle graphs of (almost) arbitrary size.

4.8 Implementation Guidelines

The following section is a short guide to the most important design rules to which the interfaces and algorithms in *htd* conform. Especially when developing own algorithms one should read these coding guidelines carefully and follow them in order to avoid breaking functionality.

Class Names Due to the fact that C++ does not use a special keyword for discriminating interfaces from (abstract) classes we define that each class name starting with the the capital letter "I" followed by another capital letter is considered an interface, i.e., a class with pure virtual functions only.

Function Arguments and Return Types Whenever a function receives a reference which is not modified this reference is marked as "const". Conversely, when a reference is not marked as "const" one can assume that the object will be modified inside the function and one should be careful in cases where it is desired to keep an unmodified variant of the input object. Similarly, when a function returns a reference, one is free to modify the underlying object.

One crucial point is the memory management in cases where function receive and/or return pointers to objects. In cases where the function argument is a non-const pointer, the function is required to take over control over the memory region. That is, the function must take care that the memory region is properly freed. This also applies to functions which take collections of non-const pointers. When a function returns a non-const pointer, one must free the resources after using them. Note that functions receiving a const-pointer will not free the pointer, so one must take care of freeing the resources.

WARNING: You must not free the memory of const-pointers returned by functions, you must not provide the same non-const pointer multiple times as a function argument and you must not free objects which were given to a function via a non-const pointer outside the respective function boundaries as this will probably lead to memory corruptions.

To be on the safe side, don't access or modify to object to which the non-const pointer points after it was used as a function argument. In order to re-use it, one can easily create a deep copy of almost all classes in *htd* via their appropriate clone() method.

Type Conversions and Casts Note that its a safe operation to up-cast from a read-only graph or decomposition type to the corresponding write-capable one. That is, one can use the dynamic_cast function provided in the C++ language specification to convert, for instance, a pointer or reference to `htd::IGraph` to a pointer or reference to `htd::IMutableGraph`. This convertibility must also be guaranteed by custom implementations in order to avoid breaking the functionality of algorithms.

WARNING: This conversion is only a safe and permitted operation for directly corresponding interfaces, i.e., although each `htd::ITreeDecomposition` is a `htd::IMultiHypergraph` one cannot cast a `htd::ITreeDecomposition` to a `htd::IMutableMultiHypergraph`. This is clearly neither possible nor permitted as hypergraphs allow cycles and trees do not.

5 Algorithm Engineering

During the development of *htd*, a lot of time was spent on implementing, extending and optimizing the algorithms which contribute to the library. The extension of (existing) algorithms was needed

to be able to incorporate all customization capabilities of *htd* and the optimization is needed to be able to efficiently deal with large input graphs.

Subsequently, we will discuss in detail our approach how to accelerate the most-crucial parts for computing tree decompositions in the context of *htd*, namely the algorithm for computing the Min-Fill vertex elimination ordering and the algorithm for computing the actual tree decomposition via Bucket Elimination [Dechter, 1999; McMahan, 2004; Schafhauser, 2006]. We chose these two algorithms for presentation here as they represent well-established techniques and we want to share our findings in order to speed up development of implementations of Min-Fill and Bucket Elimination in future projects.

5.1 Accelerating Min-Fill

Min-Fill is a prominent heuristic for computing vertex elimination orderings which often produces good results, i.e., tree decompositions of low width, in practice [Koster *et al.*, 2001]. Like any greedy triangulation algorithm, Min-Fill follows the schema that, given an input graph $G = (V, E)$, in each iteration a vertex $v \in V$ is chosen based on a given criterion and then a clique incorporating all of v's neighbors is formed in G by adding the so-called *fill edges*. The vertex v is then removed from G and stored in the next position of the resulting ordering. These simple steps are repeated until G finally is empty.

In the case of Min-Fill, the criterion for selecting the vertex to be removed is the following: In each iteration, we select and eliminate the vertex which requires the least amount of fill edges to be added in order to form a clique of all its neighbors. As, in general, there are multiple vertices with the same amount of required fill edges, ties are broken randomly.

Figure 2 shows an example input graph for the Min-Fill graph triangulation algorithm. The vertex labels denote the number fill value of the corresponding vertex. For instance, Vertex a has a fill value of 2 as we need two additional edges, namely (b, d) and (b, e), in order to create a clique containing all of a's neighbors which are given by the vertices named b, d and e.

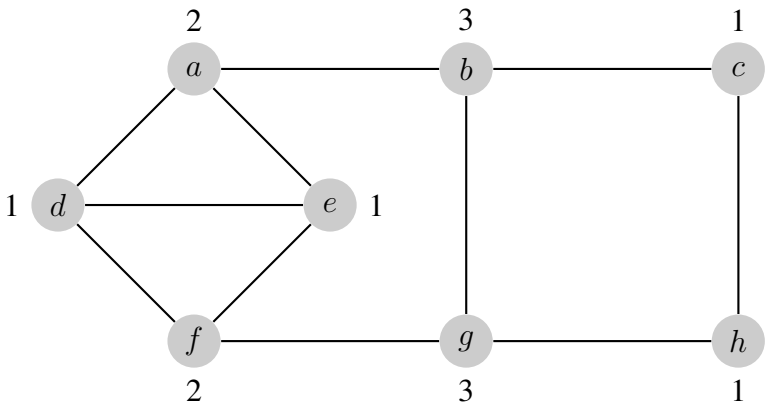


Figure 2 Example Input Graph for Min-Fill

The pseudo-code of the Min-Fill heuristic is shown in Algorithm 1. Note that, although our definition of triangulation algorithms given before refers to graphs, the pseudo-code in Algorithm 1 takes an arbitrary (constraint) hypergraph as input. This is a valid move because we can easily transform any (constraint) hypergraph to its corresponding primal graph by introducing an edge between each pair of vertices in a hyperedge.

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
Result: A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V

Let $\mathcal{G} = (V, E)$ be the primal graph of $\mathcal{H}$.
$\sigma \leftarrow []$; `// List σ is initially empty.`
while $\mathcal{G}$ *is not empty* **do**
  `/* Ties are broken randomly! */`
  Select a vertex $v_x \in V$ whose elimination requires the least amount of edges to be added;
  `/* Add necessary fill-in edges. */`
  $neighbors \leftarrow (\{v \mid (v_x, v) \in E\} \cup \{v \mid (v, v_x) \in E\}) \setminus \{v_x\}$;
  for $v_1 \in neighbors$ **do**
    for $v_2 \in neighbors$ **do**
      if $(v_1, v_2) \notin E$ **then**
        $E \leftarrow E \cup \{(v_1, v_2)\}$;
      end
    end
  end
  `/* Remove` v_x `from` $\mathcal{G}$`. */`
  $V \leftarrow V \setminus \{v_x\}$;
  $E \leftarrow E \setminus \{(v_x, v) \mid (v_x, v) \in E\}$;
  $E \leftarrow E \setminus \{(v, v_x) \mid (v, v_x) \in E\}$;
  Append v_x to σ;
end
return σ;

Algorithm 1: Min-Fill (Simple Pseudo-Code)

Basically, the pseudo-code does not tell us how to efficiently determine the fill value of a vertex. By the term *fill value* we refer to the amount of fill edges that need to be added in order to form a clique incorporating all neighbors of the respective vertex. Subsequently we will give detailed insights on how to accelerate the computation of a Min-Fill vertex elimination ordering.

First, let us define the data structures which the algorithm will rely on and fix the notation we will use throughout the following explanation: As underlying data structure for storing the graph we use a simple adjacency list. Additionally we maintain a dictionary which holds the current fill

value for each vertex which is not yet removed and we maintain set of vertices which we refer to as the *pool* and which contains all vertices with minimum fill value.

We define that, given a graph $G = (V, E)$, the (one-hop) neighborhood $N_1(v)$ of a vertex $v \in V$ is represented by the set $\{v_x|(v_x, v) \in E\} \cup \{v_x|(v, v_x) \in E\} \cup \{v\}$ and the two-hop neighborhood $N_2(v)$ of a vertex $v \in V$ is given by the set $N_1(v) \cup (\bigcup_{v_x \in N_1(v)}(v_y|(v_y, v_x) \in E\} \cup \{v_y|(v_x, v_y) \in E\}))$. That is, $N_1(v)$ contains all vertices which are reachable from v in at most one hop and $N_2(v)$ contains all vertices which are reachable from v within not more than two hops. Based on these definitions, we define the following notions with respect to a removed vertex $v_r \in V$:

Definition 3. *For a vertex $v \in N_2(v_r)$, we call the vertices in $N_e(v) = (N_1(v) \cap N_1(v_r)) \setminus \{v_r\}$ the* existing neighbors *of v. Furthermore, we call the vertices $N_u(v) = N_1(v) \setminus N_1(v_r)$ the* unaffected neighbors *of v and we refer to $N_a(v) = (N_1(v_r) \setminus N_1(v)) \setminus \{v_r\}$ as the* additional neighbors *of v. Indeed, $N_e(v) \cup N_u(v) \cup N_a(v) \equiv N(v) \setminus \{v_r\}$ holds and the sets are disjoint, that is, existing, unaffected and additional neighbors of v form a partition of v's neighborhood with v_r removed.*

The intuition behind the terms *existing, unaffected* and *additional vertex* is the following: The existing vertices of a vertex v are those vertices which are directly manipulated, i.e., the vertices whose neighborhood will be updated. The additional vertices are those vertices which are to be added to the neighborhood of v in order to create a clique between the neighbors of v_r and the unaffected vertices are those neighbors of v which are not adjacent to v_r. Note that $N_a(v) = \emptyset$ holds for all vertices $v \in N_2(v_r) \setminus N_1(v_r)$ where v_r is the vertex eliminated in the current iteration. This is because of the fact that only vertices which are direct neighbors of the eliminated vertex v_r will potentially get additional neighbors due to the creation of the clique.

As mentioned before, in order to perform the selection step in Line 4 of Algorithm 1 efficiently, we decide to keep track of the fill value of each vertex by storing it in a dictionary data structure which allows for fast lookup. That is, instead of computing the fill value of a vertex from scratch in each iteration, we just update the fill value accordingly. In this way, we can easily manage a pool of vertices which currently have the lowest fill values without having to re-compute the same information over and over for vertices which were not affected in a previous iteration. From this pool of vertices with minimum fill value we can then simply select a vertex at random in order to perform the next iteration.

To illustrate how to efficiently compute the necessary fill value changes, we refine the pseudo-code provided in Algorithm 1. The enhanced version of the pseudo-code is given in Algorithm 2. While the procedures of eliminating a vertex and adding it to the ordering stay unchanged, the extended algorithm now illustrates how one can easily update the fill value of those vertices (and only of those vertices) which are affected by a vertex elimination step.

In the first iteration of the algorithm, we need to compute the actual fill value for each vertex. With this information we can initialize the variables $fill$, $minfill$ and $pool$ (see Line 6 of Algorithm 2). After doing so we never need to compute the full fill value of a vertex again. We completely rely on updating the fill values as this significantly improves performance in practice. This is based on the fact that for updating the fill value it often suffices to consider at most two out of the three partitions – $N_e(v)$, $N_u(v)$ and $N_a(v)$ – of the neighborhood relation.

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
Result: A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V

1 Let $\mathcal{G} = (V, E)$ be the primal graph of $\mathcal{H}$.

2 $\sigma \leftarrow []$; `// List` σ `is initially empty.`
3 *fill* $\leftarrow []$; `// The dictionary for the fill value of each vertex.`
4 *pool* $\leftarrow \emptyset$; `// The set of vertices with minimum fill value.`
5 *minfill* $\leftarrow \infty$; `// The minimum fill value.`

6 Initialize *fill*, *minfill* and *pool* based on $\mathcal{G}$.

7 **while** $\mathcal{G}$ *is not empty* **do**
8 **if** *pool* $== \emptyset$ **then**
9 Initialize *minfill* and *pool* based on $fill$.
10 **end**
11 Select randomly a vertex $v_x \in$ *pool*;
12 **if** *fill*$[v_x] == 0$ **then**
13 **for** $v \in N_1(v_x) \setminus \{v_x\}$ *where* *fill*$[v] > 0$ **do**
14 *fill*$[v] \leftarrow$ *fill*$[v] - |N_1(v) \setminus N_1(v_x)|$;
15 **end**
16 **else**
17 **for** $v \in N_1(v_x) \setminus \{v_x\}$ **do**
18 $fill[v] \leftarrow$ updateNeighbor($\mathcal{G}, v_x, v, fill[v]$);
19 **end**
20 **for** $v \in N_2(v_x) \setminus (\{v_x\} \cup \{u | u \in N_1(v_x), N_u(u) == \emptyset \,||\, N_a(u) == \emptyset\})$ **do**
21 **for** $u \in N_e(v)$ **do**
22 *fill*$[v] \leftarrow$ *fill*$[v] - |\{t | t \in N_e(v), t > u\} \cap N_a(u)|$;
23 **end**
24 **end**
25 **end**
26 Add necessary fill-in edges;
27 Remove v_x from $\mathcal{G}$;
28 Append v_x to σ;
29 **end**
30 **return** σ;

Algorithm 2: Min-Fill (Verbose Pseudo-Code)

In Lines 8–10 of Algorithm 2 we update the pool if it is empty. This can occur whenever the last vertex with fill value equal to *minfill* was eliminated in the iteration before or when the all vertices which were in the pool in the iteration before got updated to a fill value which is higher than *minfill*. Note that we omit in Algorithm 2 the code for updating the pool content and the value of *minfill* due to space reasons. Basically, whenever the fill value of a vertex is updated, also the pool and, potentially, also the value *minfill* need to be updated.

In Lines 12–15 of Algorithm 2 the case is handled in which the eliminated vertex v_x has a fill value of 0, that is, all its neighbors already form a clique. In this case, we can simply subtract the amount of unaffected neighbors $N_u(v)$ from the fill value of each neighbor v of v_x. This update is sufficient because v does not have any additional neighbors (because the neighbors of v_x already form a clique) and the existing neighbors of v do not contribute to its fill value (for the same reason). Hence, a change of the fill value of v can be only be caused by the fact that now all missing edges between v_x and v's neighbors are no longer relevant after the elimination of v_x.

More interesting is the case in which the fill value of the eliminated vertex is greater than 0. In those cases we first enter the loop at Lines 17–19 of Algorithm 2 for each neighbor of the eliminated vertex v_x. To update the fill value of these vertices, we use Algorithm 3. This helper algorithm takes as input an undirected graph, the eliminated vertex v_x, the vertex v whose fill value shall be updated and the old fill value of v. The result of the algorithm is the new fill value of v after eliminating v_x.

Input: An undirected graph $\mathcal{G} = (V, E)$
A vertex $v_r \in V$ which is eliminated in the current iteration
A vertex $v \in V$ whose fill value shall be updated
The old fill value f of v
Result: The new fill value of v

```
1  if |N_u(v)| > 0 then
2  |  if |N_a(v)| > 0 then
3  |  |  f ← f + (|N_a(v)| − 1) ∗ |N_u(v)| − Σ_{u∈N_a(v)} |N_u(u) ∩ N_u(v)|;
4  |  else
5  |  |  f ← f − |N_u(v)|;
6  |  |  for u ∈ N_e(v) do
7  |  |  |  f ← f − |{t|t ∈ N_e(v), t > u} ∩ N_a(u)|
8  |  |  end
9  |  end
10 else
11 |  f ← 0;
12 end
13 return f;
```

Algorithm 3: Procedure *updateNeighbor* for Algorithm 2

In the case where there are no unaffected neighbors of v, Algorithm 3 will simply return 0 as the new fill value because after the elimination of v_x all of v's neighbors will be together in a clique. When there exists at least one unaffected neighbor of v, we distinguish two cases: When there exist, additional neighbors of v, we update the fill value of v according to the formula shown in Line 3 of Algorithm 3. That is, we first increase its fill value by the maximum amount of edges that can exist between the additional and the unaffected vertices and then we subtract the amount of existing edges between the additional and unaffected vertices.

In the second case, that is, when v has some unaffected but no additional vertices, we first subtract the amount of unaffected neighbors from v's fill value because v_x is a neighbor of v and the missing edges between v_x and the unaffected neighbors no longer matter. Then we iterate over all $u \in N_e(v)$ and subtract the size of the set intersection of all additional vertices of u and of all existing neighbors of v greater than u. By taking only vertices greater than u into consideration, we do not count any edge twice. In this way, we can efficiently compute the amount of edges between the existing neighbors of v which were missing but which are to be added due to the construction of the clique between the neigbors of v_x.

This brings us back to Lines 20–23 of Algorithm 2. In this loop, we iterate over all two-hop neighbors of the eliminated vertex v_x excluding v_x and which are either not directly adjacent to v_x or which have both unaffected and additional neighbors. That is, we consider again those two-hop neighbors v_x which are also direct neighbors and which were handled by Line 3 of Algorithm 3. This helps to avoid redundant code and can easily be achieved by maintaining a dictionary holding a status value of the neighbors of v_x. Line 22 of Algorithm 2 follows the same idea as Line 7 of Algorithm 3. All what remains in an iteration of the Min-Fill algorithm is to add the necessary fill-in edges, remove v_x from the graph and append it to the ordering.

Note that one can achieve additional improvements by keeping track of the total fill value – the sum of all fill values – and abort the main loop earlier when this counter reaches 0. This is possible because a total fill value of 0 implies that the remaining graph is a clique. The ordering can then be completed by appending the vertices in the remaining graph in arbitrary order.

5.2 Extending Bucket-Elimination

Bucket elimination [Dechter, 1999] was originally presented as a unifying framework for reasoning. Based on this concept, we can easily compute a tree decomposition of a given input hypergraph as shown in Algorithm 4. The pseudo-code in Algorithm 4 was presented in [McMahan, 2004; Schafhauser, 2006] and it illustrates the process of computing a tree decomposition when given a input hypergraph and a corresponding vertex elimination ordering. The algorithm first creates an empty bucket for each vertex of the input graph (Line 2) and then it proceeds by initializing the bags based on the (hyper-)edges which are contained in the graph (Lines 3–6). This happens by assigning the vertices of a given hyperedge h to the bucket which corresponds to the lowest ranked vertex in h with respect to the given elimination ordering σ. Afterwards we have to iterate over the vertex elimination ordering σ in order to obtain the complete content of each bucket and the edges between the buckets by which the final tree is constructed (Lines 7–12).

Subsequently, we propose an extension of Algorithm 4 which can be extremely helpful for the development of dynamic programming algorithms. For dynamic programming on tree decompositions we need always need the information about the (hyper-)edges which are induced by a bag. A naive approach where we check in each bag of the tree decomposition the induced edges soon becomes a bottleneck when the input graph contains thousands of edges. To overcome this issue, we use the following trick in *htd*, so that all the work is done directly by the Bucket Elimination procedure:

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V
Result: A tree decomposition $\langle T, \chi \rangle$ of $\mathcal{H}$

1 $E = \emptyset$;
2 Create an empty bucket B_{σ_i} for each vertex $\sigma_i \in \sigma$: $\chi(B_{\sigma_i}) \leftarrow \emptyset$.
3 **for** $h \in H$ **do**
4 Let v be the vertex in h which is ranked at lowest position in σ among all vertices in h;
5 $\chi(B_v) \leftarrow \chi(B_v) \cup vertices(h)$;
6 **end**
7 **for** $i \in \{1..n\}$ **do**
/* Create temporary vertex set A based on bucket B_{v_i}. */
8 Let $A = \chi(B_{v_i}) \setminus \{v_i\}$;
/* Determine lowest ranked vertex in A based on σ. */
9 Let $v_j \in A$ be the next in A following v_i in σ;
10 $\chi(B_{v_j}) = \chi(B_{v_j}) \cup A$;
11 $E = E \cup \{(B_{v_i}, B_{v_j})\}$;
12 **end**
13 **return** $\langle (B, E), \chi \rangle$ where $B = \{B_1 \dots B_n\}$;

Algorithm 4: Bucket Elimination

In fact, at Line 5 of Algorithm 4 we can store the target bucket of a (hyper-)edge, that is, we can use a dictionary which for each (hyper-)edge holds the bucket in which it is fully contained. The speedup comes from the fact that while finding the first bucket in a given tree decomposition in which an edge is contained can be tedious, internally in the context of the Bucket Elimination algorithm we get this valuable information for free.

After implementing this small change, all that remains in order to find the induced edges for each bag is to start for each edge in the input graph a kind of depth limited search in the tree decomposition starting from the target bag of the edge. The limit in this case is given by the bucket where the edge is no longer fully contained. That is, we follow a branch until we reach a bag which does no contain the edge as a whole any more and then we backtrack, instead of backtracking as soon as a certain depth is reached.

6 *htd* at Work

In this section we will have an in-depth look at how to use *htd* optimally in different situations and how to adapt it according to the actual needs. The focus of this explanation is not only to introduce the developer to the concrete classes and interfaces which can be useful for implementing dynamic programming algorithms but also to shed some light at the interplay between them. This section will first cover details on loading input data and then we will have a detailed look on how to

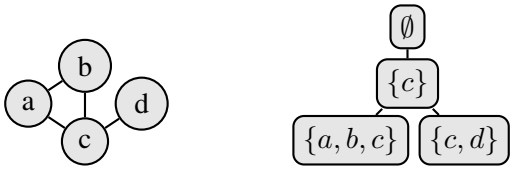
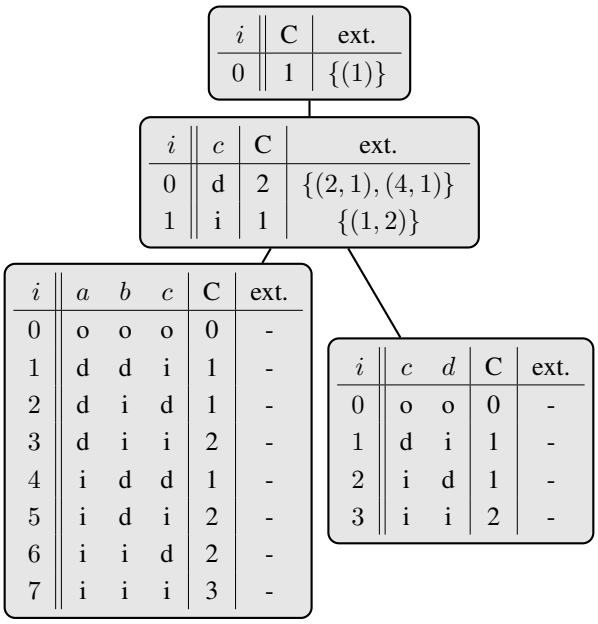


Figure 3 Example graph and a possible tree decomposition.



i	C	ext.
0	1	$\{(1)\}$

i	c	C	ext.
0	d	2	$\{(2,1),(4,1)\}$
1	i	1	$\{(1,2)\}$

i	a	b	c	C	ext.
0	o	o	o	0	-
1	d	d	i	1	-
2	d	i	d	1	-
3	d	i	i	2	-
4	i	d	d	1	-
5	i	d	i	2	-
6	i	i	d	2	-
7	i	i	i	3	-

i	c	d	C	ext.
0	o	o	0	-
1	d	i	1	-
2	i	d	1	-
3	i	i	2	-

Figure 4 Solving MINIMUM DOMINATING SET via DP for the problem instance in Figure 3.

customize the way decompositions. Finally we will make use of *htd*'s utility functions to show how the information stored in the resulting decomposition can be efficiently incorporated in the dynamic programming algorithm.

6.1 The Dynamic Programming Algorithm

For the remainder of this section, let's assume that we want to solve the MINIMUM DOMINATING SET problem via dynamic programming on tree decompositions. That is, we try to find cardinality-minimal subsets $S \subseteq V$ of a graph $G = (V, E)$ such that each vertex $v \in V$ is either contained in S or adjacent to at least one vertex in S. Furthermore, assume that we are interested in determining the complete set of all such solutions.

Figures 3 and 4 illustrate our example problem by mean of a simple input graph, a possible tree decomposition of the input graph and the corresponding dynamic programming tables. Note that the decomposition is of minimal width but not optimal, as we could simply omit the join node and the empty root node and directly connect the two leaf nodes of the given decomposition and obtain a smaller one without join nodes. Nevertheless, for explanatory purposes we stick with the

tree decomposition shown at the right-hand side of Figure 3.

In Figure 4 we can then see the dynamic programming tables for each node of the given tree decomposition. Each row within a table is identified by its index i. Furthermore, each table row stores the value assigned to each vertex contained in the corresponding bag, its total cost C and the extension pointer tuple which tells us from which child row(s) the current row originates. For the problem of MINIMUM DOMINATING SET the values which can be assigned to a vertex are i (the vertex is contained in S), d (the vertex is not contained in S but adjacent to at least one vertex in S) and o (the vertex is neither contained in S nor a neighbor of a vertex which is element of S).

To ensure correctness during the bottom-up traversal of the tree decomposition, a row may extend only compatible child rows. Rows are compatible if and only if all vertices which are part of the child bag but not of current bag have a value of either i or d. Furthermore, in join nodes, two child rows (from two different tables) are compatible if and only if the projection of the respective child row to the vertices in the current bag agrees on the vertices that carry the assignment i. Note that the value d dominates the value o, that is, if in a join node a table row of the child assigns the value d to a vertex and a compatible table row in another child assigns the value o to the same vertex, the assignment of the resulting table row is d. This is because it is sufficient for a vertex to be neighbor of a single vertex in S and, obviously, this vertex was already forgotten.

The cost of a row is computed by summing up the number of introduced vertices assigned the value i and the cost of the child row from which the row originates. In leaf nodes, the cost of a row is equal to the number of vertices with value i. In join nodes, we have to follow the inclusion-exclusion principle to avoid counting the same vertex multiple times. To do so, we simple sum up the costs of all child table rows which are extended by the current row and subtract the product of the number of children minus 1 and the number of vertices in the current bag which are assigned the value i. Clearly, as we are interested in the cheapest solution(s), in cases where table rows coincide, we only need to extend the cheapest one of them.

When we follow this schema, we obtain exactly the tables shown in Figure 4. After we have computed all the information about table rows we can construct a solution simply by following the extension pointers starting from the root node down to the leaves. In our example, we can see that the only optimal solution is to select $S = \{c\}$, as by selecting c we automatically dominate all other vertices of the input graph.

6.2 Loading the Input Data

In general, there are many ways to parse input data and even the origin of the data stream from which to read may vary between different scenarios. For the following example code we assume that the input is stored in files conforming to the following variant of DIMACS graph format which is also used as the official input format of Track A of the "First Parameterized Algorithms and Computational Experiments Challenge 2016" ("PACE16", see `https://pacechallenge.wordpress.com/track-a-treewidth/`):

- The file starts with a header matching the pattern "`p tw` *Vertices Edges*", where the placeholder *Vertices* represents the number of vertices of the input graph and *Edges* is a place-

holder for the number of edges which are contained in the input graph. The vertices are numbered between 1 and $Vertices$.

- All lines starting with the letter 'c' are treated as comments.
- Each remaining line (there must be exactly $Edges$ such lines) represents an edge. An edge information consists of exactly two vertex identifiers (numbers between 1 and $Vertices$) separated by a single space.

Subsequently we will sketch how one can simply and effectively create a new instance of `htd::IMultiGraph` which contains all relevant information of a given input graph. Note that the following C++ code snippets are not optimized and that we omit error handling here for brevity. An efficient implementation for parsing input files of the aforementioned format, also containing error handling routines, is provided via the class `htd::GrFormatImporter` which can be found in the folder "src/htd_main" located in the main folder of the *htd* project.

The project *htd_main*, which is providing a command-line executable for the *htd* project, also contains a collection of other useful classes allowing to import additional, often-used graph types. These classes can act also as a starting point for the development of custom importers.

To come back to our example, let's assume that there exists a pointer to an object of type `htd::LibraryInstance` called "manager" and assume furthermore that we already parsed the first non-comment line of an input file in the aforementioned format. At this point, we already know the total amount of vertices the new graph will contain and we can store it in the variable $Vertices$. With this information we can then create a new instance of the desired graph type using the code shown in Listing 4.

```
// Get a new instance of the default multi-graph implementation.
htd::IMutableMultiGraph * g =
    manager->multiGraphFactory().getMultiGraph(Vertices);
```

Listing 4 Creating a new instance of `htd::IMutableMultiGraph` (Variant 1).

Another way of achieving the same result, that is creating a new graph instance of non-zero size, is shown in Listing 5. While the code in Listing 4 automatically initializes the graph to the requested size, the second variant first creates an empty graph and afterwards adds the desired number of vertices to it via a bulk operation. In both cases, the created vertices are numbered between 1 and $Vertices$. When the method `addVertices` is called repeatedly, the range of the newly inserted vertices starts from the last vertex identifier + 1. Alternatively, if it is needed to add a single vertex to a graph, one can use the method `addVertex`. To add a vertex to a tree or path, *htd* offers the methods `insertRoot` (for creating the first vertex of the tree or path), `addChild` and `addParent`. Note that the creation and initialization of all other (labeled) graph types and decomposition types works analogously.

```
// Get a new, empty instance of the default multi-graph implementation.
htd::IMutableMultiGraph * g =
    manager->multiGraphFactory().getMultiGraph();
```

```
// Add all vertices to the graph
g->addVertices(Vertices)
```

Listing 5 Creating a new instance of `htd::IMutableMultiGraph` (Variant 2).

In our example, after the graph is properly initialized to the right size one can simply go ahead and for each edge with endpoints *V1* and *V2* which is read from the input file we call the code shown in Listing 6.

```
// Add a new edge with the two endpoints V1 and V2.
htd::id_t edgeId = g->addEdge(V1, V2);
```

Listing 6 Creating a new edge.

Note that for adding hyperedges, one can also use the method `addEdge` and call it with a vector or any other collection instead of the two endpoints shown in the example code. The function always returns the ID of the corresponding edge. For graph types which do not allow duplicate edges, the function returns the ID of the unique edge with the same endpoints.

6.3 Decomposing the Input Graph

After the input file is parsed successfully, we can now decompose the graph. As already mentioned, *htd* supports a variety of decomposition types together with their corresponding decomposition algorithms. As the general schema for obtaining a decomposition of a given graph in *htd* is very similar for all decomposition types, we pick here the example of a "plain" tree decomposition of type `htd::ITreeDecomposition`.

If we just need a simple tree decomposition we can use the code from Listing 7 and we will obtain a non-normalized tree decomposition where each bag is subset-maximal with respect to the other bags in the same decomposition. As mentioned earlier, the algorithms in *htd* are required to be able to deal with empty and disconnected input graphs. This means that one can feed them any input graph and the result will be a valid decomposition of the requested type.

```
// Obtain a new instance of the default tree decomposition algorithm.
htd::ITreeDecompositionAlgorithm * algorithm =
    manager->treeDecompositionAlgorithmFactory()
        .getTreeDecompositionAlgorithm(manager);

// Compute a new tree decomposition of the given graph.
htd::ITreeDecomposition * decomposition = algorithm.computeDecomposition(*g);
```

Listing 7 Computing a tree decomposition.

We can see that obtaining a plain tree decomposition of a graph for a dynamic programming algorithm is a very simple task using *htd*. This is a nice observation, but often an arbitrary tree decomposition is not good enough as the absence of structural guarantees makes the design of the dynamic programming algorithms much more complex and often this increased code complexity has negative effects on the performance.

To illustrate that *htd* also makes applying modifications to decompositions very convenient, let's have a look at Listing 8. The code snippet is to be placed before `computeDecomposition`

is called. In the given example we ensure that each join node only has children for which the bag content is equal to the bag content of the respective join node. Having this guarantee is often very useful as it makes join operations much easier to implement.

```
// Ensure that each child bag of a join node matches the join node's bag.
algorithm->addManipulationOperation
    (new htd::JoinNodeNormalizationOperation(manager));
```

Listing 8 Computing a tree decomposition.

Clearly, one can also request multiple manipulation operations to be applied automatically by a decomposition algorithm. In this case, the manipulations are applied in the order they were provided. For convenience, in addition to the possibility to apply manipulations globally to all decompositions computed by an algorithm instance, one can also request manipulations directly in the call to `computeDecomposition`. In this case, all manipulation operations provided in the call to `computeDecomposition` are applied after all operations are provided to the decomposition algorithm.

6.4 Decomposing the Input Graph with Optimization

Sometimes, the manipulation of a tree decomposition like shown before is not enough and we want to obtain an optimized decomposition with respect to a custom fitness function. Also for scenarios of this kind, *htd* offers an easy-to-use workflow.

To illustrate this workflow in the context of our working example let's assume that we want to obtain a tree decomposition which is of lowest width and whose height is minimal. Furthermore, let's assume that minimizing the width is more important than minimizing the height of the tree decomposition. To achieve this goal, we first have to define the simple fitness function shown in Listing 9.

```
class FitnessFunction : public htd::ITreeDecompositionFitnessFunction
{
public:
    htd::FitnessEvaluation * fitness(const htd::IMultiHypergraph &,
        const htd::ITreeDecomposition & decomposition) const
    {
        return new htd::FitnessEvaluation(2,
                            -(double)(decomposition.maximumBagSize()),
                            -(double)(decomposition.height()));
    }

    FitnessFunction * clone(void) const { return new FitnessFunction(); }
};
```

Listing 9 Fitness function for minimizing the width and height of a tree decomposition.

A fitness function basically is a class with a method `fitness` which takes two parameters, the input graph and the tree decomposition, and which returns a fitness evaluation consisting of an arbitrary number of levels (corresponding to the priorities) each represented by a value of type `double`. The constructor of `htd::FitnessEvaluation` takes a (non-empty) parameter list

where the first argument of the constructor is an integer value determining the number of levels the evaluation will contain. A decomposition A is considered better than a decomposition B if the fitness evaluation of A is lexicographically greater than B's fitness evaluation. In our example, as we want to minimize the width and height of the tree decomposition, we have to negate the corresponding values before we create the fitness evaluation object. In Listing 9 we can also see that there is a method `clone`. This function is required by almost all classes in *htd* as we often need deep copies of an object and this is exactly the purpose of the `clone` function.

```
// Obtain a new instance of the default tree decomposition algorithm.
htd::ITreeDecompositionAlgorithm * baseAlgorithm =
    manager->treeDecompositionAlgorithmFactory()
        .getTreeDecompositionAlgorithm(manager);

// Create a new fitness function object.
FitnessFunction function;

// Create a new optimization operation for selecting the optimal root.
htd::TreeDecompositionOptimizationOperation * operation =
    new htd::TreeDecompositionOptimizationOperation
        (manager.get(), function);

// Consider at most ten randomly selected vertices as new root node.
operation->setVertexSelectionStrategy
    (new htd::RandomVertexSelectionStrategy(10));

// Ensure that each child bag of a join node matches the join node's bag.
operation->addManipulationOperation
    (new htd::JoinNodeNormalizationOperation(manager));

// Compute a new tree decomposition of the given graph.
htd::ITreeDecomposition * td = algorithm.computeDecomposition(*g);

// Apply the optimization operation to the tree decomposition algorithm.
baseAlgorithm->addManipulationOperation(operation);

// Create a new instance of a tree decomposition algorithm
// which iteratively calls the base algorithm and returns
// the decomposition with optimal fitness.
htd::IterativeImprovementTreeDecompositionAlgorithm algorithm(manager.get(),
                                                              baseAlgorithm,
                                                              function);

// Compute at most ten decomposition of the input graph.
algorithm.setIterationCount(10);

// Abort the optimization process before the iteration limit is
// reached if no improvement was found in the last five iterations.
algorithm.setNonImprovementLimit(5);
```

Listing 10 Computing a tree decomposition.

After implementing the fitness function, the ten lines of code presented in Listing 10 suffice to enhance the code from Listing 7 in such a way that the fitness function will be maximized. This happens in two steps: The tree decomposition optimization operation takes a tree decomposition and tries to change its root in such a way that the fitness function is maximized and the iterative improvement algorithm calls the base algorithm (which automatically applies the optimization operation) repeatedly and returns the decomposition having optimal fitness. Clearly, one can also use different fitness functions in the two algorithms or nest the algorithms, but for our example the code is completely sufficient.

For the use in own implementations, one can freely customize the set of vertices which shall be considered in the search for the optimal root by choosing a different vertex selection strategy which is used by the tree decompostion optimization operation. *htd* offers a set of built-in selection strategies but one can also define custom strategies depending on the actual needs of the dynamic programming algorithm.

Note that in the listing above, the optimization algorithm takes care of applying the join node normalization operation which is also shown in Listing 8. Moving the responsibility for applying the manipulations from the base algorithm towards the optimization operation is necessary because we want the fitness function to be optimal for the total outcome. Applying the normalization operation after the optimization operation likely would destroy the property of optimality as the height could change when normalizing the join nodes.

In order to track the progress of optimization, an interesting aspect of the iterative improvement algorithm is the fact that we can call the function `computeDecomposition` with an additional lambda expression, like shown in Listing 11. In this example, the code outputs the width and height of each new decomposition.

```
htd::ITreeDecomposition * decomposition =
    algorithm.computeDecomposition(*g,
        [&](const htd::IMultiHypergraph &,
            const htd::ITreeDecomposition &,
            const htd::FitnessEvaluation & fitness)
    {
        std::size_t width = -fitness.at(0) - 1;
        std::size_t height = -fitness.at(1);

        std::cout << "Width: " << width << "    Height: " << height << std::endl;
    }
});
```

Listing 11 Tracking the optimization progress.

Finally, it is important to notice that the iterative improvement tree decomposition algorithm is safely interruptible. That is, one can call the `terminate` method of the corresponding manager and the iterative improvement tree decomposition algorithm will immediately return the best tree decomposition found so far or a `nullptr` in case that no decomposition was computed so far.

During the experiments we made in the context of the publication [Abseher *et al.*, 2015] it turned out that D-FLAT often benefits from a reduction of the bag size of join nodes. This is most probably caused by the fact that then the number of possible solution candidates which have to

be joined is potentially reduced. Due to the fact that the bag size of join nodes is not affected by changing the root of a normalized tree decomposition, in this case we can even omit the class `htd::TreeDecompositionOptimizationOperation` or leave away the fitness function in its constructor in order to make it a transparent "non-operation". Because the join node bag sizes in a normalized decomposition are not influenced when selecting a different vertex as root node, we decided to present the reduction of the decomposition height in the example code in Listing 10. The height of a tree decomposition is highly dependent on the actual node which acts as its root and so it is a much more intuitive application of the decomposition optimization operation of *htd*.

6.5 Working with the Decomposition

After the tree decomposition is computed, we still have to execute the dynamic programming algorithm on it. Due to the fact that there are various ways to implement a dynamic programming algorithm for a given problem, we do not go into details of the concrete algorithm for MINIMUM DOMINATING SET at this point. Instead, we provide in Listing 12 a short example how a computed tree decomposition together with the induced edges for the bags can be printed in a human-readable form as this code can act as a general starting point to get an idea how *htd* supports the implementation of dynamic programming algorithms also beyond the plain decomposition of input graphs.

```
htd::PreOrderTreeTraversal traversal;

traversal.traverse(*decomposition, [&](htd::vertex_t vertex,
                                       htd::vertex_t parent,
                                       std::size_t distanceToRoot)
{
    for (htd::index_t index = 0; index < distanceToRoot; ++index)
    {
        outputStream << "    ";
    }

    std::cout << "NODE " << vertex << ": " <<
                 decomposition.bagContent(vertex) << std::endl;

    for (const htd::Hyperedge & e : decomposition.inducedHyperedges(vertex))
    {
        for (htd::index_t index = 0; index < distanceToRoot + 1; ++index)
        {
            outputStream << "    ";
        }

        std::cout << e.id() << ": " << e.elements() << std::endl;
    }
});
```

Listing 12 Printing a tree decomposition.

The code above traverses the tree decomposition in preorder and outputs for each node its ID together with its corresponding bag content. The very helpful and powerful feature for easily deriving the (hyper-)edges induced by a bag is also illustrated in the example source code shown in Listing 12. With this feature one automatically has direct access to all hyperedges whose elements are a subset of the current bag content. In this way one can save huge portions of the total runtime compared to tree decomposition libraries which do not provide this feature as in the latter case one has to do a subset check for each (hyper-)edge in each node's bag. This can be very expensive in the presence of graphs with millions of edges.

6.6 Upgrading the Dynamic Programming Algorithm

At this point we presented all aspects of *htd* which are needed to solve the problem of MINIMUM DOMINATING SET. In this section we will have a glance on how we can exploit the full potential of *htd* when we want to upgrade the dynamic programming algorithm so that it we can easily make the algorithm work with vertex weights. The idea is to solve the problem of MINIMUM WEIGHTED DOMINATING SET by using *htd* support for arbitrary labels.

First, let's recall the definition of MINIMUM WEIGHTED DOMINATING SET: The problem is defined on a graph $G = (V, E)$ with vertex weights $W : V \to \mathbb{R}$ and we want to find all sets $S \subseteq V$ of minimum cost such that each vertex in V is either contained in S or adjacent to at least one vertex inside S. The cost of a dominating set S is defined as the sum of all vertex weights of the vertices in S.

To adapt the basic algorithm for MINIMUM DOMINATING SET so that it is able to deal with vertex weights, we just have to sum up the proper weight instead of the value 1 for each selected vertex. Clearly, we can achieve this modification of the dynamic programming algorithm presented previously by maintaining a mapping between the vertices and the corresponding weights in the algorithm, but this would involve using a global mapping variable or passing the mapping variable as a parameter to the function which computes the dynamic programming tables, probably causing high maintenance effort. As a workaround and in order to keep the code clean and highly maintainable, one can use *htd*'s functionality to add, remove and access vertex and edge labels in the context of any available decomposition and labeled graph type. An example is given in Listing 13 where we add a vertex label to Vertex 1 of type double, representing its weight.

```
// Get a new instance of the default multi-graph implementation.
htd::IMutableLabeledMultiGraph * g =
    manager->labeledMultiGraphFactory().getLabeledMultiGraph(Vertices);

// Add a vertex label "Weight" to the vertex with ID 1.
g->setVertexLabel("Weight", 1, new htd::Label<double>(3.14159));

// Access the vertex label "Weight" assigned to the vertex with ID 1.
double weight = htd::accessLabel<double>(g->vertexLabel("Weight", 1));

// Remove the vertex label "Weight" from the vertex with ID 1.
g->removeVertexLabel("Weight", 1);
```

Listing 13 Using *htd*'s support for labels.

Following this example, all what remains to do in the context of the dynamic programming algorithm is to cast the reference to `htd::IMultiHypergraph` which is used by the algorithms of *htd* as the basic graph interface to the labeled graph type at hand (which is clearly a valid and permitted up-cast) and call the desired access methods of the graph for the vertex and edge labels that are needed by the dynamic programming algorithm. We can see that the extension of the algorithm just involves changing a few lines of code while fully maintaining the readability and maintainability of the original code.

7 Performance Characteristics

In this section we give first results regarding the performance characteristics of our framework. In order to have an indication for the actual efficiency, we compare *htd* 0.9.9 [Abseher, 2016] to the other participants Track A ("Treewidth") of the "First Parameterized Algorithms and Computational Experiments Challenge 2016" ("PACE16", see `https://pacechallenge.wordpress.com/track-a-treewidth/`).

The experiments are based on the following algorithms submitted to the PACE16 challenge:

- 1: "tw-heuristic"
  Available at `https://github.com/mrprajesh/pacechallenge`.
  GitHub-Commit-ID: 6c29c143d72856f649de99846e91de185f78c15f
- 5 (htd): "htd_gr2td_minfill_exhaustive.sh"
  Available at `https://github.com/mabseher/htd`
  GitHub-Commit-ID: f4f9b8907da2025c4c0c6f24a47ff4dd0bde1626
- 6: "tw-heuristic"
  Available at `https://github.com/maxbannach/Jdrasil`
  GitHub-Commit-ID: fa7855e4c9f33163606a0677485a9e51d26d7b0a
- 9: "tw-heuristic"
  Available at `https://github.com/elitheeli/2016-pace-challenge`
  GitHub-Commit-ID: 2f4acb30b5c48608859ff27b5f4e217ee8346ca5
- 10: "tw-heuristic"
  Available at `https://github.com/mfjones/pace2016`
  GitHub-Commit-ID: 2b7f289e4d182799803a014d0ee1d76a4de70c1f
- 12: "flow_cutter_pace16"
  Available at `https://github.com/ben-strasser/flow-cutter-pace16`
  GitHub-Commit-ID: 73df7b545f694922dcb873609ae2759568b36f9f

The list of algorithms contains all participants of the sequential heuristics track of the PACE treewidth challenge in the variant in which they were submitted to the challenge. For each of the algorithms we provide its ID that was used in the challenge (and also in our experiments), the name of the binary, the location of its source code and the exact identifier of the program version in the GitHub repository. Note that for *htd* (ID 5), four different configurations exist. In our experiments here we only consider the best-performing variant, namely "htd_gr2td_minfill_exhaustive.sh". Its implementation iteratively calls the minfill algorithm in order improve the width of obtained the tree decomposition using the class `htd::IterativeImprovementTreeDecompositionAlgorithm`.

The evaluation is done separately for two data sets. The first data set is the official data set of the PACE challenge[5] and the second one is DataSet 1 from the QBFEVAL'16 competition of solvers for quantified boolean formulae[6]. Note that the second data set had to be converted in order to comply to the input format of the PACE competition. This was done in the following way: For each QBF formula we ignore the quantifier information and we introduce a clique between the vertices of each clause, that is, we consider the clauses in the QBF formula as hyperedges and work on the primal graph of the QBF formula. The "PACE" data set contains 282 instances and the "QBF" data set contains 825 instances.

All our experiments were performed on a single core of an Intel Xeon E5-2637@3.5GHz processor running Debian GNU/Linux 8.3 and each test run was limited to a run-time of at most 100 seconds and 32 gigabyte of main memory. For the actual evaluation we use the testbed of the PACE challenge which is available at `https://github.com/holgerdell/PACE-treewidth-testbed`.

Due to the fact that the QBF instances often contain large clauses, the converted instances sometimes have file sizes of several gigabytes and also the output of the algorithms might be large, we increased the kill time, i.e., the time after which the process is killed when the output stream is not closed by the algorithm itself after sending the terminating request, to 30 seconds. For repeatability of the experiments we provide the complete testbed and the results for each data set and each algorithm under the following link: `www.dbai.tuwien.ac.at/proj/dflat/evaluation_htd099.zip`

Figures 5 and 6 summarize the outcome of our evaluation. The first figure shows the results for the instances of the "PACE" data set and the right-hand side is dedicated to the results for the "QBF" data set. Each of the plots is constructed by running *htd* and each of the five other algorithms of the competition on each instance. Afterwards, we plot the cumulative frequency of the obtained width after 100 seconds.

The gray line indicates the performance of *htd* and the dashed lines illustrate the width achieved by its competitors. A point $p = (x, y)$ on a line in the figures represents the fact that y instances could be successfully decomposed within the time limit of 100 seconds and each decomposition had a width not higher than x. Hence, it is good to minimize x with respect to y, that is, the optimal algorithm reaches a certain point on the y-axis not exceeding the widths of its competitors on the x-axis.

[5] Available at `https://github.com/holgerdell/PACE-treewidth-testbed`

[6] Available at `http://www.qbflib.org/TS2016/Dataset_1.tar.gz`

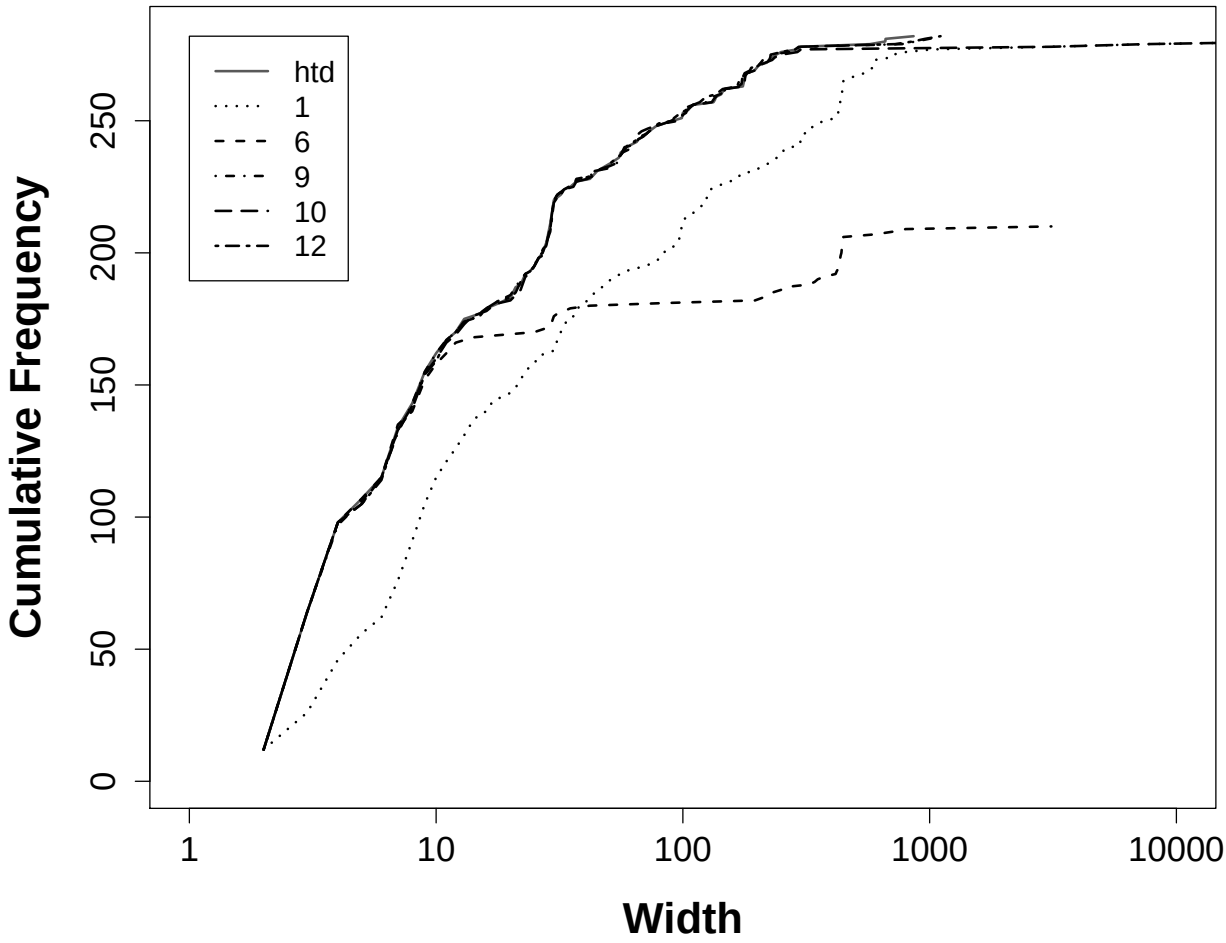


Figure 5 Benchmark-Instances “PACE”

In Figure 5 we can see that *htd* 0.9.9 achieves a similar or even slightly better width on the PACE data set and the worst width achieved by *htd* is better than the worst width of all other algorithms. This indicates that *htd* is well suited for computing decompositions of small width on the given instances. We can also see that four out of six algorithms were able to decompose all instances within 100 seconds and that the difference between the four algorithms is very small.

When we look at the “QBF” data set, illustrated in Figure 6, the picture is slightly more diverse. We can see that about 100 instances cannot be decomposed by any of the algorithms and that two algorithms (9 and 12) actually are able to decompose slightly more instances than *htd*. In the fragment of the figure around width 1000 we observe that *htd* performs significantly better than the other algorithms. When the width gets closer to 5000 and also in the case of higher widths, two competitors (9 and 12) can decompose more instances.

The effect that *htd* performs well on instances up to a certain width and that it gets harder for the library to finish decomposing graphs of high treewidth can be explained by the fact that the decompositions in the context of *htd* always carry the information of induced edges in order to support dynamic programming algorithms. The higher the width gets, the farther the edge

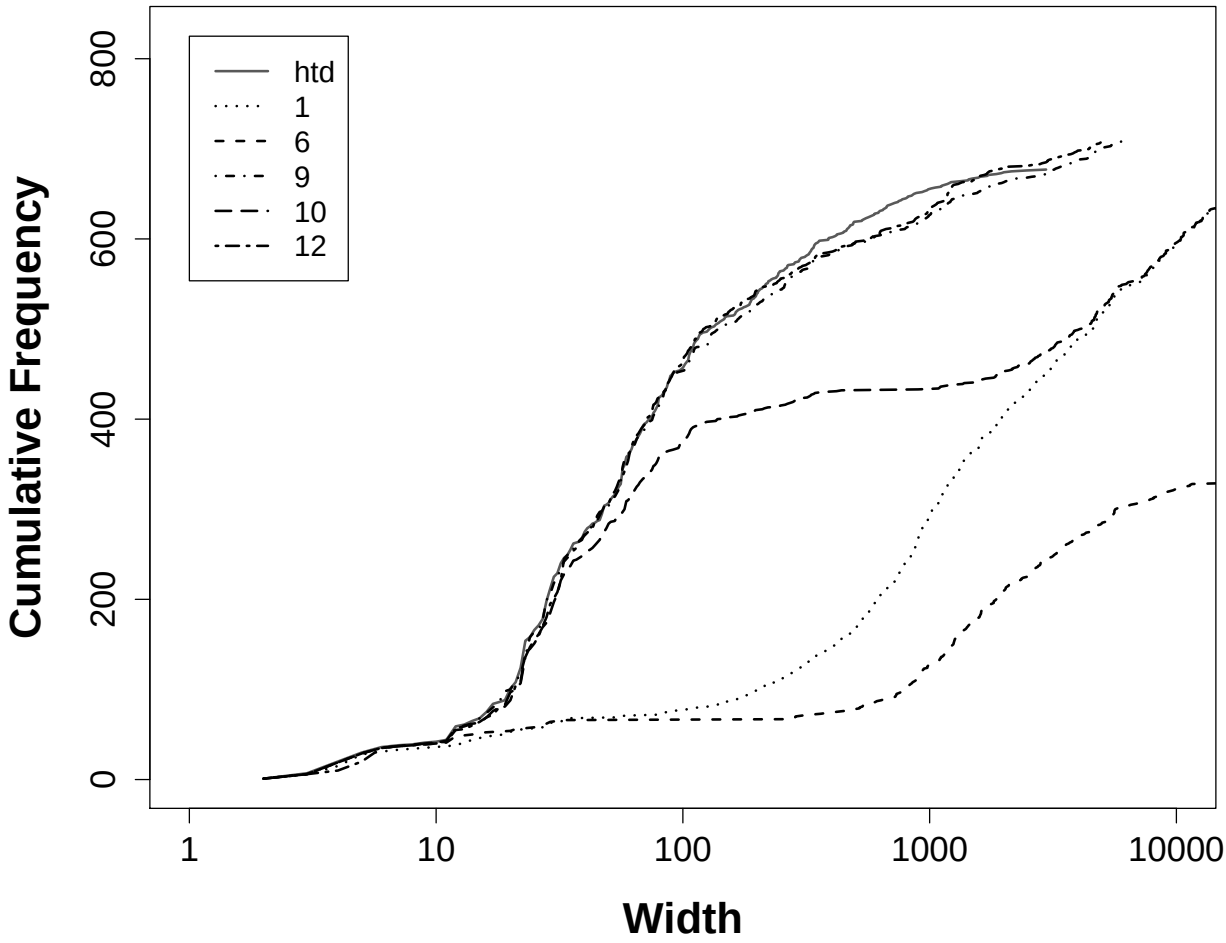


Figure 6 Benchmark-Instances “QBF”

information has to be distributed in the decomposition (see Section 5.2), causing a significant overhead when the width is very high. Additionally, the decompositions computed by *htd* contain only subset-maximal bags by default, which also can be time-consuming in the context of very large graphs. When we omit these steps and go for “pure” decomposition performance, *htd* likely will be able to decompose significantly more instances within the time limit.

8 Conclusion

In this report we presented a free, open-source C++ framework for graph decompositions. We show the most important features, give an introduction on how to use the library and highlighted issues we faced during the implementation phase and provide insights on how we coped with them. Furthermore, we evaluated our approach by comparing our library with the participants of the ”First Parameterized Algorithms and Computational Experiments Challenge”. The outcome of the evaluation indicates that the performance characteristics of the new framework are very promising.

For future work we clearly want to further improve the built-in heuristics and algorithms in order to enhance the capabilities for generation of decompositions of small width. Furthermore we are currently working on refining the algorithms allowing to automate the process of computing customized tree decompositions.

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.

References

[Abseher *et al.*, 2014] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT System for Dynamic Programming on Tree Decompositions. In *Proc. JELIA*, volume 8761 of *Lecture Notes in Artificial Intelligence*, pages 558–572. Springer, 2014.

[Abseher *et al.*, 2015] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. In *Proc. IJCAI*, pages 275–282. AAAI Press, 2015.

[Abseher *et al.*, 2016a] Michael Abseher, Marius Moldovan, and Stefan Woltran. Providing built-in counters in a declarative dynamic programming environment. In *Proc. KI 2016: Advances in Artificial Intelligence*, pages 3–16. Springer, 2016.

[Abseher *et al.*, 2016b] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. Technical Report DBAI-TR-2016-94, TU Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-94.pdf`.

[Abseher, 2016] Michael Abseher. htd 0.9.9, 2016. Available at `https://github.com/mabseher/htd/tree/f4f9b8907da2025c4c0c6f24a47ff4dd0bde1626`.

[Arnborg and Proskurowski, 1989] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[Bachoore and Bodlaender, 2006] Emgad H. Bachoore and Hans L. Bodlaender. A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth. In *Proc. AAIM*, volume 4041 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2006.

[Berry *et al.*, 2003] Anne Berry, Pinar Heggernes, and Geneviève Simonet. The minimum degree heuristic and the minimal triangulation process. In *Graph-Theoretic Concepts in Computer Science: 29th International Workshop, WG 2003*, pages 58–70. Springer, 2003.

[Bertelè and Brioschi, 1973] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973.

[Bodlaender and Koster, 2008] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial Optimization on Graphs of Bounded Treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[Bodlaender and Koster, 2010] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[Charwat and Woltran, 2016] Günther Charwat and Stefan Woltran. Dynamic Programming-based QBF Solving. In *Proceedings of the International Workshop on Quantified Boolean Formulas*, 2016.

[Clautiaux *et al.*, 2004] François Clautiaux, Aziz Moukrim, Stèfane Négre, and Jaques Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Operations Research*, 38:13–26, 2004.

[Dechter, 1999] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1):41 – 85, 1999.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Dermaku *et al.*, 2008] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin McMahan, Nysret Musliu, and Marko Samer. Heuristic Methods for Hypertree Decomposition. In *Proc. MICAI*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2008.

[Downey and Fellows, 1999] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[Fichte *et al.*, 2016] Johannes Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. DynASP 2.0: System Specification. Technical Report DBAI-TR-2016-101, TU Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-101.pdf`.

[Fulkerson and Gross, 1965] Delbert R. Fulkerson and Oliver A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[Gavril, 1972] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum coloring cliques and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1:180–187, 1972.

[Gogate and Dechter, 2004] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proc. UAI*, pages 201–208. AUAI Press, 2004.

[Gottlob *et al.*, 2002] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[Gottlob *et al.*, 2009] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM*, 56(6):1–32, 2009.

[Gutin, 2015] Gregory Gutin. Should we care about huge imbalance in parameterized algorithmics? *The Parameterized Complexity Newsletter*, December 2015.

[Halin, 1976] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[Hammerl and Musliu, 2010] Thomas Hammerl and Nysret Musliu. Ant colony optimization for tree decompositions. In *Proc. EvoCOP*, volume 6022 of *Lecture Notes in Computer Science*, pages 95–106. Springer, 2010.

[Hammerl *et al.*, 2015] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic Algorithms and Tree Decomposition. In *Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.

[Jégou and Terrioux, 2014] Philippe Jégou and Cyril Terrioux. Bag-Connected Tree-Width: A New Parameter for Graph Decomposition. In *Proc. ISAIM*, pages 12–28, 2014.

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

[Kloks, 1994] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[Koster *et al.*, 1999] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving Frequency Assignment Problems via Tree-Decomposition 1. *Electronic Notes in Discrete Mathematics*, 3:102–105, 1999.

[Koster *et al.*, 2001] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. Treewidth: Computational experiments. *Electronic Notes in Discrete Mathematics 8*, 2001.

[Larranaga *et al.*, 1997] Pedro Larranaga, Cindy M.H. Kujipers, Mikel Poza, and Roberto H. Murga. Decomposing bayesian networks: Triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7 (1):19–34, 1997.

[Lauritzen and Spiegelhalter, 1988] Steffen L. Lauritzen and David J. Spiegelhalter. 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.

[McMahan, 2004] Ben McMahan. Bucket elimination and Hypertree Decompositions, 2004. Implementation Report, Institute of Information Systems (DBAI), TU Wien.

[Morak *et al.*, 2010] Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A Dynamic-Programming Based ASP-Solver. In *Proc. JELIA*, volume 6341 of *Lecture Notes in Artificial Intelligence*, pages 369–372. Springer, 2010.

[Morak *et al.*, 2012] Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In *Proc. Learning and Intelligent Optimization Conference, LION 2012*, volume 7219 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2012.

[Musliu and Schafhauser, 2007] Nysret Musliu and Werner Schafhauser. Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, 1(3):317–340, 2007.

[Musliu, 2008] Nysret Musliu. An iterative heuristic algorithm for tree decomposition. *Studies in Computational Intelligence, Recent Advances in Evolutionary Computation for Combinatorial Optimization*, 153:133–150, 2008.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[Robertson and Seymour, 1984] Neil Robertson and Paul Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Ser. B*, 36(1):49–64, 1984.

[Robertson and Seymour, 1991] Neil Robertson and Paul Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153 – 190, 1991.

[Schafhauser, 2006] Werner Schafhauser. New heuristic methods for tree decompositions and generalized hypertree decompositions. Master's thesis, TU Wien, Wien, Austria, 2006.

[Shoikhet and Geiger, 1997] Kirill Shoikhet and Dan Geiger. A Practical Algorithm for Finding Optimal Triangulations. In *Proc. AAAI/IAAI*, pages 185–190. AAAI Press / The MIT Press, 1997.

[Tarjan and Yannakakis, 1984] Robert E. Tarjan and Mihalis Yannakakis. 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.

[Tarjan, 1972] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Xu *et al.*, 2005] Jinbo Xu, Feng Jiao, and Bonnie Berger. A tree-decomposition approach to protein structure prediction. In *Computational Systems Bioinformatics Conference, CSB 2005*, pages 247–256, 2005.