Technische Universität Dresden
Fakultät Informatik

Bachelor Thesis

Visualizing Dynamic Programming on Tree Decompositions

Author:
Martin Röbke
Matrikel-Nr. 3949819

Supervisor:
Dr. Johannes Fichte
Second Supervisor:
Prof. Dr. Stefan Gumhold

International Center For Computational Logic

July 27, 2020

Erklärung zur Urheberschaft

Hiermit versichere ich, dass diese Arbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweis oder als Leistung, die als Prüfungsvoraussetzung zu erbringen war, andernorts bereits eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften oder Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Martin Röbke, geb. 04.03.1995

01309 Dresden, Müller-Berset-Straße 29

Dresden, ...

(Unterschrift)

Abstract

Answering questions that can be expressed using graph theory is increasingly interesting in scientific work. Many problems like Boolean satisfiability or traffic related problems can be translated into connections and nodes. We think that the use of graph structures can help to further develop algorithms in different areas.

The algorithms we visualize in this thesis use dynamic programming on tree decompositions of graphs. We preprocess the input graph into a customized tree-decomposition of small tree-width. This gives us a description of the processing sequence for the algorithm, and allows with right hindsight for good parallelization while keeping the solving times competitive even on on larger instances.

Since the design of dynamic programming can be quite error-prone, a fast and uncomplicated visualization is needed for the solution process.

To help further refine and visualize the dynamic programming, we specified a JSON template for communication between solvers and the newly created visualization tool TDVisu.

As two reference implementations of dynamic programming on tree decompositions we selected the existing solvers GPUSAT and dpdb.

As a side note:
Of course the information we visualize is mostly in the solver and could be read out as text. But as the saying goes: "a picture is worth a 100 spreadsheets".

Contents

1. Introduction

1.1. Motivation

The goal of this thesis is to improve the progress in developing dynamic programming solutions on tree decomposition through visualization.

Graph-algorithms and problems regarding graph-related data are increasingly interesting in scientific work, as the applications of interconnected datasets grow. One interesting paper about graphs in (natural) language processing is [JGJ13]. Some use cases include fields of interest like cited in [Neo16]:

- Network and Database Infrastructure
- Recommendation Engines
- Artificial Intelligence and Analytics

As illustrations of the possibilities for an application, smaller examples from the problem types Boolean satisfiability **SAT**, model counting SAT **#SAT** and **minimal vertex cover** are presented. We also show an example of finding and visualizing a faulty tree-decomposition that occurred during development.

The target audience for this work includes:

- Developers of dynamic programming on tree decompositions for debugging,
- Researchers of such algorithms for comparisons and visualizations,
- Teachers and students looking for automatic visualization of their problem instances and the solving process.

1.2. Research Question

When developing techniques for dynamic programming for various problems or platforms there can be errors in actual implementations that will be hard to pin down.

There is some evidence and experience in e.g. [Die07; HJW14] that visualization of intermediate and final results can help to understand various issues - such as correctness and efficiency of the algorithms.

As there was no tailored tool for our use case yet, it should be created and integrated into existing tools with a minimum of additional effort.

So our main research questions we want to answer with this thesis are:

1. How could a prototype visualization and debugging tool for solvers of MSO logic problems using dynamic programming on tree decompositions look like?
2. How could an implementation of visualization in existing solvers look like?

To answer these questions we have implemented TDVisu, a visualization software based on Graphviz. Our proposed solution is inspired by previous manual visualizations like the two examples in Figure 1.

The Figure 2 shows a draft of the process creating the visualization with TDVisu.

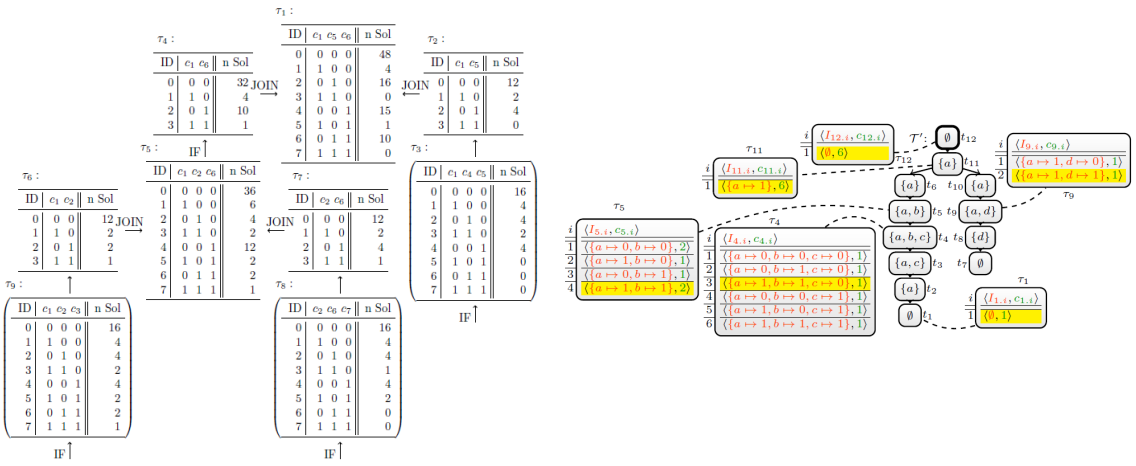


Figure 1: Two templates from published project documentations which show similar visualizations as TDVisu. The left graphic from [Zis18] Figure 4.3 on page 29 and the right graphic from [Fic+20] Figure 2, page 4.

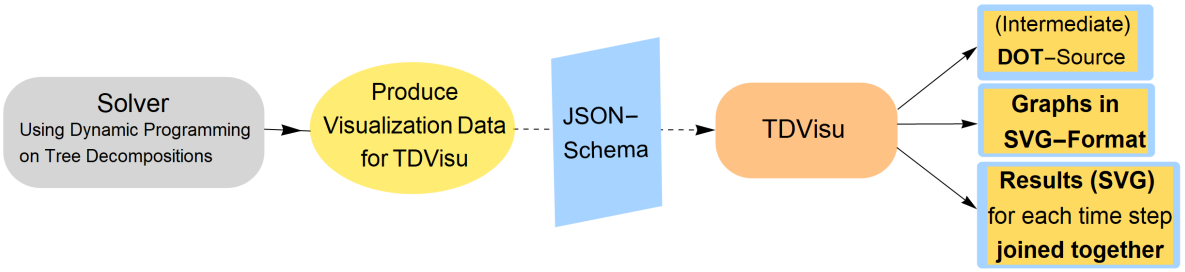


Figure 2: An overview of the intended use of the visualization. Starting with a solver for DP on TD, over the extraction of the desired data, labels and parameters to the result files.

1.3. Methodology

The algorithms whose runs we visualize basically work so that the input graph is used to find the best possible processing sequence adapted to the hardware. With this order it is known which parts of the problem can be solved separately, and which intermediate results have to be stored and for how long. To visualize these algorithms for different problem types, we need to be flexible enough with the data we process for visualization. To accomplish this, we almost everywhere expect simple text to be included in various places. Mostly the internal identifiers for nodes or variables in Boolean formulas are not processed based on strings, but on integers.

The graphs are assembled using the object-oriented style of the Python interface [1] to Graphviz.

[1] `https://pypi.org/project/graphviz/`

For our visualizations we chose Graphviz[2] as an open source graph visualization software, which offers customizable visualization for directed and undirected graphs.

The tree decompositions in every tested application were provided by the utility htd (small but efficient C++ library for computing (customized) tree and hypertree decompositions) [AMW17].

The source code for TDVisu is available under the GPL3 license
on `github.com/VaeterchenFrost/tdvisu`. Our tool is written in python and published on pypi.

We defined a JSON-format specification for portability and customization of the visualization in one file and two reference implementations in practical solvers.
It is explained in more detail in the implementation chapter, or directly in the *tdvisu/TDVisu.schema.json* file, a JSON-schema [3] with examples and an almost complete validation of the format.

The implementation currently does not support hyper-graphs and assumes that each node in the tree decomposition has either one or two children. The visualization output consists by default of scalable-vector-graphics (SVG), a very flexible text-based standard for describing images that can be easily compressed (and modified) without loss of quality [Web20].

1.4. Related Work

To the best of our knowledge no comparable tool exists that is capable of out-of-the-box visualization of dynamic programming on tree decompositions. In this chapter, however, we would like to mention works that deal with the surroundings of our problem.

First two papers dealing with additional solvers exploiting small tree-width and Courcelle's Theorem:

- "Implementing Courcelle's Theorem in a declarative framework for dynamic programming" in [BPW16]
  and
- "Evaluation of an MSO-solver" from [Lan+12]

Additional some references on algorithm visualization and visual debugging can be found in

- "ELVIZ: A query-based approach to model visualization" about an approach to visualization, generic regarding both the source model, and the kind and content of the visualization [HJW14],
- "Visualizing Tree Structures in Genetic Programming" about methods to visualize the structure of trees that occur in genetic programming [Dai+05],

[2] `https://graphviz.org/`
[3] `http://json-schema.org/draft-07/schema`

- The book “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software” - the first textbook on software visualization, Stephan Diehl, Springer 2007 [Die07]

- the overview on

Current advancements in dynamic programming on tree-decompositions itself can be found in [Fic+20] and [HTW20]. One of the latest papers for *gpusat2* discussed a series of experiments using several benchmark sets for model counting #SAT and weighted model counting. It compared the then latest versions of publicly available #SAT solvers, where *gpusat2* could solve the vast majority of the instances and ranked second place [FHZ19, Ch. 5].

1.5. Thesis Outline

The rest of the thesis is structured as follows:
Chapter 2 gives the reader the necessary background for this thesis. In Chapter 4, we describe our approach to the visualization and the functionalities of the software. Next, Chapter 5 describes the integration of *tdvisu* in the solver *gpusat* [Zis18]. In Chapter 6, we describe the integration with the database-based solver *dpdb* [Fic+20]. Chapter 7 shows, limited by the page size, small examples for the tested use cases of visualization, starting with a *SAT* example, then a *#SAT* problem visualized, followed by a *minimal-vertex-cover* example. After these three use cases we show the possibilities of joining single result images together. Finally the chapter closes with a subsection on how our tool can be used for successful debugging. In Chapter 8 of this thesis we give a summary of our work and hints for future progress.

2. Background

In this chapter we provide a brief background for this work.
We will start with a short introduction to logic and MSO in 2.2. Then we describe the Boolean satisfiability problem which in it's variants is a popular application for solvers. Then we introduce Courcelle's theorem, which describes important properties of the problems and related bounds for the solvers. Eventually, tree decompositions are described as a basis for dynamic programming in the next section.

2.1. Graphs

In this thesis we use the word **graph** for a pair $G = (V_G, E_G)$, where V_G is a nonempty set of vertices also called **nodes** and $edg_G = E_G$ is the binary relation $E_G \subseteq V_G \times V_G$, such that $(x, y) \in E_G$ if and only if there exists an edge from x to y if G is directed, and an edge between x and y if G is undirected. For further information on graphs we refer to the common literature, for example [Bro+15, p. 401–412] or more casually in [Car17].

Example 1. G is an undirected graph with

$$\begin{aligned} G &= (V_G, E_G) \\ V_G &= \{1, 2, 3, 4, 5, 6, 7\} \\ E_G &= \{(1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (2,3), (3,4), (4,5), (5,6), (6,7), (7,2)\} \end{aligned}$$

2.2. (Monadic Second-Order) Logic

Monadic Second-Order (MSO) logic is a logical language that is suitable for expressing numerous graph properties [CE12, p. 41]. In the following summary we would like to introduce propositional logic and the extensions first order logic and MSO. The concepts and notations from propositional logic are needed for the example problems around Boolean satisfiability; MSO will be used as the language that describes the possibilities of the dynamic programming on tree decompositions.

2.2.1. Propositional Logic

Propositional logic is a formal system $\mathcal{L} = \mathcal{L}(\mathrm{A}, \Omega, \mathrm{Z}, \mathrm{I})$ with

- the set A as a countably infinite set of elements called proposition symbols or propositional variables. These are also called terminal elements.
- The set Ω is a finite set of elements called operator symbols or logical connectives. It is partitioned into disjoint subsets as follows:

  $\Omega = \Omega_0 \cup \Omega_1 \cup \ldots \cup \Omega_j \cup \ldots \cup \Omega_m$..

  In this partition, Ω_j is the set of operator symbols of arity j.

  Ω is typically partitioned as follows:

$\Omega_0 = \{\bot, \top\}$.
$\Omega_1 = \{\neg\}$,
$\Omega_2 \subseteq \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

- The set Z is a finite set of transformation rules that are called inference rules when they receive logical applications.
- The set I is a countable set of initial points that are called axioms when they receive logical interpretations.

2.2.2. Boolean Satisfiability Problem

In this thesis we use the following symbols for expressions of Boolean algebra:

$\Omega_0 = \{0, 1\}$.
$\Omega_1 = \{\neg\}$,
$\Omega_2 = \{\wedge, \vee\}$.

A propositional logic formula, or Boolean expression, is then built from variables, the logical values Ω_0, the operators Ω_1 and Ω_2 as well as parentheses.

Example 2. B is a Boolean expression with

$B = (\text{v1} \vee \text{v4} \vee \text{v6}) \wedge (\neg\text{v5} \vee \text{v1}) \wedge (\neg\text{v1} \vee \text{v7}) \wedge (\text{v2} \vee \text{v3}) \wedge (\text{v2} \vee \neg\text{v5}) \wedge (\neg\text{v6} \vee \text{v2}) \wedge (\text{v3} \vee \neg\text{v8}) \wedge (\text{v4} \vee \neg\text{v8}) \wedge (\neg\text{v4} \vee \text{v6}) \wedge (\neg\text{v4} \vee \text{v7})$

To describe the usual notation used in solvers for Boolean satisfiability we also use the following definitions to describe the problem instances denoted in conjunctive normal form (CNF).

A literal is a Boolean variable v or its negation $\neg v$. A *clause* is a finite set of literals interpreted as their disjunction. A clause c is called *unit* if $|c| = 1$. A CNF $formula$ is a set of clauses and is interpreted as the conjunction of its clauses. We define $var(C)$ as the set of variables contained in the clause or clause set C. As $assignment\alpha$ maps variables in a formula to 0 or 1, $\alpha : var(C) \rightarrow 0, 1$. A clause is satisfied by an assignment if for some variable $v \in var(c)$ we have $v \in c \wedge \alpha(v) = 1$ or $\neg v \in c \wedge \alpha(v) = 0$. Otherwise the assignment falsifies the clause. An assignment satisfies a formula if each clause in the formula is satisfied by the assignment.

Example 3. For visualization samples we will use the formula from Ex. 2 with the following clause set: $C = \{c_1 = \{v_1, v_4, v_6\}, c_2 = \{v_1, \neg v_5\}, c_3 = \{\neg v_1, v_7\}, c_4 = \{v_2, v_3\}, c_5 = \{v_2, v_5\}, c_6 = \{v_2, \neg v_6\}, c_7 = \{v_3, \neg v_8\}, c_8 = \{v_4, \neg v_8\}, c_9 = \{\neg v_4, v_6\}, c_{10} = \{\neg v_4, v_7\}\}$

The formula C is for example satisfied by the assignment that maps all variables $var(C) \rightarrow 1$ to 1. All 22 satisfying assignments can be seen in Table 5 on page 65.

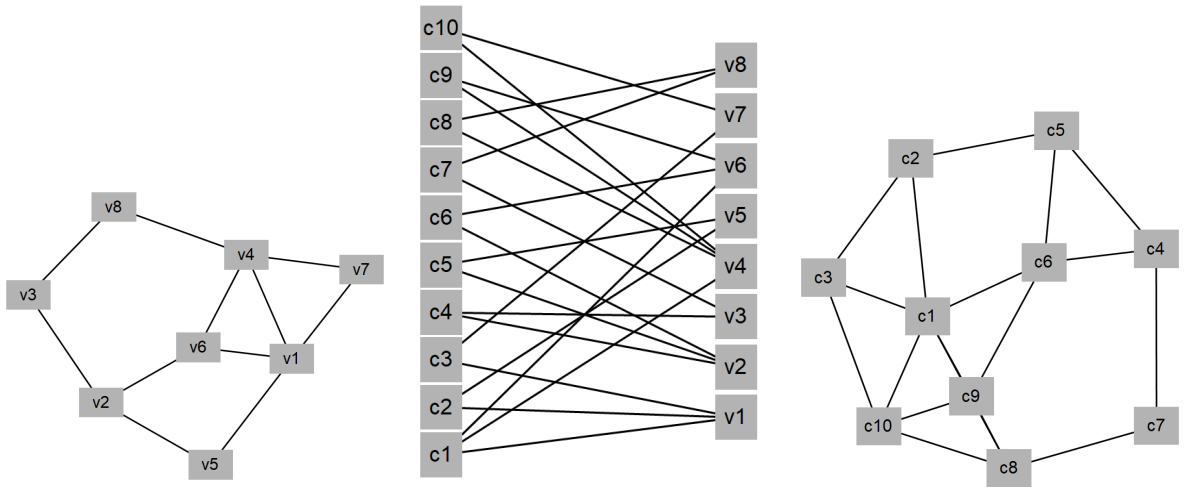


Figure 3: From left to right: primal, incidence and dual graph

The Boolean satisfiability problem **SAT** consists of the question: Given a propositional formula ϕ, is there an assignment of the variables in ϕ for which ϕ is satisfied? The **#SAT** problem consists of the question: Given a propositional formula π, for how many assignments of the variables in π is π satisfied?

To later create a tree decomposition for Boolean formulas, solvers use the structure of the clause set to compute at least one of the three following graphs. For more details see for example [Zis18, Chapter 2.1].

The *primal graph* of a SAT formula contains a vertex for each variable of the SAT formula, and only has an edge between two vertices if they both occur in one clause together.

The *incidence graph* of a SAT formula if bipartite and contains a node for each variable and clause in the SAT formula. It only has edges between a variable node and a clause node if the variable occurs in the clause.

The *dual graph* of a SAT formula contains a node for each clause in the SAT formula. This graph only has an edge between two vertices if the two clauses have at least one common variable.

Those three graphs for our Example 3 are shown in Fig. 3 .

2.2.3. From Propositional Logic to MSO Logic

First-order logic adds relations and quantifiers to propositional logic. For further information on logic see for example [Bro+15, Chapter 5].
The quantifier symbols are

- $\exists$ for the existential quantification and
- $\forall$ which expresses that a propositional function can be fulfilled by any member of a domain.

While first-order logic only quantifies variables that range over individuals (elements of the domain of discourse), **Second-order logic**, in addition, also quantifies over relations.

Monadic second-order logic (MSO) is a restriction of second-order logic in which only quantification over unary relations (i.e. sets) is allowed.

2.2.4. MSO (Graph) Properties

MSO logic can express graph properties and mappings from (labeled) graphs to (labeled) graphs [KAI11]. Typical MSO properties include:

- **K-colorability**
  Example graph is 3-colorable:
  $\exists X, Y\ (X \cap Y = \emptyset \wedge \forall u, v\ (edg(u, v) \implies [(u \in X \implies v \notin X) \wedge (u \in Y \implies v \notin Y) \wedge (u \notin X \cup Y \implies c \in X \cup Y)]))$
- **Connectivity**
  Example graph is **not** connected:
  $\exists Z\ (\exists x \in Z \wedge \exists y \notin Z \wedge (\forall u, v\ [u \in Z \wedge edg(u, v) \implies v \in Z]))$
- **Vertex Cover** [Fic+20, Ch. 4.2]
  Given a graph $G = (V, E)$ a vertex cover is a set
  $C \subseteq V : edg(u, v) \implies \{u, v\} \cap C \neq \emptyset)$
  Then minimal vertex cover (MinVC) asks to find the minimum cardinality $|C|$ among all vertex covers of G.

2.3. Tree Decomposition

A *tree decomposition* (TD) of a graph G is a pair (T, χ). T is a tree and χ is a mapping which assigns each node $n \in V(T)$ a set $\chi(n) \subseteq V(G)$ called a *bag*. Then (T, χ). T is a TD if the following conditions hold:

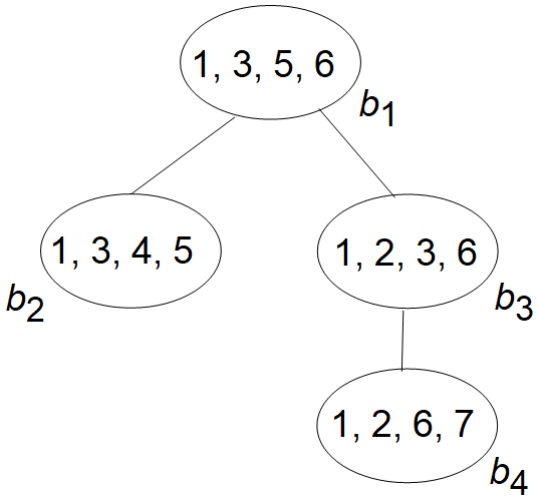
1. for each vertex $v(n) \in V(G)$ there is a node $n \in V(T)$ such that $v \in \chi(n)$
2. for each edge $(x, y) \in E(G)$ there is a node $n \in V(T)$ such that $x, y \in \chi(n)$
3. if $x, y, z \in V(T)$ and y lies on the path from x to z then $\chi(x) \cap \chi(z) \subseteq \chi(y)$. The set of bags that contain the variable v induce a connected sub-graph of T.

The width $width(T)$ of a tree decomposition T is $max_{n \in V(T)}(|\chi(n)|) - 1$. The tree width of a graph is the *minimal width* over all tree decompositions of the graph.

MSO queries on tree decomposable structures are computable with linear delay [Bag06]. There exist many "easy" problems on tree decomposable graphs [ALS91].

We use tree decompositions throughout our visualizations in this thesis. One detailed explanation is also introduced in [Fic19] at page 169. The tree decomposition for our "wheelgraph" example 17 can be viewed in listing 1. The conditions for tree decompositions outlined above are easy to verify for this small example, and we get a tree width $width(T) = 3$ for this graph.

Figure 4: Tree decomposition of a wheel graph (Fig. 17)
with seven nodes and one node in the center:



Listing 1: Tree decomposition in DIMACS format for Figure 4

```
s td 4 4 7
c r 1
b 1 1 3 5 6
b 2 1 3 4 5
b 3 1 2 3 6
b 4 1 2 6 7
1 2
1 3
3 4
```

Some projects might use so called *nice* tree decompositions in order to simplify the cases in the used algorithm [Zis18, Ch. 2.2]. For every tree decomposition a nice tree decomposition can be computed within linear time without increasing the width [Klo94]. Non-nice tree decomposition, however, have advantages in terms of solving time. In the application *gpusat* for example they combined three single operations into one introduce forget (IF) operation. Thereby, reducing the overhead for memory operations on the device and the overhead for copying and retrieving tables between devices. Another advantage is that this reduces the size of the tables they need to store by forgetting the variables which do not occur in the next bag after each introduce operation [Zis18, Ch. 4.2.1]. We see the results of this combined IF operation throughout our visualizations, where for the most part only the results needed for the next bag are included.
Similar techniques regarding non-nice TD are also used in the solver dpdb, see [Fic+20, Ch. 4.1].

As stated in [Zis18, Ch. 2.2], Arnborg et. al. [ACP87] have shown that finding a tree decompositions of minimal width is only feasible for small graphs. There are exact methods for obtaining minimal width tree decompositions, e.g. [GD12; BB06]. The solvers for which we implemented the visualization use heuristics for generating tree

decompositions, therefore the width of larger examples than shown in this thesis might not be minimal.

2.4. Dynamic Programming on Tree Decompositions

In this chapter we want to give a short overview over the basics of dynamic programming. For specific information on the visualized algorithms we refer to [Zis18; SS10; Fic+20].

With dynamic programming, a problem is broken down into parts and the parts are then solved individually. The solving algorithm evaluates the formula along the path of the tree decomposition in a bottom up order starting in one of the leafs. The solution for each assignment in the bag is then stored in a table. The size of the table is exponential to the size of the bag. The tables of the child nodes can be deleted as soon as the table of the current node is generated. [Zis18, Ch. 3.1].

The dynamic programming approach for a given graph works as follows:

1. Create a tree decomposition T of the graph.
2. Apply dynamic programming for each bag in bottom up order of T
   a) visit the next node n of T,
   b) apply the solving algorithm to the bag and
   c) store the results in a table.
3. Output the result based on the table of the root node of the tree decomposition.

The basic algorithmic procedure is presented in Fig. 5. It does include a step one to build a graph in case the type of problem (like SAT) requires it.

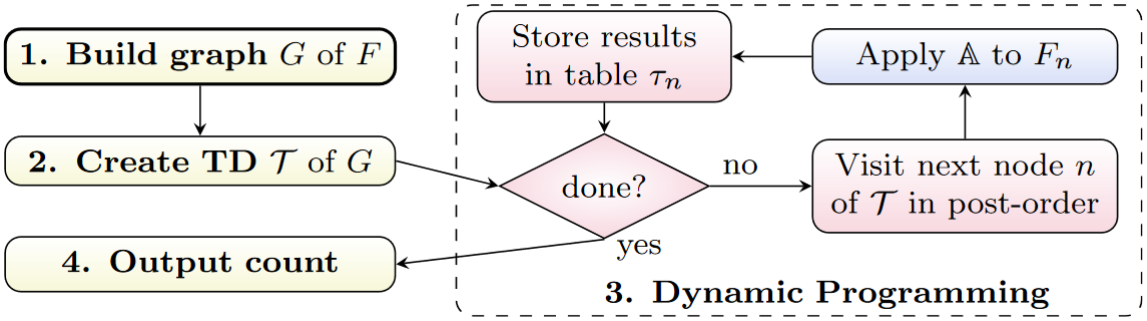


Figure 5: Basic procedure for dynamic programming on TD. [Zis18, Figure 3.1]

2.5. Parametrized Complexity

In the following two sections we want to introduce the reader with some computational bounds for the underlying concept of solving MSO-problems with the current approaches.

2.5.1. Fixed-Parameter Tractable (FTP-)Algorithm

FPT for model checking: Some problems can be solved by algorithms that are exponential only in the size of a fixed parameter while polynomial in the size of the input. Such an algorithm is called a fixed-parameter tractable (FTP-)algorithm, and the problem can be solved efficiently for small values of the fixed parameter. [Gro99]

There are efficient algorithms for enumerating and for counting the number of solutions of a MSO formula, if it is ensured that the input data is preprocessed in linear time, and that each solution is then produced in a delay linear in the size of each solution [Bag06; ALS91]. MSO graph properties are "fixed-parameter-tractable" with respect to tree-width. So are MSO counting and optimizing functions. [KAI11]

DETAILS MinVC:

2.5.2. Courcelle's Theorem

Courcelle's theorem is a constructive meta-theorem and thus does provide algorithms for evaluating MSO formulas over graphs of bounded treewidth. Courcelle's theorem [CE12, p. 54]:

Theorem 1
Every graph property definable in monadic second-order logic (MSO) is decidable in linear time on graphs of bounded tree-width.

To be more specific: It holds that for all $k \in \mathbb{N}$ and MSO-formulas F is the decision problem for a given graph G, whether $G \models F$ is true, in time $2^{p(tw(G))} \cdot |G|$ with a polynomial p decidable.

A generic workflow implementing Courcelle's theorem looks like we see in Figure 6.

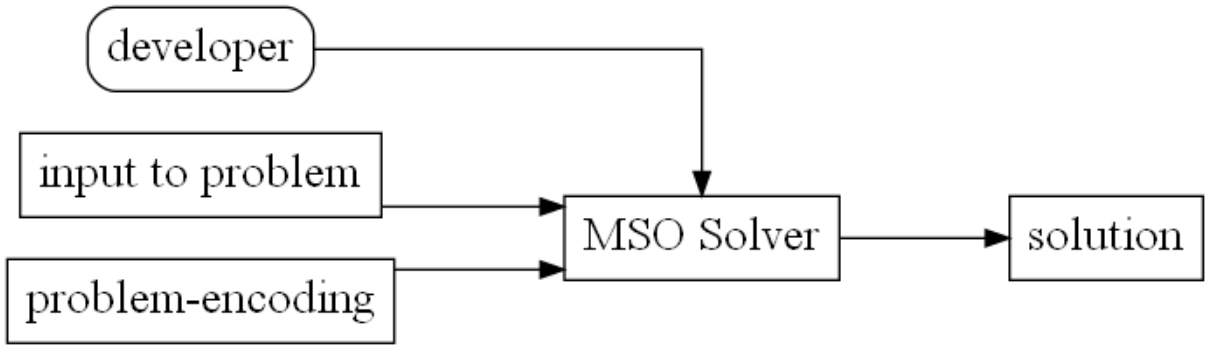


Figure 6: Implementation of the theorem

Even if linear in the size of the input, naive implementations are still expensive and the constant factor can be very significant. An efficient implementation will require several tricks to make this approach practical. Developing and debugging these "tricks" is not always that easy, so a visualization can help at this point.

Auswirkung, Aktuelle Grenzen.

3. Practical Requirements

The exchange of intermediate results of solvers and parts of the visualization is done via two defined file formats. In the following two chapters we want to introduce the reader with the concepts of the DIMACS and DOT format. These formats are either produced or required as input at various stages of the application.

3.1. DIMACS format

Inputs for solvers and in some cases for preparing the visualization of dpdb are usually in a DIMACS format. There exist several standardized formats for different cases, for example CNF clauses, graph edges and graph decompositions. Several file formats for these purposes were developed at "DIMACS" (the Center for Discrete Mathematics and Theoretical Computer Science) [DIM20] beginning in 1993 at Rutgers University. They are partly supported in several math-related software.

The underlying concept is one line-based ASCII file and for all different formats specifies comments as lines starting with the character "c", a problem line starting with "p" (rarely with "s") and following the problem line usually multiple lines specifying the data in a format depending on the problem type. The formats used for this work are:

DIMACS CNF: This format is used to define a Boolean expression, written in conjunctive normal form. The problem line specifies the type, **number of variables** and **number of clauses**. The following lines specify the clauses a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. Each clause is followed by the character zero, so 0 should not occur as a variable. Instead variables are expected to start at one.

DIMACS tw: This format is used to describe a single undirected graph. The problem line specifies the type, **number of nodes** and **number of edges**. The following lines specify the edges with two nodes separated by a space.

DIMACS td: This format is used to describe a tree decomposition. The problem line specifies the type, **number of bags**, **maximum size of the bags** and **number of nodes**. The following lines describe the bags starting for each with "b", the **bag number** and **nodes in this bag**. Following these bags are lines not prefixed with a "b". Now each line describes one **edge between the bags** as two bag numbers separated by a space.

Some examples of this format can be seen in the appendix on page 65.

3.2. DOT format

The graph description language DOT can be used to describe directed or undirected graphs and specify layout details and various attributes for graphs, edges and nodes. It is similar to the Graph Modeling Language [4] that is used in Chapter **??** as a text

[4] https://gephi.org/users/supported-graph-formats/gml-format

based file format for describing graphs. Keeping this in mind, in this chapter we restrict ourselves to a short introduction of the DOT properties that were used the most.

The complete abstract grammar for DOT can be viewed at the website of the DOT language `https://graphviz.gitlab.io/_pages/doc/info/lang.html`.

The nodes in the DOT-language are *labeled*, so creating a node takes one string identifier and might additionally be provided with a text label. Valid examples for IDs include all combinations of letters and digits.

The optional text labels an have various formats. One example with the setting "shape=box"

The visualizations presented in the thesis are constructed as undirected graphs, but would be easily extendable to directed representations, since the order of the edge endpoints remains intact in almost all operations.

Another concept utilized were the sub-graphs and clusters available in DOT. To get a well structured (bipartite) incidence graph, each partition is placed in an individual cluster, making adjustments to various distances simple.

DOT has different components (see `http://www.graphviz.org/doc/info/attrs.html`) which can be modified with attributes: edges, nodes, the root graph, subgraphs and cluster subgraphs, respectively.

We want to introduce the reader with the attributes that are modifiable in several places of the visualization:

- **rankdir -** "TB", "LR", "BT", "RL", corresponding to directed graphs drawn from top to bottom, from left to right, from bottom to top, and from right to left, respectively.
- **fillcolor -** The color used as a node-background in different situations, for different values see `https://graphviz.org/doc/info/attrs.html#k:color`. `https://graphviz.org/doc/info/colors.html` By default this is a linear fill; If the value is a colorList, a gradient fill is used.
- **fontcolor -** Color for text in nodes or graphs.
- **style -** Drawing style of the nodes. At present, the recognized style names are "dashed", "dotted", "solid", "invis" and "bold" **for nodes and edges**, and "filled", "striped", "wedged", "diagonals" and "rounded" **for nodes** only.
- **margin -** Used for horizontal and vertical margins around graphs.
- **fontsize -** Font size, in points.
- **penwidth -** Specifies the width of the pen in points. This is used to draw lines and curves, including the boundaries of edges and clusters.
- **nodesep -** In dot, this specifies the minimum space between two adjacent nodes in the same rank in inches.

- **shape -** Controls how nodes are drawn. For a full list of shapes see `https://graphviz.org/doc/info/shapes.html`. There are three main ways of specifying shapes, and all are applied in our visualization:

  (1) **polygon-based**; for example as a box, ellipse or diamond.

  Example: shape = diamond label = v_1

  (2) **record-based** nodes that are basically boxes stacked vertically and horizontally and each filled with text.

  Example: shape = record label = {sol bag 2|{{$v1$|1|1|1|1|0}|{$v3$|1|0|0|1|1}
  |{$v5$|0|1|0|1|1}|{size|3|3|2|3|3}}|min-size: 2}

  (3) **HTML-like** labels are specified in a own grammar and can contain tables, text-styles like italic, sub- and superscript, vertical and horizontal rules.

  Example: shape = box label =

  <<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
  <TR><TD BGCOLOR="white">bag 1</TD></TR>
  <TR><TD PORT="anchor"></TD></TR><TR><TD>[1, 3, 5, 6]</TD>
  </TR><TR><TD>dtime=0.0009s</TD></TR></TABLE>>

4. Implementation

In this section we want to give a thorough insight into the functionalities of TDVisu.

As a rough overview our software TDVisu works as follows:

1. We generate data using a special solver extension or extra programs and extract arguments for the visualization into one file following our JSON schema.
2. We read the data and parameters into the visualization.
3. The program creates a graph-layout for the bags of the tree decomposition and calculates which parts are to be shown at each step of the provided solving process.
4. It creates a graph-layout for each additional graph to visualize alongside the steps in the tree decomposition.
5. Each graph for individual time steps is saved as a SVG file.
6. If desired, the individual parts of a time step can be merged into a single SVG file.

The programming language chosen is Python because of it's rich dependency environment, fast prototyping, simple tooling for debugging with pdb, optional static analysis with mypy, easy to follow code-style using pylint and autopep8, and modern packaging with pip and pypi.

Python 3.8 was the newest python version at the beginning of the project, released on October 14th 2019. The innovation most frequently applied in this project was the f-string support for shorter and easier to read string-building.

The development process was for most parts of the final software driven by evolutionary prototyping with the help of small and well understood examples such as 31. It helped to understand the possibilities of visualization in this domain and gather user input and requirements early [Ove91]. Some artifacts of the early prototypes with different graph-description languages can be still seen in the class *Graphoutput* in **??**.

The first versions were developed in the repository `https://github.com/VaeterchenFrost/gpusat-VISU` and the first releases and later development of the source code was outsourced to the `https://github.com/VaeterchenFrost/tdvisu` repository.

4.1. Commandline and Configuration

The *tdvisu.visualization* expects the command line parameters in a format described by Table 1.

Table 1: Usage visualization.py

[-h] [–version] [–loglevel LOGLEVEL] [infile] outfolder	
infile=stdin	Input file for the visualization must conform with the JsonAPI.md
outfolder	Foldername to output the visualization results to

--loglevel	set the minimal loglevel for the root logger
--version	show program's version number and exit
-h, --help	show the help message and exit

We see that this input is very simple, and that the heavy lifting is done with the input file given in *infile*.

One extra possibility for configuration comes with the method **logging_cfg** from tdvisu.utilities. There are two example configurations provided with our project, one in the .yml, one in the .ini format. The implementation is very flexible in detecting which parser has to be applied - either via a dictionary-like or a configuration-like function. Both possibilities are documented in python's logging configuration.

Our default configuration in tdvisu/logging.yml and tdvisu/logging.ini provides one handler, two formatters and six loggers.

The **handler** is a stream handler to sys.stdout with level *WARNING* and the the 'full'-formatter to format messages.

The **full-formatter** includes the full date and time up to milliseconds. After that we can expect the logging-level, filename and line where it was generated, and the message itself.

The **loggers** we use in our project are located in

- root, level: WARNING
- visualization.py, NOTSET
- svgjoin.py, NOTSET
- reader.py, NOTSET
- construct_dpdb_visu.py, NOTSET
- utilities.py, NOTSET

and can be individually customized using one configuration file. With the command line parameter *--loglevel* we can modify the level of *root* and it's associated handlers.

4.2. Initialization and Tree Decomposition

The main purpose of the initialization is parsing the input file containing visualization information.

To get a small overview of the type and structure of the data processed in the software, let us take a quick look at the initialization with the API. We instantiate a *Visualization* object as shown in Listing 2 , which parses the *VisualizationData* with the help from our *inspect_json* method.

Next we want to extract information into several objects:

- the instance variables
  - *timeline*, describing the time steps on the tree decomposition
  - *tree_dec*, describing the TD itself

  - *bagpre*, *joinpre*, *solpre* and *soljoinpre* as names for different nodes in the produced visualization

- The *VisualizationData* object containing the data for
  - IncidenceGraphData in Listing 7 p. 61
  - GeneralGraphData in 8 p. 62
  - SvgJoinData in 6 p. 61
  - adjustable parameters, which influence additional aspects of the visualization

Listing 2: Overview of data initialization

```
def __init__(self, infile, outfolder) -> None:
  self.data: VisualizationData = self.inspect_json(infile)
  self.outfolder = outfolder
  self.tree_dec_digraph = None

def inspect_json(self, infile) -> VisualizationData:
  visudata = read_json(infile)

  [...]
  _incid = visudata['incidenceGraph']
  _general_graph = visudata['generalGraph']
  _svg_join = visudata.get('svg_join', None)

  [...]

  return VisualizationData(incidence_graph=incid_data,
                           general_graph=general_graph_data,
                           svg_join=svg_join_data,
                           **visudata)
```

Our next step is to begin visualizing the time-steps on the tree decomposition. Direct results of this task can be seen in Chapter 7.1, Fig. 12, Tab. 4 and Figure 13.

First, a quick setup is performed for the tree decomposition as a directed graph that

- is *strict*, meaning a simple graph where equal edges are merged into one
- has an orientation where it grows with each "rank" of the nodes
- has a shape and a fill-color for it's nodes
- has a margin around it's bounding box.

Second, it creates the basic bag structure by adding nodes and edges for all bags of the provided tree decomposition.

Next comes the iteration over the requested time steps, adding the provided solutions and the edges connecting them to the existing bags. We do this in two passes. A first one in which we move all nodes to their final position. A special case occurs when two bags

are joined together. In this case, we remove all old edges between the children and the parent node, add the result to the graph, and add edges from the children to the result and from the result to the parent node. Details of this step can be seen in Listing 10 on page 63. And one in which we create the actual time step images backwards by masking solutions calculated later. The goal is to have all parts of the graph in optimal position when they first appear, and not to move when additional features are added. For details of these steps, see the Listing 11 on page 63.

An automatically inserted join node is shown in Figure 31 on page 52. The provided data for this example to layout the bags can be seen in Listing3. There we see that bags 2 and 3 have an edge to bag 1 and therefore a join-operation will occur between the results of bags 2 and 3:

Listing 3: Structure provided for bags of example 31

```
"edgearray" :
[
  [ 1, 0 ],
  [ 2, 1 ],
  [ 3, 1 ],
  [ 4, 3 ]
]
```

For rendering the tree decomposition we have added a convenience option to *view* the result files automatically, which is disabled by default.

4.3. Visualizing Additional Graphs

To gain a more comprehensive insight into the solution process, we want to see the graphs that best describe the problem instance that the solver has been working on.

Because the data in the API does not directly include details about highlights corresponding to time steps in additional graphs, we will construct this information on the fly by referencing with the tree decomposition.

With additional data from the *IncidenceGraphData* class we are further able to reconstruct the

- incidence graph,
- primal graph,
- dual graph

for Boolean formulas.

For general problems and with the data from the *GeneralGraphData* class we can construct a simple graph that logically should include the nodes which are in the bags of the TD.

Because graph representations of Boolean formulas are not necessarily connected, we make sure to include potentially isolated nodes into our graphs as well and not only

draw some components of it. For example the formula
$(\neg a \vee \neg b \vee \neg c \vee \neg d) \wedge (b \vee c \vee d) \wedge g$

with its set of clauses
$\{c_1 = \{-a, -b, -c, -d\}, c_2 = \{b, c, d\}, c_3 = \{g\}\}$
will create the dual graph in Fig. 7. This happens with no pre-processing and if the variable g is only included in the unit c_3.

Figure 7: Disconnected (dual) graph

4.4. Incidence Graph

The idea is to visualize the incidence graph as a bipartite graph for clauses and variables that can represent the formula the solver did work on in those problem instances.

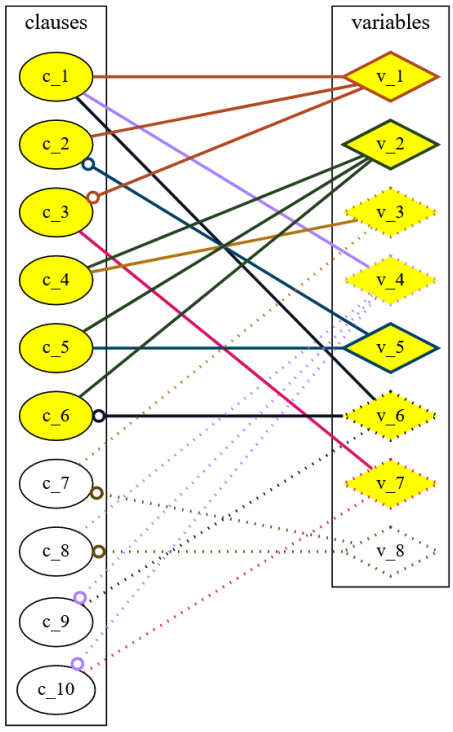


Figure 8: Example for the incidence graph constructed from Ex. 3.

We can see the incidence graph visualization for our example Boolean formula in Figure 8. On the left side we see the clauses ordered by their label, each in an ellipse-shaped node. The other partition does contain the sorted variables in diamond-shaped

nodes. As an addition to its visualization we did include the negations as the little circles on the left end of each edge where a negated variable is included in the respective clause. The right-hand partition does get the different *colors* applied to its nodes and their adjacent edges.

We emphasize the clauses their nodes used in each timestep with a changing background and the not used dotted edges. This allows the correlation of the parts of the formula with the bags.

The representation of the bipartite graphs can be adjusted in many parameters; including colors, fontsize, fillcolor and more, Listing 7 on page 61.

4.5. General Graph

This type of graph can represent the underlying graph for different problems, as well as primal and dual graph for Boolean formulas. Because of its wider scope, the layout for the general graph was not as simple to create as for the incidence graph. We did prepare two different layouts that should cover most cases and are toggled by the parameter *do_sort_nodes*. For smaller and dense graphs of up to 20 vertices it might be helpful to sort the nodes on a circle, while for larger or sparse graphs an organic layout may be more appropriate.

To layout these two options we chose the engine

- *do_sort_nodes* true: circo (see [ST99])
- *do_sort_nodes* false: sfdp (see [Hu05]) with the spring constant 'K' set to 2.

Additional parameters used in both layouts are used to not overlap nodes, as well as drawing the edges first. This should provide a minimally cluttered layout compared to the defaults, but could make edges ambiguous in some cases.

The highlighting for each time step is basically the same as in the incidence graph previously.

We allow an additional option, which depends on the concrete algorithm to be visualized, namely the flag *do_adj_nodes*. In case the algorithm uses adjacent nodes they can be visualized with this flag using a third color for emphasis.

A more detailed presentation of the results with these visualizations is presented in Section 7.3. Figure 9 shows a graph visualization for a graph with 16 nodes and the subgraph $\{11, 12, 14\}$ emphasized.
Figure 10 shows the same graph and emphasis visualized with the nodes sorted and placed on a circular layout. In this layout we find it easier to identify individual nodes by their position and to cross-reference them with further representations.

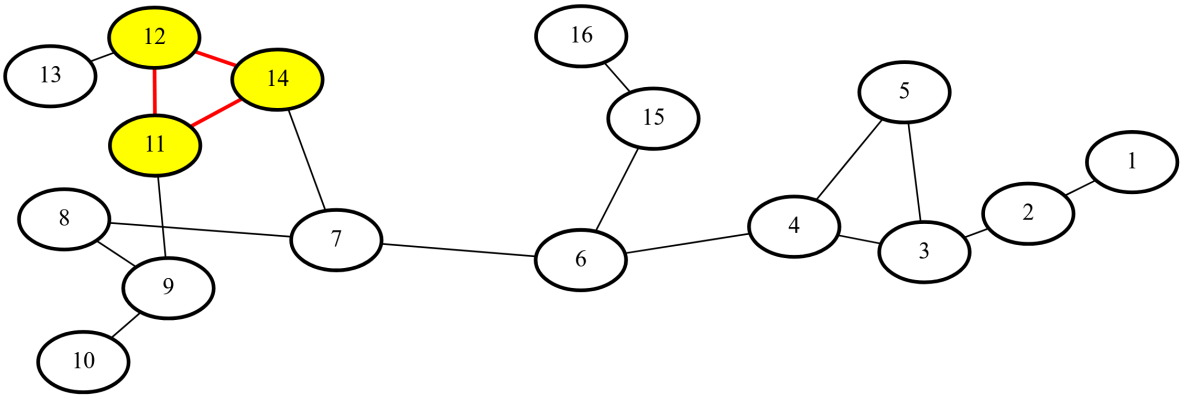


Figure 9: A graph with 16 nodes and highlights for the subgraph $\{11, 12, 14\}$ in yellow and red edges.

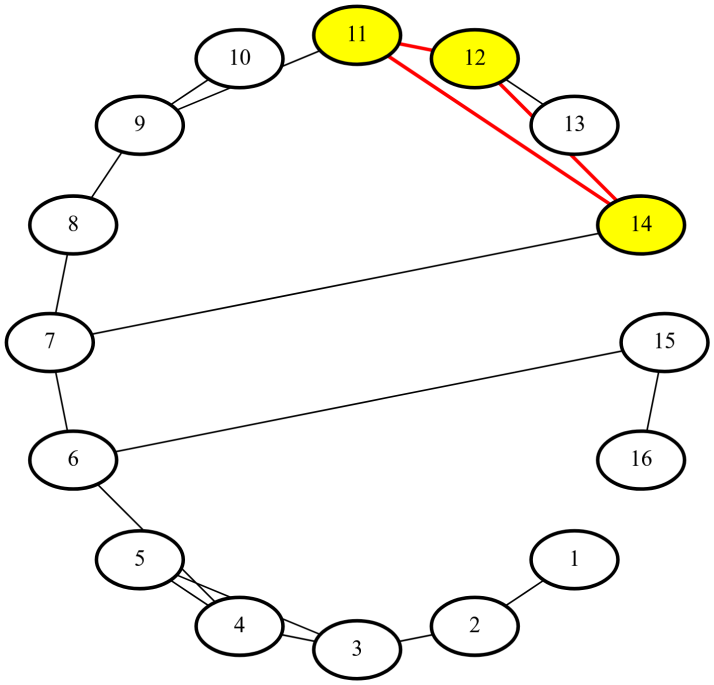


Figure 10: A graph with 16 nodes and highlights for the subgraph $\{11, 12, 14\}$ in yellow and red edges. The nodes are arranged in a circular layout and sorted by their labels.

4.6. Joining SVG

Once all user defined images for one timeline are created, it would be nice to have all graphs combined in one file for each step. All created graphs for each time step together create a overall impression of the work done during solving this step. It showed to be inconvenient to manually open and view the tree decomposition along the primal, dual, incidence or fundamental graph for each time step and problem instance, so we automated this task. To provide some basic scaling and adjusting, we prepared a section

in the API for joining the resulting SVG graphics. Its parameters can be viewed in *SvgJoinData* in Listing 12 on page 64.
We have prepared some examples at the end of this section. The currently implemented functionality could be further extended upon using SVG animations[5].

With the default settings, it aligns the top edge of both images and applies no scaling to either image. The four parameters, each with its expected type and default value in [unit]

* "padding": int = 0, [image coordinates]
* v_bottom: float = None, [size of the *first* image]
* v_top: float = None, [size of the *first* image]
* scale2: float = 1, [size of the *second* image]

allow flexible **vertical**, **horizontal** and **scaling** transformations. Note that the images are placed in a Cartesian coordinate system and their origin is the upper left corner of their bounding box. All images fill a rectangle in this coordinate system with height and width respectively. Since the order of the parameters *v_bottom* and *v_top* could be confused due to the inverted y-axis, we make sure that *v_top* is correctly set to the smaller and *v_bottom* to the larger value whenever possible.

A special case is obtained by setting both parameters *v_bottom* and *v_top* to the same number. Then they are interpreted as the position of the vertical centerline for the second image in units of the first. So setting both parameters to $\frac{1}{2}$ would result in both images being vertically centered.

If we want to position more than two images, it is possible to specify the parameters in a list. The list can be of different length for each parameter - if it is exhausted, the last parameter in the list will be repeatedly used until all images are joined together.

[5] `https://www.w3.org/TR/SVG11/animate.html`

5. Integration in GPUSAT

The integration of visualization into the C++ based `https://github.com/daajoe/GPUSAT` did happen in two parts.

The first part is mostly included in the class *Graphoutput*, and includes several experimental steps into visualization. This class has 186 lines of code (LOC) in its source file, and 44 lines of code in its header file.

The second part was done in the class *Visualization*, and fulfills the API to *TDVisu*. This class consists of 203 LOC in its source file, and 86 LOC in its header file.

The integration into the existing classes from *gpusat* is listed in Tab. 2.

GPUSAT	main	Solver
Graphoutput	6	10
Visualization	5	7

Table 2: Lines of code referencing the classes Graphoutput and Visualization from the main-method or the Solver class.

With utilizing streams[6] for all string operations we tried to keep the impact on performance during the solving process small. When running on the same thread, especially for larger problem instances, the creation of the visualization files could impact the performance of the run. With the relatively small examples we used to visualize during our development process this was not noticeable.

The forked repository, with visualization and some minor fixes included, can be seen on `https://github.com/VaeterchenFrost/GPUSAT`.
The changes in 142 commits made when developing this project can be seen on `https://github.com/daajoe/GPUSAT/compare/master...VaeterchenFrost:master`.

Table 3: Usage: ./gpusat [OPTIONS]

-s, --seed INT	number used to initialize the pseudorandom number generator
-f, --formula TEXT	path to the file containing the sat formula
-d, --decomposition TEXT	path to the file containing the tree decomposition
--CPU	run the solver on a cpu
--NVIDIA	run the solver on an NVIDIA device
--AMD	run the solver on an AMD device
--weighted	use weighted model count
--noExp	don't use extended exponents
-v, --verbose	print additional program information
-p, --nopreprocess	skips the preprocessing step for debugging and visualization-purposes

[6] `http://www.cplusplus.com/reference/sstream/stringstream`

-w, --combineWidth INT=20	maximum width to combine bags of the decomposition
-g, --graph TEXT	filename for saving the decomposition graph
--visufile TEXT	filename for saving the visualization file

An example call with *./gpusat -f ../examples/test_da4_1.cnf -v -p -d ../examples/td4p1.txt -g ../examples/graphfileda41.txt --visufile ../examples/visufileda41.json* enabled verbose output, disabled pre-processing to prevent the creation of bags with too many variables at once to be visualized, and creates full visualization output. The console output produced by this example call is listed in **??**.

5.1. Class Graphoutput

This class does include the first steps to automatically visualize the tree decomposition of the solving process with its solutions. It's main funcionality can output visualizations specified in gml (Graph Modeling Language) [Him10]. We see one small example output in Listing 17 p. 68 with labeled nodes and two edges. The output can be seen in Figure 11.

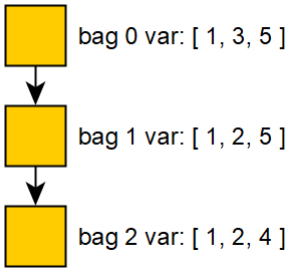


Figure 11: GML source from Listing 17 when plotted (*layout modified*) with yEd `https://www.yworks.com/products/yed` Version 3.20 powered by the yFiles Graph Visualization Library `http://www.yWorks.com`

Additional methods exist to evaluate the possibilities of using Graph-Databases like Neo4j [RWE15]. It is possible to create visualizations of the initial situation for CNF Clauses and the computed tree decomposition of the primal graph as two *Neo4j Cypher*[7] queries with:

- One graph representing the SAT formula and queries to construct incidence, dual and primal representations.
- one graph representing the tree decomposition of the primal graph with it's bags containing variables.

5.2. Class Visualization

To include the extraction of all necessary visualization information into the solver we created a separate fork. To simplify the creation of valid json we selected the actively de-

[7] `https://neo4j.com/docs/cypher-refcard/current/`

veloped c++ library JsonCpp `https://github.com/open-source-parsers/jsoncpp` version 1.9.2 from the open-source-parsers repository.

6. Integration in dpdb

The integration of the API with dpdb was easier to implement after the solving process instead of hooking into the solving, provided that all necessary information got persisted in the database. The integration is included in the TDVisu project as a separate python file. A complete workflow can be accomplished using the arguments

- --store-formula as a problem specific option for Sat and SharpSat
- --gr-file GR_FILE for problems like VertexCover with graph input

when calling dpdb.py
The integration for SharpSat was the first implemented in the file *construct_dpdb_visu.py* with the main parameters

- **problemnumber** the problem-id to select in the database
- **--twfile** TWFILE tw-file containing the edges of the graph
- **--outfile** OUTFILE file to write the output to, default 'dbjson%d.json'
- **--loglevel** LOGLEVEL set the minimal loglevel for the root logger
- **--pretty** pretty-print the JSON
- **--inter-nodes** Calculate and animate the shortest path between successive bags in the order of evaluation. To accomplish this task, an efficient implementation of the bidirectional Dijkstra's algorithm [Gol+06] based on the implementation by NetworkX [HSS08] in bidirectional_dijkstra was adapted. In our use case the weight function is always one, as the edges have no weight associated with them.

After the arguments are parsed by pythons argparse it is possible to adjust logging output by either providing a configuration file (template provided) or giving a minimal logging level per program argument.

Next the *create_json* 9 function connects to the database driver with the number of the stored problem. The "problem type" is available as a string in table *public.problem*, and the appropriate class to prepare the json will be instantiated. Currently the solver handles the problems of *satisfiability* (Sat), *count solutions to a Boolean formula* (SharpSat) as well as *minimum vertex cover* (VertexCover).

7. Application and Images

7.1. SAT Example

As a first complete visualization output we show the full output of checking the Boolean formula (Example 3 in Chapter 2.2.2) for solvability. We have six time steps included in this run. First we will look at the bags in the tree-decomposition, where as simple debugging information we added the time to solve each bag individually into the labels. See Fig. 12, Tab. 4 and Figure 13. We see the nodes gathered in each bag as an array indicated in the TD. The solver goes from bottom to top through the bags. The active bag in each step is indicated by its changing color. When a solution was found, it gets connected as a new node by an edge to its bag. The order the bags got solved is 5, 4, 3, 2 and 1. The last solution added, containing more than zero (partial) assignments in Fig. 13, provides the answer "yes" to the question of SAT.

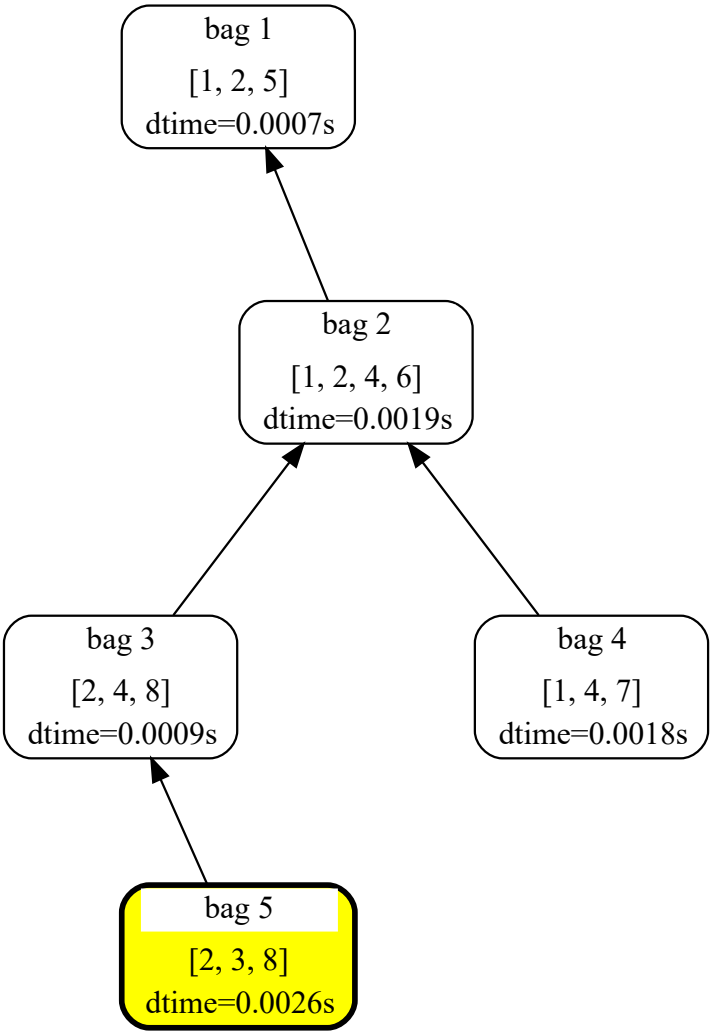


Figure 12: Tree decomposition for solving Example 3.
We see a yellow highlighting for the first leaf (bag 5) to solve.

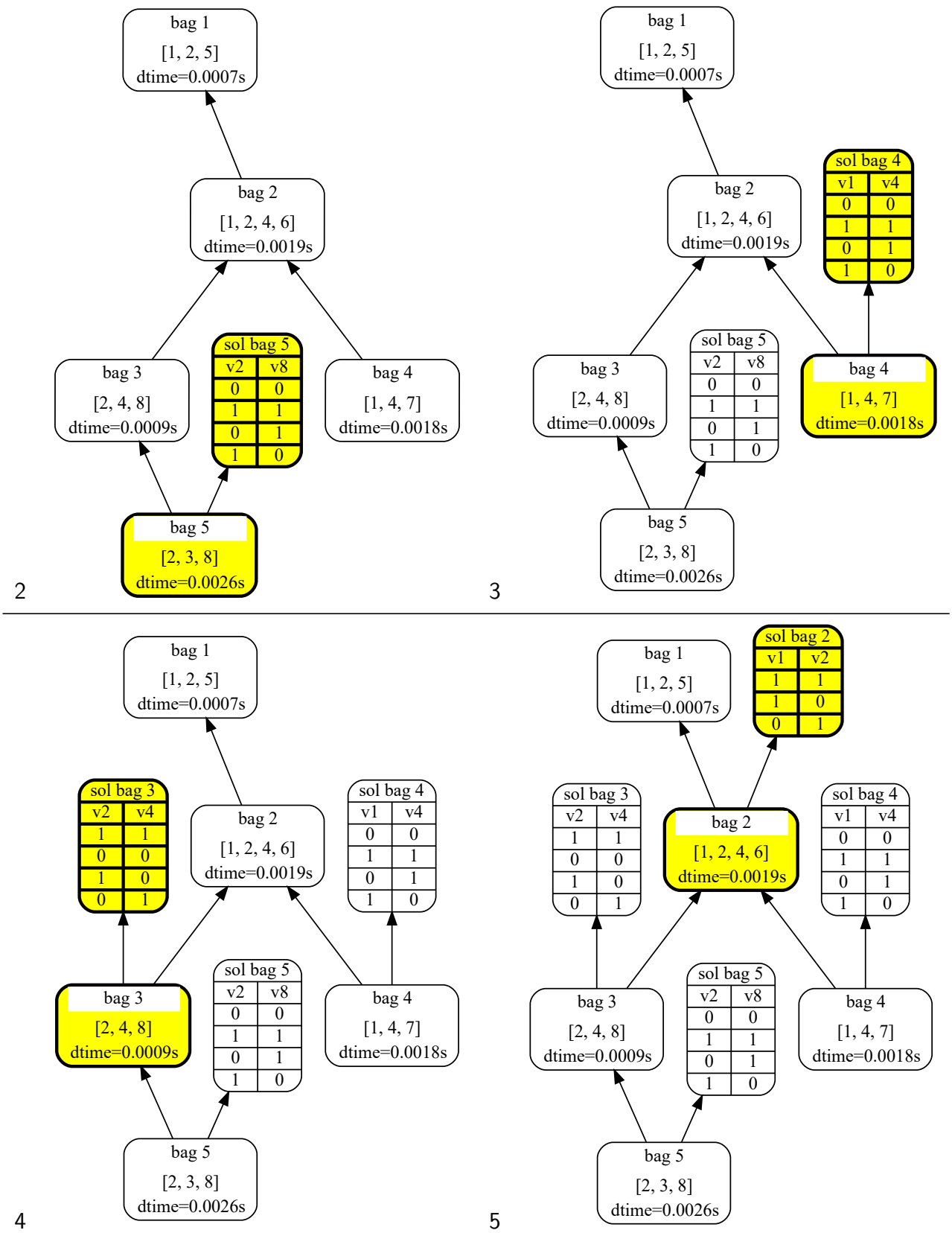


Table 4: Tree decomposition for solving example 3 . Images for steps two to five as labeled from top left to bottom right.

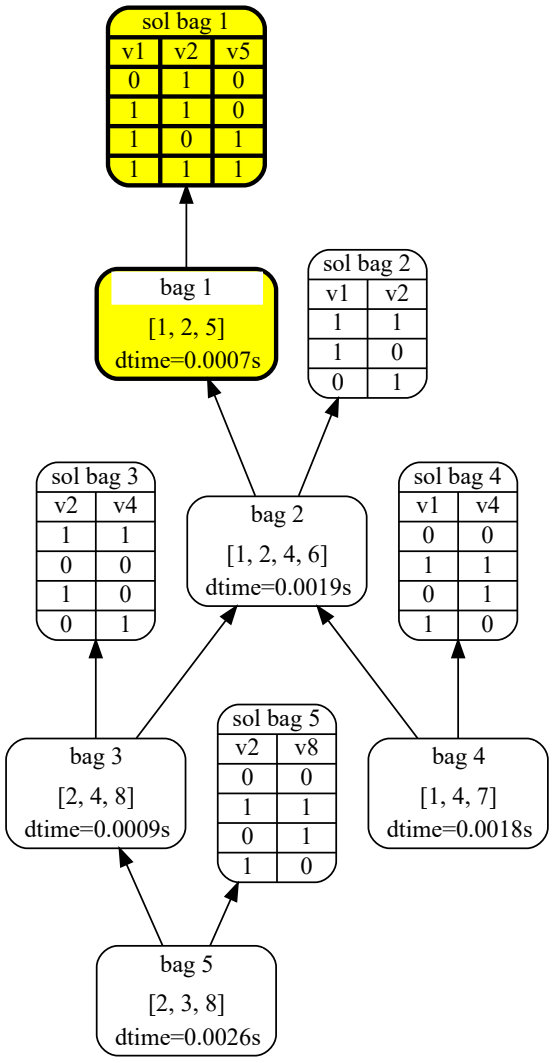
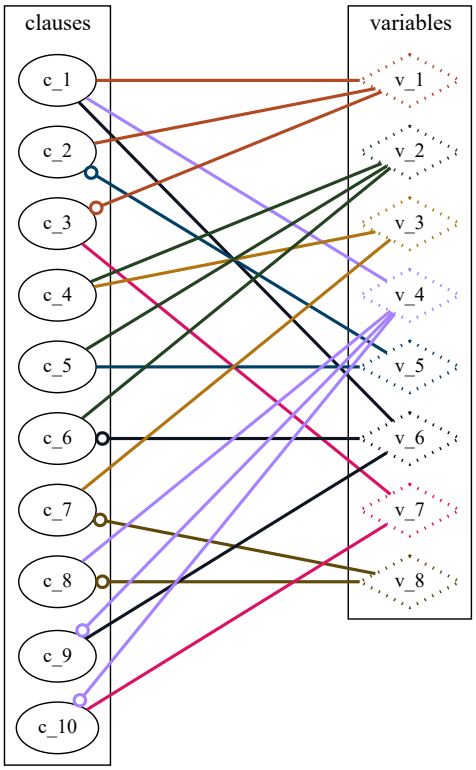
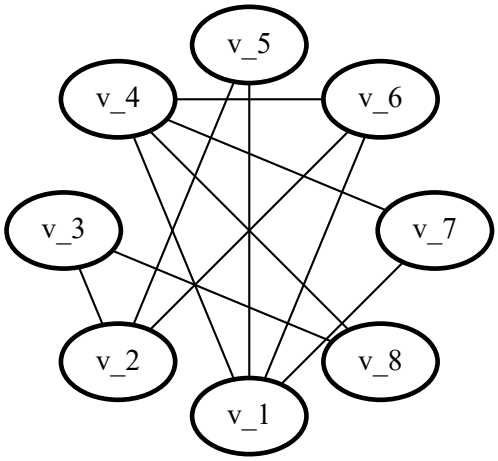


Figure 13: Tree decomposition for solving example 3 .
Final result with yellow highlighting for the last bag (1) solved.

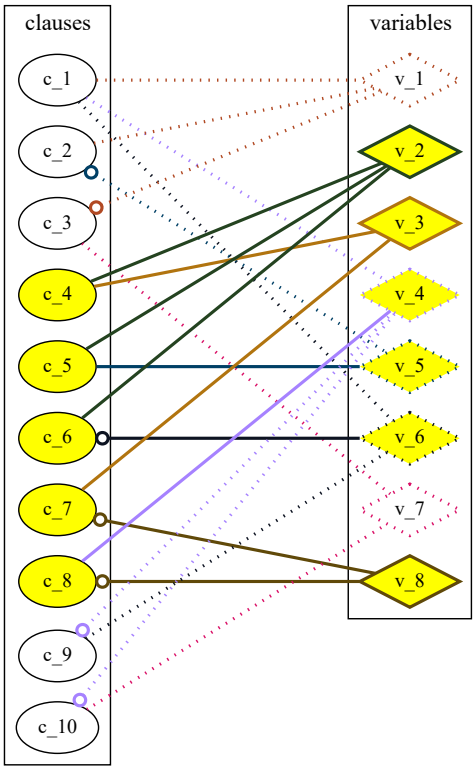
In the following Figure 14 we see the first two visualizations for incidence and primal graph with highlights corresponding to the same run as before. The first step has no highlighting, but we can compare the processed Boolean formula indicated with the clauses as nodes sorted on the left-hand side. The edges either are solid lines or starting with a little circle that indicates the negation of the connected variable at the right-hand side in this clause. The nodes in the variables have three states. As this run operated on the primal graph, there are only variables collected in the bags. When calculating the solutions we do however need whole clauses to be processed together, so additional variables are emphasized. The three states are *1) not included* in any clause needed, *2) indirectly included* emphasized with a dotted border and yellow highlighting in the incidence graph, and *3) directly included* in the bag emphasized in yellow and solid border.
The primal graph is also calculated directly from the Boolean formula and shows one of the same three states for each variable in its own layout.

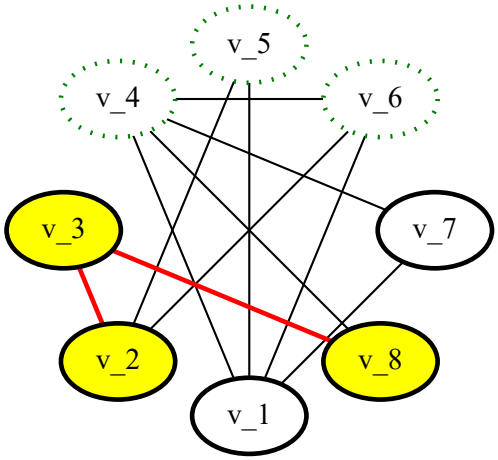
step 1:





step 2:





Visualization of the incidence graph including information for the sat formula

Visualization of the primal graph

Figure 14: Incidence graph and primal graph of example 3 .

7.2. #SAT Example

Like the previous example section we are interested in solutions to example 3. This time we want to solve #SAT and count the number of solutions, that is the number of satisfying assignments. While the tree decomposition and SAT formula stay the same, the solver added one column to our solution-tables and label this column *mc* for "*model-count*", compared to pure SAT solving. It also included a footer with the API to display the sum of all models considered up to this bag. We see two time steps, beginning and end of the run, in Figures 15, 16. Again, the empty top of the reserved space is cut out of the images compared to the direct output.

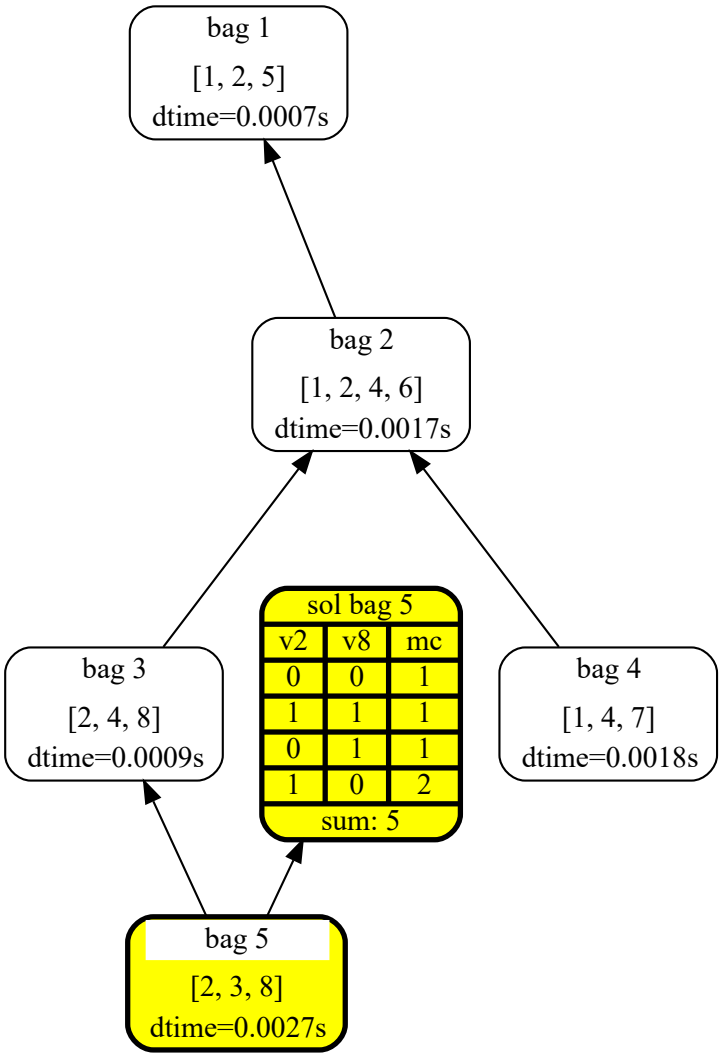


Figure 15: Tree decomposition for solving example 3 with yellow highlighting of the solution for the first leaf.

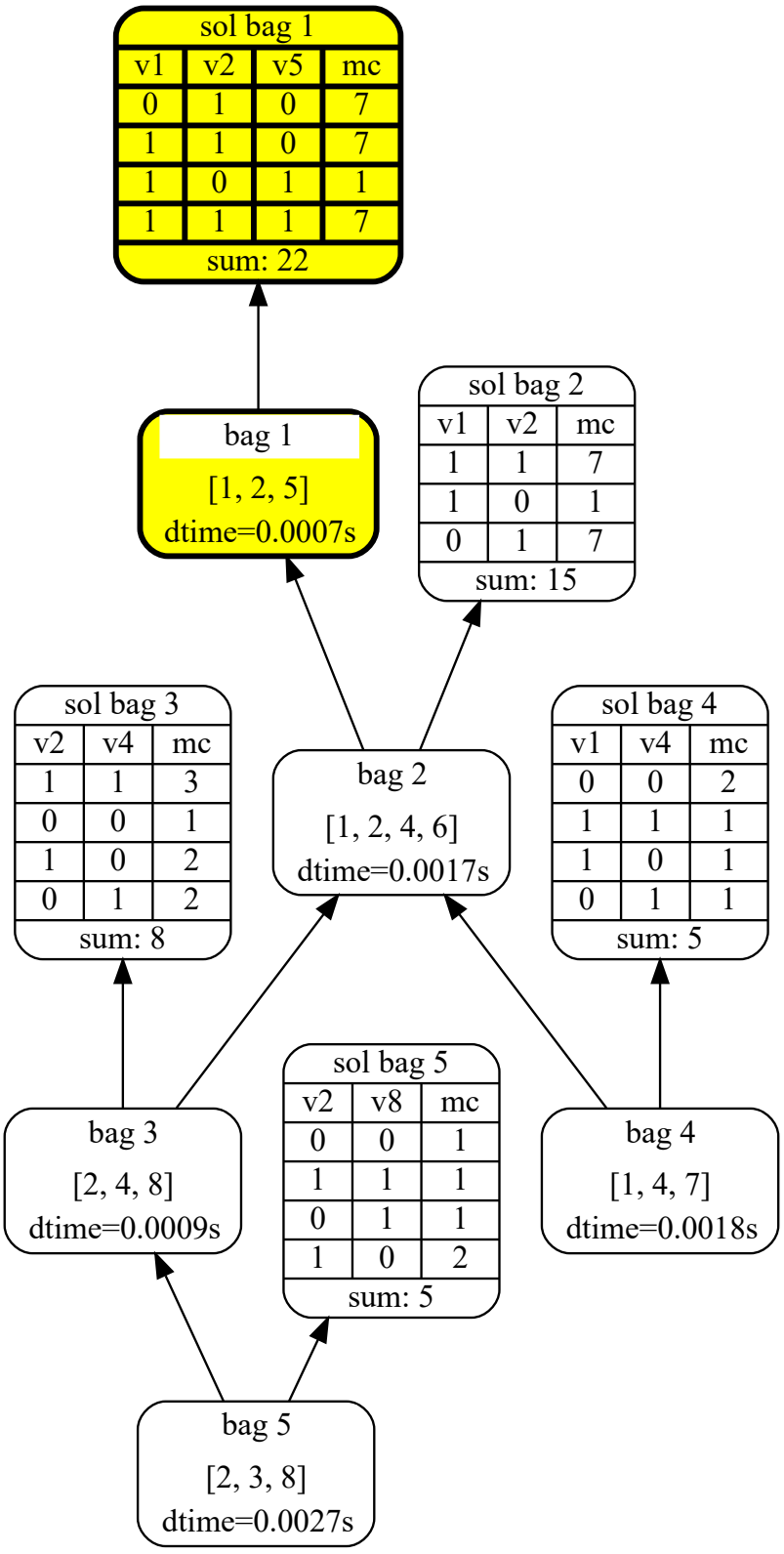


Figure 16: Tree decomposition for solving example 3. The highlighted bag 1 points to the solution of the problem, containing 22 solutions and satisfying variable assignments for v_1, v_2, v_5 contained in *sol bag 1*.

7.3. Vertex Cover Example

After showing solutions around Boolean formulas, this chapter discusses a problem that is itself formulated directly on a graph. As a very small example we will try to solve *minimal vertex cover* for the following graph in Figure 17 with edges seen in Listing 15.

For details on the algorithms used by the solvers see also [Fic+20, Ch. 4.2] on MinVC and its Listing 3 for a template for general problems.

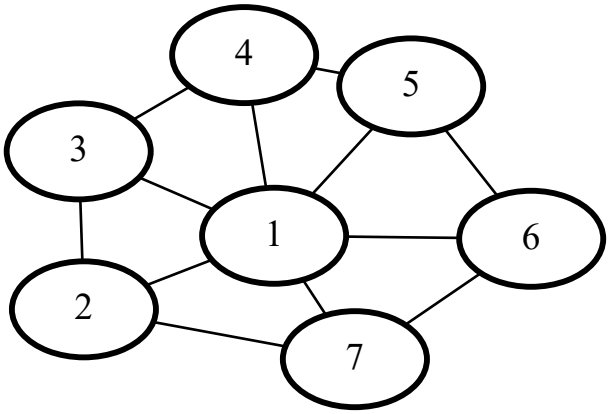
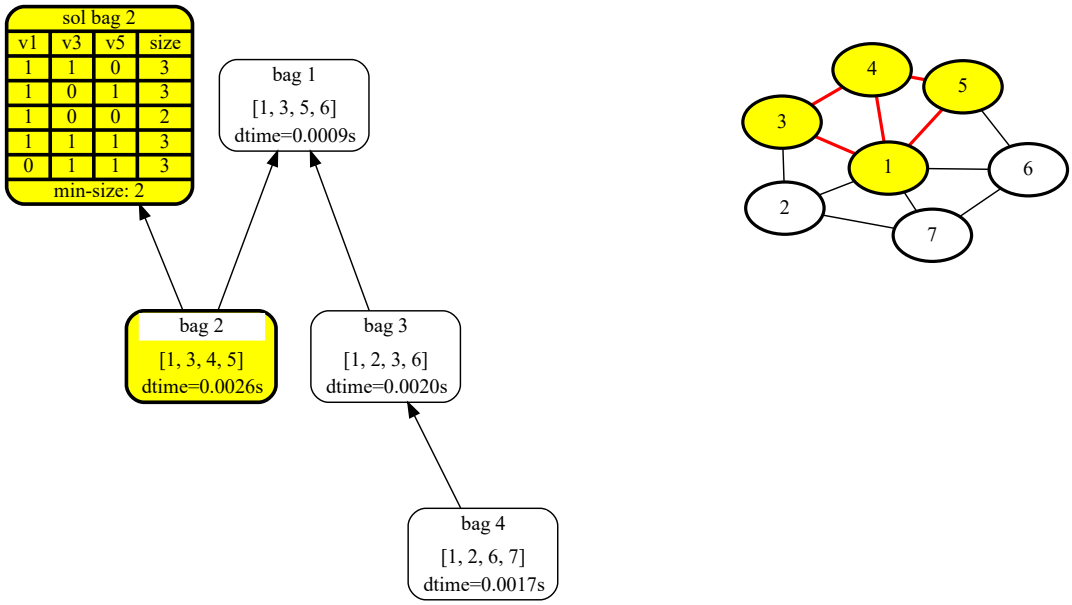


Figure 17: Wheel graph with 7 vertices.

The tree decomposition used created four bags with four nodes each. The images in 18 and 19 are documented with the bag solved. The order here is 2, 4, 3, 1.
The content in the solution nodes is very similar to the previous example on #SAT. The differences are the changed right-most column now reporting the size of the intermediate result if its assignment was used to cover the current graph seen up to this step. The footer now indicates the minimum of those values in each solution. Again solving *bag 1* contains the answer, this time to the question of MinVC, that the size of the minimal vertex cover is four.

To the right of each time step is the currently used subgraph marked with yellow nodes and red edges.

First step solving **bag 2**



Second step solving **bag 4**

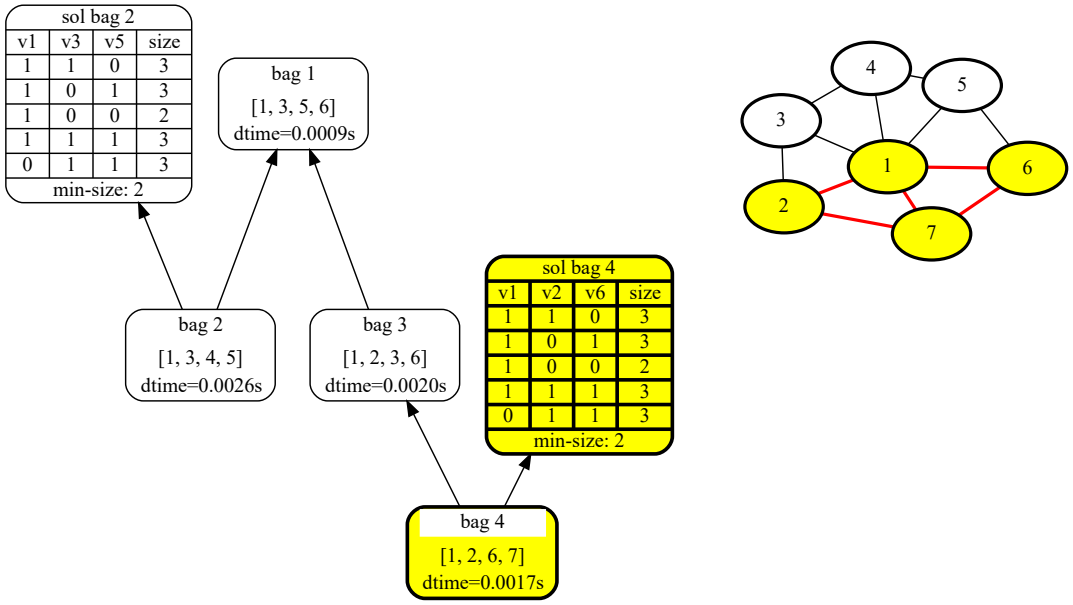
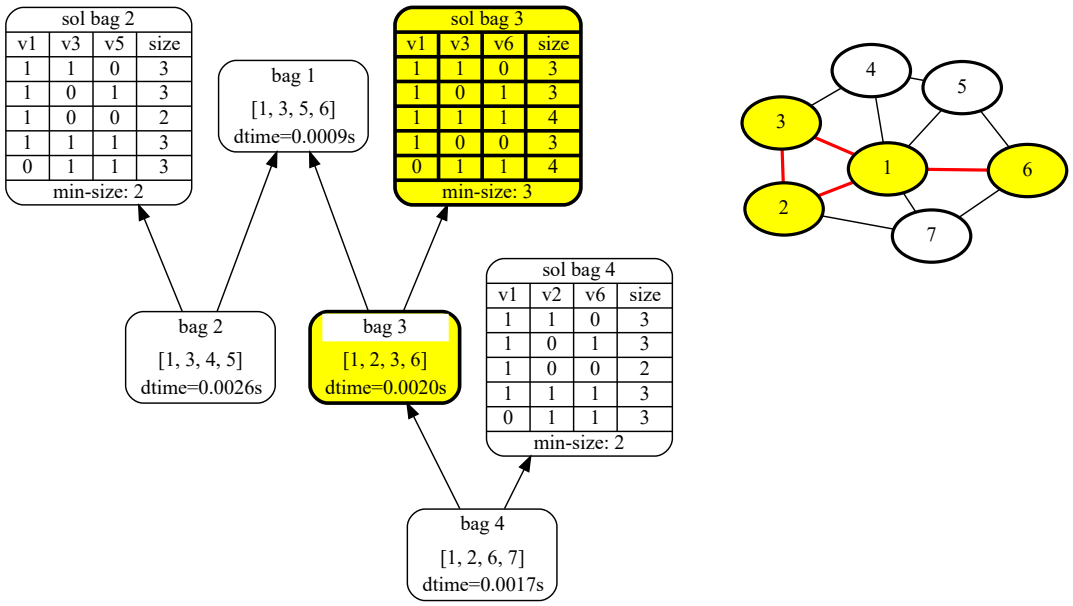


Figure 18: First steps to solve *minimal vertex cover* for example graph 17 on it's tree decomposition.

Third step solving **bag 3**



Fourth step solving **bag 1**

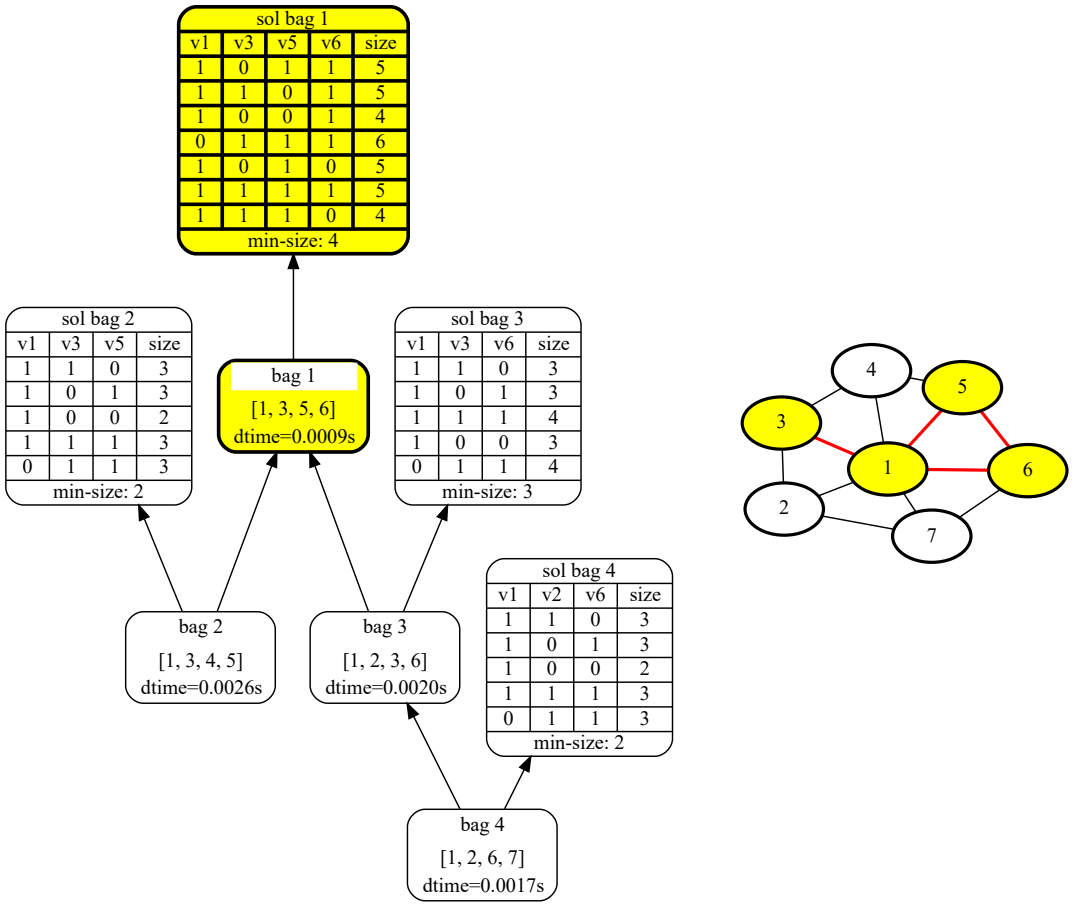


Figure 19: Last steps to solve *minimal vertex cover* for example graph 17 on it's tree decomposition.

7.4. SVG Join Example

After creating several images at each visualization step all images from this step can be automatically combined into one SVG. In this chapter we expand on Section 4.6 with some practical applications of this concept.

With the four parameters *padding*, *v_bottom*, *v_top* and *scale2* we can specify the order in which images will be placed next to each other. To explain some configurations we will take the generated images from the previous chapter.
We will first look at some constructed examples using rectangles in the Figures 20-25. Then we see some actual applications in Figures 26-28 at page 45.

Possibilities for joining images with these four parameters include joining with:

- default *padding*, Fig. 20.
- *padding* set to 200, Fig. 21.
- scaling both images to the same size, Fig. 22.
- aligning both images for vertical centering, Fig. 23.
- uniformly scaling the second image, Fig. 24.
- scaling and aligning at the bottom edge of the image, Fig. 25.

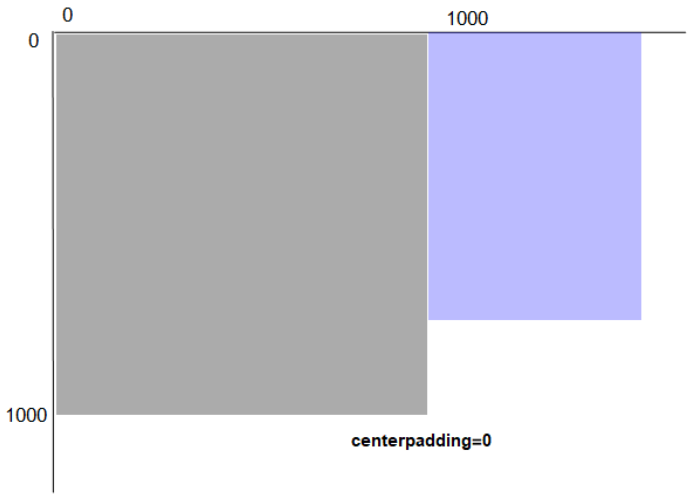


Figure 20: Example for joining with no *padding*.

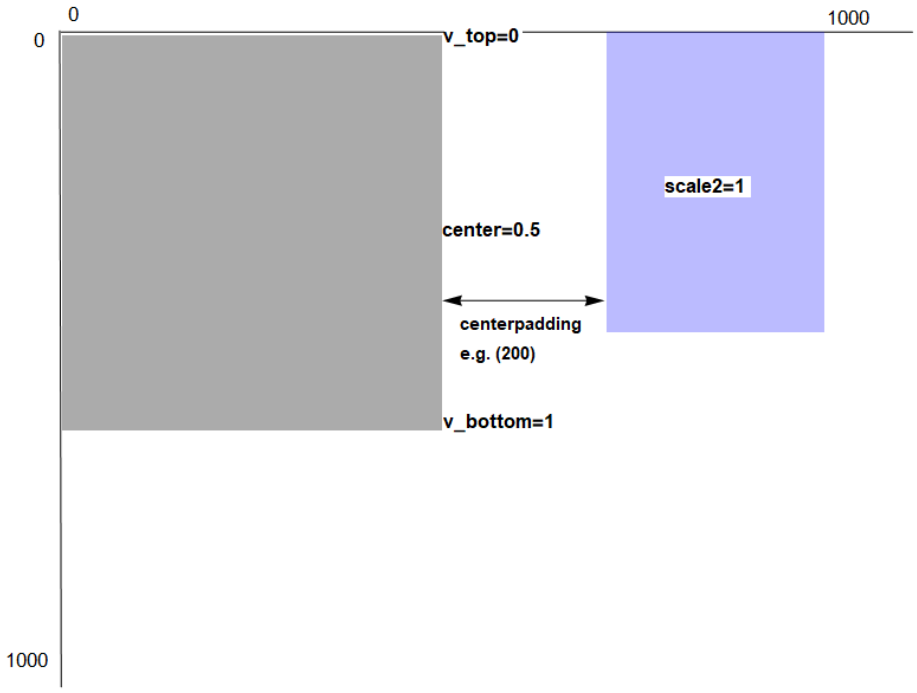


Figure 21: Joining the blue (right side) image to the left gray image with only one parameter *padding* set to 200. We see the default vertical position v_top=0, and the implied coordinate system with an origin in the top left corner.

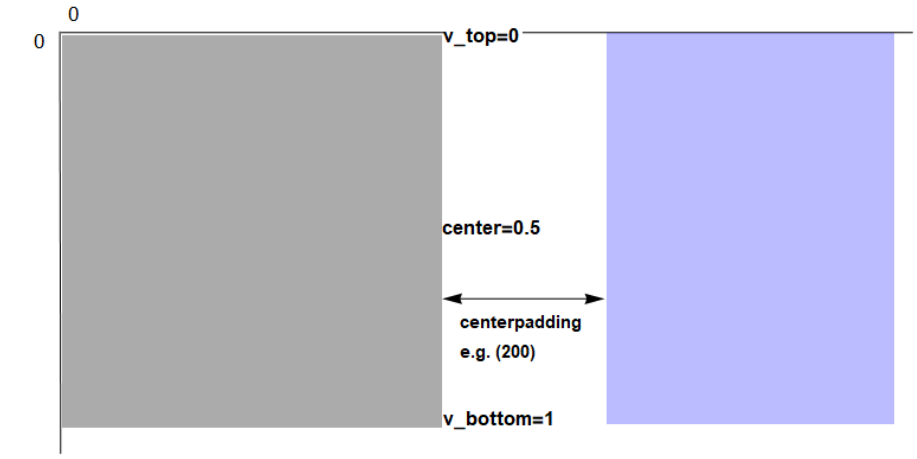


Figure 22: The second image is scaled to the same size as the first image. This can be conveniently achieved by setting *v_top* to 0 and *v_bottom* to 1. Parameter *padding* is 200 as hinted.

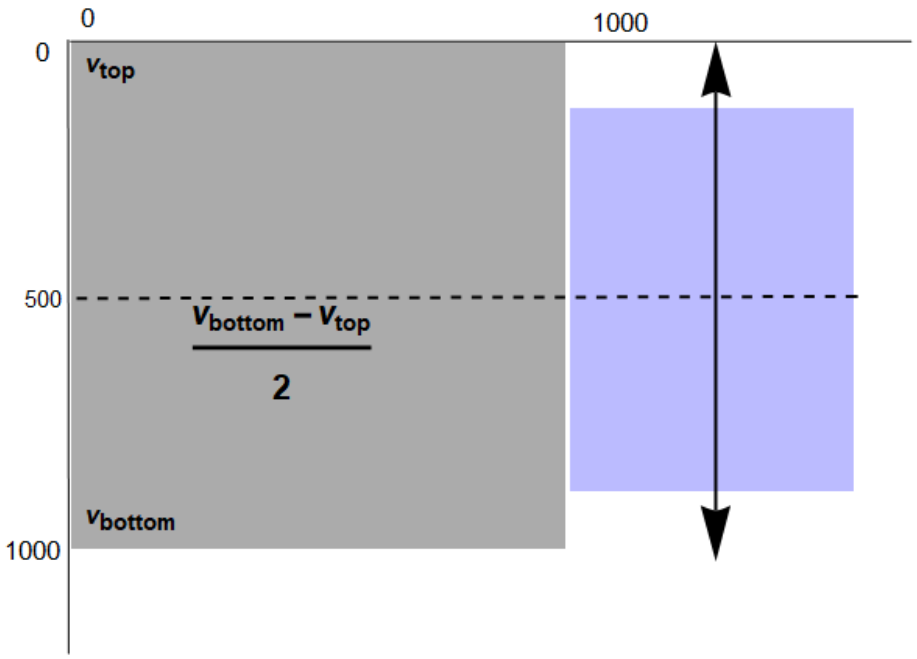


Figure 23: Align both images for vertical centering.
This can be achieved by setting $v\_bottom = v\_top = 0.5$.

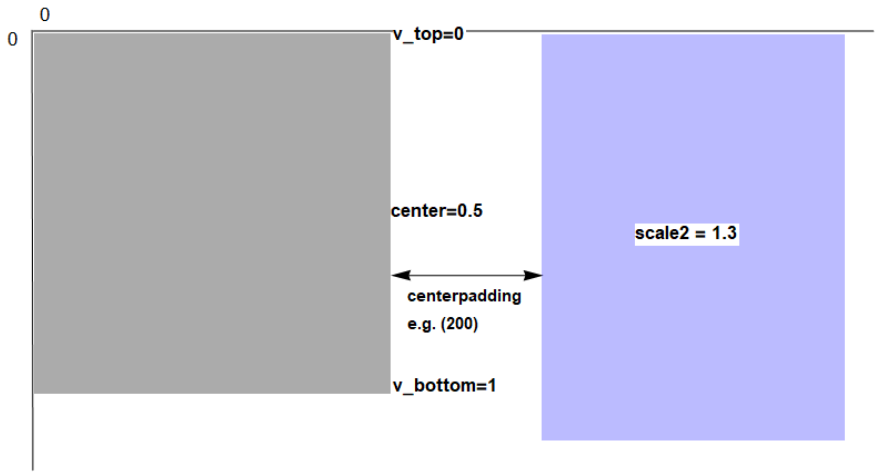


Figure 24: Setting *scale2* to 1.3 to scale the blue (right) image uniformly. Parameter *padding* is 200 as hinted.

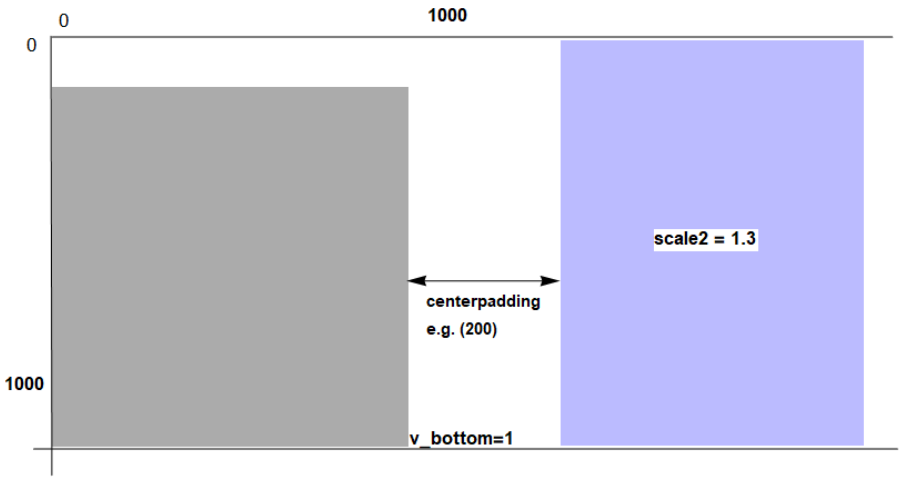
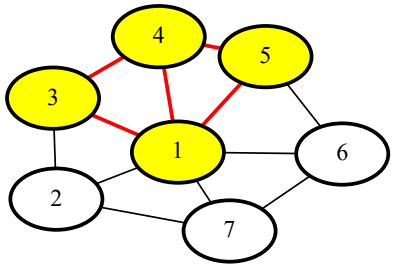


Figure 25: As in figure 24 we use *scale2* to scale the blue rectangle. To align both images at the bottom edge, we use the value $v\_bottom = 1$.

Now for the applied examples we see in Figure 26 the two images are joined with default values for all parameters, in order (*TDStep, graph*).

As we can see the solutions for bags other than 2 are hidden in the output but are taken into account in the image size. The right graph is aligned to the upper edge of the left image. With no padding between the images *graph* image is attached to the right edge of the first image visualizing the tree decomposition.



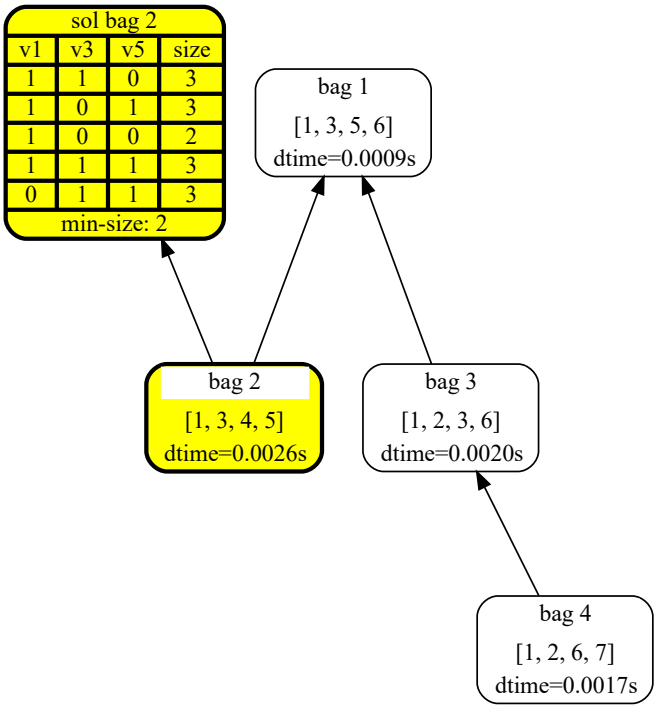
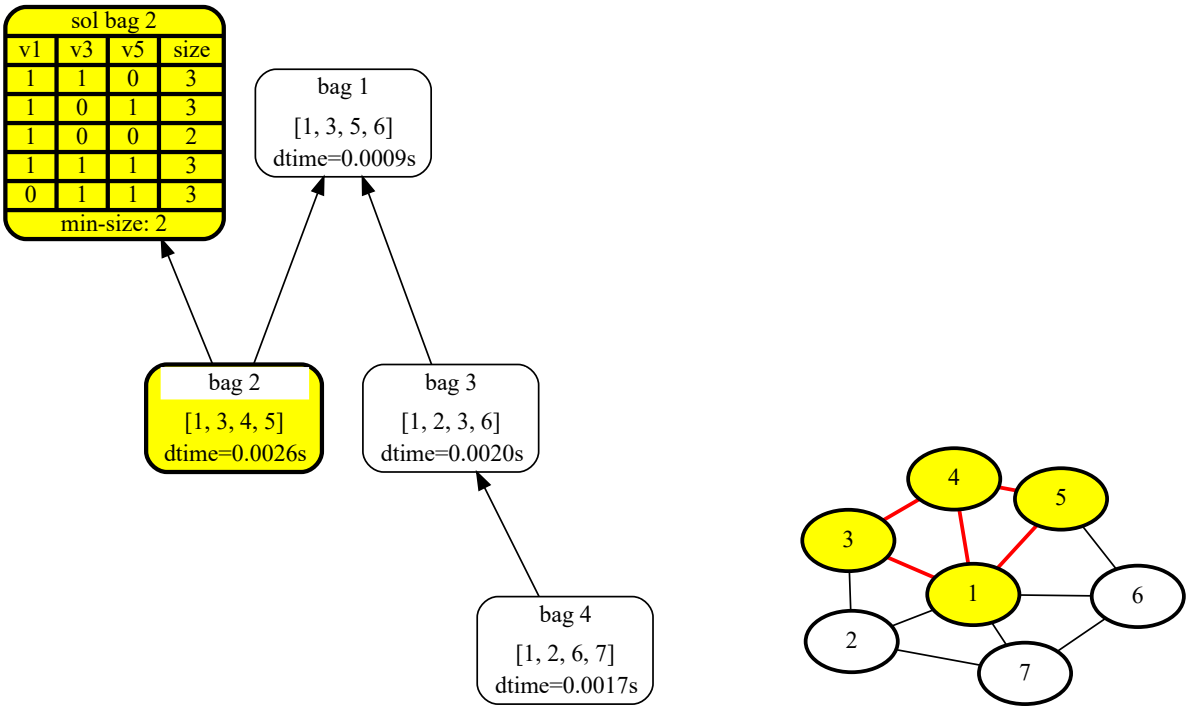


Figure 26: Joining results from Section 7.3 with default parameters at step 2.

v_bottom = "bottom"



v_bottom = "center"

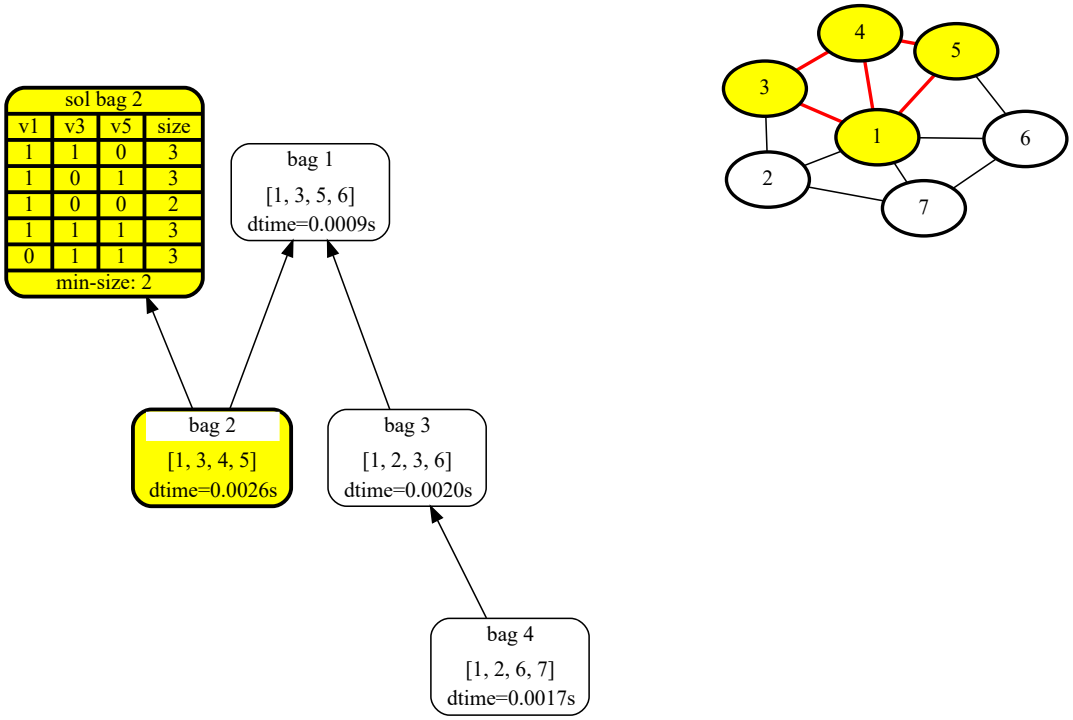


Figure 27: Joining results from Section 7.3 at step 2 and setting v_bottom. The invisible (empty) parts of the graphics is cropped from the top of each.

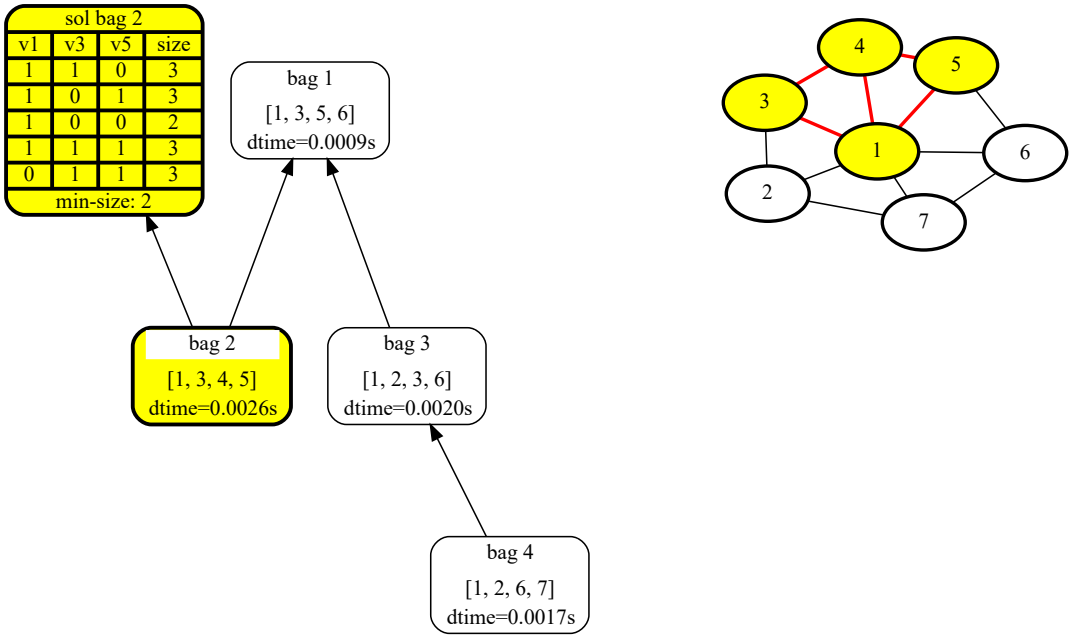


Figure 28: Joining results from Section 7.3 at step 2 and shifting bottom edge of second image to 60% height of the first image.

In Figure 27 the parameter v_bottom is set to "bottom" to align both graphics on the bottom edge; set to "center" elevates the right image to 50% of the height of the first. It is also possible to provide arbitraty floating-point values for vertical adjustment as presented in Figure 28 with v_bottom = 0.6 .

if we want the right graphic a bit larger compared to the default size, we can add additional scaling by providing the argument **scale2**. In figures 32 to 36 the complete time series for solving *minimal vertex cover* for example 15 with join-parameters

padding = [0, 0, 0, 40]

v_bottom = 0.6,

scale2 = 1.5

can be seen.
To further visualize the progress of the algorithm in the results one could use the possibility to specify multiple values for all the transformation parameters *padding*, *v_bottom*, *v_top* and *scale2* and elevate the second image with the values for
v_bottom = [1, 0.85, 0.7, 0.55, 0.4].
This is done in figures 37 to 41 on page 55.

7.5. Visualization of Defects

As stated in the introduction, most of the data we visualize is already present in the solvers. Given the right testing frameworks it is possible to produce a report for structural or even randomly occurring defects in the solver. As the software we used for our

experimental visualizations shown in this work does not itself contain a test environment, we were able to reproduce some undiscovered "bugs" using and developing our visualization tool.

One specific program invocation did not give the expected result solving a *vertex cover* with one hundred nodes. The problem was first found when inspecting the immediate output of the program *dpdb.py*, which did report the wrong result size. The problem could be localized in the tree decomposition, leading to the whole result table being pruned - see the visualization in Figure 29. It is obvious that "bag 59" is disconnected from its parent "bag 57", even if its nodes occurred in other bags in the tree. This violates our rule 3 of the TD ("the set of bags that contain the variable v induce a connected sub-graph of T."). This particular bug occurred only with a specific seed and on the windows operating system.

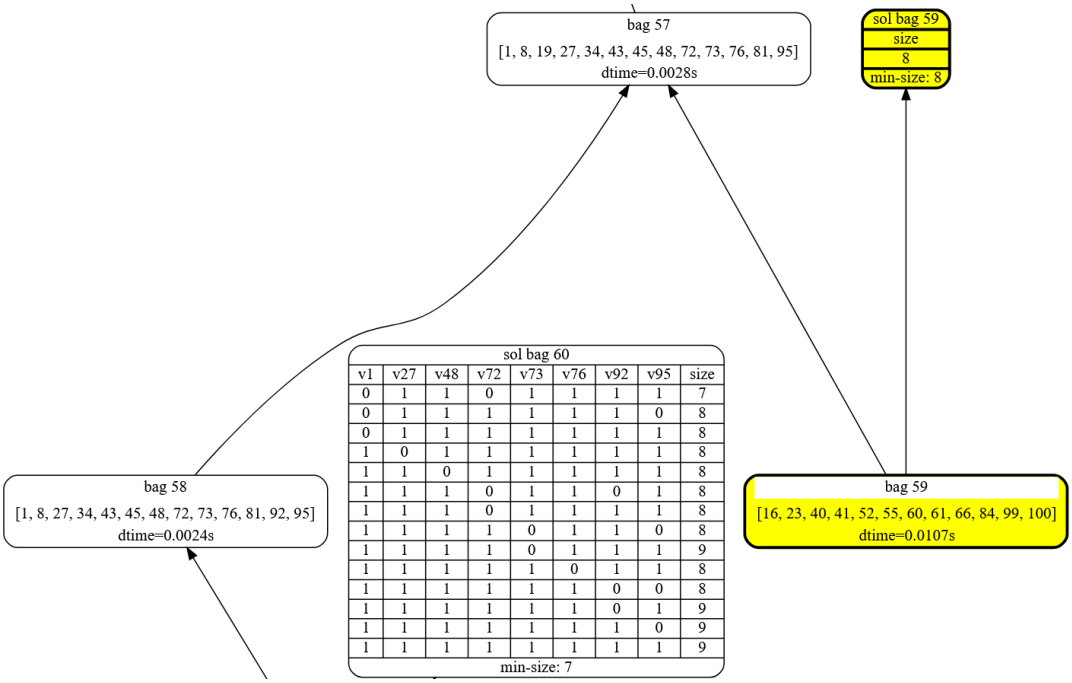


Figure 29: Part of interest to find a problem with bag 59 in visualization.

The problem was located in the version of `https://github.com/TU-Wien-DBAI/htd` used to create the tree decomposition, which in this instance and for the seed 0 did not place bag 59 in the right place in the tree-decomposition on our Windows 10 machine. We can see the bags of the tree decomposition in Figure 30, where the bags containing variable 100 are indicated in red.

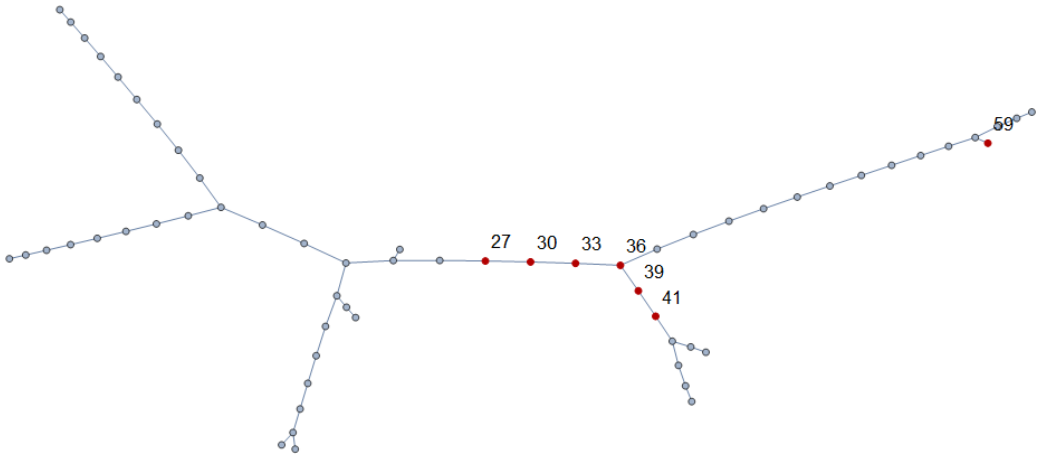


Figure 30: Bag 59 can be seen on the far right of the graphic.

8. Conclusion

8.1. Summary

We created a program which can automate the visualization of dynamic programming on tree decompositions. All tree decomposition nodes are prepared to display multiple user-defined strings, which can contain various debugging information about the run. With SVG we support by default a fast and easily editable image format.

The visualization is implemented in two actively developed solvers for the problem types SAT, #SAT and minimal vertex cover. We have tested the visualization of each problem type with graphs of different sizes - with nodes in the order of $10^{0..2}$ for MinVC and $10^{0..3}$ for SAT and #SAT.

During the development we could already find some bugs in the solvers.

We have defined a data exchange API to give the visualization the information it needs and provided meaningful default values for most parameters. The API supports the interchange of:

- One tree decomposition
- Time steps that add solutions to the TD
- Incidence graphs for problems on boolean formulas, from which primal and dual graphs can be created automatically
- Simple graphs for general graph problems
- The merging of several visualizations of a time step into one image

8.2. Future Work

In the future our graphs could provide even more parameters to the user.

Different colors visualizing attributes of each node are a consideration.

Show path of various solutions from leaf to bag.

The next step would be to expand the API to multiple bipartite or simple graphs, which right now is limited to one each with the option to create primal- and dual-graphs too.

One addition would be the inclusion of hypergraphs (graphs where one edge does connect multiple nodes) which could be of great interest for future solvers.

Right now even with the *svg-join* tool we need multiple files to represent all time steps. SVG however would be able to only create one file where the time steps will be animated. Animations might get toggled by the user or change over a specified time span.

A. Images

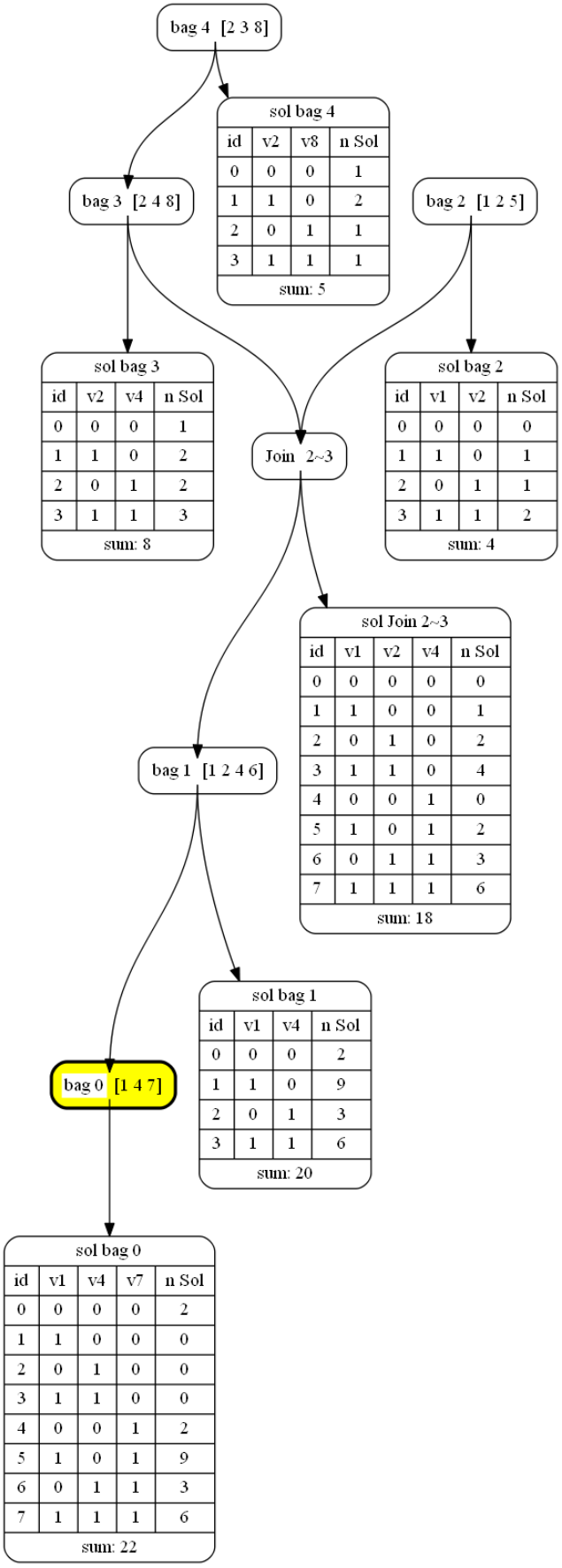


Figure 31: Created scalable-vector-graphic directly from 16

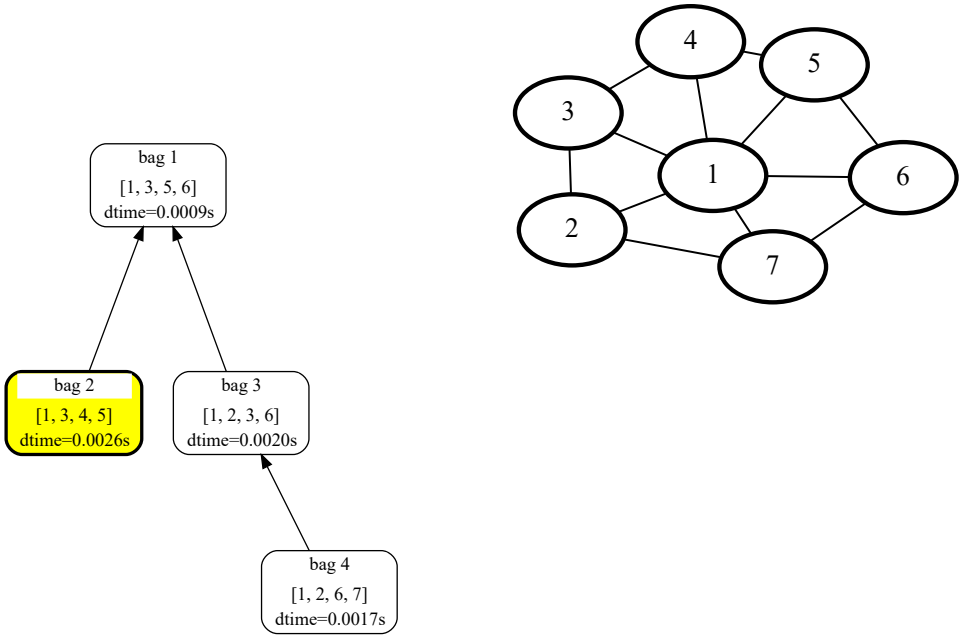


Figure 32: Joining results from Section 7.3 at step 1. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

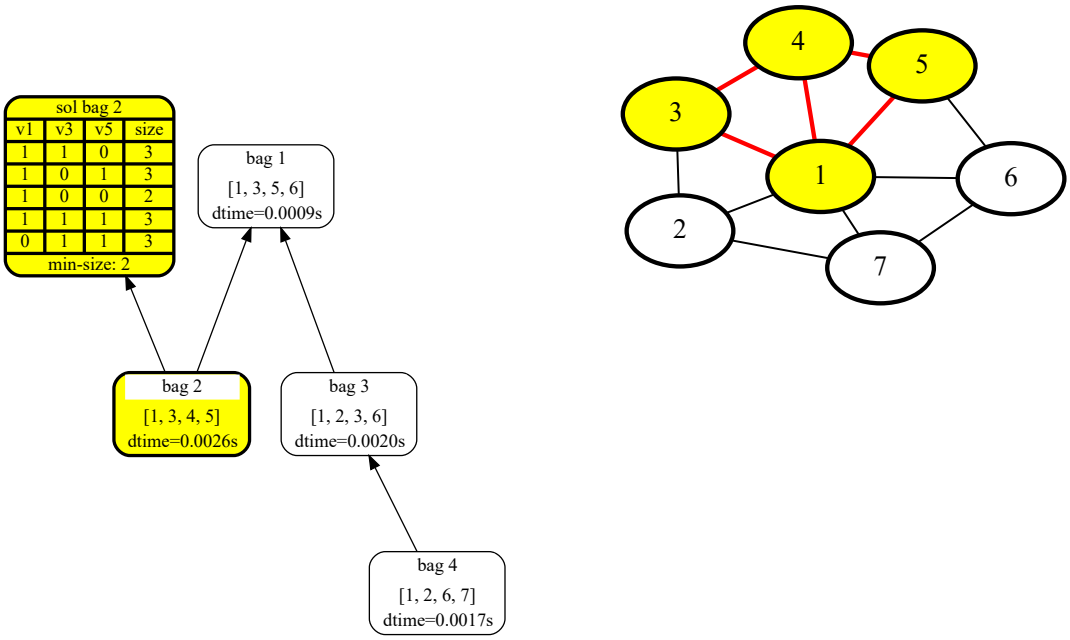


Figure 33: Joining results from Section 7.3 at step 2. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

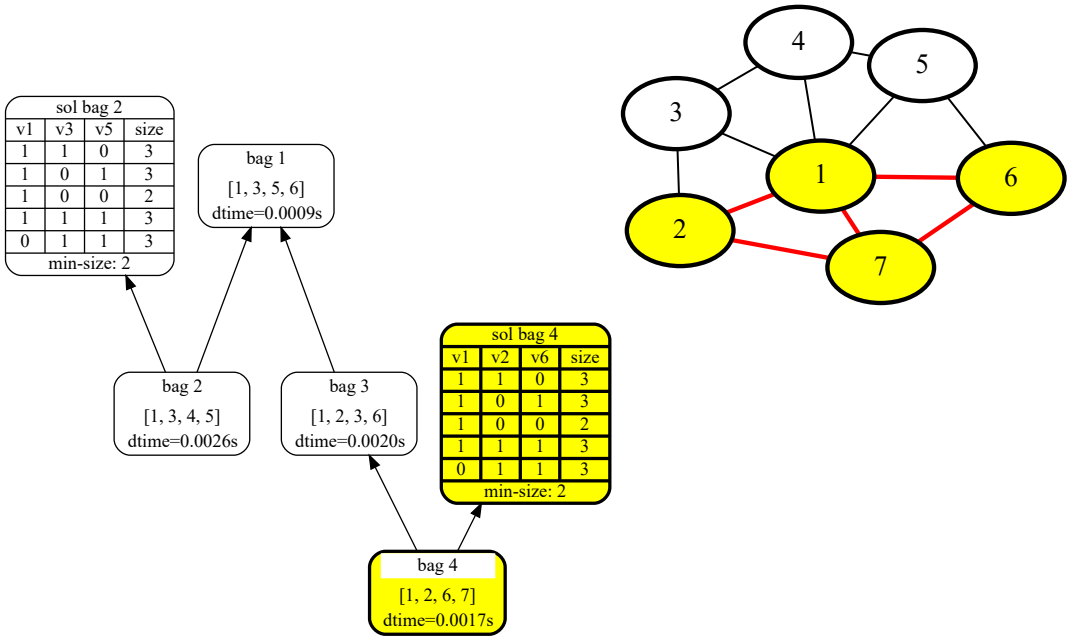


Figure 34: Joining results from Section 7.3 at step 3. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

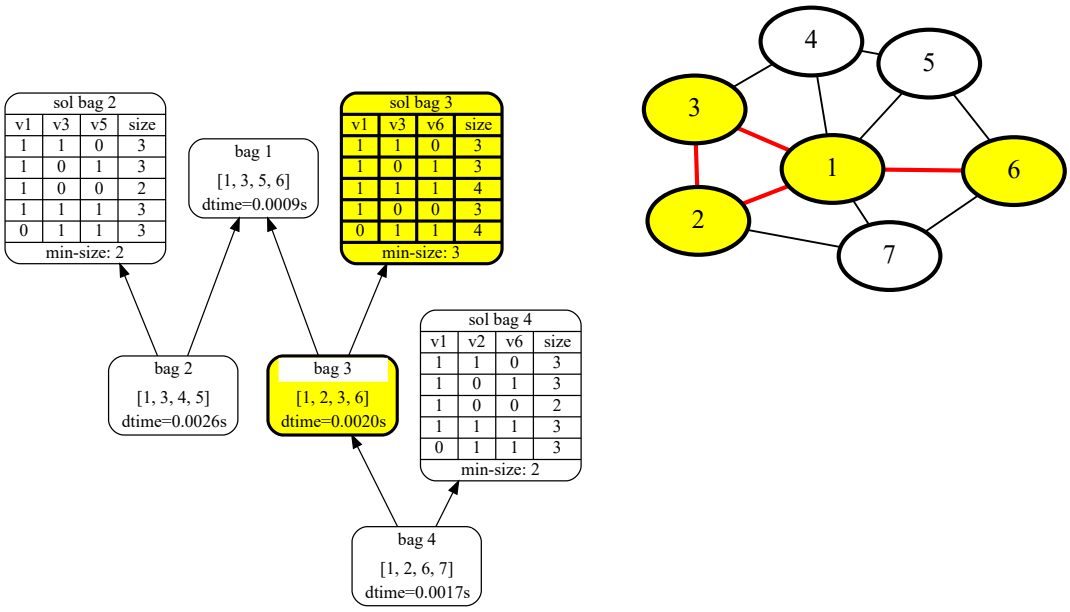


Figure 35: Joining results from Section 7.3 at step 4. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

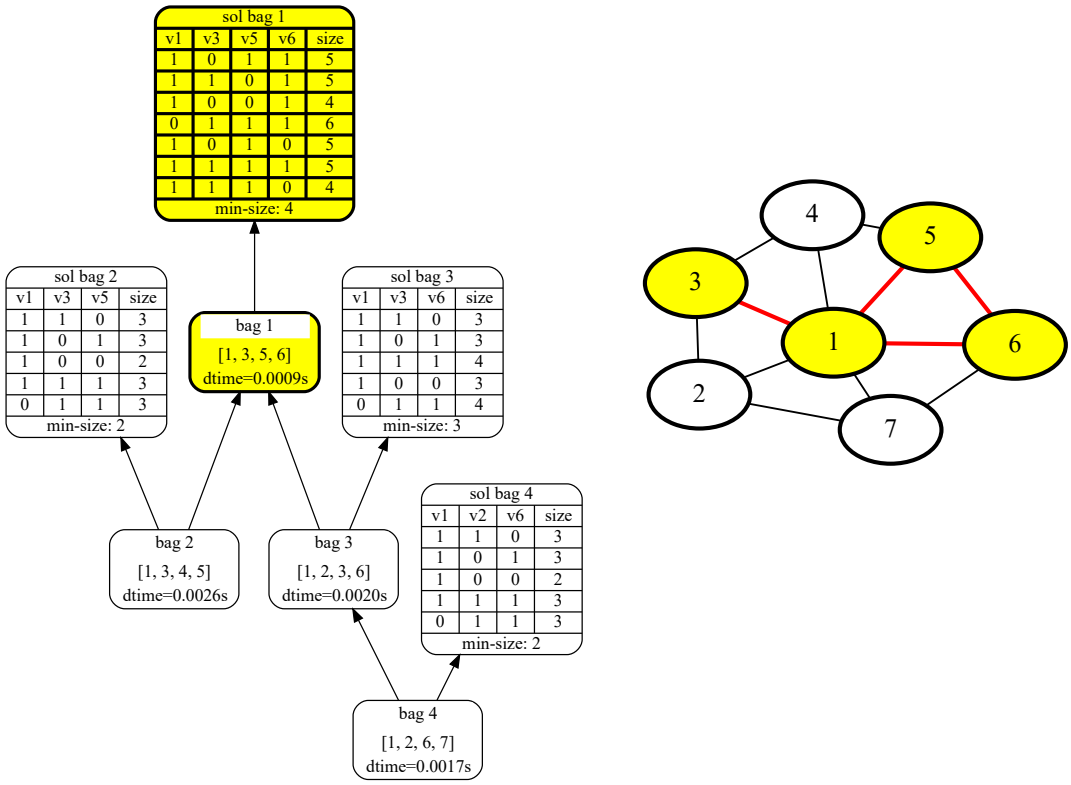


Figure 36: Joining results from Section 7.3 at step 5. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

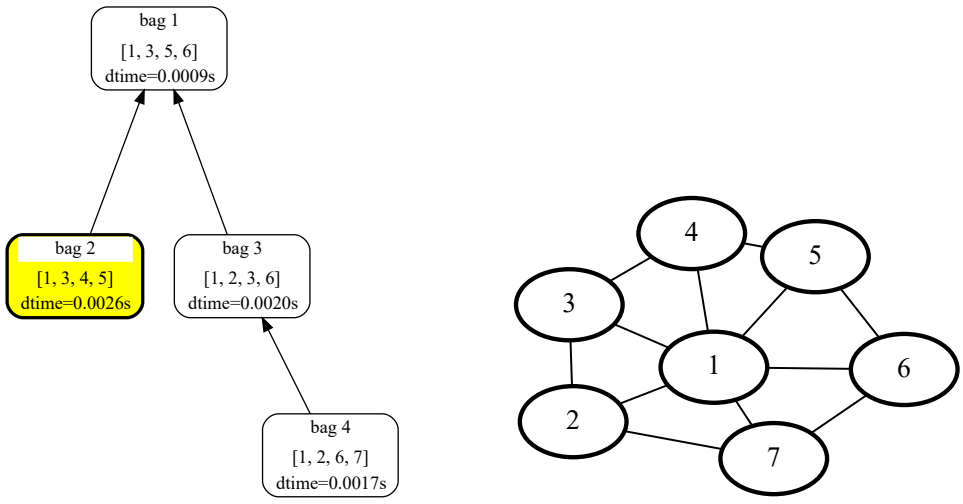


Figure 37: Joining results from Section 7.3 at the first step. Also shifting the second graphic to the bottom edge of the first image and scaling with scale2 = 1.5 .

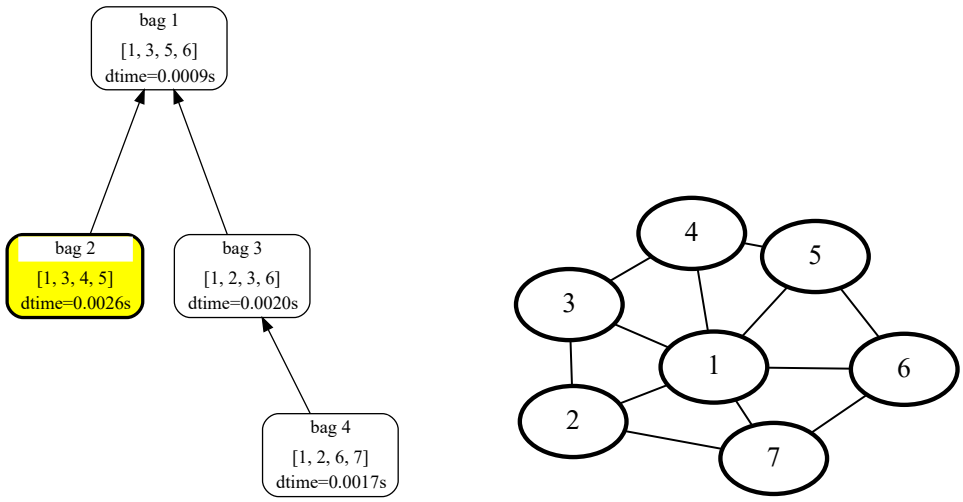


Figure 38: Joining results from Section 7.3 at the second step. Also shifting the second graphic to 15% of the height of the first image and scaling with scale2 = 1.5 .

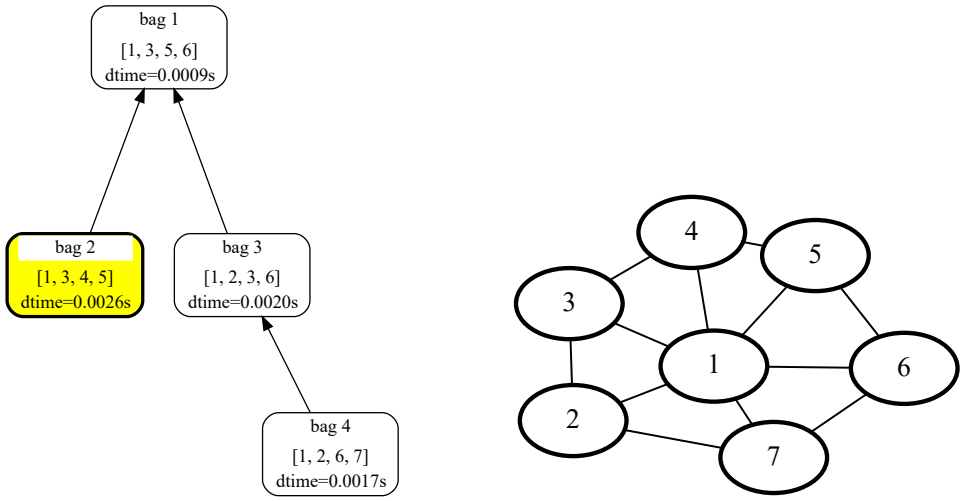


Figure 39: Joining results from Section 7.3 at the third step. Also shifting the second graphic to 30% of the height of the first image and scaling with scale2 = 1.5 .

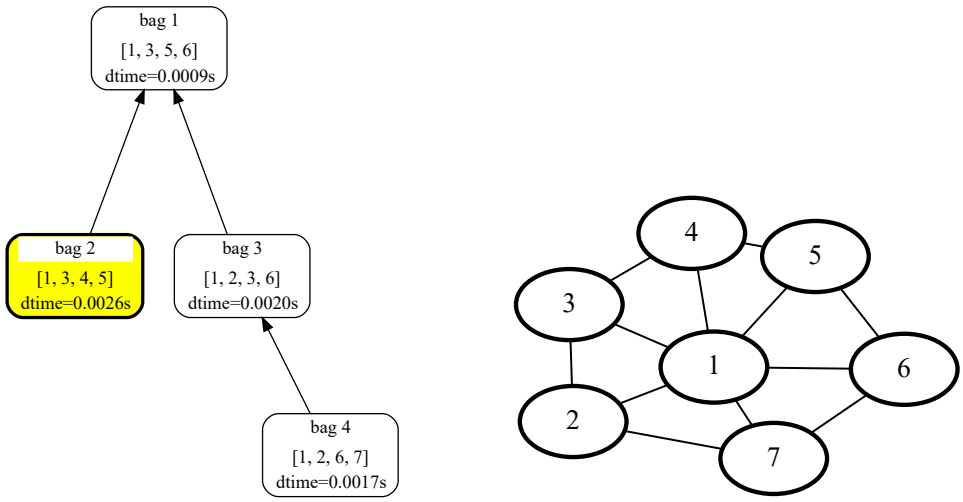


Figure 40: Joining results from Section 7.3 at the fourth step. Also shifting the second graphic to 45% of the height of the first image and scaling with scale2 = 1.5 .

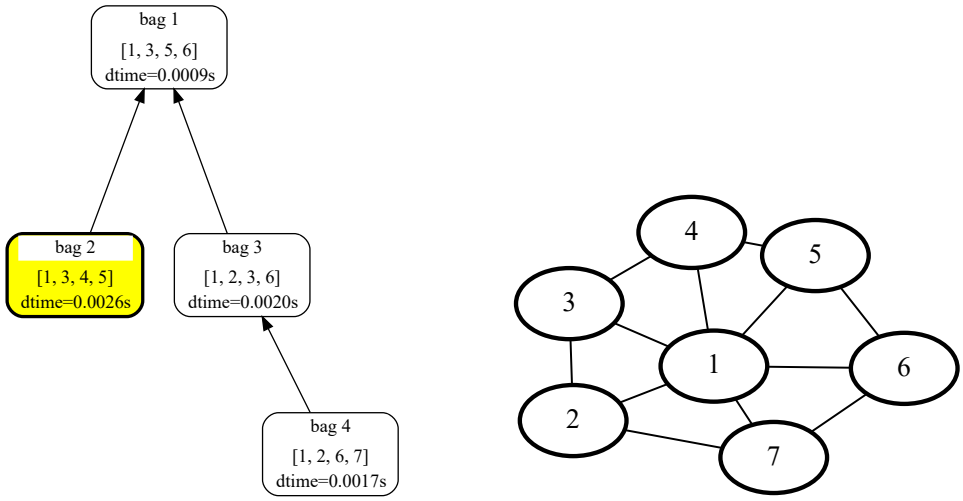


Figure 41: Joining results from Section 7.3 at the final step. Also shifting the second graphic to to 60% of the height of the first image and scaling with scale2 = 1.5 .

B. Code Snippets

Listing 4: The JSON format used to describe MSOL visualization on tree decompositions

```
{
    "incidenceGraph" : false or
    {
        Optional("subgraph_name_one" : STR, default='clauses'),
        Optional("subgraph_name_two" : STR, default='variables'),

        Optional("var_name_one" : STR, default=''),
        Optional("var_name_two" : STR, default=''),

        Optional("infer_primal" : BOOLEAN, default=false),
        Optional("infer_dual" : BOOLEAN, default=false),
        Optional("fontsize" : INT, default=16),
        Optional("second_shape" : STR, default='diamond'),
        Optional("column_distance" : FLOAT, default=0.5),

        "edges" : [
            {"id" : INT (subgraphOneId),
            "list" : [INT...]
            }
            ...
        ]
    },

    "generalGraph" : false or
    {
        Optional("graph_name" : STR, default='graph'),
        Optional("var_name" : STR, default=''),
        Optional("sort_nodes" : BOOLEAN, default=false),
        Optional("need_adj_nodes" : BOOLEAN, default=false),
        Optional("extra_nodes" : LIST, default=[]),
        Optional("fontsize" : INT, default=20),
        Optional("first_color" : STR/COLOR, default ='yellow'),
        Optional("first_style" : STR, default ='filled'),
        Optional("second_color : STR/COLOR, default='green'),
        Optional("second_style : STR, default='dotted,filled'),

        "edges" : [
            [INT, INT],
            ...
        ]
    },

    "tdTimeline" :
    [
```

```
        [INT (bagId)] or
        [INT (bagId) or [INT(bagId), INT(bagId)],
            [[
                [Any],
                [Any],
                ...
            ]
            ,STR (header)
            ,STR (footer)
            ,BOOL (transpose)
            ]
        ]
        ...
    ],

    "treeDecJson" :
    {
        "bagpre" : STR,
        "num_vars" : INT,
        Optional("joinpre" : STR, default= 'Join %d~%d'),
        Optional("solpre" : STR, default= 'sol%d'),
        Optional("soljoinpre" : STR, default= 'solJoin%d~%d'),

        "edgearray" :
            [[INT, INT]...],
        "labeldict" :
            [
                {
                    "id" : INT (bagId),
                    "items" : [ INT... ],
                    "labels" : [ STR... ]
                }
                ...
            ],
    },

    Optional("orientation" : Any['BT', 'TB', 'LR', 'RL'] , default='BT'),
    Optional("linesmax" : INT, default=100),
    Optional("columnsmax" : INT, default=20),
    Optional("bagcolor" : STR, default='white'),
    Optional("fontsize" : INT, default=20),
    Optional("penwidth" : FLOAT, default=2.2),
    Optional("fontcolor" : STR, default='black'),

    Optional("emphasis" : DICT, default=
        {
            "firstcolor" : STR/COLOR, default='yellow',
            "secondcolor" : STR/COLOR, default='green',
            "firststyle" : STR, default='filled',
```

```
            "secondstyle" : STR, default='dotted,filled'
        }
    )

    Optional("svgjoin" :
        {
            "base_names" : [STR],
            Optional("folder" : STR/NULL, default=null),
            Optional("outname" : STR, default='combined'),
            Optional("suffix" : STR, default='%d.svg'),
            Optional("preserve_aspectratio" : STR, default='xMinYMin'),
            Optional("num_images" : INT, default=1),
            Optional("padding" : [INT], default=0),
            Optional("scale2" : [FLOAT], default=1),
            Optional("v_top" : [FLOAT/STR], default='top'),
            Optional("v_bottom" : [FLOAT/STR]/NULL, default=null),
        }
    )
}
```

Listing 5: Initializing a Visualization object

```
def __init__(self, infile, outfolder) -> None:
"""Copy needed fields from arguments and create VisualizationData.
"""
self.data: VisualizationData = self.inspect_json(infile)
self.outfolder = outfolder

self.tree_dec_digraph = None

def inspect_json(self, infile) -> VisualizationData:
"""Read and preprocess the needed data from the infile into
VisualizationData.
"""
LOGGER.debug("Reading␣from:␣%s", infile)
visudata = read_json(infile)
LOGGER.debug("Found␣keys:␣%s", visudata.keys())

try:
_incid = visudata['incidenceGraph']
_general_graph = visudata['generalGraph']
_svg_join = visudata.get('svg_join', None)

incid_data: IncidenceGraphData = None
if _incid:
_incid['edges'] = [[x['id'], x['list']]
for x in _incid['edges']]
incid_data = IncidenceGraphData(**_incid)
visudata.pop('incidenceGraph')
general_graph_data: GeneralGraphData = None
```

```
if _general_graph:
general_graph_data = GeneralGraphData(**_general_graph)
visudata.pop('generalGraph')
svg_join_data: SvgJoinData = None
if _svg_join:
svg_join_data = SvgJoinData(**_svg_join)
if 'svg_join' in visudata:
visudata.pop('svg_join')

self.timeline = visudata['tdTimeline']
visudata.pop('tdTimeline')
self.tree_dec = visudata['treeDecJson']
self.bagpre = self.tree_dec['bagpre']
self.joinpre = self.tree_dec.get('joinpre', 'Join␣%d~%d')
self.solpre = self.tree_dec.get('solpre', 'sol%d')
self.soljoinpre = self.tree_dec.get('soljoinpre', 'solJoin%d~%d')
visudata.pop('treeDecJson')
except KeyError as err:
raise KeyError(f"Key␣{err}␣not␣found␣in␣the␣input␣Json.")
return VisualizationData(incidence_graph=incid_data,
general_graph=general_graph_data,
svg_join=svg_join_data,
**visudata)
```

Listing 6: SvgJoinData

```
@dataclass
class SvgJoinData:
"""Class holding different parameters to join the results."""
base_names: Union[str, Iterable[str]]
folder: Optional[str] = None
outname: str = 'combined'
suffix: str = '%d.svg'
preserve_aspectratio: str = 'xMinYMin'
num_images: int = 1
padding: Union[int, Iterable[int]] = 0
scale2: Union[float, Iterable[float]] = 1.0
v_top: Union[None, float, str,
Iterable[Union[None, float, str]]] = None
v_bottom: Union[None, float, str,
Iterable[Union[None, float, str]]] = None
```

Listing 7: IncidenceGraphData

```
@dataclass
class IncidenceGraphData:
"""Class holding different parameters for the incidence graph."""
  edges: list
  subgraph_name_one: str = 'clauses'
  subgraph_name_two: str = 'variables'
```

```
  var_name_one: str = ''
  var_name_two: str = ''
  infer_primal: bool = False
  infer_dual: bool = False
  primal_file: str = 'PrimalGraphStep'
  inc_file: str = 'IncidenceGraphStep'
  dual_file: str = 'DualGraphStep'
  fontsize: int = 16
  penwidth: float = 2.2
  second_shape: str = 'diamond'
  column_distance: float = 0.5
```

Listing 8: GeneralGraphData

```
@dataclass
class GeneralGraphData:
"""Class holding different parameters for the general graph."""
edges: list
extra_nodes: Optional[list] = None
graph_name: str = 'graph'
file_basename: str = 'graph'
var_name: str = ''
sort_nodes: bool = False
need_adj_nodes: bool = False
fontsize: int = 20
first_color: str = 'yellow'
first_style: str = 'filled'
second_color: str = 'green'
second_style: str = 'dotted,filled'
```

Listing 9: Construct_dpdb_visu.py

```
def create_json(problem: int, tw_file=None, intermed_nodes=False):
  """Create the JSON for the specified problem instance."""
  with connect() as connection:
    # get type of problem
    with connection.cursor() as cur:
        cur.execute("SELECT␣name,type,num_bags␣FROM␣"
        "public.problem␣WHERE␣id=%s", (problem,))
        (name, ptype, num_bags) = cur.fetchone()
    # select the valid constructor for the problem
    constructor: IDpdbVisuConstruct

    if ptype == 'Sat':
      constructor = DpdbSatVisu(
        connection, problem, intermed_nodes)
    elif ptype == 'SharpSat':
      constructor = DpdbSharpSatVisu(
        connection, problem, intermed_nodes)
    elif ptype == 'VertexCover':
```

```
      constructor = DpdbMinVcVisu(
        connection, problem, intermed_nodes, tw_file)

    return constructor.construct()
  return {}
```

Listing 10: forward_iterate_tdg

```
def forward_iterate_tdg(self, joinpre, solpre, soljoinpre) -> None:
"""Create the final positions of all nodes with solutions."""
  tdg = self.tree_dec_digraph                  # shorten name

  for i, node in enumerate(self.timeline):
    if len(node) > 1:
      # solution to be displayed
      id_inv_bags = node[0]
      if isinstance(id_inv_bags, int):
        last_sol = solpre % id_inv_bags
        tdg.node(last_sol, solution_node(
        *(node[1])), shape='record')
        tdg.edge(self.bagpre % id_inv_bags, last_sol)

      else:    # joined node with 2 bags
        suc = self.timeline[i + 1][0]   # get the joined bags

        LOGGER.debug('joining␣%s␣to␣%s␣', node[0], suc)

        id_inv_bags = tuple(id_inv_bags)
        last_sol = soljoinpre % id_inv_bags
        tdg.node(last_sol, solution_node(
        *(node[1])), shape='record')

        tdg.edge(joinpre % id_inv_bags, last_sol)
        # edges
        for child in id_inv_bags:  # basically "remove" current
          tdg.edge(
              self.bagpre % child
              if isinstance(child, int) else joinpre % child,
              self.bagpre % suc
              if isinstance(suc, int) else joinpre % suc,
              style='invis',
              constraint='false')
          tdg.edge(self.bagpre % child if isinstance(child, int)
                   else joinpre % child,
                   joinpre % id_inv_bags)
        tdg.edge(joinpre % id_inv_bags, self.bagpre % suc
          if isinstance(suc, int) else joinpre % suc)
```

Listing 11: backwards_iterate_tdg

```
def backwards_iterate_tdg(self, joinpre, solpre, soljoinpre,
                          view=False) -> None:
  """Cut the single steps back and update emphasis acordingly."""
  tdg = self.tree_dec_digraph     # shorten name
  last_sol = ""

  for i, node in enumerate(reversed(self.timeline)):
    id_inv_bags = node[0]
    LOGGER.debug("%s: Reverse traversing on %s", i, id_inv_bags)

    if i > 0:
      # Delete previous emphasis
      prevhead = self.timeline[len(self.timeline) - i][0]
      bag = (self.bagpre % prevhead if isinstance(prevhead,int)
          else joinpre % tuple(prevhead))
      base_style(tdg, bag)
      if last_sol:
        style_hide_node(tdg, last_sol)
        style_hide_edge(tdg, bag, last_sol)
        last_sol = ""

    if len(node) > 1:
      # solution to be displayed
      if isinstance(id_inv_bags, int):
        last_sol = solpre % id_inv_bags
        emphasise_node(tdg, last_sol)
        tdg.edge(self.bagpre % id_inv_bags, last_sol)
      else:  # joined node with 2 bags
        id_inv_bags = tuple(id_inv_bags)
        last_sol = soljoinpre % id_inv_bags
        emphasise_node(tdg, last_sol)

    emphasise_node(tdg,
      self.bagpre % id_inv_bags if isinstance(id_inv_bags, int) else
      joinpre % id_inv_bags)
    _filename = self.outfolder + self.data.td_file + '%d'
    tdg.render(
      view=view, format='svg', filename=_filename %
      (len(self.timeline) - i))
```

Listing 12: SvgJoinData

```
@dataclass
class SvgJoinData:
  """Class for holding different parameters to join the results."""
  base_names: Union[str, Iterable[str]]
  folder: Optional[str] = None
  outname: str = 'combined'
  suffix: str = '%d.svg'
  preserve_aspectratio: str = 'xMinYMin'
```

```
  num_images: int = 1
  padding: Union[int, Iterable[int]] = 0
  scale2: Union[float, Iterable[float]] = 1.0
  v_top: Union[None, float, str,
               Iterable[Union[None, float, str]]] = 'top'
  v_bottom: Union[None, float, str,
                  Iterable[Union[None, float, str]]] = None
```

C. More Examples

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	0
3	1	1	1	1	0	1	1	1
4	1	1	1	1	0	1	1	0
5	1	1	1	0	1	1	1	0
6	1	1	1	0	1	0	1	0
7	1	1	1	0	0	1	1	0
8	1	1	1	0	0	0	1	0
9	1	1	0	1	1	1	1	0
10	1	1	0	1	0	1	1	0
11	1	1	0	0	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	0	0	1	1	0
14	1	1	0	0	0	0	1	0
15	1	0	1	0	0	0	1	0
16	0	1	1	1	0	1	1	1
17	0	1	1	1	0	1	1	0
18	0	1	1	0	0	1	1	0
19	0	1	1	0	0	1	0	0
20	0	1	0	1	0	1	1	0
21	0	1	0	0	0	1	1	0
22	0	1	0	0	0	1	0	0

Table 5: The satisfying assignments for the formula from Example 3.

Listing 13: CNF clauses from example 4.1 in [Zis18] page 27

```
p cnf 8 10
1 4 6 0
```

```
1 -5 0
-1 7 0
2 3 0
2 5 0
2 -6 0
3 -8 0
4 -8 0
-4 6 0
-4 7 0
```

Listing 14: CNF clauses from random example with 12 units

```
p cnf 18 24
-1 0
-2 0
-3 0
-4 0
-5 0
-6 0
-7 0
-8 0
-9 0
-10 0
-11 0
-12 0
-13 -14 -15 0
-13 -14 16 0
-13 -15 -16 -18 0
-13 -15 -17 0
13 14 16 -17 18 0
13 15 -16 -18 0
-14 -15 16 17 0
-14 15 -17 18 0
-14 15 17 -18 0
-15 -16 -17 18 0
15 -16 -17 -18 0
15 16 17 -18 0
```

Listing 15: Edges for example 17

```
c
p tw 7 12
1 2
1 3
2 3
1 4
3 4
1 5
4 5
1 6
```

```
5 6
1 7
2 7
6 7
```

Listing 16: DOT source for visualization of example 4.1

```
strict digraph g41dot {
  node [fillcolor=white shape=box style="rounded,filled"]
  bag4 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD>bag 4</TD><TD PORT="anchor"></TD>
    <TD>[2 3 8]</TD></TR></TABLE>>]
  bag3 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD BGCOLOR="white">bag 3</TD><TD PORT="anchor"></TD>
    <TD>[2 4 8]</TD></TR></TABLE>>]
  join1 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD BGCOLOR="white">Join</TD><TD PORT="anchor"></TD>
    <TD>2~3</TD></TR></TABLE>>]
  bag2 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD BGCOLOR="white">bag 2</TD><TD PORT="anchor"></TD>
    <TD>[1 2 5]</TD></TR></TABLE>>]
  bag1 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD BGCOLOR="white">bag 1</TD><TD PORT="anchor"></TD>
    <TD>[1 2 4 6]</TD></TR></TABLE>>]
  bag0 [label=<<TABLE BORDER="0" CELLBORDER="0" CELLSPACING="0">
    <TR><TD BGCOLOR="white">bag 0</TD><TD PORT="anchor"></TD>
    <TD>[1 4 7]</TD></TR></TABLE>>]
    node [shape=record]
  sol2 [label="{sol bag 2|{{id|0|1|2|3}|{v1|0|1|0|1}|{v2|0|0|1|1}
    |{n Sol|0|1|1|2}}|sum: 4}"]
  sol4 [label="{sol bag 4|{{id|0|1|2|3}|{v2|0|1|0|1}|{v8|0|0|1|1}
    |{n Sol|1|2|1|1}}|sum: 5}"]
  sol3 [label="{sol bag 3|{{id|0|1|2|3}|{v2|0|1|0|1}|{v4|0|0|1|1}
    |{n Sol|1|2|2|3}}|sum: 8}"]
  solJoin1 [label="{sol Join 2~3|{{id|0|1|2|3|4|5|6|7}
    |{v1|0|1|0|1|0|1|0|1}|{v2|0|0|1|1|0|0|1|1}
    |{v4|0|0|0|0|1|1|1|1}|{n Sol|0|1|2|4|0|2|3|6}}
    |sum: 18}"]
  sol1 [label="{sol bag 1|{{id|0|1|2|3}|{v1|0|1|0|1}|{v4|0|0|1|1}
    |{n Sol|2|9|3|6}}|sum: 20}"]
  sol0 [label="{sol bag 0|{{id|0|1|2|3|4|5|6|7}
    |{v1|0|1|0|1|0|1|0|1}|{v4|0|0|1|1|0|0|1|1}
    |{v7|0|0|0|0|1|1|1|1}|{n Sol|2|0|0|0|2|9|3|6}}|sum: 22}"]
  bag4:anchor -> bag3:anchor
  bag2:anchor -> join1:anchor
  bag3:anchor -> join1:anchor
  join1:anchor -> bag1:anchor
  bag1:anchor -> bag0:anchor
  bag4:anchor -> sol4
  bag3:anchor -> sol3
```

```
  bag2:anchor -> sol2
  bag1:anchor -> sol1
  bag0:anchor -> sol0
  join1:anchor -> solJoin1
  bag0:anchor -> sol0
  bag0 [fillcolor=yellow penwidth=2.5]
}
```

Listing 17: Small GML output from class *Graphoutput*, Section 5.1

```
graph
[
  node
  [
   id 0
   label "bag 0 var: [ 1, 3, 5 ]"
  ]
  node
  [
   id 1
   label "bag 1 var: [ 1, 2, 5 ]"
  ]
  node
  [
   id 2
   label "bag 2 var: [ 1, 2, 4 ]"
  ]
  edge
  [
   source 0
   target 1
  ]
  edge
  [
   source 1
   target 2
  ]
]
```

References

[ACP87] Stefan Arnborg, Derek Corneil, and Andrzej Proskurowski. "Complexity of Finding Embeddings in a k-Tree". In: *SIAM J. Alg. Disc. Meth.* 8 (Apr. 1987), pp. 277–284. DOI: 10.1137/0608024.

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. "Easy problems for tree-decomposable graphs". In: *Journal of Algorithms* 12.2 (1991), pp. 308–340. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(91)90006-K. URL: http://www.sciencedirect.com/science/article/pii/019667749190006K.

[AMW17] Michael Abseher, Nysret Musliu, and Stefan Woltran. "htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond". In: May 2017, pp. 376–386. ISBN: 978-3-319-59775-1. DOI: 10.1007/978-3-319-59776-8_30.

[Bag06] Guillaume Bagan. "MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay". In: *Computer Science Logic*. Ed. by Zoltán Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 167–181. ISBN: 978-3-540-45459-5.

[BB06] Emgad Bachoore and Hans Bodlaender. "A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth". In: June 2006, pp. 255–266. DOI: 10.1007/11775096_24.

[BPW16] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. "Implementing Courcelle's Theorem in a declarative framework for dynamic programming". In: *Journal of Logic and Computation* 27.4 (Jan. 2016), pp. 1067–1094. ISSN: 0955-792X. DOI: 10.1093/logcom/exv089. eprint: https://academic.oup.com/logcom/article-pdf/27/4/1067/17659880/exv089.pdf. URL: https://doi.org/10.1093/logcom/exv089.

[Bro+15] I.N. Bronshtein et al. *Handbook of Mathematics*. English. 6th ed. Berlin: Springer Verlag Berlin Heidelberg, 2015. 1207 pp. ISBN: 978-3-662-46220-1. DOI: 10.1007/978-3-662-46221-8. eprint: 978-3-662-46221-8.

[Car17] Stephan C. Carlson. *graph theory — Problems & Applications — Britannica*. May 2017. URL: https://www.britannica.com/topic/graph-theory (visited on 07/02/2020).

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. 1st ed. Cambridge: Cambridge University Press, 2012. ISBN: 978-0-521-89833-1.

[Dai+05] Jason Daida et al. "Visualizing Tree Structures in Genetic Programming". In: *Genetic Programming and Evolvable Machines* 6 (Mar. 2005). DOI: 10.1007/s10710-005-7621-2.

[Die07] Stephan Diehl. *Software Visualization. Visualizing the Structure, Behaviour, and Evolution of Software*. English. Springer, 2007. 199 pp. ISBN: 978-3540465041.

[DIM20] DIMACS. *DIMACS :: Implementation Challenge*. 2020. URL: http://dimacs.rutgers.edu/programs/challenge/ (visited on 06/11/2020).

[FHZ19] Johannes Fichte, Markus Hecher, and Markus Zisser. "An Improved GPU-Based SAT Model Counter". In: Sept. 2019, pp. 491–509. ISBN: 978-3-030-30047-0. DOI: 10.1007/978-3-030-30048-7_29.

[Fic+20] Johannes Fichte et al. "Exploiting Database Management Systems and Treewidth for Counting". In: Jan. 2020, pp. 151–167. ISBN: 978-3-030-39196-6. DOI: 10.1007/978-3-030-39197-3_10.

[Fic19] Johannes K. Fichte. *Parameterized Complexity and its Applications in Practice. From Foundations to Implementations*. pdf. Summer 2019 (May 6th – May 16th). Jakarta, Indonesia: TU Dresden, Germany, May 6, 2019, pp. 162–174.

[GD12] Vibhav Gogate and Rina Dechter. "A Complete Anytime Algorithm for Treewidth". In: *UAI* (July 2012).

[Gol+06] Andrew V. Goldberg et al. *Efficient Point-to-Point Shortest Path Algorithms*. https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPPshortestpathalgorithms.pdf. [Online; accessed 14-June-2020]. 2006.

[Gro99] Martin Grohe. "Descriptive and Parameterized Complexity". In: *Computer Science Logic*. Ed. by Jörg Flum and Mario Rodriguez-Artalejo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 14–31. ISBN: 978-3-540-48168-3.

[Him10] Michael Himsolt. *GML: A portable Graph File Format*. 2010.

[HJW14] Marie-Christin Harre, Jan Jelschen, and Andreas Winter. "ELVIZ: A query-based approach to model visualization". In: *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)* (Jan. 2014), pp. 105–120.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[HTW20] Markus Hecher, Patrick Thier, and Stefan Woltran. "Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology". In: June 2020, pp. 343–360. ISBN: 978-3-030-51824-0. DOI: 10.1007/978-3-030-51825-7_25.

[Hu05] Yifan Hu. "Efficient and high quality force-directed graph drawing". In: *Mathematica Journal* 10 (Jan. 2005), pp. 37–71.

[JGJ13] Bevan Keeley Jones, Sharon Goldwater, and Mark Johnson. "Modeling Graph Languages with Grammars Extracted via Tree Decompositions". In: *Proceedings of the 11th International Conference on Finite State Methods and Natural Language Processing*. St Andrews, Scotland: Association for Computational Linguistics, July 2013, pp. 54–62.

[KAI11] KAIST. *On the constructive power of monadic second-order logic*. English. Oct. 2011. URL: https://www.youtube.com/watch?v=hZI-wANHO1w (visited on 06/11/2020).

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*. Jan. 1994. ISBN: 3-540-58356-4. DOI: 10.1007/BFb0045375.

[Lan+12] Alexander Langer et al. "Evaluation of an MSO-solver". In: *Proc. of ALENEX 2012* (Jan. 2012). DOI: 10.1137/1.9781611972924.5.

[Neo16] Inc. Neo4j. *Graph Database Use Cases and Solutions*. Aug. 2016. URL: http://neo4j.com/use-cases (visited on 06/11/2020).

[Ove91] Scott P. Overmeyer. "Revolutionary vs. Evolutionary Rapid Prototyping: Balancing Software Productivity and HCI Design Concerns". In: *Proceedings of the Fourth International Conference on Human-Computer Interaction*. Elsevier Science, Sept. 1991, pp. 303–308.

[RWE15] Ian Robinson, Jim Webber, and Emil Eifrém. *Graph Databases. New Opportunities for Connected Data*. English. 2nd ed. O'Reilly Media, June 10, 2015. 365 pp. ISBN: 978-1491930892.

[SS10] Marko Samer and Stefan Szeider. "Algorithms for propositional model counting." In: *J. Discrete Algorithms* 8 (Jan. 2010), pp. 50–64.

[ST99] Janet M. Six and Ioannis G. Tollis. "A Framework for Circular Drawings of Networks". In: *Graph Drawing*. Ed. by Jan Kratochvíyl. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 107–116. ISBN: 978-3-540-46648-2.

[Web20] MDN Web Docs. *SVG: Scalable Vector Graphics — MDN*. Apr. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/SVG (visited on 07/02/2020).

[Zis18] Markus Zisser. *Solving the #SAT problem on the GPU with dynamic programming and OpenCL*. English. Technische Universität Wien, 2018.