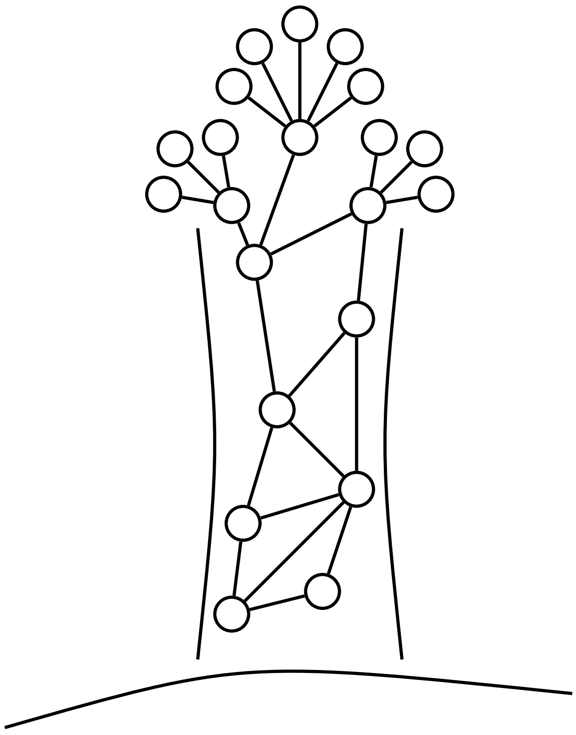


Jdrasil Manual

For Version: 0.1

Compiled: February 14, 2017

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 = {tba},
  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 GitHub at `https://github.com/maxbannach/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
```

With the same `.jar` file, *J*drasil can also be used as library: simply add the `.jar` to the classpath of the desired project.

1.2 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>
```

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

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.

1.2.1 Build the Executable and Library

To compile the core source code of Jdrasil we use the following command:

```
./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 Jdrasil, 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 Jdrasil as library, we can simply add the created `.jar` file to the classpath of our desired project.

1.3 Build the Documentation

The documentation of Jdrasil 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 Jdrasil. 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`.

1.4 Installing Upgrades

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

All upgrades have in common, that Jdrasil 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 Jdrasil 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
```

1.5 Building Startscripts (not only) for PACE

As Jdrasil 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 Jdrasil in exact mode (meaning it reads an graph from stdin, computes an optimal decomposition, and prints it on stdout); while the second script runs Jdrasil 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 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 Jdrasil with approximation algorithms:

```
./gradlew approximation
```

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

Part II: Graphs

Since Jdrasil 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 Jdrasil 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 Jdrasil'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 Jdrasil. 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 Jdrasil 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 Jdrasil'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 Jdrasil, 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.

$$\begin{tikzcd}
5 & & \\
& 4 \arrow[ul] \arrow[r] & 6 \arrow[d] & \\
& & 1 \arrow[ul] \arrow[d] \arrow[dr] & \\
& & 2 \arrow[r] & 3
\end{tikzcd}
\qquad
\begin{tikzcd}
E & & \\
& D \arrow[ul] \arrow[r] & F \arrow[d] & \\
& & A \arrow[ul] \arrow[d] \arrow[dr] & \\
& & B \arrow[r] & C
\end{tikzcd}$$

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

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

This is essentially the way Jdrasil 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, Jdrasil 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 $1 — 2$; to create the *directed* graph $1 \longrightarrow 2$, we can use the function `Graph.addDirectedEdge` :

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

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

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 Jdrasil 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);
```

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.

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.

```java
// 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 a collection of "easier" graphs. Easier here is meant with respect to computing a tree decomposition and often just means "smaller", but could also refer to adding structures to the graph that improve pruning potential.

The class `Preprocessor` models a preprocessor by providing the methods `computeGraphs`, `addbackTreeDecomposition`, `glueDecompositions`, and `getTreeDecomposition`. The first method represents the actual preprocessing and computes a collection of graphs from the input graph. The following two methods can be used to add a tree decomposition of one of the produced graphs back, and to combine these tree decompositions to 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) loop over the generated graphs, and c) add back a tree decomposition for every graph. For instance, if we wish to iterate over safe components (connected components, bi-connected components, etc.) we can do the following:

```
// a) generate instance of preprocessing algorithm
GraphSplitter<T> splitter = new GraphSplitter<>(graph);

// b) iterate over all generated graphs
for (Graph<T> component : splitter) {
  // c) compute decomposition of the component
  TreeDecomposition<T> decomposition = ...
  // and add it to the preprocessor object
  splitter.addbackTreeDecomposition(decomposition);
}

// we can now access the final decomposition of the original graph
splitter.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 set of graphs. So it is always save to simply loop over the generated graphs, without caring about this special case.

5.2 Splitting up the Graph

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 bi-connected 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.

In *J*drasil, 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.

5.3 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 $\mathrm{tw}(H) \leq \mathrm{tw}(G)$. 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 $2k + 1$.

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

Chapter 6: 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.

6.1 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 [12] to compute the degeneracy of a graph.

6.2 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 [11] 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].

6.3 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 7: 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 Jdrasil, 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.

7.1 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.

7.2 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 Jdrasil, this strategy is implemented by the class `StochasticGreedyPermutationDecomposer`.

7.3 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.

7.4 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.

Chapter 8: Approximation

An approximation algorithm as the one by Robertson and Seymour [13] 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`.

8.1 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 9: Exact

At the heart of Jdrasil is a collection of algorithms that compute optimal tree decompositions. As different algorithms may be preferable on different graphs, the class `ExactDecomposer` 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 the problem with `SATDecomposer` if a SAT solver is available;
4. solve with `CopsAndRobber`.

In Jdrasil, all exact 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 stand alone access to the `ExactDecomposer` class. This class can be compiled and used with

```
./gradlew exact
./tw—exact
```

9.1 Branch and Bound

The class `BranchAndBoundDecomposer` implements a classical branch and bound algorithm based on QuickBB [11] and its successors. The algorithm searches through the space of elimination orders and utilizes dynamic programming, reducing the search space to $O(2^n)$.

9.2 Cops and Robber

A classic result of Seymour and Thomas [14] 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.

9.3 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].

9.4 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`.

9.5 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 10 for details.

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 10: 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.

10.1 The Formula Class

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

10.1.1 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.

10.1.2 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.

10.1.3 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.

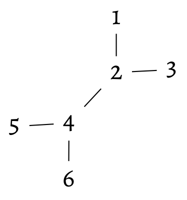
Jdrasil provides the concept of *auxiliary variables* to mark variables as helper variables, that are not directly connected to the modeled problem. The variable x_3 can be marked as auxiliary with the following command:

```
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.

10.1.4 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, Jdrasil 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.

10.2 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.

Jdrasil implements core functionality of SAT4J completely over reflections. This is done in the *intern* class `jdrasil.sat.SAT4JSolver`, which is a `jdrasil.sat.ISATSolver`. If the SAT4J library is found in Jdrasil's classpath, this class will be used by `jdrasil.sat.Formula` (see 10.1) 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 `jdrasil.sat.Formula` will return true, if SAT4J is also used (this depends on Jdrasil and other loaded upgrades), the method `init()` will return the String "SAT4J".

10.2.1 Installation

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

```
./gradlew upgrade_sat4j
```

10.2.2 Usage

Just add the SAT4J core library to the classpath:

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

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

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

```
./tw—exact
```

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

10.3 Native IPASIR Solver

Today's most advanced SAT solvers are mostly implemented in C/C++. Jdrasil 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, Jdrasil 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.

Jdrasil can use an IPASIR solver as upgrade using the class `jdrasil.sat.NativeSATSolver`, which implements the interface `jdrasil.sat.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 corresponding C interface of `jdrasil.sat.NativeSATSolver` can be found in the header file `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 Jdrasil and the actual solver in sync, allowing Jdrasil to use multiple "instances" of the solver, and allows Jdrasil to kill the solver.

10.3.1 Installation

To compile an IPASIR upgrade for Jdrasil, 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,

[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

`libjdrasil_sat_NativeSATSolver.dylib` . In order to do so, we have to link against an exiting implementation of an IPASIR solver.

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
```

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

[12] 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.

[13] 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.

[14] 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.