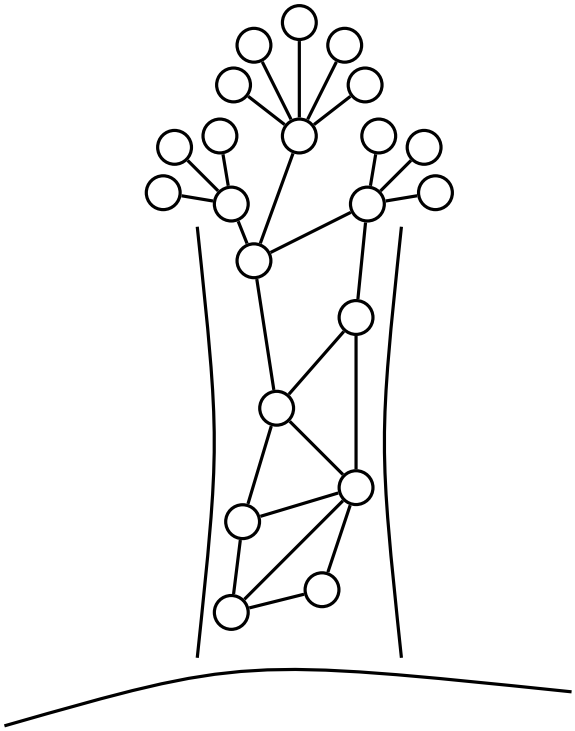


Jdrasil Manual

For Version: 0.2

Compiled: April 22, 2018

Part I:
Getting Started

0.1 Technical Overview and Design Principles

Jdrasil is a modular Java library for computing tree decompositions both, exact and heuristically. The goal of Jdrasil is to allow other projects to add tree decompositions to their applications as easy as possible. In order to achieve this, Jdrasil is designed in a very modular way: Every algorithm is implemented as interchangeable as possible. At the same time, algorithms are implemented in a clean object oriented manner, making it easy to understand and extend the implementation. We hope that this approach makes it easier to study the practical aspects of tree width algorithms, and that it helps to push the practical usage of tree decompositions.

To make the usage of Jdrasil as easy as possible, the whole library can be compiled and used without any dependencies. Everything boils down to a single, platform independent `jar` file that can freely be used with the mentioned copyright.

However, this design principle comes with the price of algorithms that are not "on the edge optimized". We believe that this is not a problem for most applications, as the algorithms run in the same principle time complexity anyway. If we, however, wish to use super optimized algorithms, Jdrasil provides the concept of *upgrades*. An upgrade is a piece of third party software (e. g., another Java library, an optimized C implementation of an algorithm, a SAT solver, ...) that can be used to *replace* some functionality of Jdrasil. An upgrade will usually boost the performance of Jdrasil in general, or on a specific target platform (for instance on a parallel machine), and will add dependencies (Jdrasil with upgrades may not be platform independent). Nevertheless, an upgrade will *not* increase the functionality of Jdrasil, only the performance and, hence, a scripted based on an upgraded version of Jdrasil is still platform independent.

0.2 Contributors

The core developer of Jdrasil are (in lexicographical order):

- Max Bannach
- Sebastian Berndt
- Thorsten Ehlers

0.3 How to Cite Jdrasil

If you use Jdrasil for your project, please refer to our GitHub page and respect the copyright. If you use Jdrasil for a research project, please cite our introduction paper:

```
@inproceedings{jdrasil,
  author = {Max Bannach and Sebastian Berndt and Thorsten Ehlers},
  title = {Jdrasil: A Modular Library for Computing Tree Decompositions},
  booktitle = {Experimental Algorithms – 16th International Symposium, {SEA} 2017,
               London, England, June 21 -- 23, 2017, Proceedings},
  year = {2017}
}
```

Chapter 1: Build and Run *J*drasil

1.1 Obtain and Use the Latest Version

The latest version of *J*drasil can be obtained from `https://maxbannach.github.io/Jdrasil/`. The obtained `.jar` file is executable, i. e., if *J*drasil should be used as standalone solver, we can simple use:

```
# this will execute the exact mode
java -jar Jdrasil.jar
```

or alternatively

```
java -cp Jdrasil.jar jdrasil.Exact
```

In addition, we can execute *J*drasil in the heuristic or approximation mode:

```
java -cp Jdrasil.jar jdrasil.Heuristic
java -cp Jdrasil.jar jdrasil.Approximation
```

Note that the heuristic mode tries to improve the solution until a `SIGTERM` is received, i. e., *J*drasil has to be stopped manually in this mode.

With the same `.jar` file, *J*drasil can also be used as library: simply add the `.jar` to the classpath of the desired project. For instance, the following simple program uses *J*drasil to compute an exact tree decomposition:

```
import jdrasil.graph.*;
import jdrasil.algorithms.*;

public class Main {
  public static void main(String[] args) {
    // create your own graph
    Graph<Integer> G = GraphFactory.emptyGraph();
    for (int v = 1; v <= 5; v++) { G.addVertex(v); }
    G.addEdge(1, 2);
    G.addEdge(1, 4);
    G.addEdge(2, 3);
    G.addEdge(2, 4);
    G.addEdge(4, 5);

    // output graph in PACE format
    System.out.println(G);

    // compute exact decomposition
    TreeDecomposition<Integer> td = null;
    try {
      td = new ExactDecomposer<Integer>(G).call();
    } catch (Exception e) {
```

```
            System.out.println("something went wrong");
        }

        // ouput tree decomposition in PACE format
        System.out.println(td);
    }
}
```

If the program is stored in a file `Main.java` , it can be compiled and used with the following commands:

```
javac –cp Jdrasil.jar: Main.java
java –cp Jdrasil.jar: Main
```

1.2 Program Arguments

The behavior of *J*drasil can be influenced by program arguments. The following tables provides an overview of all available arguments:

Argument	Effect	Exact	Heuristic	Approximation
-h	prints a help dialog	✓	✓	✓
-s <seed>	seeds the RNG	✓	✓	✓
-parallel	enables parallel processing (the `GraphSplitter` will split and handle atoms in parallel)	✓		✓
-instant	heuristic will output a solution as fast as possible		✓	
-log	prints log information during computation	✓	✓	✓

Some of the arguments may only make sense if *J*drasil is used as stand alone program, others may also be useful in an scenario where *J*drasil is used as a library. For instance, the argument "-parallel" modifes some algorithms to run in parallel, this is useful in other applications as well. *J*drasil handles such properties in the class `JdrasilProperties` and to enable an property "-x" (for instance "-parallel") we can do from source:

```
// set value of property "x" to "some value"
JdrasilProperties.setProperty("x", "some value");
// just add property "parallel"
JdrasilProperties.setProperty("parallel", "");
```

Note that some properties (as the random seed) need an actual value, this is the second argument in the example code above.

1.3 Build *J*drasil from Source

*J*drasil uses Gradle[1] as building tool. Thereby it takes advantage of the Gradle wrapper: in order to build *J*drasil the only requirements are an up-to-date JDK[2] and an active internet connection. The Gradle wrapper will download everything needed by itself.

To get started, we need the latest version of *J*drasil and switch to its root directory:

```
git clone https://github.com/maxbannach/Jdrasil.git
cd Jdrasil
```

The folder contains a directory `subprojects`, which contains the source code of *J*drasil. Besides this, there are a couple of Gradle files, most of which do not require our attention. The only important file is `gradlew` (or `gradlew.bat` on Windows). This is the Gradle wrapper that we will use to build *J*drasil. In order to use it, we have to execute on of the following commands (depending on our operating system):

```
# on Unix
./gradlew <task>
# or, if gradlew is not executable
sh gradlew <task>
# on Windows
gradlew <task>
```

In the rest of the manual, we will use the syntax for an Unix system. But everything can be done in the same way on a Windows machine.

1.3.1 Build the Executable and Library

To compile the core source code of *J*drasil we use one of the following commands:

```
# just build
./gradlew assemble
# or build and test
./gradlew build
```

This will create a directory `build`, containing the directories `classes/main` and `jars`. The first directory will contain the compiled class files, the second will contain `Jdrasil.jar`.

In order to run the freshly build *J*drasil, we can do one of the following:

```
java -jar build/jars/Jdrasil.jar
java -cp build/jars/Jdrasil.jar jdrasil.Exact
java -cp build/classes/main jdrasil.Exact
```

To use the present build of *J*drasil as library, we can simply add the created `.jar` file to the classpath of our desired project.

[1] `www.gradle.org`
[2] *J*drasil needs at least Java 8.

1.4 Build the Documentation

The documentation of *J*drasil consists of two parts: the classic JavaDocs, which provide detailed information about the individual classes, and this manual, which provides an high level view on some design principles used by *J*drasil. The JavaDocs can, without further requirements, be build by the following command:

```
./gradlew javadoc
```

This will create the folder `builds/docs/javadoc` containing the documentation.

This manual is written in LaTeX and in order to compile it, an up-to-date LuaLaTeX installation must be available. If this is the case, the manual can be typeset by

```
./gradlew manual
```

This command will place this manual as `.pdf` file in `builds/docs/manual`.

1.5 Installing Upgrades

As mentioned earlier, *J*drasil can be build and used without any dependencies. This makes it easy to update, distribute, and use *J*drasil. However, sometimes third-party software may provide a significant speed up in the process of solving some subproblems. In such scenarios, the speed of *J*drasil can be improved by *upgrades*. This topic will be discussed in detail in section IV.

All upgrades have in common, that *J*drasil uses the following convention to build and use them. To {download, build, install} (depending on the upgrade) an upgrade `x`, we can execute the following command:

```
./gradlew upgrade_x
```

Which will {download, build, install} the required files and place them in `build/upgrades`. To run *J*drasil with the installed upgrade, either do (if the upgrade is a Java library):

```
java -cp build/jars/Jdrasil.jar:build/upgrades/x.jar  jdrasil.Exact
```

or (if the upgrade is a native library):

```
java -Djava.library.path=build/upgrades -jar build/jars/Jdrasil.jar
```

The start scripts of *J*drasil (see next section) will automatically look for upgrades in these locations.

1.6 Building Startscripts (not only) for PACE

As *J*drasil was developed for the *Parameterized Algorithms and Computational Experiments Challenge* (PACE) [8] in the first place, it naturally provides the interfaces required by PACE. To build them, we can simply use:

```
./gradlew pace
```

This will, if not already done, build the Java files and will place two shell scripts in the root directory: `tw-exact` and `tw-heuristic`. The first script will execute *J*drasil in exact mode (meaning it reads an graph from stdin, computes an optimal decomposition, and prints it on stdout); while the second script runs *J*drasil in heuristic mode, meaning it will run in an infinite loop trying to heuristically find a good decomposition – the decomposition will be printed if a SIGINT or SIGTERM is received. Example usage is:

```
./tw-exact -s 42 < myGraph.gr > myGraph.td
./tw-heuristic -s 42 < myHugeGraph.gr > myHugeGraph.gr.td
```

For more details about the usage of these scripts, take a look at the PACE website: `https://pacechallenge.wordpress.com/pace-2017/track-a-treewidth/`.

These scripts are available both, as Shell script for UNIX systems and as `.bat` script for Windows. They can also be build directly via:

```
./gradlew exact
./gradlew heuristic
```

Finally, there are also scripts available for running *J*drasil with approximation algorithms:

```
./gradlew approximation
```

This will generate `tw-approximation`, which can be used as the scripts from above.

Part II:
Graphs

Since *J*drasil is a tool for computing tree decompositions, it implements a variety of graph algorithms. Hence, many graphs and "graph objects" will be handed throughout the library. It is, therefore, worth to spend some time and study the design principles *J*drasil follows when it handles graphs.

All classes and methods that are designed to deal with graphs directly are stored in the package `jdrasil.graph`. One (not that small) exception is the collection of algorithms and procedures that are meant to compute tree decompositions – as this is *J*drasil's main task, they obtain some extra packages. Within the graph package, the working horse is the class `Graph` which represents all graphs that occur within *J*drasil. The following sections will capture the design of this class, how we can obtain and store objects of the class, and how we can modify existing `Graph` objects. Furthermore, we will discuss how *J*drasil implements algorithms for computing graph properties and invariants, and, most importantly, how tree decompositions are managed.

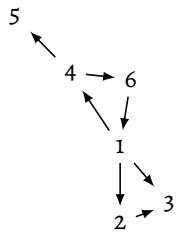
Chapter 2: The Graph Class

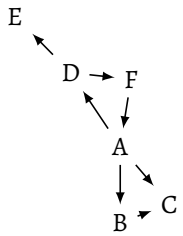
The work horse of *J*drasil's graph engine is the class `jdrasil.graph.Graph`. There are a couple of design decisions that were made during the development of this class, which grant some advantages in the context of computing tree decompositions, but which may result in disadvantages in other algorithmic tasks. We will discuss these implementation details in the following.

An object of the `Graph` class represents a directed graph $G = (V, E)$ where V is a set of *arbitrary objects*, and $E \subseteq V \times V$. An undirected graph is represented as directed graph with a symmetric edge relation. Since, in the context of computing tree decompositions, we will often deal with undirected graphs, the class provides most methods in an additional symmetric implementation, so that the `Graph` class can be used to work with undirected graphs in a natural way.

Note: This generic approach nevertheless allows the usage of vertex classes. Indeed, even the vertex class from, say another library, can be plugged in.

As stated above, the `Graph` represents a directed graph where V is a set of arbitrary objects. Consequently, there is no vertex or node class in *J*drasil, instead, the `Graph` class is generic and *any class* can be used as vertex type – the only restriction is that the class has a natural order, i. e., it is *comparable*. For instance, the following graphs are of the types `Integer` and `Character`, respectively.





We could create the graph from above with any other comparable Java class; and also run all the algorithms implemented by *J*drasil on it. Especially, for a class representing subsets of vertices, we can represent the following graphs:

$$\{1,2,3\} - \{1,4,6\} \text{ —— } \{4,5\} \qquad \{A,B,C\} - \{A,D,F\} \text{ —— } \{D,E\}$$

This is essentially the way *J*drasil represents tree decompositions, see Chapter 4 for more details.

The second part we mentioned earlier is that we simply consider E as $E \subseteq V \times V$, and in fact, *J*drasil does not have a class representing an edge. The `Graph` class only represents adjacency information about its stored vertices. This makes working with the `Graph` class more natural in many situations and keeps algorithms simple. In the few cases where an edge class would be useful, for instance if we work with weighted graphs, we interpreted the edge label function, say $\lambda\colon E \to \Sigma$, as $\lambda\colon V \times V \to \Sigma \cup \{\bot\}$, where we have $\lambda(u,v) = \bot \Leftrightarrow (u,v) \notin E$.

2.1 Implementation Details

In the core of the `Graph` class, the graph is stored as adjacency list. Hence, we may iterate over the vertex and edge set in time $O(|V| + |E|)$. Furthermore, the edge relation is stored in an hash map, allowing us to perform adjacency tests in $O(1)$.

Adding vertices to the graph is performed in $O(1)$ as well, adding an edge (u, v) costs $O(\delta(v))$[1]. Adding directed edges actually is (practically) faster, as this realized by array manipulation. When an undirected edge is added, however, this time bound is strict, as some additional information about the neighborhood is gathered. This information can be used to get $O(1)$ access to some vertex properties, for instance, testing if a vertex is simplicial, or computing the fill-in value of a vertex.

2.2 Working with Graphs

To work with graphs, we first of all need one. Objects of the class `Graph` are generated by the `GraphFactory` – a simple empty graph can be obtained by:

```
Graph<Integer> myGraph = GraphFactory.emptyGraph();
```

Vertices can be added by the method `Graph.addVertex` , or by directly adding edges. The following two code fragments are equivalent:

```
// variant 1
myGraph.addVertex(1);
myGraph.addVertex(2);
myGraph.addEdge(1, 2);

// or simply
myGraph.addEdge(1, 2);
```

The above code constructs the *undirected* graph $1 \text{ —— } 2$; to create the *directed* graph $1 \longrightarrow 2$, we can use the function `Graph.addDirectedEdge`:

```
// directed version
myGraph.addDirectedEdge(1, 2);
```

Note: Since *J*drasil mainly works on undirected graphs, methods for directed graphs are marked with *directed*, while the methods for undirected graph have no prefix.

We can also mix the commands and create a mixed graph. However, usually, and in the following, we will stick to undirected graphs.

As we have added them, we can also remove vertices and edges from the graph:

```
// removes the vertex and all incident edges, i.e., the one from 1 to 2
myGraph.removeVertex(1);

// removes the specific undirected edge
myGraph.removeEdge(2, 3);
```

Removing an undirected edge will remove both directed edges, even if only one of them is present (i. e., if the edge is actually directed). Concretely, deleting a directed edge can be done by:

```
// directed version
myGraph.removeDirectedEdge(1, 2);
```

[1] $\delta(v)$ denotes the *degree* of v: the number of nodes connected to v.

2.2.1 Iterating Over the Graph

Many graph algorithms have to iterate over the vertices and edges of the graph. The `Graph` implements an iterator for its vertices, so to iterate over the vertices we can simply do (here, the graph has vertices of type `Integer`):

```
for (Integer v : myGraph) {
  // do something with v
}
```

As *J*drasil does not know edge objects, there is no direct iteration over edges. Instead, the neighborhood of a vertex is iterable as well. In the directed case, this is straight forward:

```
for (Integer v : myGraph) {
  for (Integer w : myGraph.getNeighborhood(v)) {
    // do something with the edge (v,w)
  }
}
```

In an undirected graph, however, we would iterate over every edge twice (remember that an undirected graph is an directed graph with symmetric edge relation). To overcome this issue, we can only pick the edge in which the first vertex is lexicographical smaller:

```
for (Integer v : myGraph) {
  for (Integer w : myGraph.getNeighborhood(v)) {
    if (v.compareTo(w) > 0) continue; // skip second edge
    // do something with the undirected edge {v,w}
  }
}
```

2.2.2 Some Special Methods

While most parts of the `Graph` class are kept as general as possible, there are some special methods, which are designed towards the computation of tree decompositions. They are defined for undirected graphs only, and will produce mixed graphs or undefined results on directed graphs. These methods are:

1. `contract(T v, T w)`: Contracts the *undirected* edge $\{v, w\}$ into the vertex v. This will connect all edges incident to w to v – this will, however, *not* create multi-edges. This method will return a `ContractionInformation` object, which can be used to revert the contraction.
2. `deContract(ContractionInformation info)`: Will revert the contraction of an edge.
3. `eliminateVertex(T v)`: Eliminating a vertex v will a) turn $N(v)$[2] into a clique, and b) remove v from the graph. This method will return an `EliminationInformation` object, which can be used to revert the elimination.
4. `deEliminateVertex(EliminationInformation info)`: Will revert the elimination of a vertex.

[2] $N(v)$ denotes the *neighbourhood* of v: all nodes connected to v.

5. `getSimplicialVertex(Set<T> forbidden)`: Get an arbitrary simplicial vertex, that is not contained in the forbidden set. A simplicial vertex is a vertex that has a clique as neighborhood. As required information are already gathered during the construction of the graph, this method runs in $O(|V|)$.
6. `getAlmostSimplicialVertex(Set<T> forbidden)`: As the last method, but will return an *almost* simplicial vertex, that is, a vertex that has a clique and one other vertex as neighborhood. This method actually computes the vertex and is more expensive than the last one.
7. `getFillInValue(T v)`: The fill-in value of an vertex v is the amount of edges that will be introduced to the graph, if v is eliminated. This method runs in $O(1)$ as well.

2.3 Reading and Writing Graphs

In the previous section we have created a graph "by hand", in real scenarios, however, we will often need to read the graph from standard input or from a file. The class `GraphFactory` provides a method to read `.gr` files (which is the graph format specified by PACE[3]. These methods are quite generic, and also work for `DIMACS` (graph) files, i. e., `.dgf` files. A typical usage would be:

```
Graph<Integer> myGraph = GraphFactory.graphFromStdin();
```

A graph, however, can not only be created from a stream. A common source for a graph is, well, another graph. The `GraphFactory` class provides methods to copy graphs:

```
Graph<Integer> myGraph = GraphFactory.copy(othergraph);
```

Another way to obtain a graph from a graph is by taking a subgraph, that is, a graph defined by a subset of the vertices. In *J*drasil, this can be achieved by the following code:

```
Set<Integer> subgraph = new HashSet<>();
// ... add vertices to subgraph ...
Graph<Integer> myGraph =
  GraphFactory.graphFromSubgraph(othergraph, subgraph);
```

Once *J*drasil did its job, we most likely want to print the graph or its tree decomposition to the standard output or a file. This can be achieved with the class `GraphWriter`, which provides many methods to print graphs. To simply write a graph, or a tree decomposition, to the standard output, we can use the following code:

```
// write a graph of any type
GraphWriter.writeGraph(myGraph);

// write a graph of any type, translate vertices to {1,...,|V|}
GraphWriter.writeValidGraph(myGraph);

// write tree decomposition
GraphWriter.writeTreeDecomposition(decomposition);
```

Additionally, *J*drasil also provides methods to write graphs and tree decomposition directly into Ti*k*Z code, which is very useful for debugging, especially combined with the *graph drawing* capabilities of Ti*k*Z:

[3] `https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`

```
// write a graph to TikZ
GraphWriter.writeTikz(myGraph);

// write tree decomposition to TikZ
GraphWriter.writeTreeDecompositionTikZ(decomposition);
```

2.4 Storing Graphs on the Bit-Level

In addition to the default `Graph` class, Jdrasil also offers the class `BitSetGraph` that stores a `Graph` (with arbitrary vertices) as bitwise adjacency matrix, i. e., as array of `BitSets` where the i'th `BitSet` corresponds to the i'th row of the adjacency matrix. This comes with all the advantages and disadvantages of an adjacency matrix.

The crucial advantage is that the graph is compact and that many operations can be performed quickly on the bit level. In particular, dynamic programming over subgraphs can be implemented efficiently, as subgraphs are also just BitSets that can efficiently be hashed. This class is essentially used by exponential time algorithms that work on prepossessed (and thus, hopefully small) graphs.

Chapter 3: Graph Invariants

A graph property is a class of graphs that is closed under isomorphisms. A graph invariant is a function that maps isomorphic graphs to the same value. Examples for graph invariants are "number of vertices", or "size of the minimum vertex-cover". An example of a graph property could be "contains a triangle". In this sense, we can model graph properties as invariants that map to boolean values.

Jdrasil implements graph invariants over the interface `Invariant`. In order to do so, it essentially provides the method `getValue` , which returns the computed invariant. Since many invariants can be represented by an additional model (for instance, a model for the vertex-cover model is the actual vertex-cover), the class also provides the method `getModel` , which returns a map from vertices to some values.

The usual usage of an invariant is as follows (here, we use vertex-cover):

```
// this will already compute the vertex cover
VertexCover vc = new VertexCover<T>(myGraph);

// the following methods then cost O(1)
vc.getValue(); // size of the vertex-cover
vc.getModel(); // the vertex-cover

// since some invariants are hard to compute, we may not
// obtain an optimal solution, we can check as follows
vc.isExact()
```

3.1 Connected Components

The class `ConnectedComponents` can be used to compute the connected components of the graph. The model maps vertices to integers, which represent the id of the connected component the vertex is. In addition, the class provides methods to obtain the connected components as sets of vertices and as subgraphs.

3.2 Vertex Cover

The class `VertexCover` computes a set of vertices that covers all edges. If an SAT solver is available, this class will compute a minimal vertex-cover. Otherwise, a 2-approximation is used.

3.3 Clique

The class `Clique` computes a set of vertices that are all pairwise adjacent. If an SAT solver is available, a maximal clique will be computed. Otherwise, a greedy strategy is used.

3.4 Twin Decomposition

The class `TwinDecomposition` computes a twin decomposition of the graph. Two vertices u and v are called twins if we have $N(u) = N(v)$. This relation defines an equivalence relation on the graph, which we call twin decomposition.

3.5 Minimal Separator

The class `MinimalSeparator` computes a minimal separator, i. e., a minimal set of vertices such that the removal of the set will increase the number of connected components of the graph.

3.6 Maximal Matching

A matching in a graph $G = (V, E)$ is a subset of the edges $M \subseteq E$ such that for every vertex $v \in V$ we have $|\{ e \mid e \in M \wedge v \in e \}| \leq 1$. A matching is *maximal* if we can not increase it by adding any edge. It is a *maximum* matching, if there is no bigger matching in the graph, and it is *perfect* if every vertex is matched.

In *J*drasil, the class `Matching` greedily computes a maximal matching, i. e., it is not guaranteed that it is a maximum matching. The matching is represented as map from vertices to vertices, i. e., a vertex is mapped to the vertex it is matched with.

3.7 Cut Vertices

A *cut vertex* or *articulation point* is a vertex whose removal disconnects the graph. If the graph is biconnected, no such vertex exists. The class `CutVertex` computes an arbitrary cut vertex using the algorithm from Hopcroft and Tarjan in time $O(n + m)$.

The class allows to forbid some vertices in the graph, this can be used to compute 2 or 3 connected components as follows: If the graph has not cut vertex, but the graph without some vertex $v \in V$ has a cut vertex c, then the pair v, c is a 2-cut.

3.8 Clique Minimal Separator

The class `CliqueMinimalSeparator` computes a clique minimal separator of the graph, that is, a set $S \subseteq V$ such that $G[S]$ is a clique, $G[V \setminus S]$ has more components then G, and such that S is a minimal separator for some vertices $a, b \in V$.

This class takes $O(nm)$ time and implements the algorithm described in "An Introduction to Clique Minimal Separator Decomposition" by Berry et al. In short, it does the following:

1. compute a minimal triangulation of the graph;
2. find minimal separators of the triangulation;
3. check which of these separators are cliques in the graph.

The class allows to specify a set of forbidden vertex, which are assumed to be not part of the graph then. In this way, the invariant can be used to compute almost clique minimal separators: take a vertex $v \in V$ and remove it, if the resulting graph has a clique minimal separator S, then $S \cup \{v\}$ is an almost clique minimal separator.

3.9 Minor-Safe Separator

A separator S is safe for tree width, if for each connected component C of $G[V \setminus S]$ the tree width of the graph $G[C \cup S]$ with S as clique is bounded by the tree width of G.

Bodlaender and Koster have shown that a separator S is safe in these terms if for each component C in $G[V \setminus S]$ the graph $R = G[V \setminus C]$ contains a clique on the vertices of S as labeled minor [3]. We call such a separator S *minor-safe*.

The class `MinorSafeSeparator` implements a heuristic to find a minor-safe separator. Whenever the heuristic outputs a separator, it will be guaranteed that it actually is a minor-safe separator, however, if the heuristic does not find such a separator there still could be one contained in G.

3.10 Minimal Vertex Separator

The class `MinimalVertexSeparator` implements an algorithm to compute a minimal vertex separator between two sets S_A, S_B in a given subgraph $G[W]$. The size of the separator can be bounded by a variable k, i.e., we will find the smallest separator less or equal to k or report that no such separator exists.

In detail, this class implements a bounded version of the Ford-Fulkerson algorithm running in time $O(k(n + m))$.

Chapter 4: Tree Decompositions

Since *J*drasil is a library for computing tree decompositions, a well thought through representation of such decompositions is required. Recall the formal definition of a tree decomposition: A *tree decomposition* of a graph G is a pair (T, ι) consists of a tree T and a mapping ι from nodes of T to subsets of vertices of G (called *bags*), such that:

- $\forall x \in V(G)$ there is a node $n \in V(T)$ with $x \in \iota(n)$;
- $\forall \{x, y\} \in E(G)$ there is a node $n \in V(T)$ with $\{x, y\} \subseteq \iota(n)$;
- $\forall x, y, z \in V(T)$ we have $\iota(y) \subseteq \iota(x) \cap \iota(z)$ whenever y lies on the unique path between x and z in T.

4.1 Design Principle

The formal definition of tree decompositions and the way *J*drasil represents graphs (remember that graphs can build up on any vertex class) gives rise to the following approach: a tree decomposition in *J*drasil is pretty much just a `Graph` with a special vertex class `Bag`. The `Bag` class on the other hand is actually just a collection of vertices.

To put it all together, the class `TreeDecomposition` represents a tree decomposition. It can store the original graph G, and a graph representing the tree T with `Bag` vertices. Furthermore, the class `TreeDecomposition` provides methods to create new nodes in the decomposition and to link existing nodes.

4.2 Building and Using Tree Decompositions

Let us assume we wish to implement a new algorithm for computing a tree decomposition. The following code snippets illustrate how we would create and store the decomposition in *J*drasil. Note that during the construction, the tree decomposition has not to be valid in any means.

Assume we read a graph with integer vertices from the standard input. We can create a tree decomposition for the graph as follows. At this point, the decomposition will be *empty* and *invalid*, i. e., it will only correspond to the graph, but will not have any information about the decomposition stored yet.

```
// read the graph G
Graph<Integer> G = GraphFactory.graphFromStdin();
// create empty tree decomposition for graph G
TreeDecomposition<Integer> td = new TreeDecomposition<>(G);

// check if the decomposition is valid
boolean isValid = td.isValid(); // returns false
```

To create a bag, we need a subset of the vertices of G that we wish to store in the bag. Finding these subsets is of course the (hard) task of the algorithm. Let us, in this example, just create the trivial tree decomposition of a single bag:

```
Set<Integer> someVertices = G.getVertices();
Bag<Integer> myBag = td.createBag(someVertices);

// decomposition now is valid
boolean isValid = td.isValid(); // returns true
```

The bag is created with the method `createBag`. This will do two things: it will create the `Bag` object and store it in the tree decomposition, and it will return a reference to this `Bag` object as well. This reference may be needed later on, for instance, if we wish to connect some bags with a tree edge:

```
// anotherBag is a reference to another bag that we have created
td.addTreeEdge(myBag, anotherBag);
```

And that is it. We have successfully created a tree decomposition. *J*drasil allows us to improve the quality of the decomposition (whenever we know that we do not have an optimal decomposition) with the separator technique of [4]. Just do the following and the decomposition may be improved:

```
td.improveDecomposition();
```

This method, however, may take some time if there are huge bags and should therefore be used on already good decompositions (and not on the trivial one as in this example).

Part III:
Algorithms

At its core engine, Jdrasil implements a bunch of algorithms to compute tree decompositions. These algorithms either directly compute such decompositions, or assist other algorithms in doing so. In particular, the implemented algorithms (that are related to the computation of tree decompositions) are partitioned into five types: *Preprocessing* algorithms, algorithms that compute *lowerbounds*, *heuristics/upperbounds*, *approximation* algorithms, and *exact* algorithms.

Chapter 5: Preprocessing

A Preprocessor is a function that maps an arbitrary input graph to an "easier" graph. Easier here is meant with respect to computing a tree decomposition and often just means "smaller", but could also reefer to adding structures to the graph that improve pruning potential.

The class `Preprocessor` models a preprocessor by providing the methods `preprocessGraph`, `addbackTreeDecomposition`, `computeTreeDecomposition`, `getTreeDecomposition`. The first method represents the actual preprocessing and computes a smaller graph from the input graph. The following two methods can be used to add a tree decomposition of the reduced graph produced by the first method back, and to use this tree decomposition to create one for the input graph. The last method is a getter for this decomposition.

The usual way to use a preprocessing algorithm is by a) initializing an instance of it, b) get the generated graph, and c) add back a tree decomposition for this graph. For instance, if we wish to apply standard reduction rules, we could do the following:

```
// a) generate instance of preprocessing algorithm
GraphReducer<T> reducer = new GraphReducer<>(G);

// b) get the generated graph
Graph<T> H = reducer.getProcessedGraph();

// c) add the decomposition of H
TreeDecomposition<T> td = ...
reducer.addbackTreeDecomposition(td);

// we can now access the final decomposition of the original graph
reducer.getTreeDecomposition();
```

5.1 Reduction Rules

There are many reduction rules for tree width known in the literature, see for instance [9]. A reduction rule thereby is a function that removes (in polynomial time) a vertex from the graph and creates a bag of the tree decomposition such that this bag can be glued to an optimal tree decomposition of the remaining graph – yielding an optimal tree decomposition. For graphs of tree width at most 3, these rules produce an optimal decomposition in polynomial time. For graphs with higher tree width, the rules can only be applied up to a certain point. From this point on, another algorithm has to be used.

In *J*drasil, the class `GraphReducer` implements several reduction rules and can be used as preprocessing for any other algorithm. As mentioned before, this class will (automatically) compute optimal tree decompositions of graphs with tree width at most 3. If this class fully reduces the input graph, it will generate an empty graph.

5.2 Contracting the Graph

It is a well known fact that tree width is closed under taking minors[1], i. e., if H is a minor of G, then $\text{tw}(H) \leq \text{tw}(G)$. Many algorithms that compute tree decompositions use this fact in some way.

The `GraphContractor` class computes a matching of the input graph, and contracts it. The result is a minor (which is of course much smaller) that is returned. Given a tree decomposition of this minor, a tree decomposition for the original graph is generated by decontracting the edges within the bags of the decomposition. The result is a valid (but not optimal) tree decomposition of the input graph. Furthermore, if the decomposition of the minor has width k, the width of the final decomposition is at most $2k + 1$.

5.3 Computing Improved Graphs

The k-{neighbor, path}-improved graph of G is the graph that is obtained by repeatedly adding edges between non-adjacent vertex u, v that have at least k {common neighbors, vertex disjoint paths between them}. Note that this operation is not necessarily safe. We have, however, $\text{tw}(G) \leq k \Leftrightarrow \text{tw}(H) \leq k$ where H is the $(k + 1)$-improved graph of G. [5].

The improved graphs are useful, as adding edges improves the quality of lower bound algorithms and may also boost the pruning potential of exact algorithms. In *J*drasil they can be computed by the classes `NeighborImprovedGraph` and `PathImprovedGraph`, respectively.

[1] A *minor* of G can be formed by deleting edges and vertices and contracting edges.

Chapter 6: Splitting up the Graph

When we compute the tree width of a graph, it is often not necessary to compute the tree width of the whole graph at once. In stead, it is often possible to *split* the graph into some components that can be handled separately. For instance, each connected component can be decomposed for its one.

A separator $S \subseteq V$ of a graph $G = (V, E)$ is a subset of the vertices such that $G \setminus S$ has more connected components than G. In particular, the connected components of G are separated by the separator $\emptyset$, while biconnected components have a separator S with $|S| = 1$.

Bodlaender and Koster have presented a list of *safe* separators for tree width [3]. A safe separator is one that does not effect the tree width, i. e., one that allows to reproduce an optimal tree decomposition for G from optimal tree decompositions of the connected components of $G \setminus S$ (in polynomial time). Having such safe separators allows us to compute tree decomposition in an divide-and-conquer manner: Split the graph using safe separators until the graph is small enough to solve it, or until there are no separators left. We call these unsplittable graphs *atoms*.

In Jdrasil, the class `GraphSplitter` implements some safe separators. The class will split the graph using such separators, and will keep track of potential glue points. Once the class is provided with tree decompositions for all components produced, it will generate a tree decomposition for the original graph.

In particular, `GraphSplitter` splits the graph using the following separators:

1. separators of size 0, i. e., it splits into connected components;
2. separators of size 1, i. e., it splits into biconnected components;
3. separators of size 2, i. e., it splits into triconnected components;
4. safe separators of size 3;
5. clique minimal separators;
6. almost clique minimal separators;
7. minor-safe separators found by a heuristic.

The `GraphSplitter` makes use of two Java concepts in order to make it easy to use and efficient.

Splitting the graph using recursive tasks. First of all, it implements its divide-and-conquer procedure using Javas `RecursiveTask` interface. This let Java handle the recursion efficiently and, if the "parallel" flag is set, leads to an direct and automatic parallelization (each atom can be handled in parallel).

Handling atoms using Lambda expressions. Splitting the graph into safe components is a generic task that may be used by many different tree decomposition algorithms (let it be exact, approximation, or heuristic algorithms). Hence, the `GraphSplitter` is designed to handle atoms with

an arbitrary function that it obtains as parameter. In particular, the `GraphSplitter` expects a functions as parameter that maps graphs to tree decompositions, and this function will be applied to all atoms. Afterwards, the created tree decompositions will be glued together. All of this will happen in parallel, if the "parallel" flag is set.

The usual way of using the `GraphSplitter` is straight forward:

```
// 1. create a splitter for the graph G
GraphSplitter<T> splitter = new GraphSplitter<T>(G, H -> {
  // 2. create a tree decomposition of the atom H
              TreeDecomposition<T> td = ...
              return td;
          },lb);

// 3. Obtain a tree decomposition of G
// this will invoke the splitting process
TreeDecomposition<T> final = splitter.call();
```

Note that, intuitive, one would first split the graph and then apply preprocessing to the atoms. However, splitting the graph is more expensive then preprocessing, so the advice is to apply preprocessing first and to split the result afterwards. However, it can also be useful to apply preprocessing to the atoms again.

Chapter 7: Lowerbounds

The `lowerbound` interface describes a class that models an algorithm for computing a lower bound of the tree width of a graph. By implementing this interface, a class has to implement the Callable interface, which computes a lower bound on the tree width a graph. In addition, a class implementing this interface has to implement the method `getCurrentSolution`, which can be used if the lower bound algorithm can already provide lower bounds during its execution. Jdrasil implements the following lowerbound algorithms.

7.1 Degeneracy of a Graph

We call a Graph $G = (V, E)$ d-degenerated if each subgraph H of G contains a vertex of maximal degree d. It is a well known fact that we have $d \leq tw(G)$ and, thus, we can use the degeneracy of a graph as lowerbound for the tree width.

The class `DegeneracyLowerbound` implements the linear time algorithm from Matula and Beck [13] to compute the degeneracy of a graph.

7.2 Lowerbounds based on Minors

It is a well known fact that for every minor H of G the following holds: $tw(H) \leq tw(G)$. To obtain a lowerbound on the tree width of G it is thus sufficient to find good lowerbounds for minors of G. The minor-min-width heuristic devoloped by Gogate and Dechter for the QuickBB algorithm [12] does exactly this. It computes a lowerbound for a minor of G and tries heuristically to find a good minor for this task.

In Jdrasil the class `MinorMinWidthLowerbound` implements this heuristic with different strategies discussed by Bodlaender and Koster [5].

7.3 Compute Lowerbound in Improved Graphs

The {k-neighbor, k-path}-improved graph H of a given graph G is obtained by adding an edge between all non adjacent vertices that have at least k {common neighbors, vertex disjoint paths}. A crucial lemma states that adding these edges will not increase the tree width. Hence, we can compute a lower bound on the tree width of G by computing lower bounds on increasing improved graphs of G. In this way, improved graphs can be used to improve the performance of any lower bound algorithm.

The class `ImprovedGraphLowerbound` implements the improved graph trick to improve some implemented lower bound algorithms. The implementation is based on [5].

Chapter 8: Heuristics

Since computing the exact tree width of a graph is NP-hard, there are many graphs for which we are not yet able to compute an optimal tree decomposition. However, it turns out that there are very powerful heuristics for this problem.

In *J*drasil, all heuristics implement the interface `TreeDecomposer` and produce tree decompositions of quality "Heuristic".

In addition to the different implemented algorithms, there is the class `Heuristic` which provides stand alone access to a combination of heuristics and is used by us for the PACE heuristic track. This class can be compiled and used with

```
./gradlew heuristic
./tw-heuristic
```

The program combines some of the implemented heuristics to get the best from each world. In particular, it does the following:

1. reduce the graph using `GraphReducer`;
2. computes a first decomposition using; `StochasticGreedyPermutationDecomposer`;
3. tries to improve the decomposition;
4. starts an infinite local search to further improve the decomposition using the algorithm implemented in `LocalSearchDecomposer`.

The program will try to improve the decomposition, until a `SIGTERM` is received.

8.1 Greedily Compute an Elimination Order

The class `GreedyPermutationDecomposer` implements greedy permutation heuristics to compute a tree decomposition. The heuristic eliminates the vertex v that minimizes some function $\gamma(v)$, while ties are broken randomly. See [4] for an overview of possible functions γ. The class implements six different value functions, which can be selected by the method `setToRun`. Experiments have shown that different functions are preferable on different graphs. It is, thus, worth to test different value function when we compute a tree decomposition of a graph.

To improve the quality of the found decomposition, we can do some sort of *look-ahead*: instead of taking the vertex that minimizes γ, we take the vertex such that the sum of the next k choices minimizes γ. A look-ahead for $k > 1$ can be set with `setLookAhead`. Already for $k = 2$ this improves the quality of the heuristic on many graphs. However, increasing k by one increases the running time by a factor of $|V|$ and, hence, the look-ahead should be used with care.

8.2 Greedily Compute an Elimination Order with Many Coins

The Greedy-Permutation heuristic performs very well and can be seen as *randomized algorithm* as it breaks ties randomly. Therefore, multiple runs of the algorithm produce different results and, hence, we can perform a stochastic search by using the heuristic multiple times and by reporting the best result. As the Greedy-Permutation heuristic implements different algorithms, we can pick different algorithms in different runs. As the performance of these algortihms differ, we choose them with different probabilities. In *J*drasil, this strategy is implemented by the class `StochasticGreedyPermutationDecomposer`.

8.3 Maximum Cardinality Search

The class `MaximumCardinalitySearchDecomposer` implements the Maximum-Cardinality Search heuristic. The heuristic orders the vertices of G from 1 to n in the following order: We first put a random vertex v at position $|V|$. Then we choose the vertex v' with the most neighbors that are already placed at position $|V| - 1$ and recurse this way. Ties are broken randomly.

8.4 Local Search

The class `LocalSearchDecomposer` implements a tabu search on the space of elimination orders developed by Clautiaux, Moukrim, Nègre, and Carlier [6].

The algorithm expects two error parameters r and s: the number of restarts and the number of steps. To find a tree decomposition the algorithm starts upon a given permutation, then it will try to move a single vertex to improve the decomposition and do this s times. If no vertex can be moved, a random vertex is moved and the process restarts. At most r restarts will be performed. At the end, the best found tree decomposition is returned.

8.5 Greedy Path Decompositions

A path decomposition can be seen as a special case of a tree decomposition that is sometimes easier to handle. In particular heuristics for path decompositions can be simpler and more flexible then the ones for tree decompositions. In *J*drasil one such heuristic is provided by the class `GreedyPathDecomposer`, which greedily computes a path decomposition by adding vertices to the current bag such that other vertices can be removed quickly from the bag.

Chapter 9: Approximation

An approximation algorithm as the one by Robertson and Seymour [14] can be interesting in two ways: it produces lower *and* upper bounds at the same time, while also provides a guarantee on its quality. This section lists the approximation algorithms implemented by Jdrasil.

In Jdrasil, all approximation algorithms implement the interface `TreeDecomposer` and produce tree decompositions of quality "Approximation".

In addition to the different implemented algorithms, there is the class `Approximation` which provides stand alone access to a combination of approximation algorithms.

```
./gradlew approximation
./tw-approximation
```

The program combines some of the implemented approximations to get the best from each world. In particular, it does the following:

1. split the graph into safe components using `GraphSplitter`
2. reduce the graph using `GraphReducer`;
3. computes a decomposition using `RobertsonSeymourDecomposer`.

9.1 Robertson and Seymour like Approximation

The class `RobertsonSeymourDecomposer` uses the standard FPT[1] approximation algorithm for tree width based on the work of Robertson and Seymour. The algorithm assumes that the graph is connected and computes in time $O(8^k k^2 \cdot n^2)$ a tree decomposition of width at most $4k + 4$.

A detailed explanation of the algorithms can be found in many text books about FPT, for instance [7, 10].

[1] FPT stands for *fixed-parameter tractable*.

Chapter 10: Exact

At the heart of Jdrasil is a collection of algorithms that compute optimal tree decompositions. All these algorithms implement the interface `TreeDecomposer` and produce tree decompositions of quality "Exact".

In addition to the different implemented algorithms, there is the class `Exact` which provides direct access to some decomposition algorithms. This class can be compiled and used with

```
./gradlew exact
./tw-exact
```

As different algorithms may be preferable on different graphs, the program selects between different algorithms, in particular it will:

1. split the graph into safe components using `GraphSplitter`;
2. reduce the graph using `GraphReducer`;
3. solve it with `CatchAndGlue`

10.1 Branch and Bound

The class `BranchAndBoundDecomposer` implements a classical branch and bound algorithm based on QuickBB [12] and its successors. The algorithm searches through the space of elimination orders and utilizes dynamic programming, reducing the search space to $O(2^n)$.

10.2 Cops and Robber

A classic result of Seymour and Thomas [15] provides a connection of the tree width of the graph and the cops-and-robber search game. Together with an algorithm by Berarducci and Intrigila [1] to evaluate such games in time $n^{O(tw(G))}$, this yields a way to compute exact tree decompositions. In Jdrasil the class `CopsAndRobber` implements this approach.

10.3 Cops and Robber (Bottom-Up)

Tree width has nice game theoretic characterisations, one of which is the node-search game played by a set of searchers and a fugitive. The variant of the game that corresponds to tree width is the one with a visible fugitive of unbounded speed (also known as the helicopter cops and robber game, defined by Robertson and Seymour).

In Jdrasil, the class `CatchAndGlue` computes a winning strategy for the cops (from which a tree decomposition can be deduced) in a bottom-up fashion. In order to do so, a queue of win-configurations is maintained which is pre-filled with trivial win-configurations, i. e., configurations in which the robber will be caught immediately. Then predecessor configurations (with respect to a winning strategy of the cops) are computed for the configurations in the queue, and if these configurations are win-configurations as well, they are added back to the queue. As larger win-configurations are "better" in the sense that we wish to find a win-configuration for the whole graph, a priority queue is used that orders configurations by size.

Computing predecessor configurations is straighted forward if the cops move does not disconnect the graph (i. e. a simple existential "fly"-move), however, it is difficult if a universal "split"-move is reconstructed (i.e., if the current configuration of the game originated from a cops move that has disconnected the graph and from a corresponding robber move whom has chosen a connected component). In this scenario multiple win-configurations (which we may or may not already have discovered) have to be glued together. These configurations correspond to branch nodes in the tree decomposition.

The asymmetrical running time of this algorithm is worse then the one of the `CopsAndRobber` class, as the "glue"-part may cost exponential time as well. However, in practice there are much less configurations considered over all if we compute the winning strategy backwards, as we can discard a lot of "lose"-configurations and sinks in the configuration-graph.

10.4 Limited Graph Search

The tree width of a graph can be characterized from a game theoretic point of view with the graph search game where a team of searchers tries to catch a fugitive, who has unlimited speed. Depending on the visibility of the fugitive, a winning strategy of the searchers corresponds to a path- or tree-decomposition of the graph. Fomin, Fraigniaud and Nisse described a mixed version of the game [11]. Here, the fugitive is invisible, but he searchers may reveal her from time to time. By fixing the number q of reveals that are performed by the searchers, the game covers path decompositions ($q = 0$) and tree decompositions ($q = \infty$), and, in particular, the area between these two extrema. In the cited paper such decompositions are called q-branched tree decompositions and they have, loosely speaking, the property that on each path from a root to the leafs there are at most q nodes with more then one children, i.e., at most q branching nodes.

In Jdrasil the class `LimitedGraphSearch` implements a dynamic program that computes a winning strategy of the searchers in the limited search game and, thus, computes a q-branched tree decomposition. The class will compute the smallest k such that there is a winning strategy for a fixed q, or alternatively the smallest k such that any q exists such that the searcher win with q reveals.

10.5 Dynamic Programming

The class `DynamicProgrammingDecomposer` implements exact exponential time (and exponential space) algorithms to compute a tree decomposition via dynamic programming. The algorithms are based on the work of Bodlaender, Fomin, Koster, Kratsch, and Thilikos [2].

10.6 Naive Brute Force

The tree width characterization via elimination order gives a very simple brute force algorithm: just check all $n!$ permutations. This is of course only feasible for very small graphs. The approach is implemented by the class `BruteForceEliminationOrderDecomposer`.

10.7 Using a SAT Solver

A very common (theoretical and practical) approach to solve intractable problems is to first represent them as *constraint satisfaction problems* (CSP) and then solve those problems via specialized solvers. The most widely used solvers are SAT solvers that work on Boolean formulas.

*J*drasil provides the classes `BaseEncoder` and `ImprovedEncoder` to encode the problem of finding an optimal tree decomposition into a logic formula. The class `SATDecomposer` constructs such formulas, solves them using a SAT solver, and extracts the corresponding tree decomposition.

The class `SATDecomposer` only works if a SAT solver is installed as upgrade, see Section 13 for details.

Chapter 11: Postprocessing

A Postprocessor is a function that maps a given arbitrary tree decomposition to another tree decomposition of the same graph. For instance a Postprocessor may ensure that the decomposition has certain properties (i. g., is a *nice* tree decomposition), but a Postprecessor may also *improve* an non-optimal decomposition.

`protected final TreeDecomposition<T> treeDecomposition`

The tree decomposition that shall be processed, i. e., the input.

`protected TreeDecomposition<T> processedTreeDecomposition`

The processed tree decomposition, i. e., the tree decomposition *after* the postprocessing. This may be a pointer to the same object as `treeDecomposition` (if the postprocessing was performed inplace), or may be a new object.

`public Postprocessor(TreeDecomposition<T> treeDecomposition)`

The constructor just initialize some internal data structures and stores the tree decomposition that should be postprocessed.

Parameter treeDecomposition: The tree decomposition to be postprocessed.

`protected abstract TreeDecomposition<T> postprocessTreeDecomposition()`

This method computes the actual postprocessing. It shall use `treeDecomposition` and shall return a postprocessed version of it. This may the same reference if the postprocessing is performed inplace.

Return Value: A postprocessed version of the given tree decomposition.

`public TreeDecomposition<T> getProcessedTreeDecomposition()`

Obtain the postprocessed tree decomposition. If this was not computed yet, this method will call `postprocessTreeDecomposition`.

Return Value: A postprocessed version of the given tree decomposition.

11.1 Improving Tree Decompositions

This postprocessor allows to improve the quality of the decomposition (whenever we know that we do not have an optimal decomposition) with the separator technique of "Treewidth computations I. Upper bounds" by Bodlaender and Koster.

`public ImproveTreeDecomposition(TreeDecomposition<T> treeDecomposition)`

The constructor just initialize some internal data structures and stores the tree-decomposition that should be postprocessed.

Parameter treeDecomposition: The tree-decomposition to be postprocessed.

`public void improveDecomposition()`

Improve the current decomposition by trying to find a suitable, improvable bag.

`public void improveBag(Bag<T> b)`

Improve a bag by computing a minimum separator for it and splitting it according to the separator. See "Treewidth computations I. Upper bounds" by Bodlaender and Koster.

Parameter b: the bag to improve

`public Graph<T> toGraph(Bag<T> b)`

Construct a graph from a bag. It consists of all vertices from the bag and all graph edges. In addition, it also contains u,v, if there is another bag that contains u and v.

Parameter b: the bag
Return Value: the graph constructed from the bag

`private boolean inAnotherBag(T u, T v, Bag<T> b)`

Computes whether another bag exists that contains u and v.

Parameter u: the first node u
Parameter v: the second node v
Parameter b: the original bag that contains u and v
Return Value: whether a bag b' != b exists that also contains u and v

11.2 Flatten Tree Decompositions

A tree-decomposition may contain useless bags. For instance, two adjacent bags with the same content, or an empty bag. This Postprocessor will *flatten* the given tree-decomposition by removing such useless bags.

`public FlattenTreeDecomposition(TreeDecomposition<T> treeDecomposition)`

The constructor just initialize some internal data structures and stores the tree-decomposition that should be postprocessed.

Parameter treeDecomposition: The tree-decomposition to be postprocessed.

`private void contractDuplicateBags()`

Contract adjacent bags with same content.

11.3 Nice Tree Decompositions

A *nice* tree decomposition is a *rooted* tree decomposition in which each bag has one of the following types:

1. A *leaf* bag has no children and stores and empty set.
2. An *introduce* bag has exactly one child bag and has the same content as its child up to one addional vertex (which was "introduced" at this bag).
3. A *forget* bag has exactly one child bag and contains all but one vertex of its child (it "forgets" this vertex).

4. A *join* bag has two children which both have the same content as the join bag.
5. An *edge* bag has one child with the same content as that child and "introduces" one edge in that bag. It is required that every edge is introduces exactly once.

It is well known that to every tree decomposition there can be found a nice tree decomposition of the same width. This class will perform this transformation for a given tree decomposition. The modifiction will be performed *inplace*, i. e., the given tree decomposition will be modified.

`public Map<T, Integer> treeIndex`

The tree-index is a mapping $\phi\colon V \to \{0, \ldots, \mathrm{tw}\}$ that maps every vertex of the graph to an index such that no two vertices appearing in a bag share the same index. Therefore, the tree-index can be used to index data structures when working on the tree decomposition.

`public enum BagType`

Every bag in a nice tree decomposition has a specific type (leaf, join, introduce, forget, edge).

`public Map<Bag<T>, BagType> bagType`

Stores for every bag which type it has.

`public Map<Bag<T>, T> specialVertex`

Introduce and forget bags also have a special vertex.

`public Map<Bag<T>, T> secondSpecialVertex`

Edge bags need two special vertices.

`private Bag<T> root`

The root bag of the nice tree decomposition.

`private boolean isVeryNice`

A very nice tree decomposition has edge bags as well.

`public NiceTreeDecomposition(TreeDecomposition<T> treeDecomposition)`

The constructor just initializes some internal data structures and stores the tree decomposition that should be postprocessed.

Parameter treeDecomposition: The tree decomposition to be postprocessed.

```
public NiceTreeDecomposition(TreeDecomposition<T> treeDecomposition,
  boolean veryNice)
```

The constructor just initializes some internal data structures and stores the tree decomposition that should be postprocessed.

Parameter treeDecomposition: The tree decomposition to be postprocessed.
Parameter veryNice: Decide whether edge bags should be computed or not.

`private Bag<T> findSuitableRoot()`

A nice tree decomposition is computed from a chosen root bag down to the leafs. The choose of this root bag may dramatically change the structure of the nice tree decomposition which may have an impact on the performance of algorithms that later work on the decomposition.

This method implements an heurictic that tries to find a "good" candidate for such an root.

Return Value: A suitable root bag from which on we can build a nice tree decomposition.

`private Bag<T> makeNice(Bag<T> suitableRoot)`

Make the tree decomposition nice starting at the given root. Note that the root bag may not be the given bag, but will lead by a path of forget-bags to it.

Parameter suitableRoot: A bag at which we shall root the tree decomposition.
Return Value: The actual root bag of the constructed nice tree decomposition.

`private void optimizeDecomposition()`

The structure of a nice tree decomposition is crucial for the performance of algorithms working on it. This method tries to optimize the structure by, for instance, rearranging join bags.

`private void classifyBags()`

In a *nice* tree decomposition, the bags are partitioned into {join, introduce, forget, leafs} bags. This function computes, given a nice tree decomposition, the type of each bag.

`private void computeTreeIndex()`

Given a bag of a tree decomposition, we may which to index data using the elements of this bag. For instance, we which to to know which vertex is "the first vertex in the bag", since we do not want to allocate space in the range of $O(n)$ per bag.

This function computes a so called *tree-index*, which is a mapping from the vertices of the graph to $\{0, \dots, k\}$, where k is the width of the given tree decomposition. It guarantees that in no bag two vertices have the same tree-index, i. e., it can be used to access data.

The tree-index can be computed by the following simple algorithm (which also shows that something like the tree-index exists at all): Start at the root with a pool of available indices ($\{0, \dots, k\}$). Whenever you encounter a forget bag (say for v), pop a index from the queue and assign it to vertex v. In every child-branch, if you encounter an introduce bag for this vertex, add the index back to the queue. Observe that there is only one forget bag for every vertex and, therefore, every vertex optains just one index. Since there can not be a chain of more then k forget vertices, we also have always an index in the queue.

`private void computeEdgeBags()`

Compute all edge bags for the decomposition. This will change the tree decomposition and the labeling of bags function. However, it will not change the tree-index.

`public Bag<T> getRoot()`

Returns the root bag of the nice tree decomposition.

Return Value: The unique root bag.

Chapter 12: Dynamic Programming on Tree Decompositions

This class implements a *general framework* for executing dynamic programs over (nice) tree decompositions. Starting with a provided tree decomposition or with a decomposition computed by `SmartDecomposer`, this class will optimize the tree decomposition and transform it into a nice one. Then it will traverse the decomposition in post-order and simulate a stack machine while doing so. On leaf bags, the stack machine will push new `StateConfigurations` using a provided `StateVectorFactory`. For all other bag types, the stack machine will only use the top of the stack and the methods of `StateVector`.

Note that the *actual* program that will be executed is encoded in the `StateVector` class, which is provided to the stack machine in form the the given `StateVectorFactory`. All operations performed by this class universally apply to all dynamic programs over tree decompositions.

`private StateVectorFactory<T> leafFactory`

The leaf factory is used to create new (usually empty) state vectors for leaf bags.

`private boolean worksOnVeryNiceTreeDecomposition`

Indicates if the program runs on a *very* nice tree decomposition, that is, a nice tree decomposition that also uses edge bags.

`private Graph<T> graph`

The graph on which we actually work.

`private TreeDecomposition<T> treeDecomposition`

The tree decomposition on which the dynamic program runs.

`private NiceTreeDecomposition<T> niceTreeDecomposition`

The nice (or very nice) tree decomposition.

`private int tw`

The tree width of the nice (or very nice) tree decomposition.

`private Stack<StateVector<T>> stateVectorStack`

While the tree decomposition is processed in a post-order traversal, we store the results of the individual bags on a stack, which simulates the "bubble-up" of the dynamic program.

```
public DynamicProgrammingOnTreeDecomposition(Graph<T> graph,
  StateVectorFactory<T> leafFactory)
```

Initialize a new instance of a program that runs over a tree decomposition. The given leaf factory is used to produce state vectors of the dynamic program for leafs. Note that this implies all other state vectors and, thus, is essentially the program code.

The program will be executed on a nice tree decomposition (which does not have edge bags).

Parameter graph: The graph for which the program shall be executed.
Parameter leafFactory: A StateVectorFactory that generates (usually empty) state vectors for leafs.

```
public DynamicProgrammingOnTreeDecomposition(Graph<T> graph,
  StateVectorFactory<T> leafFactory, boolean veryNice)
```

Initialize a new instance of a program that runs over a tree decomposition. The given leaf factory is used to produce state vectors of the dynamic program for leafs. Note that this implies all other state vectors and, thus, is essentially the program code.

The program may be executed on a *very* nice tree decomposition (which does have edge bags).

Parameter graph: The graph for which the program shall be executed.
Parameter leafFactory: A StateVectorFactory that generates (usually empty) state vectors for leafs.
Parameter veryNice: Indicates if the program should be executed on a *very* nice tree decomposition.

```
public DynamicProgrammingOnTreeDecomposition(Graph<T> graph,
  StateVectorFactory<T> leafFactory, boolean veryNice,
  TreeDecomposition<T> treeDecomposition)
```

Initialize a new instance of a program that runs over a tree decomposition. The given leaf factory is used to produce state vectors of the dynamic program for leafs. Note that this implies all other state vectors and, thus, is essentially the program code.

The program may be executed on a *very* nice tree decomposition (which does have edge bags).

The given tree decomposition will be transformed to an optimized nice (or very nice) tree decomposition on which the program then is executed. The given tree decomposition may be null, in which case a new one will be computed.

Parameter graph: The graph for which the program shall be executed.
Parameter leafFactory: A StateVectorFactory that generates (usually empty) state vectors for leafs.
Parameter veryNice: Indicates if the program should be executed on a *very* nice tree decomposition.
Parameter treeDecomposition: A given tree decomposition on which the program shall run.

```
private void initialize()
```

Initialize the dynamic program. This will

1. Compute a tree decomposition if no one is given;
2. Make the decomposition nice or very nice;
3. Heuristically optimize the decomposition.

```
public StateVector<T> run()
```

Runs the dynamic program and "bubbles-up" state vectors from the leafs up towards the root. Once all state vectors are computed, the state vector of the root is returned.

Return Value: The state vector computed for the root.

```
private void handleBag(Bag<T> bag)
```

Handle a bag of the nice tree decomposition. This functions assumes that the children of the bag where already handled and stored on the state vector stack.

Parameter bag: The bag to be processed.

12.1 State Vectors

A state vector describes the state of a dynamic program working on a tree decomposition at a specific bag. For instance, in a program that checks if the graph is 3-colorable, the state vector of a bag could be the set of all possible 3-colorings of that bag that are consistent to colorings of the subgraph of the subtree rooted at the bag.

In *J*drasil the whole dynamic program that works on a tree decomposition is defined over the `StateVector` interface. In more detail, the dynamic program will traverse the tree decomposition in a bottom-up fashion and compute `StateVector` objects for each bag, using the `StateVector` objects of the children of the bag. For doing so, it will, depending on the type of the bag, call a method of the `StateVector` object of the child (for instance to indicate that a vertex is *introduced* in the next bag) that then produces the `StateVector` for its parent. How the `StateVector` for the parent is produced is highly problem specific and can be seen as the *actual program*.

We shall note that all methods of this interface do not have to produce a new `StateVector` object, but may also just modify itself and return a reference to itself. With this pattern, the dynamic program uses less space while traversing the tree decomposition.

All methods of the interface are always called with the parent bag, eventually some special vertices (for instance the introduced vertex), and a so called *tree-index*. The later is a mapping $\phi\colon V \to \{0, \ldots, \mathrm{tw}\}$ that maps an integer to every vertex such that no two vertices that appear in a common bag share an index. With other words, the tree-index can be used, for instance, to index an array that stores information for the vertices in a bag.

```
public StateVector<T> introduce(Bag<T> bag, T v,
  Map<T, Integer> treeIndex)
```

This method is called whenever the parent of the bag that corresponds to this `StateVector` is an *introduce bag*, which means it has the same content and exactly one additional vertex.

Parameter bag: The parent bag.
Parameter v: The introduced vertex.
Parameter treeIndex: A reference to the tree-index.
Return Value: The `StateVector` for the parent.

```
public StateVector<T> forget(Bag<T> bag, T v, Map<T, Integer> treeIndex)
```

This method is called whenever the parent of the bag that corresponds to this `StateVector` is an *forget bag*, which means it has the same content and exactly one vertex less.

Parameter bag: The parent bag.
Parameter v: The forgotten vertex.
Parameter treeIndex: A reference to the tree-index.
Return Value: The `StateVector` for the parent.

```
public StateVector<T> join(Bag<T> bag, StateVector<T> o,
  Map<T, Integer> treeIndex)
```

This method is called whenever the parent of the bag that corresponds to this `StateVector` is an *join bag*, which means it has to children with the same content.

Parameter bag: The parent bag.
Parameter o: The `StateVector` of the other child.
Parameter treeIndex: A reference to the tree-index.
Return Value: The `StateVector` for the parent.

```
public StateVector<T> edge(Bag<T> bag, T v, T w,
  Map<T, Integer> treeIndex)
```

This method is called whenever the parent of the bag that corresponds to this `StateVector` is an *edge bag*, which means it has the same content and is labeled with an edge.

Parameter bag: The parent bag.
Parameter v: The first vertex of the introduced edge.
Parameter w: The second vertex of the introduced edge.
Parameter treeIndex: A reference to the tree-index.
Return Value: The `StateVector` for the parent.

```
public boolean shouldReduce(Bag<T> bag, Map<T, Integer> treeIndex)
```

A state vector can eventually be further reduced (for instance, if it contains equivalent states). A typical example is the rank-based reduction for steiner tree. However, such a reduction is eventually expensive and we do not want to compute it at every node of the tree decomposition.

This method shall (quickly) test if it is a good idea to try to reduce the state vector. If it returns true, the method `reduce` is executed. Otherwise nothing happens.

A state vector that does not support reduction may always return false and may not implement `reduce`.

Parameter bag: The current bag.
Parameter treeIndex: A reference to the tree-index.
Return Value: True if the state vector should be reduced.

```
public void reduce(Bag<T> bag, Map<T, Integer> treeIndex)
```

A state vector may contain equivalent states and can be reduced by removing some of these. This method test if such states exists and, if so, filters them out.

This function may be expensive, as it is only called if `shouldReduce` returns true.

Parameter bag: The current bag.
Parameter treeIndex: A reference to the tree-index.

12.2 State Vector Factory

The `StateVectorFactory` is used to produce `StateVector` objects for *leafs* of the tree decomposition. From there on, other `StateVector` objects are created directly by calling methods of the `StateVector` interface.

Use an initialization of this interface to provide a dynamic program (i. e., a implementation of `StateVector`) to `DynamicProgrammingOnTreeDecomposition`.

```
public StateVector<T> createStateVectorForLeaf(int tw)
```

Generate a `StateVector` object for a leaf of the tree-decomposition. These are usually empty, but may store the tree width of the graph or initialize some data structures.

Parameter tw: The tree width of the decomposition on which the dynamic program is executed.
Return Value: A `StateVector` object.

12.3 Example: Graph Coloring

To illustrate the usage of dynamic programming over tree decompositions in *J*drasil, we will turn to the most simple example: graph coloring. More precisely, we wish to implement an algorithm that determines if a given graph can be vertex-colored with 3 colors such that no two adjacent vertices get the same color. To get started, we have to implement a class that implements the `StateVector` interface. This class should represent the states of the dynamic program at a single node of the tree decomposition. A state in the graph coloring problem is a specific coloring of the vertices of the bag, so `ColoringStateVector` should represent a collection of possible colorings of the vertices in the bag represented by the node.

To represent a single coloring, we will introduce a simple wrapper class `State`, which essentially manages an array that assigns colors to vertices. Observe that the array has only size bounded by the tree width, as we may use the tree-index to index it.

```
class State {
  int[] colors;

  public State(int tw) {
    this.colors = new int[tw+1];
  }

  public State(State o) {
    this.colors = Arrays.copyOf(o.colors, o.colors.length);
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    State that = (State) o;
    return Arrays.equals(colors, that.colors);
  }

  @Override
  public int hashCode() {
    return Arrays.hashCode(colors);
  }
}
```

Given the wrapper class, our `ColoringStateVector` essentially boils down to mange a set of states, i. e., `Set<State> states` . We now have to implement the behaviour for introducing or forgetting vertices, joining bags, and introducing edges. We handle *introduce vertex* first. This is quite easy, as we just have to assign the new vertex one of the three possible colors. This means that we have to create three copies of every old state and assign the new vertex a different color in each of these copies. Observe that we use the given tree-index to assign a color in the state. Also observe that we assign the colors 1, 2, and 3 – we use "color" 0 to indicate that a vertex with this index is not in the bag.

Note: For simplicity, we do not care about adjacent vertices getting the same color here, as we will use edge-bags for this purpose. Of course, we could also perform this test directly here and do not use edge-bags at all.

```
@Override
public StateVector<Integer> introduce(Bag<Integer> bag, Integer v,
    Map<Integer, Integer> treeIndex) {
  Set<State> newStates = new HashSet<>();
  for (State state : states) {
    for (int color = 1; color <= 3; color++) {
```

```
      State newState = new State(state);
      newState.colors[treeIndex.get(v)] = color;
      newStates.add(newState);
    }
  }
  this.states = newStates;
  return this;
}
```

The next thing on our checklist are forget-bags in which we have to *forget a vertex*. This essentially means that we set the entry of its tree-index to 0 (meaning that there is no vertex with this index in the bag). The crucial point is that this may result in multiple states being equivalent (for instance, assume we have a bag with four vertices and the states $(1,2,3,3)$ and $(1,2,3,4)$ – if we delete the last vertex both states are the same). In order to guarantee the usual run time of a dynamic program over tree decompositions, we are not allowed to carry duplicates around. Therefore, we have to detect all duplicates created by the forget bag and remove them from the state set. Fortunately for us, this property can automatically be enforced by the underling hash set.

```
@Override
public StateVector<Integer> forget(Bag<Integer> bag, Integer v,
    Map<Integer, Integer> treeIndex) {
  Set<State> newStates = new HashSet<>();
  for (State state : states) {
    state.colors[treeIndex.get(v)] = 0;
    newStates.add(state);
  }
  this.states = newStates;
  return this;
}
```

The third bag type we have to handle are join-bags, which connect two children with the same content. Recall that a state is a coloring of the current bag that is valid for the whole subtree rooted at this bag. Hence, valid states at a join-bag are exactly the once's that appear in both children. This means that all we have to do at a join-bag is to simply compute the intersection of the two state vectors.

```
@Override
public StateVector<Integer> join(Bag<Integer> bag,
StateVector<Integer> stateVector,
    Map<Integer, Integer> treeIndex) {
  ColoringStateVector o = (ColoringStateVector) stateVector;

  Set<State> newStates = new HashSet<>();
  if (this.states.size() < o.states.size()) {
    for (State state : this.states) {
      if (o.states.contains(state)) newStates.add(state);
    }
  } else {
    for (State state : o.states) {
      if (this.states.contains(state)) newStates.add(state);
    }
  }

  this.states = newStates;
  return this;
```

```
}
```

The last thing on our checklist are edge-bags, which *introduce an edge* to the bag. Recall that every edge is introduced exactly once. Since we do not check if the coloring is valid when we introduce a new vertex, we have to do this now, i. e., given the introduced edge we have to verify that the two connected vertices have different colors. States in which this property is not satisfied have to be removed.

```
@Override
public StateVector<Integer> edge(Bag<Integer> bag, Integer v, Integer w,
    Map<Integer, Integer> treeIndex) {
  Set<State> newStates = new HashSet<>();
  for (State state : states) {
    if (state.colors[treeIndex.get(v)] != treeIndex.colors[map.get(w)])
      newStates.add(state);
  }
  this.states = newStates;
  return this;
}
```

We are actually almost done. The class `ColoringStateVector` does everything it has to. The only thing missing is a way to tell *J*drasil to use this state vector class. This is done by providing a class that implements the `StateVectorFactory` interface and that is able to produce state vectors for the leafs. Usually these state vectors are just empty – this is true for the coloring problem. We can therefore use a quite simple implementation for `StateVectorFactory`.

Note: If we wish to compute the chromatic number of the graph (i. e., the smallest number of colors needed), a convenient way is to give the coloring state vector a parameter q of allowed colors and to provide different factories to the solver.

```
public class ColoringStateVectorFactory
    implements StateVectorFactory<Integer> {

  @Override
  public StateVector<Integer> createStateVectorForLeaf(int tw) {
    return new ColoringStateVector(tw);
  }

}
```

We are now finally ready to run the algorithm. Essentially, all we have to do is to provide the `DynamicProgrammingOnTreeDecomposition` class with the input graph and an instance of our factory. If we run the program the "magic" happens in the background, i. e., a tree decomposition is computed, optimized, and traversed. The `ColoringStateVector` class is internally used to transfer information from the leafs to the root and, finally, we obtain the state vector assigned to the root. If this vector is not empty, the graph is 3 colorable (as the valid colorings of the bag correspond to valid colorings of the whole subtree). On the other hand, if the vector is empty the graph can not be colored with 3 colors.

Note: If we would provide "false" to the solver it would not compute edge-bags. Our algorithm would then not work. However, if we would check the coloring when we introduce a vertex we would obtain a correct algorithm again.

```
Graph<Integer> G = GraphFactory.emptyGraph();
// fill G with custom content

DynamicProgrammingOnTreeDecomposition<Integer> solver
  = new DynamicProgrammingOnTreeDecomposition<Integer>(
    G, new ColoringStateVectorFactory(), true
  );
ColoringStateVector rootVector = (ColoringStateVector) solver.run();

if (rootVector.states.size() >= 0) {
  System.out.println("3 colorable");
```

```
} else {
  System.out.println("not 3 colorable");
}
```

Part IV:
Upgrades

Jdrasil is designed as a modular and platform independent library, which provides all the advantages discussed earlier, but also comes with a couple of problems. In particular, in order to compute a tree decomposition of a graph, Jdrasil internally solves many different combinatorial optimization problems. Some of these problems may be solved more efficiently on a specific target platform, rather than on Javas virtual machine. For instance, one may want to use present graphic cards for massive parallelization. On the other hand, for many of these problems there are excellent and optimized libraries available, which we want to use. For instance, Jdrasil will rather use existing SAT solvers to solve the boolean satisfiability problem, instead of implementing its own. The concept of *upgrades* is Jdrasil's way to use external code or libraries.

We use the term "upgrade", in contrast to something like "library", as Jdrasil will always be fully functional and platform independent without any upgrade. In particular, it can be compiled and shipped without any upgrade. On the other hand, an upgrade will, as the name suggests, speed up Jdrasil on certain instances or platforms. In other words, an upgrade will not increase the functionality of Jdrasil, but will provide tools for Jdrasil such that it can execute its functionality faster.

The default location of upgrades for Jdrasil is the folder `build/upgrades/` . Since Jdrasil comes without any upgrade, this folder, at default, does not contain much. However, the gradle file of Jdrasil provides some targets to obtain upgrades.

> **Licence Warning**
>
> While Jdrasil stands under the open MIT licence, the third party software installed through upgrades may not. Whenever we install an upgrade, we may have to restrict the licence. Please read the licence of used third party software before you install an upgrade.

Chapter 13: Boolean Satisfiability

The boolean satisfiablity problem SAT is the most canonical NP-complete problem, and "simply" asks if a boolean formula in CNF has a satisfying model. Many NP-complete problems can naturally be stated as a SAT-problem, and hence, can be naturally solved by finding a model for a CNF formula. This is the reason why *SAT solvers*, i. e., tools that solve the boolean satisfiablity problem, have received a lot of research effort. In particular, there are annual challenges[1] that try to find the fastest solver. The result of this effort is that modern SAT solver can solve hard problems on many instances very quickly.

The power of SAT solvers makes it interesting to use them while computing a tree decomposition. Equipped with a SAT solver, *J*drasil can directly encode the problem of finding a tree decomposition (or more precisely, an elimination order) into a SAT-formula. On the other hand, *J*drasil can also use the SAT solver to solve different subproblems while computing the tree decomposition. For instance, if *J*drasil computes an elimination order, it can always put a clique of the graph at the end of the permutation. If the clique is large, this can reduce the search space dramatically. However, finding large cliques in a graph is NP-hard as well and *J*drasil will thus use a SAT solver to find the largest clique in the graph.

13.1 The Formula Class

The main interface of *J*drasil to use boolean logic is the class `Formula`, which represents a boolean formula in CNF. This class is always available and can always be used to create and manage logic formulas.

13.1.1 Specifying a Formula

A formula ϕ is always represented in CNF and in the classic DIMACS format, that is, variables are positive integers $x \in \mathbb{N}$, and negated variables are simply stored as $-x$. We can specify a formula by adding clauses to it, for instances $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge x_3$ can be created as follows:

```
Formula phi = new Formula();
phi.addClause(1, -2, -3);
phi.addClause(2, -3);
phi.addClause(3);
```

We can also "concatenate" two formulas by combining them with a logic "and", i. e., we can compute $\phi \wedge \psi$:

```
Formula psi = new Formula();
psi.addClause(-1);

phi.and(psi);
```

[1] `http://baldur.iti.kit.edu/sat-competition-2016/`

We can always add clauses to an existing formula or concatenate it with another formula. With other words, we can always further restrict the solution space of a formula. Sometimes, however, we may wish to remove a clause, which can be done by:

```
psi.removeClause(-1);
```

But this operation should be used with caution: first of all it is much more expensive to remove a clause than adding one; and, furthermore, we are not always allowed to remove a clause (the method can throw an exception). The reason for this is that SAT solvers that solve a formula incrementally often only allow to restrict the formula further. This means, once we started to "solve" the formula, we can not remove clauses anymore.

13.1.2 Solving a Formula

Note: We can only solve formulas if a SAT solver is installed as upgrade.

So, how to actually find a model for the formula, i. e., how to "solve" it. In order to check if a formula has a satisfying assignment, *J*drasil uses external SAT solvers which have to be installed as upgrade (see the following sections for possible solvers). If a SAT solver is installed, we can register it to the formula:

```
String sig = phi.registerSATSolver();
```

This method will register an arbitrary SAT solver that *J*drasil has found as upgrade. If no SAT solver is installed, this method will throw an exception; otherwise the signature of the solver is returned. Once a solver is registered, the following will happen:

1. The formula "as is" will be transfered to the solver, i. e., all clauses stored will be send to the solver.
2. The formula and the solver will be kept in sync, that is, clauses added to the formula will directly be added to the solver.
3. The method `removeClause()` can not be called anymore.
4. The method `isSatisfiable()` can now be called.

Once a solver was registered to the formula, we can check if there is a satisfying assignment:

```
phi.isSatisfiable();
```

This method will use the SAT solver to solve the formula. The whole API is incremental, so we can modify the formula between calls of this method (which will be faster than recreating new formulas). A typical scenario would look like:

```
while (phi.isSatisfiable()) {
  phi.addClause(...);
  ...
}
```

Sometimes, we actually would like to remove clauses between calls (which we are not allowed to do, as mentioned earlier). To overcome this issue, most incremental SAT solvers support the concept of *assumptions*. An assumption is an unit clause that is added to the solver for a single run. We can for instances say $x_1 = \text{true}$ and check if the formula is satisfiable *under this assumption*. After a call of `isSatisfiable()` , all assumptions are removed. To check if a formula is satisfiable under a set of assumptions, simply add them to the method call:

```
phi.isSatisfiable(1, -3);
```

Once we have defined a formula and solved it using `isSatisfiable()`, we are most likely interested in an actual satisfying model. A model is a mapping from the variables to boolean values, i. e., a `Map<Integer, Boolean>` and can be obtained with the following call:

```
Map<Integer, Boolean> model = phi.getModel();
System.out.printf("Value of %d is %b\n", 1, model.get(1));
System.out.printf("Value of %d is %b\n", 2, model.get(2));
System.out.printf("Value of %d is %b\n", 3, model.get(3));
```

Note that we can only obtain a model after a call to `isSatisfiable()`, and only if this call has returned true. Otherwise the code from above will throw an exception.

13.1.3 Auxiliary Variables

When we model a problem as CNF formula, we often need a lot of additional variables, which do not directly model parts of the problem (as vertex is selected or not), but that model structural things of the formula (to allow us to write them in short CNF). These variables arise a lot and will be added by different methods to the formula. However, if we talk about the formula on a higher level, we actually do not want these variables. For instance, we do not want have variables in our model that we do not know.

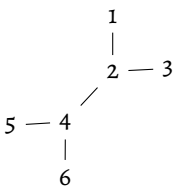
Jdrasil provides the concept of *auxiliary variables* to mark variables as helper variables, that are not directly connected to the modeled problem. The variable x_3 can be marked as auxiliary with the following command:

```
phi.markAuxiliary(3);
```

Once a variable x is marked as being auxiliary, the following will happen:

1. The variable list of the formula will not contain x.
2. A model will not contain an entry for x.
3. The auxiliary variable list will contain x.
4. The behavior of the formula, a registered SAT solver, and the satisfiability of the formula will *not* change. The variable is still part of the formula.

13.1.4 Cardinality Constraint



Many problems can naturally be encoded into an CNF – *when* we can restrict the number of variables that we are allowed to set to true. For instance, a *vertex cover* of a graph is a subset of its vertices, such that every edge is incident to one of these vertices. The graph at the border, for instance, has the vertex cover $\{2, 4\}$. For a given graph $G = (V, E)$, it is easy to write down a formula that states that the graph has a vertex cover:

$$\phi = \bigwedge_{\{u,v\} \in E} (x_u \vee x_v).$$

However, as simple this formula is, as uninteresting it is as well: every graph contains a vertex cover – just take all the vertices. To make the problem interesting (and difficult), we have to

restrict the number of vertices that we are allowed to set to true. This is exactly what a *cardinality constraint* does.

Jdrasil provides two ways to add cardinality constraints to a formula. In both cases, we first of all need so specify the set of variables (or literals) that we wish to restrict:

```
// build the formula
Formula phi = new Formula();
phi.addClause(1, -2, -3);
phi.addClause(2, -3);
phi.addClause(3);

// define a set that we wish to restrict
Set<Integer> vars = new HashSet<>();
vars.add(1);
vars.add(2);
vars.add(3);
```

Note that ϕ is satisfiable and has only one model, which sets all variables to true. So we get:

```
phi.isSatisfiable()  -----> true
```

We can now restrict the number of variables that are allowed to be set to true, for instance, to 2:

```
phi.addAtMost(2, vars);
```

We now obtain, as expected:

```
phi.isSatisfiable()  -----> false
```

In a similar manner, we can also enforce that a certain amount of variables *must be set*, but for our formula this has no effect:

```
phi.addAtLeast(2, vars);
```

For the interested reader: These methods use sequential counters and introduced $O(kn)$ auxiliary variables *per call*.

Both methods, `addAtMost` and `addAtLeast`, add clauses and auxiliary variables to the formula. This should be used for cardinality constraints that are used only once, since these methods will add these clauses for every call again (even if the set of variables does not change).

However, when we solve an optimization problem, we often wish to add a cardinality constraint for the same set of variables again and again. For instances, a typical routine to solve vertex cover would look like:

```
Formula phi = ...       // as in the example
phi.registerSolver();

Set<Integer> vars = ... // all variables
int k = vars.size() - 1;

phi.addAtMost(k, vars);
while (phi.isSatisfiable()) {
  k = k - 1;
  phi.addAtMost(k, vars);
}

System.out.println(k+1);
```

In such an incremental setup, the methods from above are not optimal, since they would add the similar auxiliary clauses and variables over and over again. To overcome this, Jdrasil also provides *incremental cardinality constraints*. The following ensures that at least 3 and at most 6 variables of the set `vars` is set to true:

```
phi.addCardinalityConstraint(3,6,vars);
```

This method will add a lot of auxiliary variables and clauses (most of the time more than a single call for of `addAtMost`), however, it will reuse this. More precisely, while the first call adds a lot of structure to the formula, incremental calls will only add single clauses. So for the algorithm from above, this method is way more efficient.

For the interested reader: This method uses sorting networks and introduces $O(n \log^2 n)$ auxiliary variables overall.

Finally, *J*drasil provides a third possibility to use cardinality constraints. While computing tree decompositions, we often deal with instances of small tree width (as the usual use case is parameterized complexity). If k is much smaller than n, a sorting network is asymptotically not optimal in sense of introduced auxiliary variables. For such scenarios, the `Formula` class provides the method `addDecreasingAtMost`. This method can be seen as a compromise between `addAtMost` (which is simple, but static), and `addCardinalityConstraint` (which is complex, but can be decreased and increased between solver calls). In contrast, the method `addDecreasingAtMost` is as simple as the first method, but allows to be decreased between calls to the solver. It is, however, only efficient for small values of k; and only works for decreasing upper bounds (and not increasing lower bounds). Note that we can interstate this as a parameterized SAT encoding, where k is the parameter.

For the interested reader: This method uses a variant sequential counter as well. It introduces $O(kn)$ auxiliary variables overall.

13.2 SAT4J

The Java Library SAT4J[2] is the most advanced and complete SAT-library for the Java platform. Although it is not the fastest solver available, it is one of the most widespread solver, as it has a clean API and a good documentation.

*J*drasil implements core functionality of SAT4J completely over reflections. This is done in the *intern* class `SAT4JSolver`, which is a `ISATSolver`. If the SAT4J library is found in *J*drasil's classpath, this class will be used by `Formula` (see 13.1) to find a model. This is fully capsuled from the user, which only has to work with the formula class.

If SAT4J is available in the classpath, `(` canRegisterSATSolver()) of `Formula` will return true, if SAT4J is also used (this depends on *J*drasil and other loaded upgrades), the method `init()` will return the String "SAT4J".

13.2.1 Installation

To use SAT4J in *J*drasil, it is sufficient to download the core library[3]. We can perform this step automatically with:

```
./gradlew upgrade_sat4j
```

13.2.2 Usage

Just add the SAT4J core library to the classpath:

```
java -cp bin:build/upgrades/org.sat4j.core.jar jdrasil.Exact
```

Or use one of *J*drasil's start scripts, which automatically looks for upgrades in these locations:

[2] `http://www.sat4j.org`
[3] `http://forge.objectweb.org/project/showfiles.php?group_id=228`

```
./tw-exact
```

Of course, the SAT4J library can be stored at another location as well. Other than that, just work with the class `jdrasil.utilities.sat.Formula` as we would otherwise.

13.3 Native IPASIR Solver

Today's most advanced SAT solvers are mostly implemented in C/C++. *J*drasil can use such "native" solver with the help of Javas JNI-API[4]. To be as general as possible and, thus, to support as many SAT solvers as possible, *J*drasil implements the IPASIR[5] interface, which is the reversed acronym for "Re-entrant Incremental Satisfiability Application Program Interface". This interface was proposed and used in recent incremental SAT challenges.

A solver that implements the IPASIR interface just has to implement the following 9 functions:

```
const char* ipasir_signature();
void* ipasir_init();
void ipasir_release(void* solver);
void ipasir_add(void* solver, int lit_or_zero);
void ipasir_assume(void* solver, int lit);
int ipasir_solve(void* solver);
int ipasir_val(void* solver, int lit);
int ipasir_failed(void* solver, int lit);
void ipasir_set_terminate(void* solver,
                          void* state,
                          int (*terminate)(void* state));
```

More details about what these functions should do can be found on the website of recent SAT challenges. The interface is closely related to the API of modern solvers as Lingeling or PicoSAT, so that such solvers can easily be linked against IPASIR.

*J*drasil can use an IPASIR solver as upgrade using the class `NativeSATSolver`, which implements the interface `ISATSolver` with native methods. Note how this interface, which we also use for other solvers like SAT4J, is closely related to IPASIR.

Note: We can always create this file with `./gradlew cinterface`

The C interface of `NativeSATSolver` can be found in `jdrasil_sat_NativeSATSolver.h`, which is located in the corresponding subproject folder at `subprojects/upgrades/ipasir/`. The corresponding C/C++ implementation `jdrasil_sat_NativeSATSolver.cpp` implements these methods and maps them against `ipasir.h`. This implementation takes care of keeping *J*drasil and the actual solver in sync, allowing *J*drasil to use multiple "instances" of the solver, and allows *J*drasil to kill the solver.

13.3.1 Installation

To compile an IPASIR upgrade for *J*drasil, we have to compile the JNI implementation in the file `jdrasil_sat_NativeSATSolver.cpp` into a dynamic library, which either should be be called `libjdrasil_sat_NativeSATSolver.so` or, depending on you operating system, `libjdrasil_sat_NativeSATSolver.dylib`. In order to do so, we have to link against an exiting implementation of an IPASIR solver.

[4] `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`
[5] `http://baldur.iti.kit.edu/sat-race-2015/downloads/ipasir.h`

In `subprojects/upgrades/ipasir` there is a `Makefile` that compiles the C++-file under the assumption that there is a library `libipasirsolver.dylib` in the same folder. This library should implement the ipasir interface with an actual SAT solver.

Using Gradle: To install a custom IPASIR solver, place a IPASIR compatible implementation called `libipasirsolver.dylib` in `subprojects/upgrades/ipasir` and run

```
./gradlew upgrade_ipasir
```

in the root folder of Jdrasil.

Jdrasil is also shipped with two default IPASIR compatible SAT solvers: glucose[6] and lingeling[7]. To upgrade Jdrasil with these state of the art solvers, use the following commands:

```
./gradlew upgrade_glucose
./gradlew upgrade_lingeling
```

13.3.2 Usage

Once we have compiled Jdrasil's IPASIR-JNI interface against an IPASIR solver, i. e., once we have a library called `libjdrasil_sat_NativeSATSolver.dylib`, we can *upgrade* Jdrasil "on the fly". Jdrasil will look for the SAT solver in its library path (not to be confused with its class path), so the following call will allow Jdrasil to use the solver:

```
java –cp bin –Djava.library.path=build/upgrades jdrasil.Exact
```

Of course, the path can be set to any location, wherever the upgrade is stored. Note that this only sets the path to location at which Jdrasil searches the upgrade. If, however, the upgrade is compiled against other dynamic libraries, these libraries are searched in the default system depending way (and not in the above specified path).

Alternatively, we can use one of Jdrasil's premade runscripts:

```
./tw–exact
```

[6] `www.labri.fr/perso/lsimon/glucose/`
[7] `www.fmv.jku.at/lingeling/`

Bibliography

[1] Alessandro Berarducci and Benedetto Intrigila. On the Cop Number of a Graph. *Advances in Applied Mathematics*, 14(4):389 – 403, 1993. doi:10.1006/aama.1993.1019.

[2] Hans L. Bodlaender, Fedor V. Fomin, Arie M.C.A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On Exact Algorithms for Treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012. doi:10.1145/2390176.2390188.

[3] Hans L. Bodlaender and Arie M. C. A. Koster. Safe separators for treewidth. *Discrete Math.*, 306(3):337–350, February 2006. URL: http://dx.doi.org/10.1016/j.disc.2005.12.017, doi:10.1016/j.disc.2005.12.017.

[4] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth Computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010. URL: http://dx.doi.org/10.1016/j.ic.2009.03.008, doi:10.1016/j.ic.2009.03.008.

[5] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations II. Lower Bounds. *Information and Computation*, 209(7):1103 – 1119, 2011. doi:10.1016/j.ic.2011.04.003.

[6] François Clautiaux, Aziz Moukrim, Stéphane Nègre, and Jacques Carlier. Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO-Operations Research*, 38(1):13–26, 2004.

[7] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[8] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The First Parameterized Algorithms and Computational Experiments Challenge. In *Proc. IPEC*, volume 63 of *LIPIcs*, pages 30:1–30:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[9] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[10] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, Heidelberg, Germany, 2006.

[11] Fedor V. Fomin, Pierre Fraigniaud, and Nicolas Nisse. Nondeterministic graph searching: From pathwidth to treewidth. *Algorithmica*, 53(3):358–373, 2009. URL: http://dx.doi.org/10.1007/s00453-007-9041-6, doi:10.1007/s00453-007-9041-6.

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

[13] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983. URL: http://doi.acm.org/10.1145/2402.322385, doi:10.1145/2402.322385.

[14] Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory*, 63(1):65–110, 1995. doi:10.1007/978-3-540-24605-3_37.

[15] Paul D. Seymour and Robin Thomas. Graph Searching and a Min-Max Theorem for Tree-Width. *Journal of Combinatorial Theory, Series B*, 58(1):22 – 33, 1993. doi:10.1006/jctb.1993.1027.