Technische Universität Dresden
Fakultät Informatik

Bachelor Thesis

Visualizing Dynamic Programming on Tree Decompositions

Author:
Martin Röbke

Supervisor:
Dr. Johannes Fichte

International Center For Computational Logic

July 3, 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
Matrikelnummer 3949819
Geburtsdatum 04.03.1995

TU Dresden E-Mail-Adresse:
Martin.Roebke@tu-dresden.de

Dresden, ...

(Unterschrift)

Abstract

"A picture is worth a 100 spreadsheets" - better Analysis

Answering questions that be expressed using graph theory is increasingly interesting in scientific work. Many problems such as Boolean satisfiability or problems related to traffic can be translated to and solved on a graph. 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. 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 and allows for faster solving times on larger instances.

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.

Contents

1. Introduction

1.1. Motivation - WIP

Graphs are increasingly interesting in scientific work, as the applications of interconnected datasets grow. Some use cases include fields of interest like

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

As illustrations of the possibilities for an application, smaller examples from the problem-types "SAT", "#SAT" and "Minimal Vertex Cover" are presented, as well as an example of a faulty tree-decomposition that occurred during development. Intended audience:

- Developer of dynamic programming on tree decompositions for debugging.
- Researcher of such algorithms for comparisons and visualizations.
- Teachers or students looking for automatic visualization of their examples and the dynamic programming.

The idea for this project comes from my supervisor Dr. Johannes Fichte, who works with many projects such as dpdb on solving monadic second order logic (MSOL) problems using highly parallelized architectures like graphics processing units or state of the art databases. One early implementation is published in [Lan+12] where for different real world examples the results looked promising These projects are very competitive ????REF????? for solving even large instances of those problems.

My experience with the topics of this work comes mainly from the two courses:

- Visualization with python from the lecture "Computational Physics" by Prof. Dr. A. Bäcker, chair of computational physics, TU Dresden 2016
- algorithms and various manipulations on graphs from the lecture "Graph Data Management and Analytics" by Hannes Voigt. [VK19]

1.2. Concept - WIP

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

1. How could a prototype visualization 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. The code is written in python and published on pypi.

Our software TDVisu works as follows:

1. We generate data using the solver or extra software like the one provided in our construct_dpdb_visu.py and arguments for the visualization into one file.
2. We read the data and parameters specified in a JsonAPI.md conform file.
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 source code for TDVisu is available under GPL3 license.

We chose Graphviz as an open source graph visualization software which offers customizable visualization for directed and undirected graphs and has good interfaces to different programming languages.

We defined a JSON-format specification for portability and customization of the visualization combined in one human-readable file and two reference implementations in practical solvers. 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 compressed and modified very easily without loss of quality [Web20].

1.3. Related Work - WIP

intro. mit motivation und related work, state of the art, advancements.

[Lan+12].

Visualization Pipeline

1.4. Thesis Outline - WIP

2. Background

In this chapter we provide a brief background for this work.

We will start with a short introduction to MSO. Then we describe BFS as a basis for SAT and SSAT problems which are a popular application for solvers. Then we introduce Courcelle's theorem, which describes important properties of the problems and thus possibilities of the solvers. Tree decompositions are also described as a basis for dynamic programming in this domain. At the end of the chapter the input and intermediate formats DIMACS and DOT are briefly described.

2.1. Monadic Second-Order Logic

Monadic Second-Order (MSO) logic is a logical language that is suitable for expressing numerous graph properties [CE12].

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 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].

Propositional logic, also called propositional calculus or zeroth-order logic, is a formal system $\mathcal{L} = \mathcal{L}(\mathrm{A}, \Omega, \mathrm{Z}, \mathrm{I})$ with

- the alpha set A as a countably infinite set of elements called proposition symbols or propositional variables. These are also called terminal elements.
- The omega 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.

  In the more familiar propositional calculi, Ω is typically partitioned as follows:

  $\Omega_0 = \{\bot, \top\}$.

  $\Omega_1 = \{\neg\}$,

  $\Omega_2 \subseteq \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.
- The zeta set Z is a finite set of transformation rules that are called inference rules when they receive logical applications.
- The iota set I is a countable set of initial points that are called axioms when they receive logical interpretations.

First-order logic further adds relations and quantifiers to the propositional logic. As a short description of both concepts we say that quantifier symbols add

- $\exists$ for the existential quantification and

- $\forall$ that expresses that a propositional function can be satisfied by every member of of a domain.

The non-logical symbols now represent relations, functions and constants on the domain of discourse.

A **relation symbol** with some arity (number of arguments) $\in \mathbb{N}$. Relations of arity 0 can be identified with propositional variables. Examples for practical interpretations for 2-ary (binary) relations are $\boldsymbol{x}$ *greater than* $\boldsymbol{y}$, $\boldsymbol{x}$ *less than* $\boldsymbol{y}$, $\boldsymbol{x}$ *divides* $\boldsymbol{y}$, $\boldsymbol{x}$ *subset of* $\boldsymbol{y}$ depending on the current domain.

A **function symbol**, with some arity $\in \mathbb{N}$. Any function $f(t_1, ..., t_n)$ of n arguments (where each argument t_i is a term and f is a function symbol of arity n) is a term. Function symbols of valence 0 are called constant symbols, and are often denoted by lowercase letters from the beginning of the alphabet a, b, c,... .

First-order logic quantifies only 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. Quantification over functions, owing to the equivalence to relations as described above, is thus also not allowed.

There are efficient enumeration algorithms [Bag06] and counting the number of solutions [ALS91] 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. MSO graph properties are "fixed-parameter-tractable" with respect to clique-width and tree-width. So are MSO counting and optimizing functions. MSO logic can express graph properties and mappings from (labeled) graphs to (labeled) graphs [11]. **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.

Theorem 1
An algorithm is FPT if it takes time $f(k) \cdot n^c$ for some constant c and a fixed function f depending only on $k \in \mathbb{N}$. The size of the input is n. The value k is a parameter of the input, in our case usually tree-width. This algorithm is then usable for small values of k.

2.2. Boolean satisfiability problem

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, $\alpha : var(C) \to 0, 1$. 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 os satisfied by the assignment.
A set C of clauses is

- *unfalsifiable if there is an assignment that falsifies all clauses in it.* This can only exist when there exists a variable $v \in var(C)$ such that $v \in C$ and $\neg a \in C$.
- *falsifiable* if there is an assignment that falsifies all clauses in C.
- *satisfiable* if there is an assignment that safisfies all clauses in C.
- *unsatisfiable* if there does not exist an assignment that safisfies all clauses in C.

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.

For more details see for example chapter 2.1 of [Zis18].

Example 1. For visualization samples we will use the formula with the following clause set: $C = \{c_1 = \{v_1, v_4, v_6\}, c_2 = \{v_1, \neg v_5\}, c_3 = \{\neg v_1, v_7\}, c_4 = \{v_2, v_3\}, c_5 = \{v_2, v_5\}, c_6 = \{v_2, \neg v_6\}, c_7 = \{v_3, \neg v_8\}, c_8 = \{v_4, \neg v_8\}, c_9 = \{\neg v_4, v_6\}, c_{10} = \{\neg v_4, v_7\}\}$

2.3. Courcelle's Theorem

Courcelle's theorem - see also [CE12] page 54:

Theorem 2
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 polynom p decidable.

A generic workflow implementing Courcelle's theorem looks like we see in figure 1.

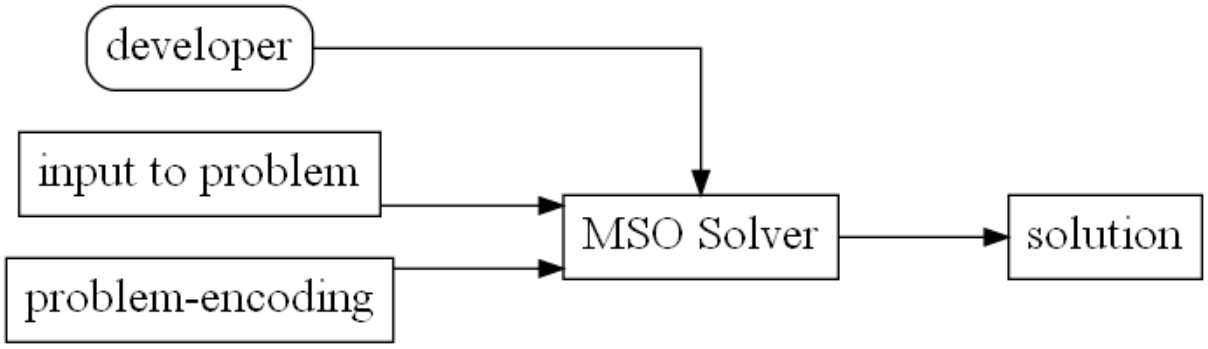


Figure 1: Implementation of the theorem

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 still 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.

2.4. 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 example "wheelgraph" 14 can be viewed as a listing in 16. 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.

2.5. 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. File formats for these purposes were developed at "DIMACS" (the Center for Discrete Mathematics and

Theoretical Computer Science) 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 at C.

2.6. DOT format - WIP

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 as a text based file format for describing graphs, and the Graphviz project includes gml2gv and gv2gml as two tools that can convert between GML and DOT files.

Dot Language: http://www.graphviz.org/doc/info/lang.html

The nodes in the dot-language are *labeled*, so creating a node takes one string identifier and can additionally be provided a string label. Valid examples for IDs include: a, b, A1, node1. The complete abstract grammar for DOT can be viewed at the DOT language.

It supports directed (*digraph* with edges indicated by '-¿') an undirected (*graph* with edges indicated by '--') graphs. The visualizations presented here are constructed as undirected graphs, but would be easily extendable to directed representations since almost all operations keep the order of edge-endpoints given as input.

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 and sorted by node-label to easier find single nodes in potentially large clusters.

DOT has different components that can be modified with attributes: edges, nodes, the root graph, subgraphs and cluster subgraphs, respectively.

In the following we want to specify the attributes we used the most in TDVisu.

- 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 `https://graphviz.org/doc/info/attrs.html#k:color` `https://graphviz.org/doc/info/colors.html` If the value is a colorList, a gradient fill is used. By default, this is a linear fill; setting style=radial will cause a radial fill.
- fontcolor Color used for text.
- style for nodes `https://graphviz.org/doc/info/attrs.html#d:style` 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 not customizable, node attr margin='0.11,0.01' `https://graphviz.org/doc/info/attrs.html#d:margin`
- fontsize Font size, in points, used for text.
- penwidth Specifies the width of the pen, in points, 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 https://graphviz.org/doc/info/shapes.html

3. My Visualization Project

Python because: Rich dependency environment. Fast prototyping. Simple tooling for debugging (pdb), static analysis (mypy), code-style (pylint, autopep8), packaging (pip, pypi).

Python 3.8 because: Python 3.8 was the newest python version at the beginning of the project, released on October 14th 2019. The change applied most times in this project would be f-string support for shorter and easier to read string-building - for a longer list see summary of release highlights.

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 24. 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 4.1.

The first steps were in `https://github.com/VaeterchenFrost/gpusat-VISU` and the first releases of the source code outsourced to`https://github.com/VaeterchenFrost/tdvisu`

The objective of this project was/is to support the visualization mainly to document and improve the development efforts of dynamic programming on tree decompositions.

The tree decompositions in every tested application were provided by the utility `https://github.com/mabseher/htd` (small but efficient C++ library for computing (customized) tree and hypertree decompositions).

3.1. 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.

3.2. Initialization and Tree Decomposition

After the configuration we instantiate a Visualization object as shown in listing 1 , which parses the VisualizationData with the help from our inspect_json method.

The main purpose of the initialization is parsing the input file containing visualization information. This is encapsulated in *read_json*.

Next we want to extract information into two places:

- 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
- *VisualizationData* containing the data for
  - IncidenceGraphData in listing 3
  - GeneralGraphData in 4
  - SvgJoinData in 10
  - adjustable parameters affecting the visuals of the visualization

Listing 1: 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 2: 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 3: 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: Union[float, str] = 2.2
  second_shape: str = 'diamond'
  column_distance: float = 0.5
```

Listing 4: GeneralGraphData

```
@dataclass
class GeneralGraphData:
  """Class holding different parameters for the general graph."""
  edges: list
  extra_nodes: Optional[list] = None
  graph_name: str = 'graph'
```

```
7    file_basename: str = 'graph'
8    var_name: str = ''
9    sort_nodes: bool = False
10   need_adj_nodes: bool = False
11   fontsize: int = 20
12   first_color: str = 'yellow'
13   first_style: str = 'filled'
14   second_color: str = 'green'
15   second_style: str = 'dotted,filled'
```

Next we call the method *Visualization.tree_dec_timeline* that will start the visualization. First, a quick setup is performed for 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 longest calculation when iterating over the time steps, adding the provided solutions and the edges connecting them to the existing bags. We do this in two passes, one in which we put all the nodes in their final position and one in which we create the final time step images. A special case occurs when two bags are joined into a new bag.

In this case, we remove all old edges between the children and the parent node, add the link result to the graph, and add edges from the children to the link result and from the link result to the parent node. Details of this function can be seen in listing 8.

An automatically inserted join node is shown in figure 24. The provided data for this example to layout the bags is listing5. Here we see that bags 2 and 3 have an edge to bag 1:

Listing 5: Structure provided for bags of example 24

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

The second run iterates backwards over all time steps to hide later time steps and emphasize the current node. When rendering the graphs there is an added option to automatically *view* the result (disabled by default). Details of this function can be seen in listing 9.

3.3. Create time steps for the underlying graph

To get a more comprehensive insight into the solving process we decided to also highlight the parts of graphs that best describe the problem instance the solver worked on.

Because the data in the API does not directly include details about highlights in those graphs, we will construct this information on the fly.

First we select only bag ids from the timeline provided that represent an IF-operation. With additional data from *IncidenceGraphData* we are able to reconstruct the

- incidence graph,
- primal graph,
- dual graph

for Boolean formulas. With input from *GeneralGraphData* we can construct a simple graph that should include the nodes we find 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 the graph as well. 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 2. This happens with no pre-processing and if the variable g is only included in the unit c_3.

Figure 2: Disconnected (dual) graph

3.4. Incidence Graph

The incidence graph is a bipartite graph that we present in a way that creates a one to one correspondence with the Boolean formula it does represent. For an example of an incidence graph visualization see 25

This bipartite graph is prepared with good default values, but is customizable in many parameters. Those values are:

1. *colors*, an iterable of colors that is used to color different nodes
2. *inc_file*, basis for the file created that gets appended with the step number
3. *view*, could automatically open the generated files with the default program
4. *fontsize*, the size of all text in this graph
5. *penwidth*, width of the lines around nodes

6. *basefill*, filling of the background for nodes
7. *sndshape*, shape of the nodes with variables
8. *neg_tail*, the shape of the edge-tail indicating a negated variable
9. *var_name_one/two*, prefix for nodes in the left (right) partition
10. *column_distance*, the distance between both partitions

We create the graph, add the necessary arguments and two sub-graphs. The first subgraph is called *cluster_clause* with it's label *clauses*. We add the clauses with their clause-ids starting at one sorted in ascending order from top to bottom.

The second sub-graph we call *cluster_ivar* labeled *variables* gets all variables added to it, starting from variable-id one. This sub-graph does get the different provided *colors* applied to its nodes and their adjacent edges.

The last step in this method is the highlighting of "active" parts in each time step beeing processed during it's dynamic programming. To accomplish this with as small overhead as possible, we apply and remove additional lines to the body of its graph source code. For highlighting there are two main cases:

- There is no active clause in this step: we only reset highlighting
- Else: we also create new highlighting for clauses, variables and edges

In each case we create one image after the step to provide the inside generated.

3.5. General Graph

The so called "general graph" can represent the underlying graph for different problems, as well as primal and dual graph for Boolean formulas. Because of its larger area of application the general graph was not as easy to layout as 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 set to not overlap nodes and easily identify each node, 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. Of course these layouts could use even more configuration than what was set for this version, a complete overview is provided in the graphviz-doc.

Only if the option to sort the nodes was chosen with *do_sort_nodes* the method

1. saves the current graph source

2. sets the layout engine to *circo*
3. adds edges between successive node-ids to form a closed circle
4. runs the layout and reads back the output in plain dot-format
5. reads the calculated positions for each node in the graph
6. resets the source to step one, removing all temporary additions
7. writes the calculated positions into each node
8. sets the layout engine to an engine that uses those calculated positions later.

We add the edges, and eventually the isolated nodes also, to the graph. Then the highlighting for each time-step is done basically like in the incidence graph 3.4. We allow one additional option that would depend on the concrete algorithm being visualized, and that is *do_adj_nodes*. In case the algorithm uses adjacent nodes they can be visualized with this flag using a third color.

EXAMPLES

In each case we create one image after the time step to provide the inside generated.

3.6. 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. To support this functionality and some basic scaling and adjusting, there is a special key in the API for joining the svg-graphs. The functionality is placed in the file *svgjoin.py* and will be called with the specifications given in the optional dictionary **svgjoin** within the JsonAPI.

We transform this information into the python *dataclass* **SvgJoinData** 10. The internally used method *svgjoin.append_svg* joins two images horizontally in each step. With default settings it will align the top of both images and apply no scaling to either of them. The possible parameters (in [unit])

- centerpad: float = 0, [image coordinates], just "padding" in the JSON API
- 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 and easy to use vertical, horizontal and scaling transformations. Note that the images are placed in a Cartesian coordinate system and its origin is the top left corner of its 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 coordinate system, we make sure to correctly set *v_top* to the smaller and *v_bottom* to the larger one if this is possible.

One special case is achieved by setting both these parameters 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 centering both images vertically.

Possibilities for joining images with these four parameters include:

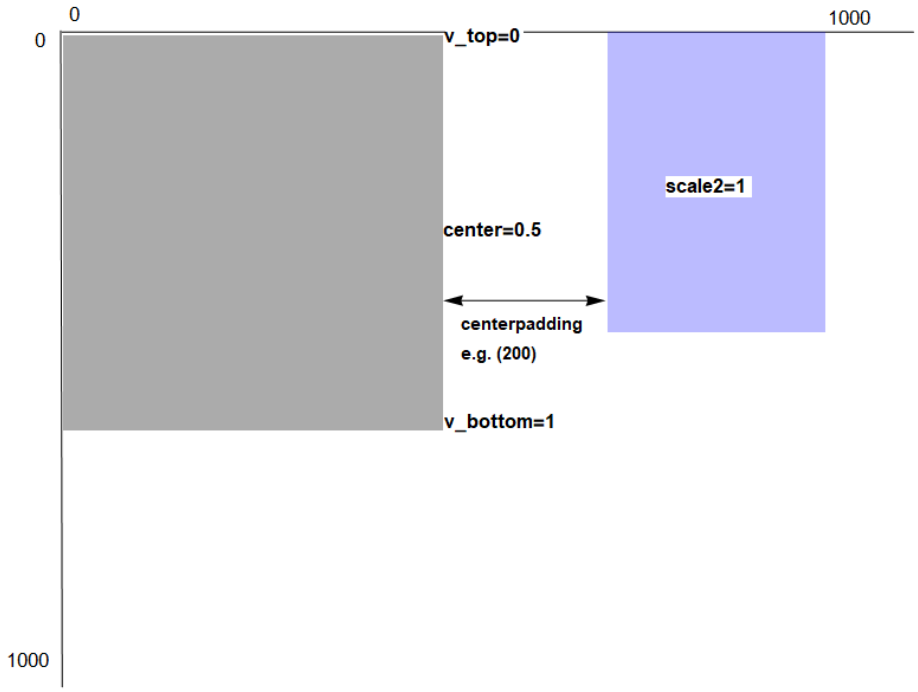


Figure 3: Joining the blue (right side) image to the left gray image with only one parameter *centerpad* set to 200. We see the default vertical position v_top=0, and the implied coordinate system with an origin in the top left corner.

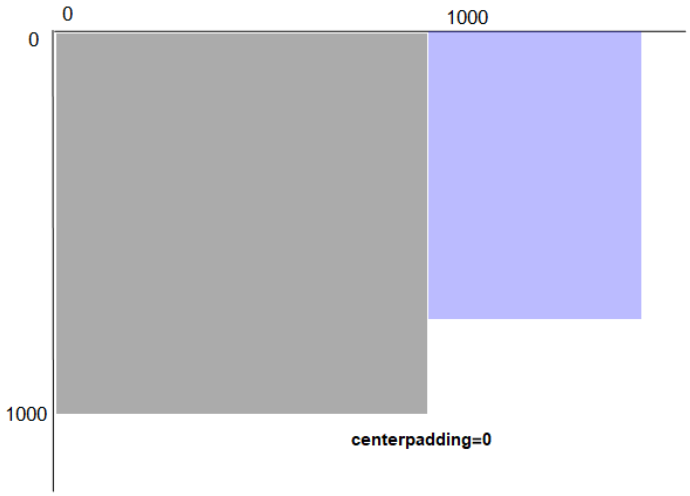


Figure 4: Example f0r joining with no *centerpad*.

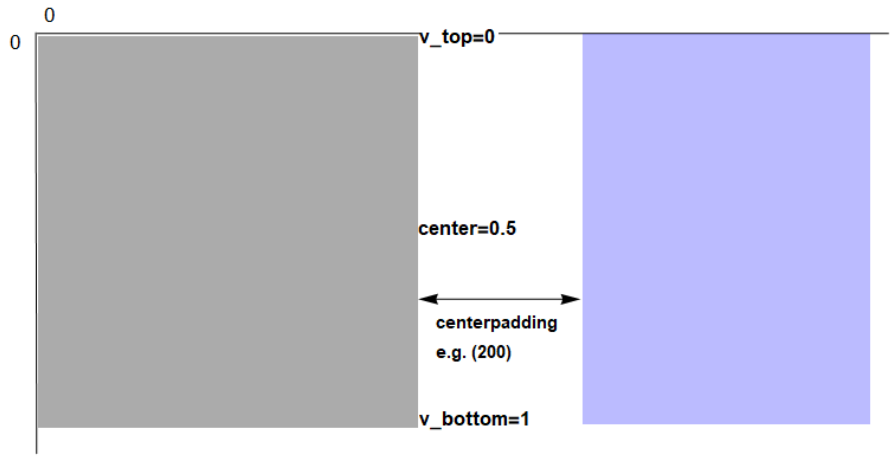


Figure 5: The second image gets scaled to the same size as the first image. This can be conveniently achieved by setting *v_top* to 0 and *v_bottom* to 1. Parameter *centerpad* is 200 as hinted.

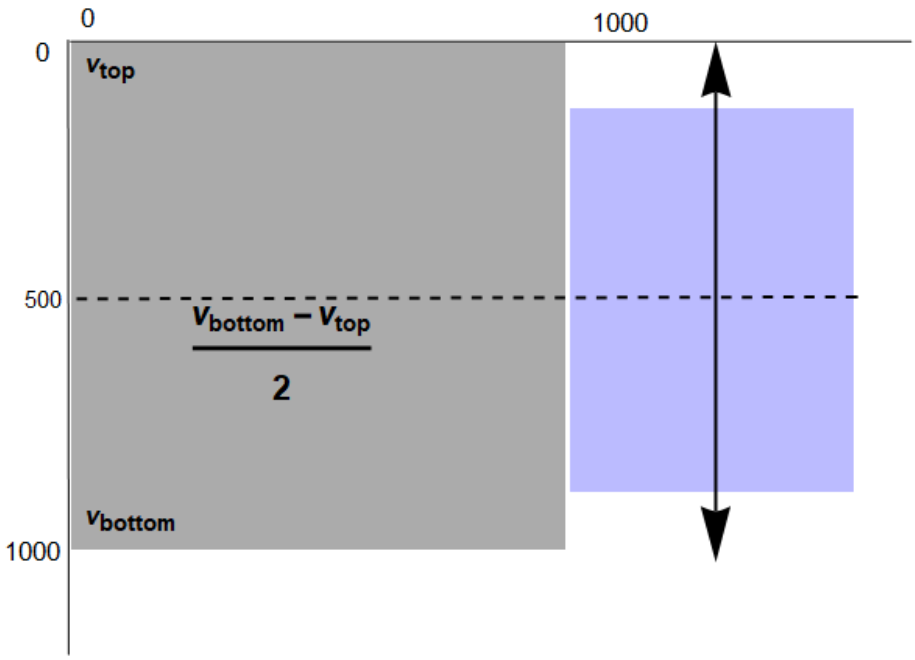


Figure 6: Aligning both images to be vertically centered.
This can be achieved by setting $v\_bottom = v\_top = 0.5$.

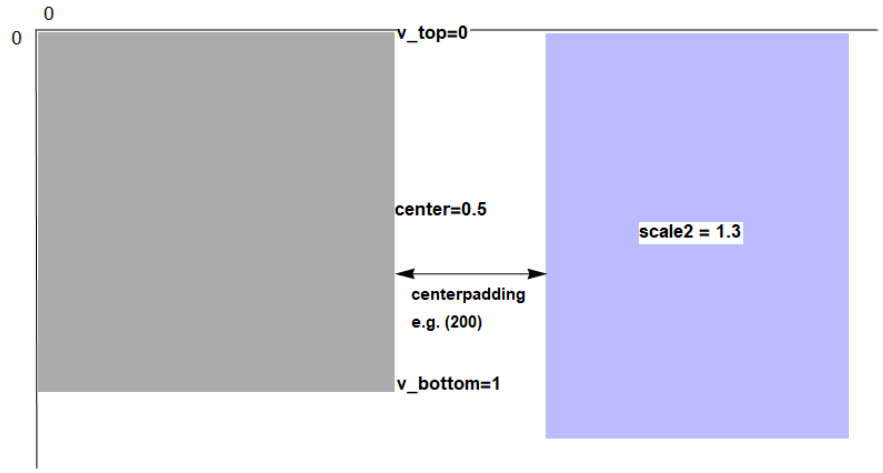


Figure 7: Setting *scale2* to 1.3 to scale the blue (right) image uniformly. Parameter *centerpad* is 200 as hinted.

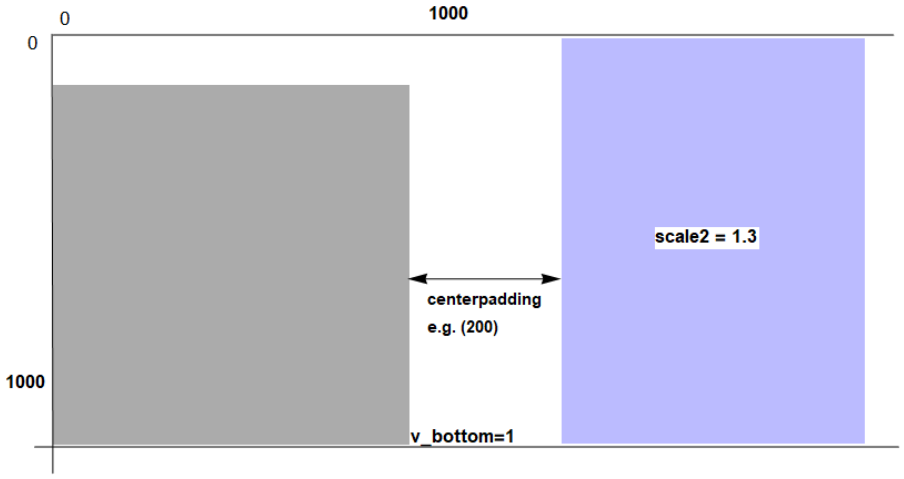


Figure 8: As in figure 7 we use *scale2* to scale the blue rectangle. To align both images at the bottom edge, we use the value $v\_bottom = 1$.

If we want to merge 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.

4. Integration in GPUSAT

To study and improve the handling of the C++ program was chronologically the first task I experimented with. Getting the program up and running proved to be more difficult than we envisioned due to a probable bug in the driver when running OpenCL Drivers from CUDA on the Windows OS.

Impact on performance: Utilizing small classes and streams we tried to keep the impact on performance during the solving process low. However running on the same thread especially for larger problem instances

Some non-functional changes made to the source were

a) some adjustments to satisfy the local compiler (shortening kernel string, replacing '*and*' with '&&')

b) some explicit casts

c) fixed documentation of command line arguments

The functional changes were:

1. allow tabs instead of spaces in input files
2. output more information about the hardware used (device_query)
3. add verbose output globally toggled by a flag
4. decide on `https://en.wikipedia.org/wiki/DOT_%28graph_description_language%29` as the intermediate format for storing graphs.
5. start collecting information that might be needed for a visualization *graphfile* for saving the decomposition graph
6. add *graphout* to solver flow to get automated insight into the structure of a run.
7. add function *solutiontable* to extract tables of variable assignments as a string
8. add labels to each node (bags and solutions at this point)
9. encapsulating the previous functions into gpusat::Graphoutput class and instantiate it in main.
10. using the class functionality in the Solver
11. changing enum to the scoped enum class for better encapsulation and strongly typed.
12. with inlining gpusatutils the current functionality of creating raw dot was completed.
13. Experimented with Neo4j, but found the visualization in particular not that presentable. The functionality to create cypher-queries for the SAT formula with primal, incidence and dual graph is still present in the class.

14. The functionality to create the cypher-query for the graph of the tree-decomposition was added.
15. Rename *visualisierung* into *visualization*
16. Using JsonCPP for processing and formatting json objects in C++
17. Setting format to BasedOnStyle: LLVM, UseTab: Never, IndentWidth: 4, TabWidth: 4, ColumnLimit: 0
18. Creating a *Grid* class for efficiently storing unsigned integer values in a two-dimensional structure based on ideas from this thread
19. create guide for remote development with *Visual Code*
20. Converted intermediate string operations to string-streams instead of files
21. Updated README.md
22. Added Doxygen for docbook, html, latex

Programm `https://github.com/VaeterchenFrost/GPUSAT`
Differences: `https://github.com/daajoe/GPUSAT/compare/master...VaeterchenFrost:master` Commits 142 Files changed 94 Nagoya talk: Graphs for performance are Ordered by used time per algorithm - gpusat quite good

Working with cmake remotly. ssh @(sg1.)dbai.tuwien.ac.at CPU branch wasn't working. Only AMD/Nvidia graphics with respective flags.

Manual configuration with the include options from cmake in CMakeLists.txt or with help from `https://marketplace.visualstudio.com/items?itemName=ms-vscode.cmake-tools` to set up for the (potentially remote) environment.

Table 2: 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
-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 14.

4.1. Class Graphoutput

First steps to automatically visualize the tree decomposition of the solving process with its solutions. Outputs a graph specified in gml[Him10] (Graph Modeling Language). For our example for #SAT 13 with the bags and respective solutions as nodes connected to the bag they solve.

Two additional functions generate a Neo4j Cypher query with:

- one graph representing the SAT formula and queries to construct incidence, dual and primal representations. output as satFile = "cypherSatFormula.txt"
- one graph representing the tree decomposition of the primal graph with it's bags containing variables. output as tdFile = "cypherTreedec.txt"

4.2. Class SolverVisualization

To include the extraction of all necessary visualization information into the solver I created a separate fork. To simplify the creation of valid json I selected the actively developed c++ library JsonCpp `https://github.com/open-source-parsers/jsoncpp` version 1.9.2 from the open-source-parsers repository.

5. 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* 7 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).

6. Application and Images

6.1. SAT Example

In this section we will visualize the process of checking the formula **??** for solvability. We have six time steps included in this visualization. First we will look at the bags in the tree-decomposition. As some simple debugging information we added the time to solve each bag individually into the labels.

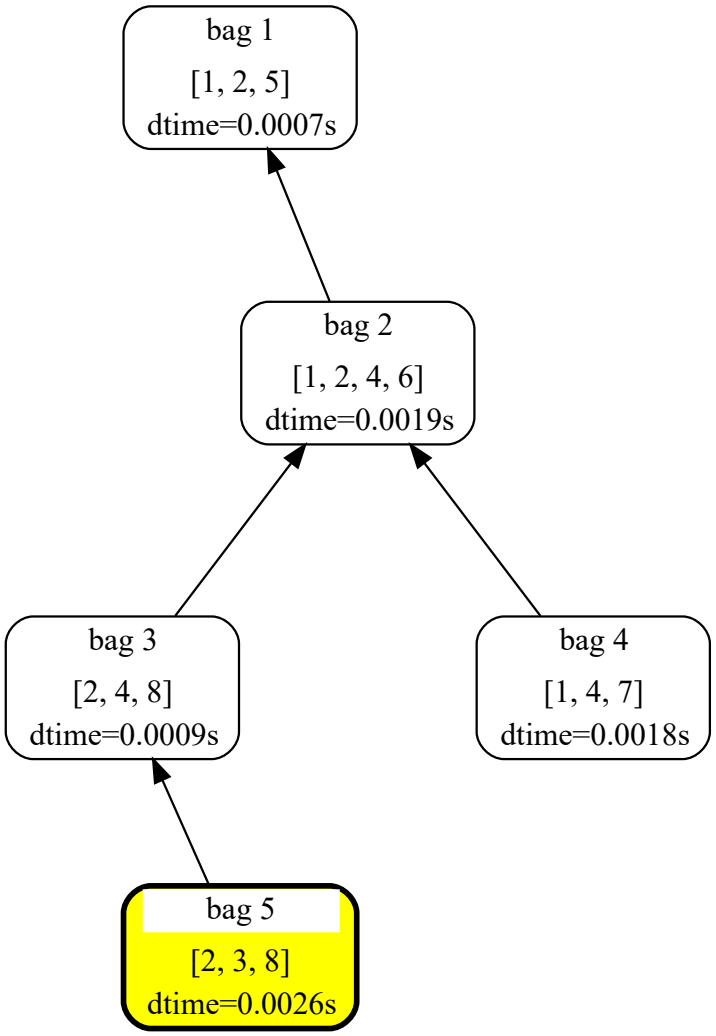


Figure 9: Tree decomposition for solving example **??** .
With yellow highlighting for the first leaf (bag 5) to solve.

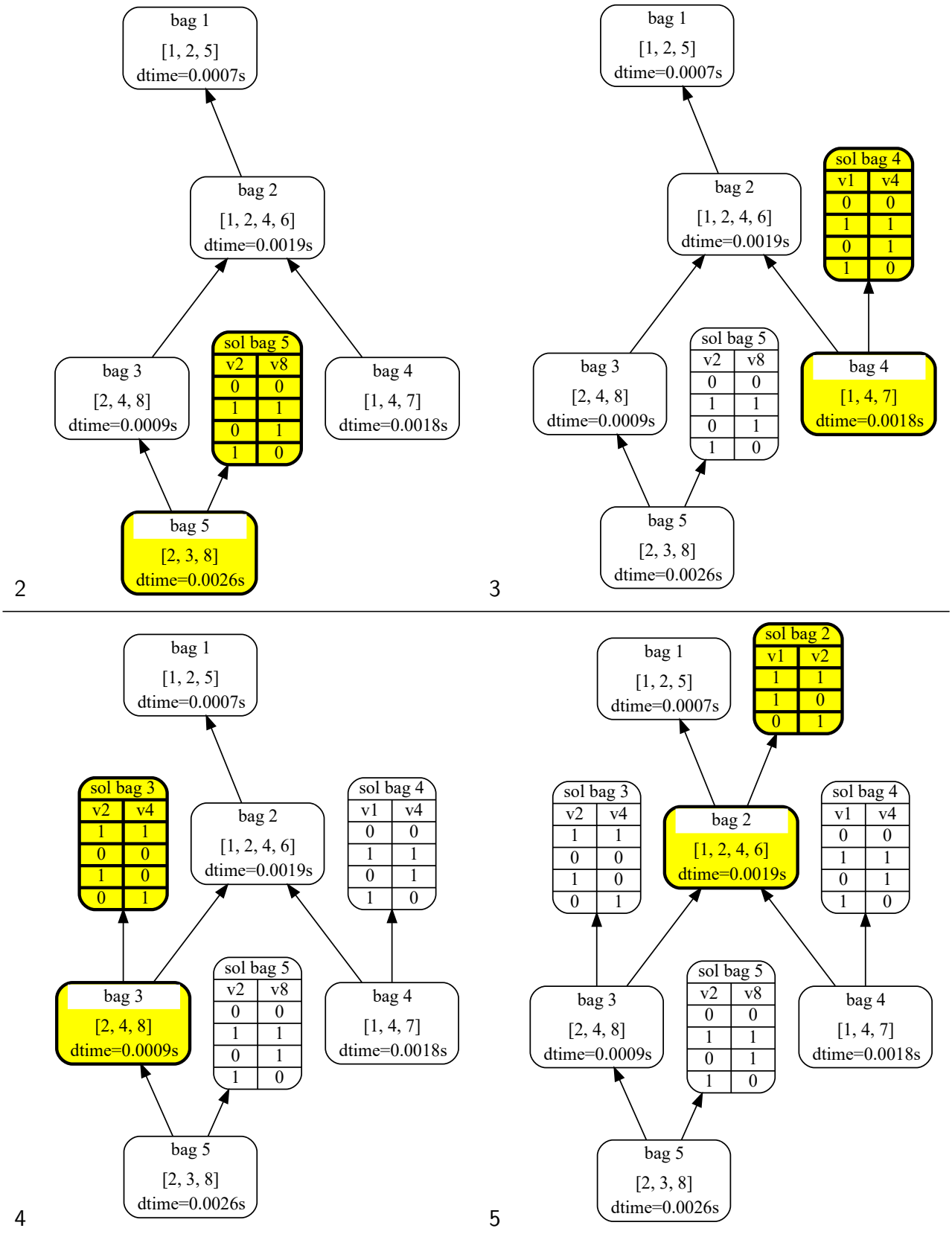


Table 3: Tree decomposition for solving example ?? . Images for steps two to five as labeled from top left to bottom right.

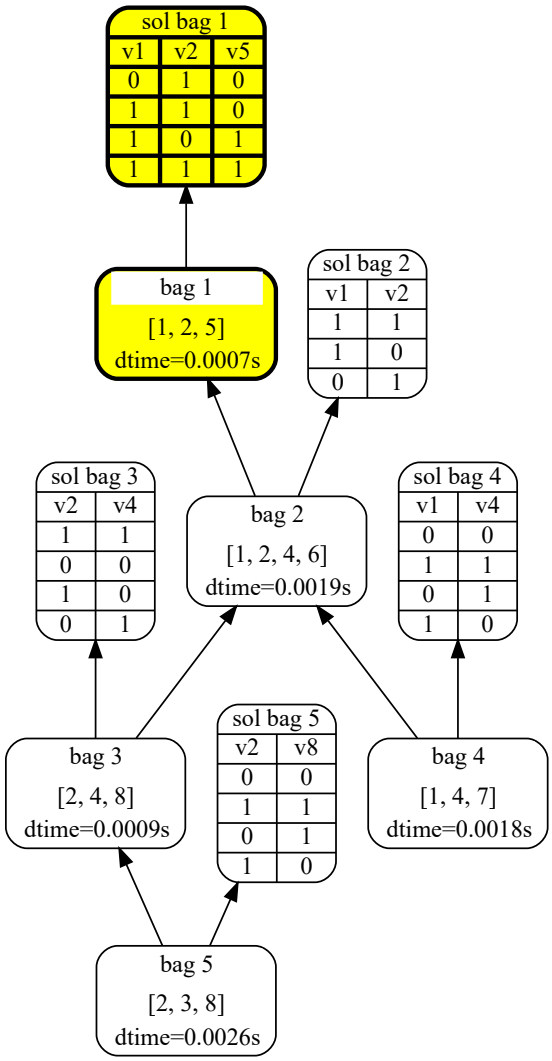
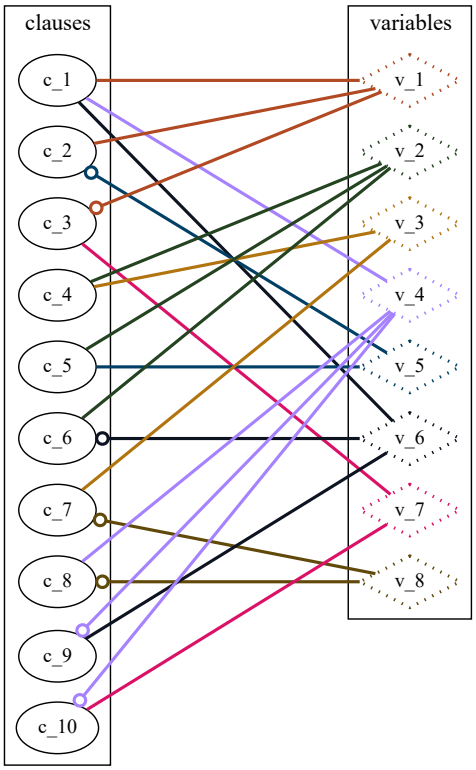
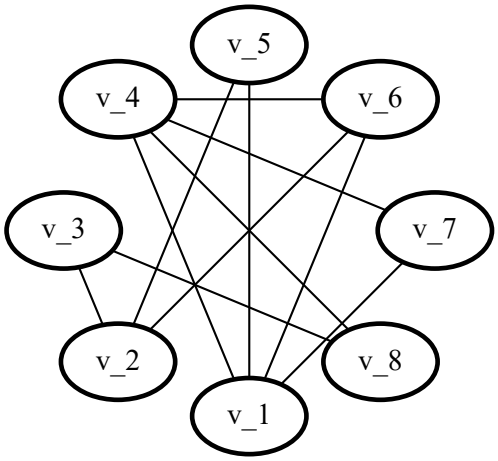


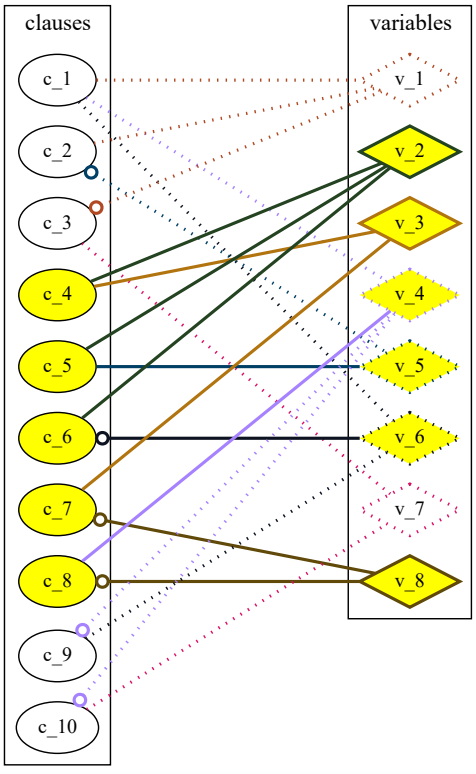
Figure 10: Tree decomposition for solving example **??** .
Final result with yellow highlighting for the last bag (1) solved.

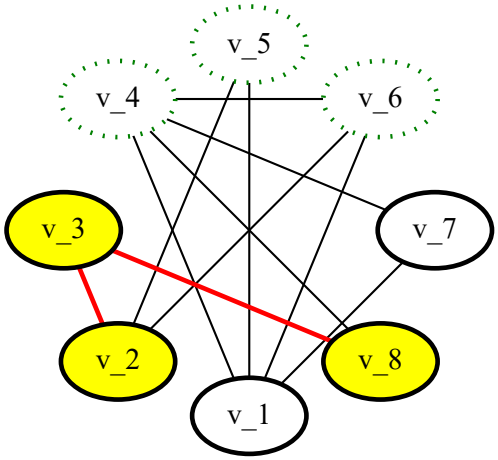
step 1:





step 2:





Visualization of the incidence graph including information for the sat formula

Visualization of the primal graph

Figure 11: Incidence graph and primal graph of example **??** .

6.2. #SAT Example

Like the previous example section we are interested in solutions to example **??**. 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, we can add one column to our solution-tables compared to pure SAT solving and label this column *mc* for "*model-count*". We also included a footer with the API to display the sum of all models considered up to this bag.

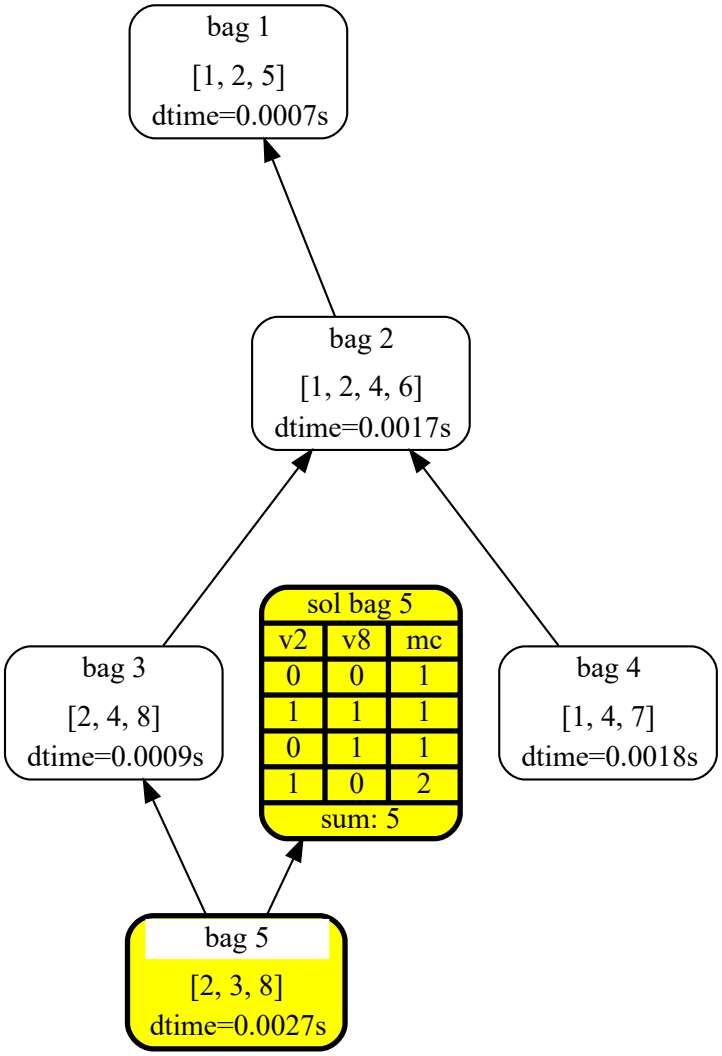


Figure 12: Tree decomposition for solving example **??** with yellow highlighting of the solution for the first leaf.

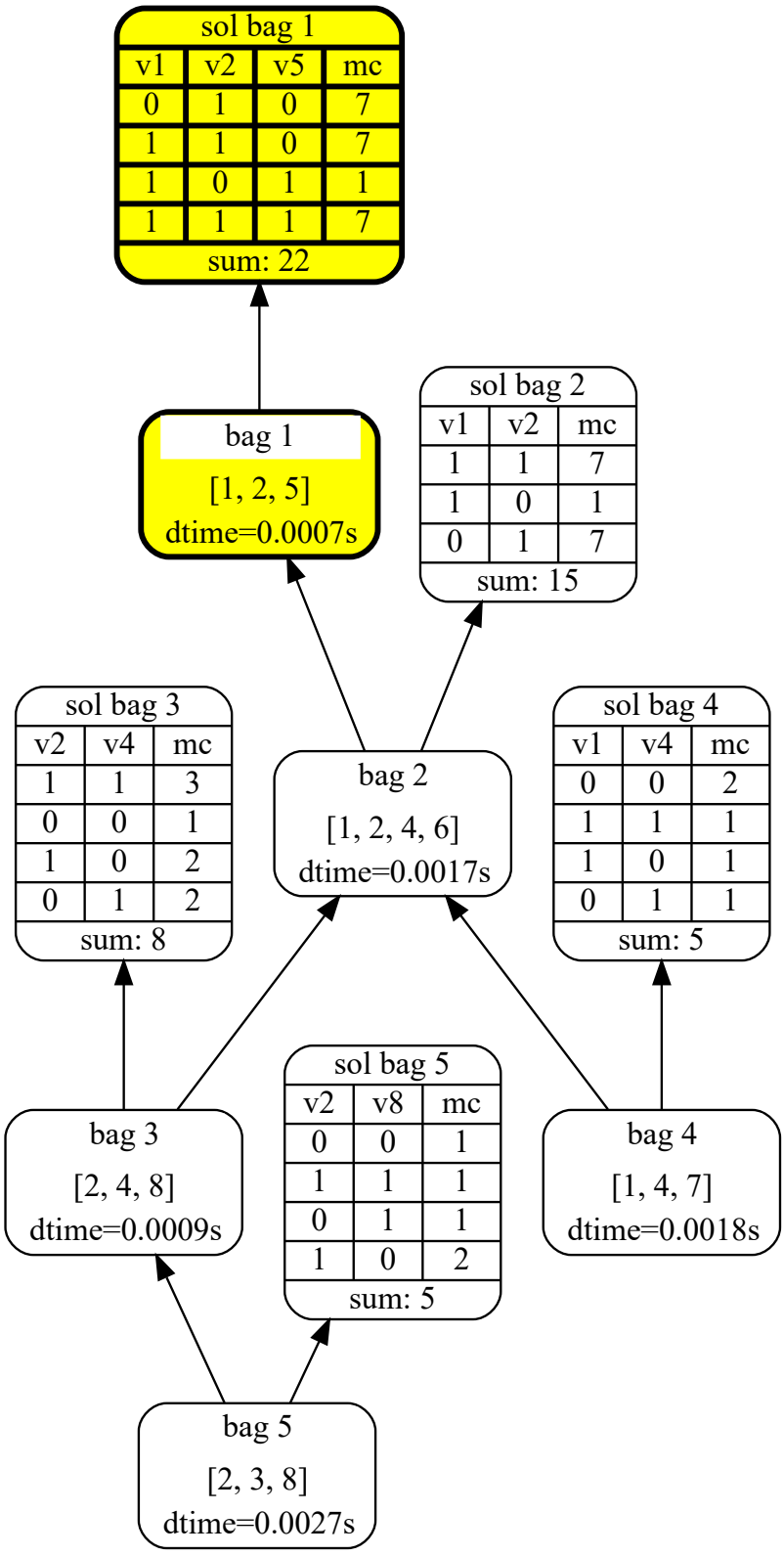


Figure 13: Tree decomposition for solving example **??**. The highlighted bag 1 points to the solution of the problem, containing 22 solutions and satisfying variable assignments for v_1, v_2, v_5 contained in *sol bag 1*.

6.3. Vertex Cover Example

Now we take a look at one problem that is inherently stated on a graph.

Vertex Cover Problem: For a given input graph $G = (V, E)$, a *vertex cover* is a set C of vertices $C \subseteq V$ so that we have $\{u, v\} \cap C \neq \emptyset$ for each edge $\{u, v\} \in E$. The problem *minimal vertex cover* asks to find the minimum cardinality among all vertex covers, i.e. $|C|$ is such that there is no vertex cover C' with $|C'| < |C|$.

An instance $G = (V, E)$ of *minimal vertex cover* parameterized by treewidth *tw* can be solved in time $\mathcal{O}(2^{tw(G)}|V|)$ [Fer05].

For details on the algorithms used see also [Fic+20] on MinVC and listing 3 for a template.

As a very small example we will try to solve *minimal vertex cover* for the following graph in figure 14 with edges seen in listing 15.

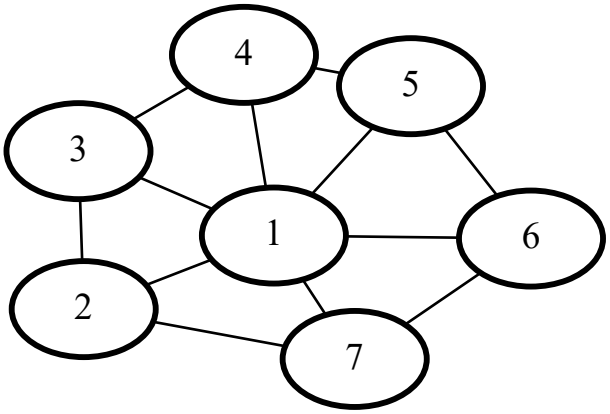
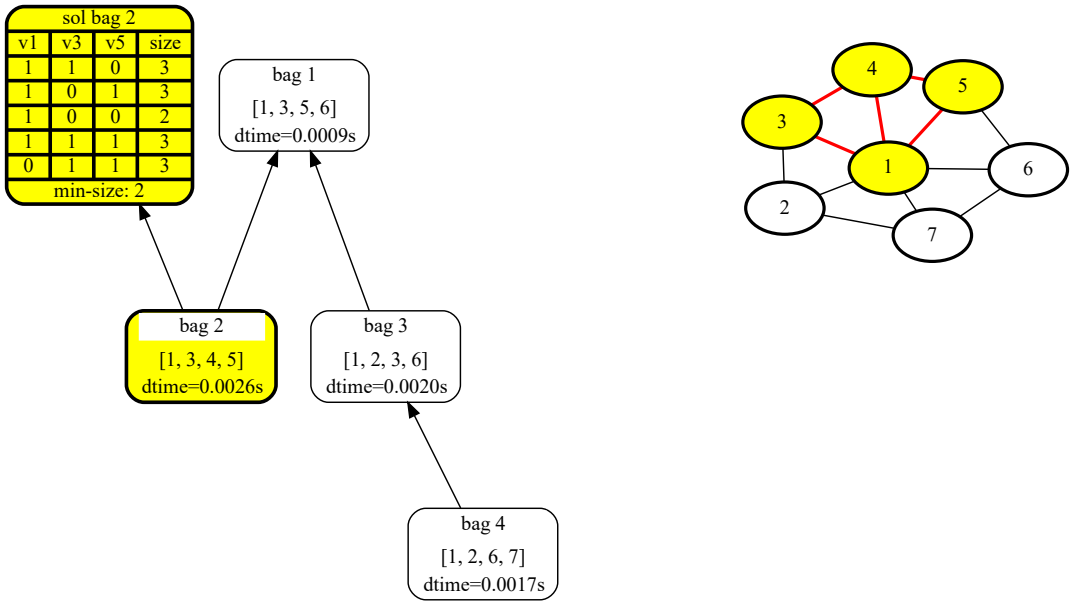


Figure 14: Wheel graph with 7 vertices.

First step solving **bag 2**



Second step solving **bag 4**

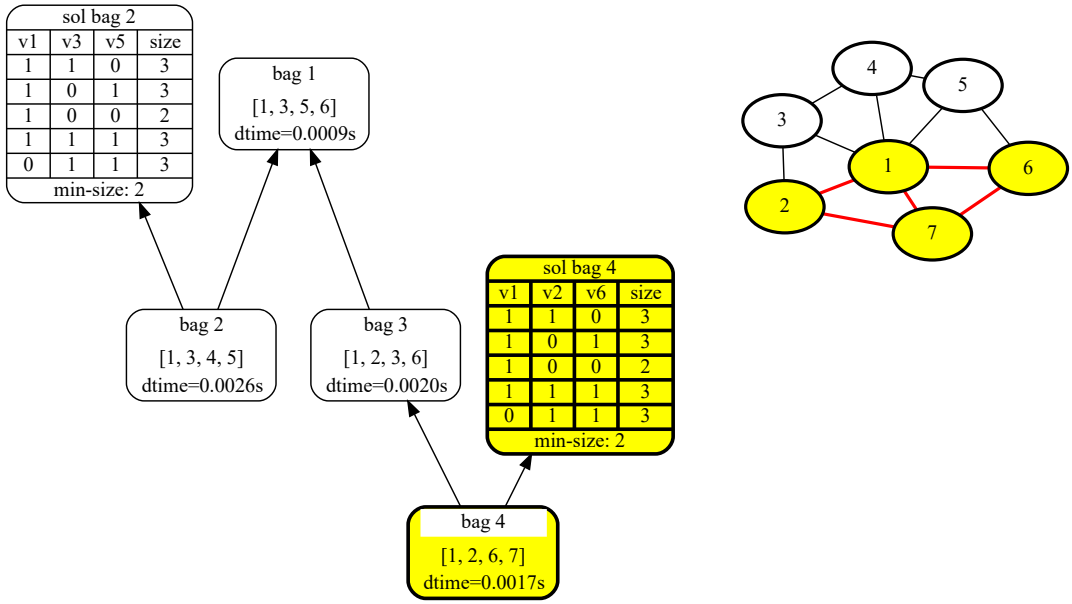
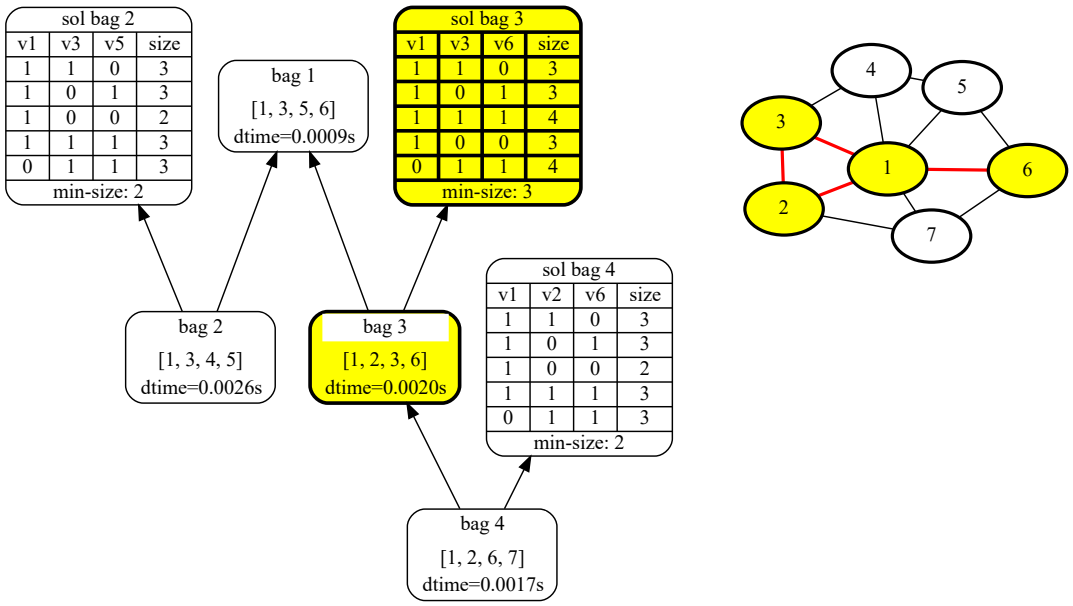


Figure 15: First steps to solve *minimal vertex cover* for example graph 14 on it's tree decomposition.

Third step solving **bag 3**



Fourth step solving **bag 1**

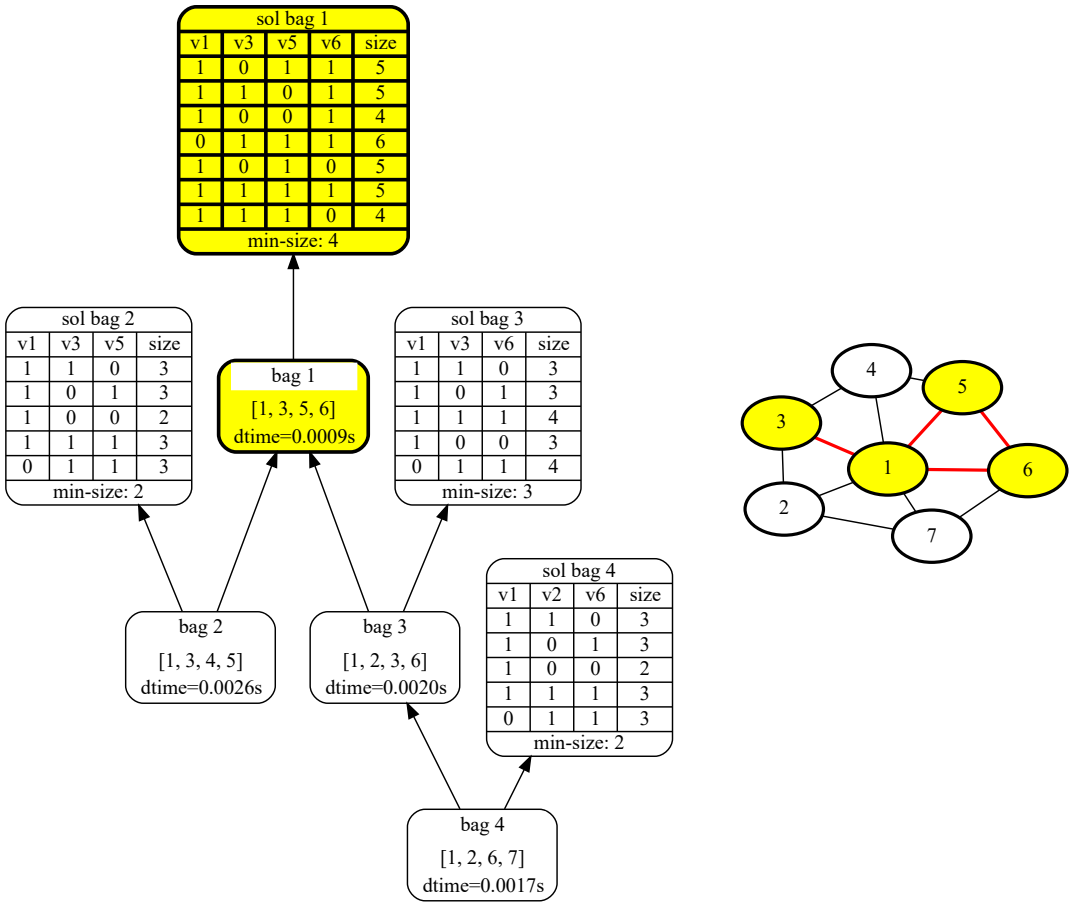


Figure 16: Last steps to solve *minimal vertex cover* for example graph 14 on it's tree decomposition.

6.4. 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 3.6 with some practical applications of this concept.

With the four parameters *centerpad*, *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.

In figure 17 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.

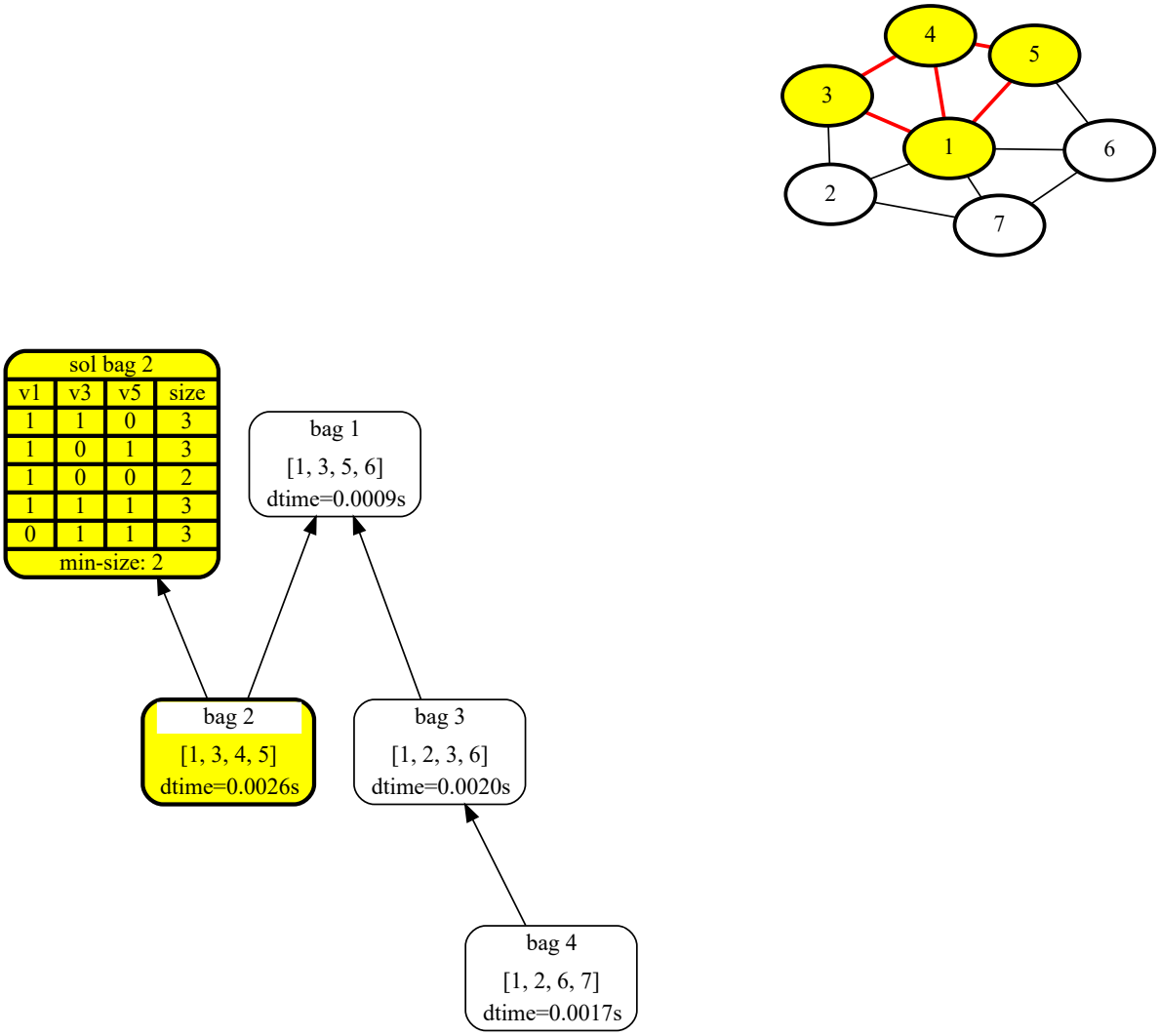


Figure 17: Joining results from section 6.3 with default parameters at step 2.

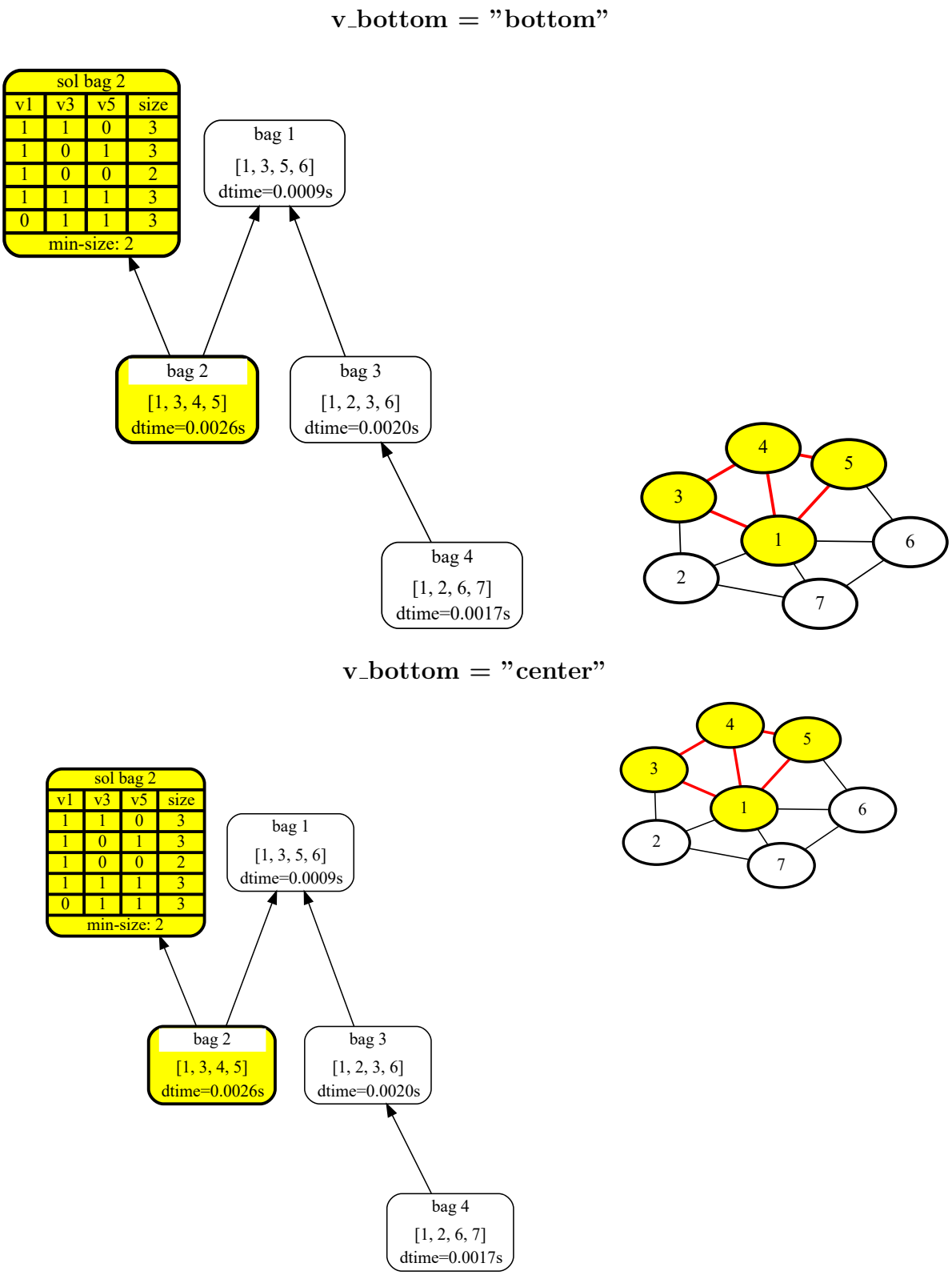


Figure 18: Joining results from section 6.3 at step 2 and setting v_bottom. The invisible (empty) parts of the graphics is cropped from the top of each.

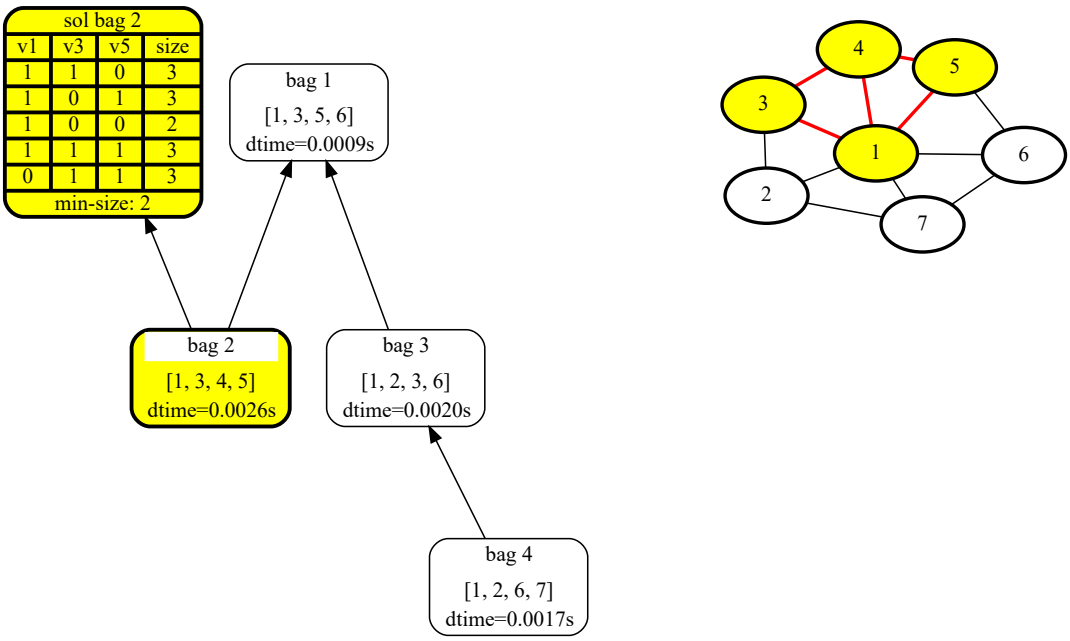


Figure 19: Joining results from section 6.3 at step 2 and shifting bottom edge of second image to 60% height of the first image.

In figure 18 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 19 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 26 to 30 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 *centerpad*, *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 31 to 35.

6.5. Visualizing Errors

One call did not give the expected result for *vertex cover*. The problem was first found expecting the immediate output from the *dpdb.py* program reporting the wrong set size. The underlying problem could found using the visualization in figure 20

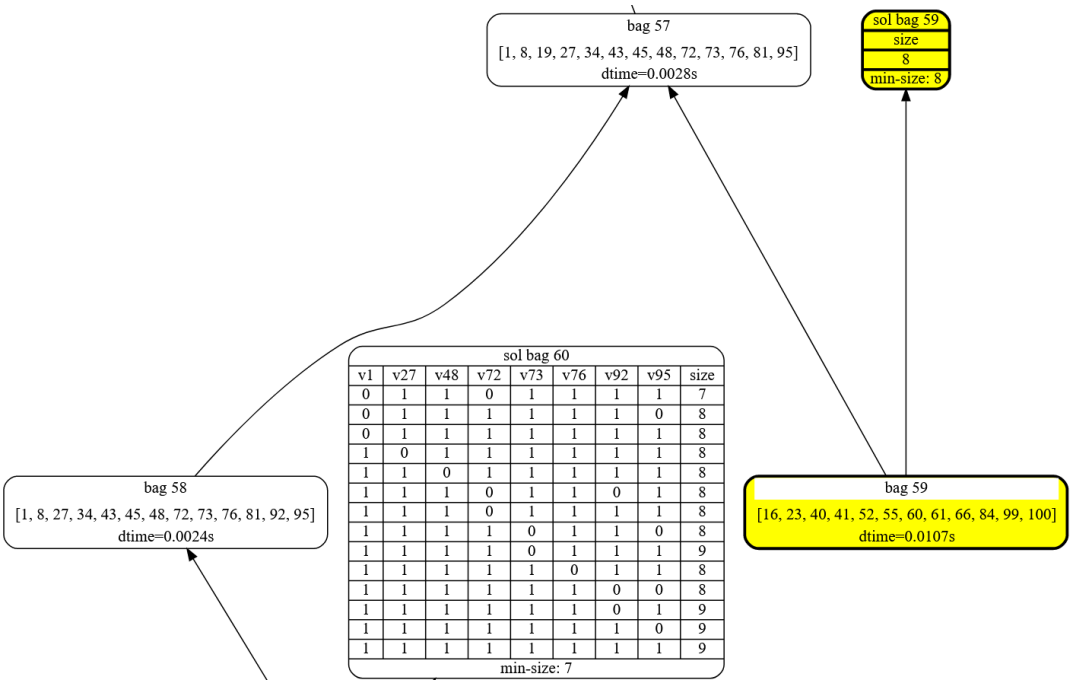


Figure 20: Part of interest to find a problem with bag 59 in visualization.

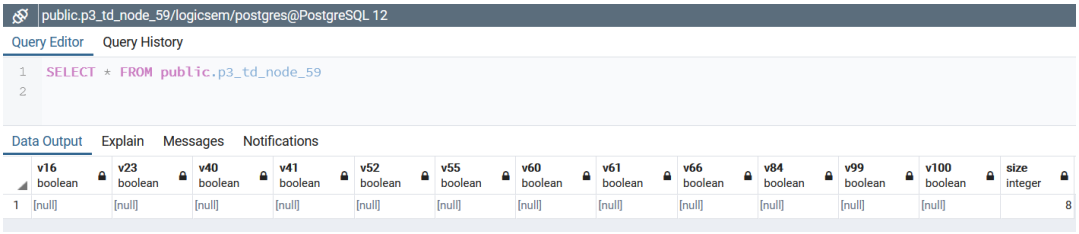


Figure 21: Bag 59 in the database table.

The bag 59 does not provide any rows with variables The problem was located in the htd version 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. The proof that the provided tree is not a tree decomposition is visualized using Mathematica [Wol] in figure 22.

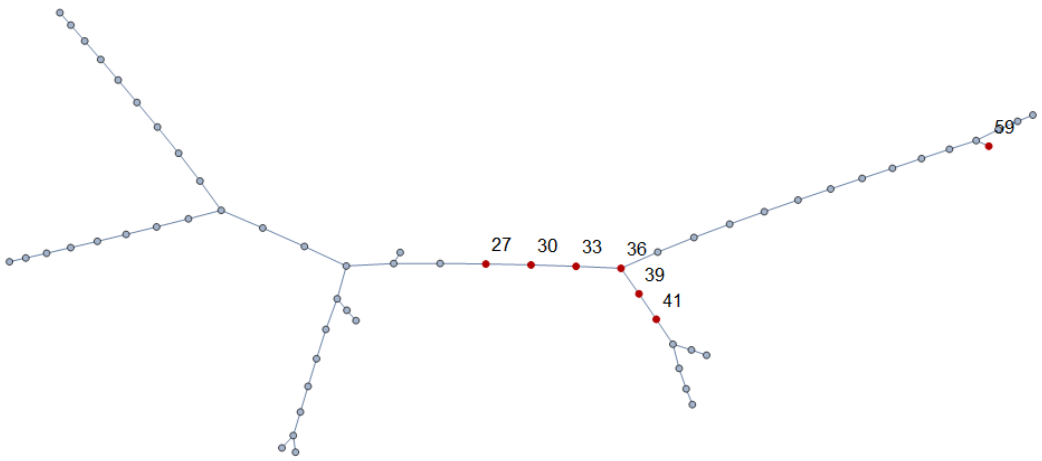


Figure 22: Bag 59 can be seen on the far right of the graphic.

7. Conclusion

7.1. Summary

We created and could for the most part automate visualization of dynamic programming on tree decompositions. With SVG we by default do support a human readable, highly adaptable data format.

We defined and developed a data-exchange form to give the visualization the information it will need and provide sensible default values for (almost) all parameters.

The visualization got implemented and tested in two actively developed solvers for the problem types **SAT**, **#SAT** and **minimal vertex cover**.

During development we already found some improvements in the solvers and easily identified possible bugs. All nodes are prepared to display arbitrary user defined strings that could include various debugging-information about the run.

7.2. 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 could be the inclusion of hypergraphs (graphs where one edge does connect multiple nodes) which could be of interest to solvers of the future.

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

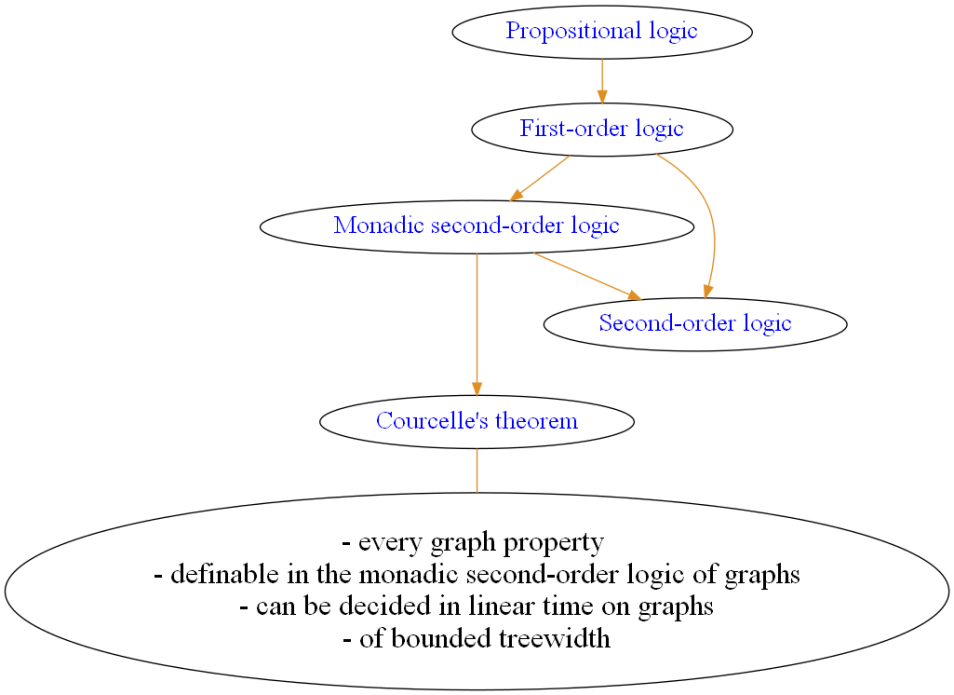


Figure 23: From propositional logic to monadic second order logic and Courcelle's Theorem

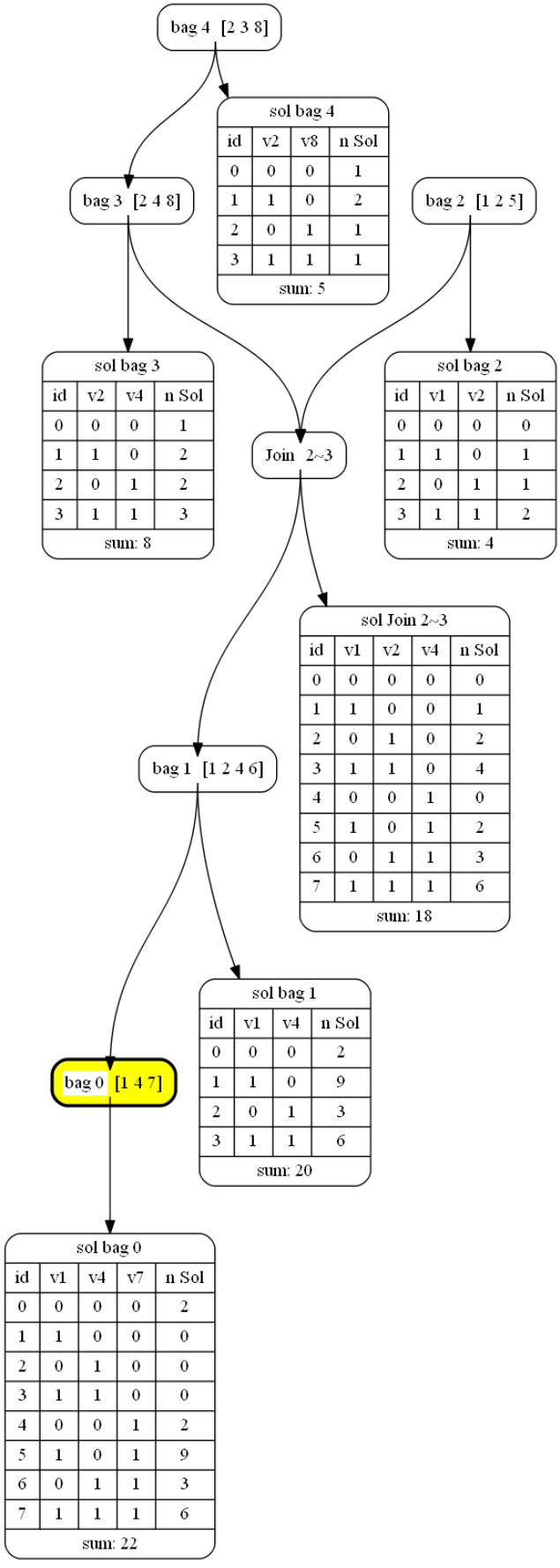


Figure 24: Created scalable-vector-graphic directly from 13

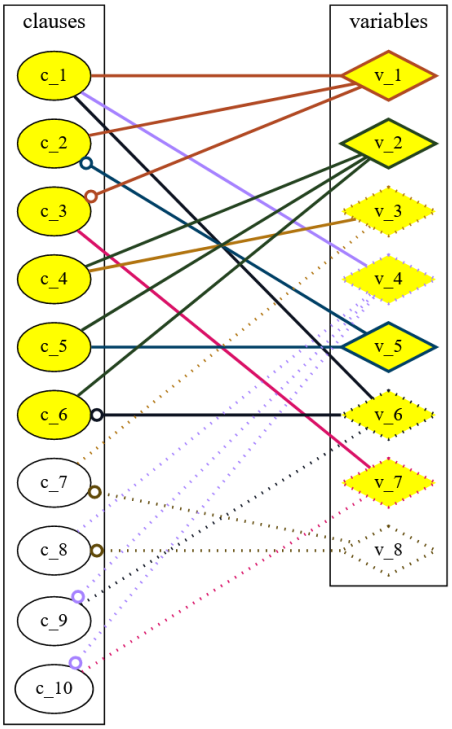


Figure 25: Example for an incidence graph from example 4.1

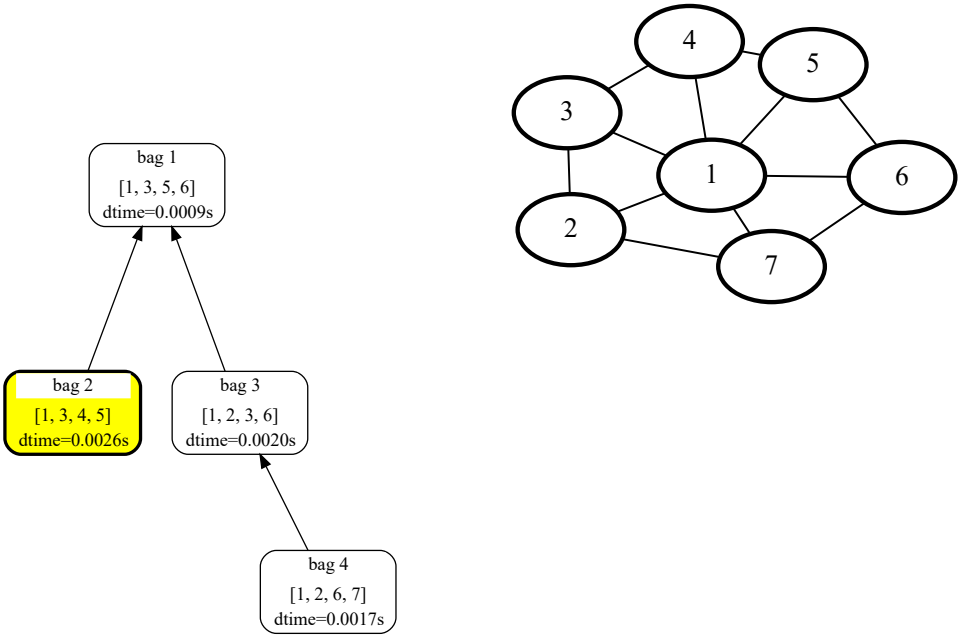


Figure 26: Joining results from section 6.3 at step 1. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

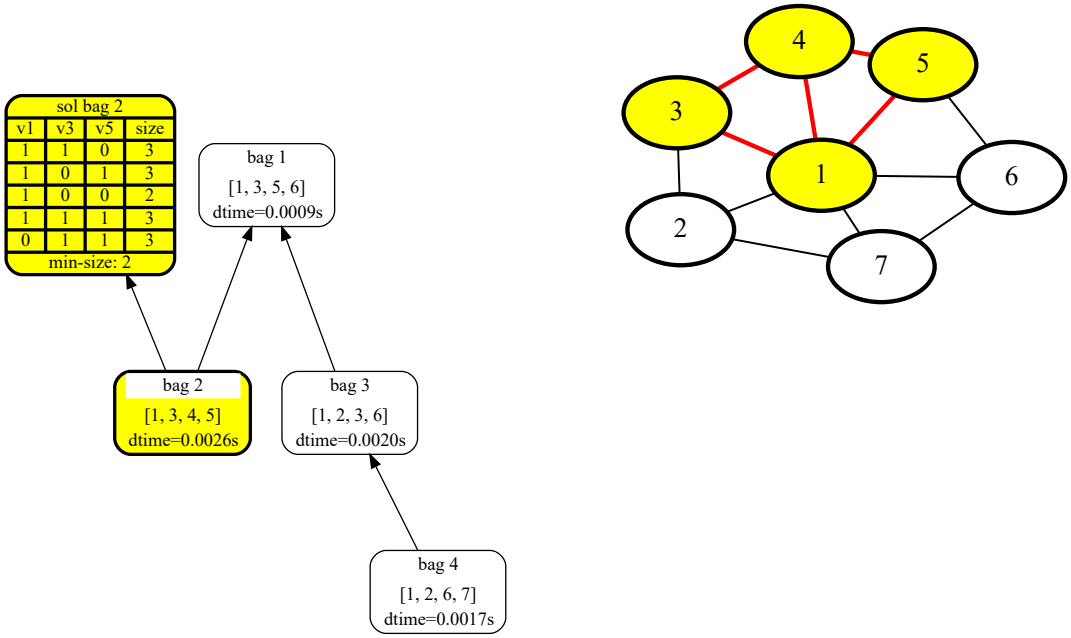


Figure 27: Joining results from section 6.3 at step 2. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

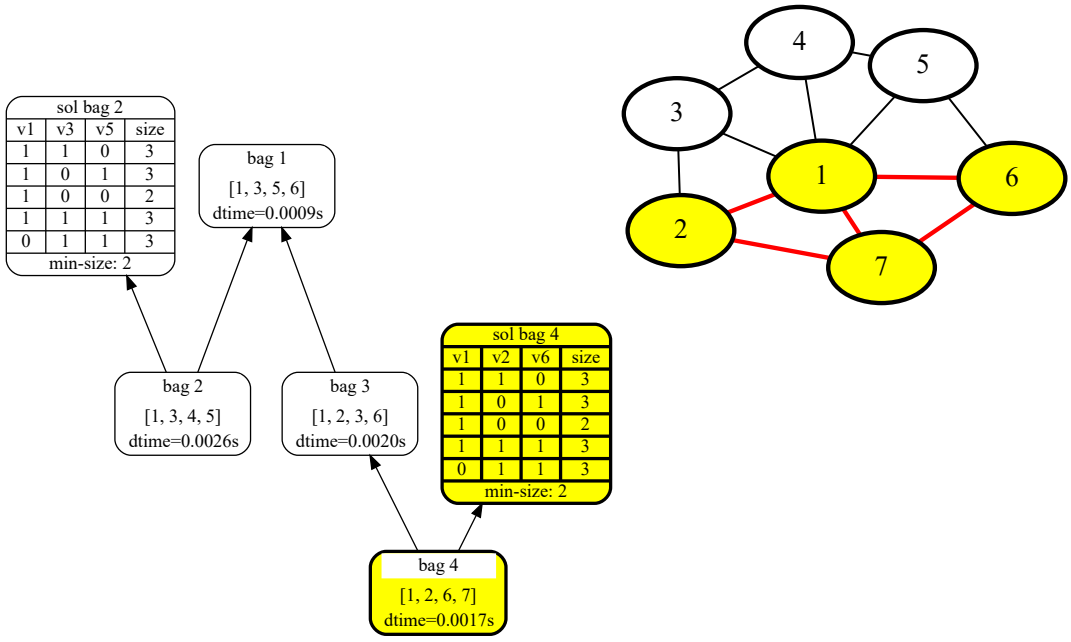


Figure 28: Joining results from section 6.3 at step 3. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

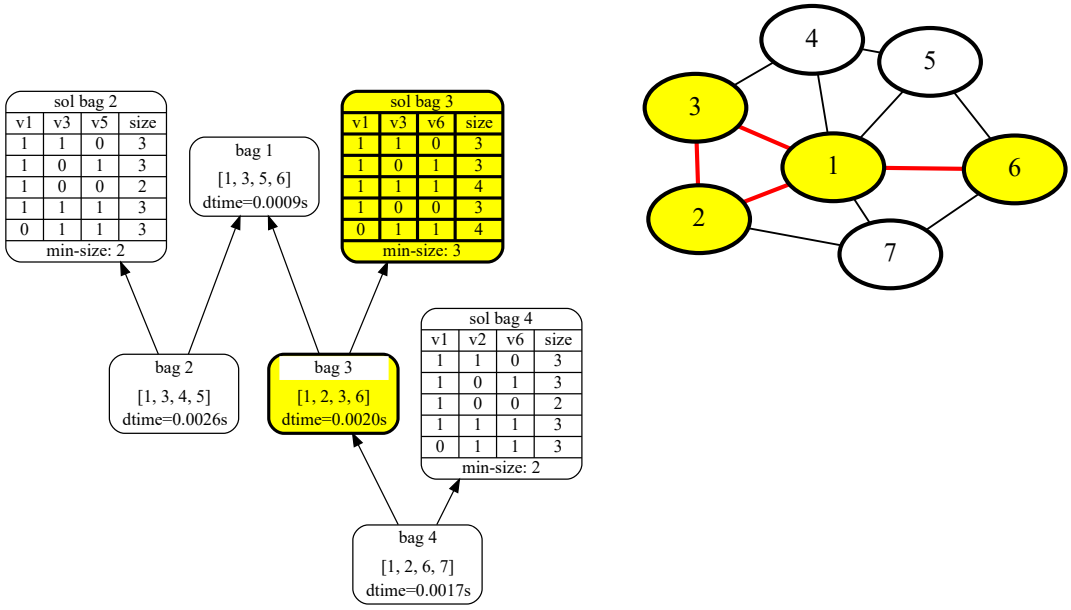


Figure 29: Joining results from section 6.3 at step 4. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

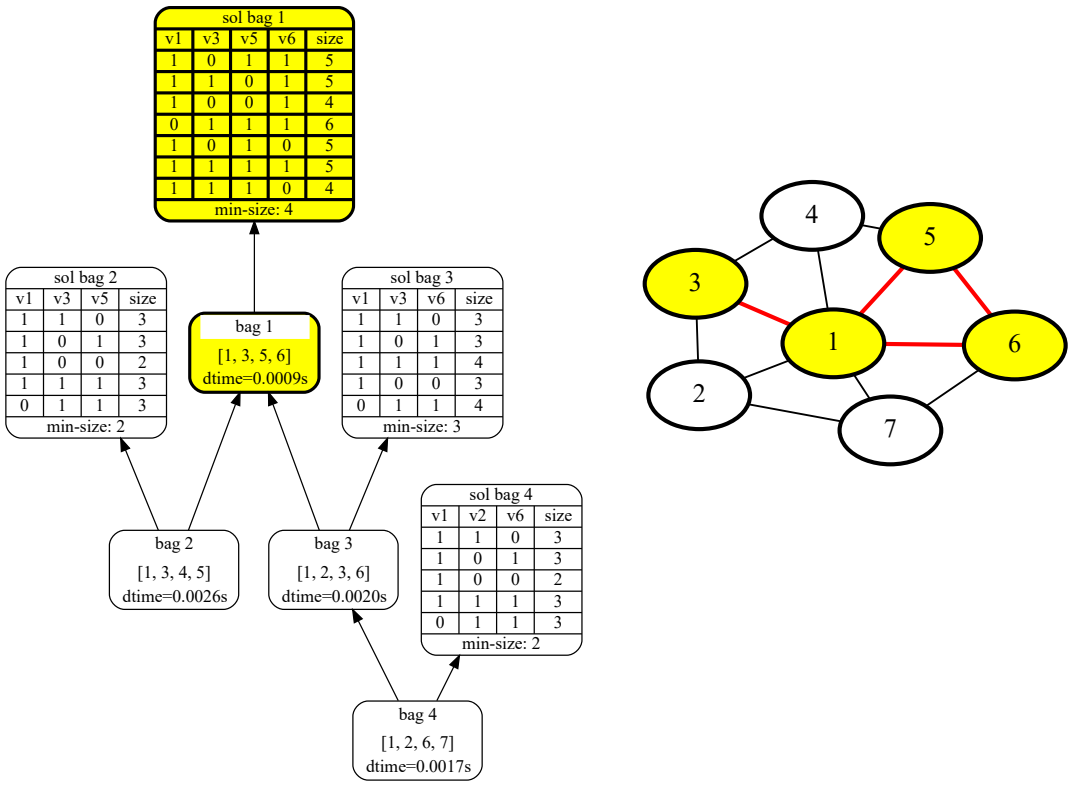


Figure 30: Joining results from section 6.3 at step 5. Also shifting the second graphic to 40% height of the first image and scaling with scale2 = 1.5 .

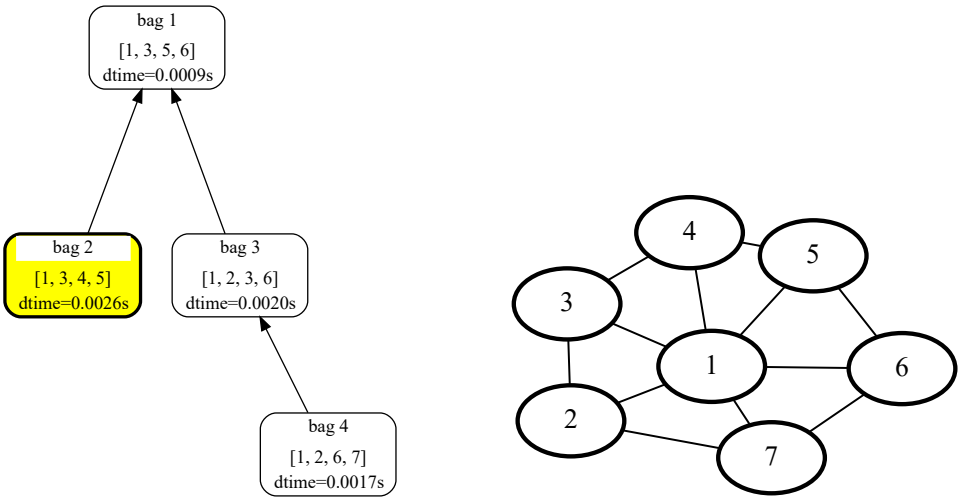


Figure 31: Joining results from section 6.3 at the first step. Also shifting the second graphic to the bottom edge of the first image and scaling with scale2 = 1.5 .

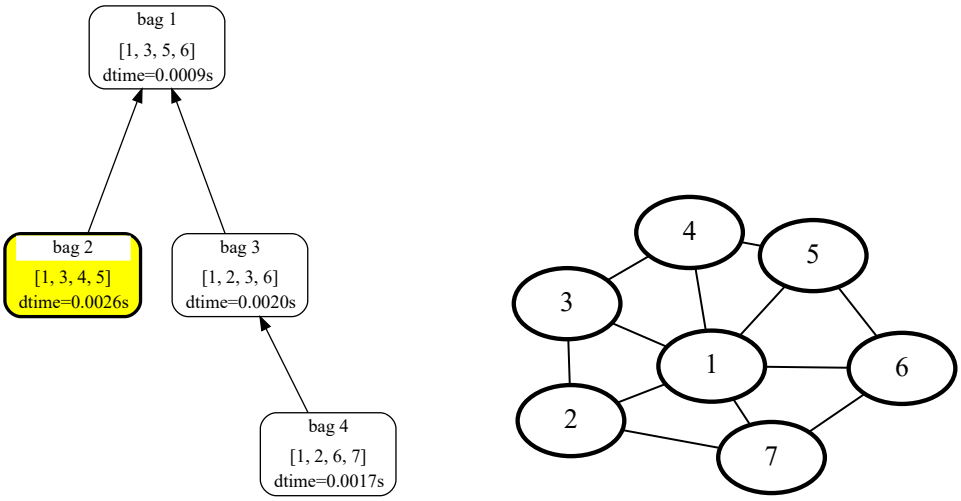


Figure 32: Joining results from section 6.3 at the second step. Also shifting the second graphic to 15% of the height of the first image and scaling with scale2 = 1.5 .

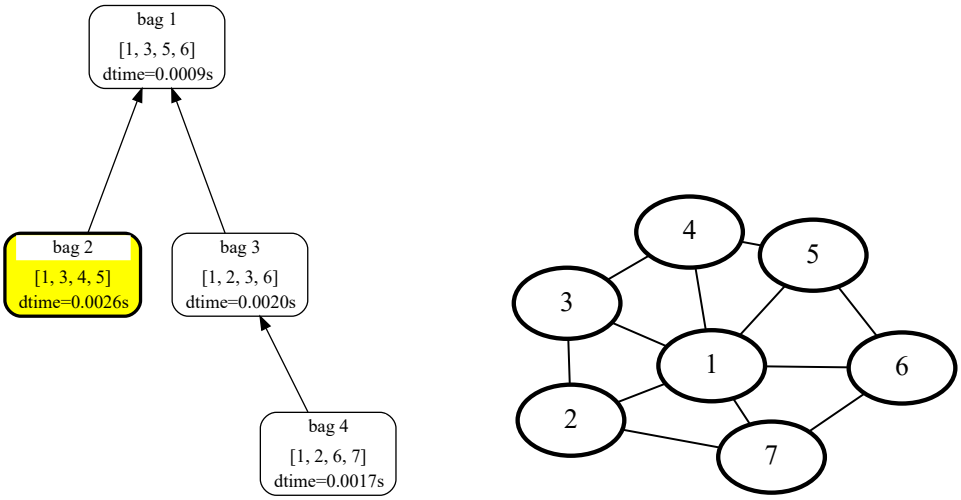


Figure 33: Joining results from section 6.3 at the third step. Also shifting the second graphic to 30% of the height of the first image and scaling with scale2 = 1.5 .

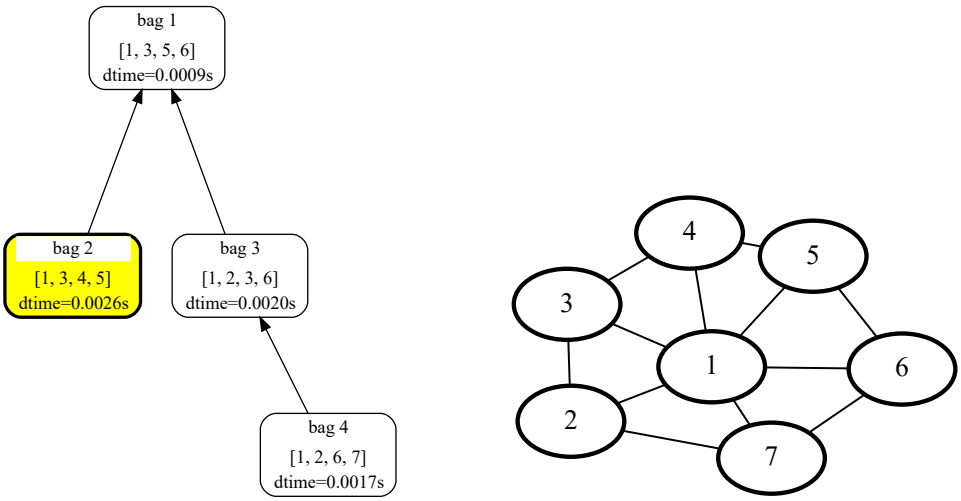


Figure 34: Joining results from section 6.3 at the fourth step. Also shifting the second graphic to 45% of the height of the first image and scaling with scale2 = 1.5 .

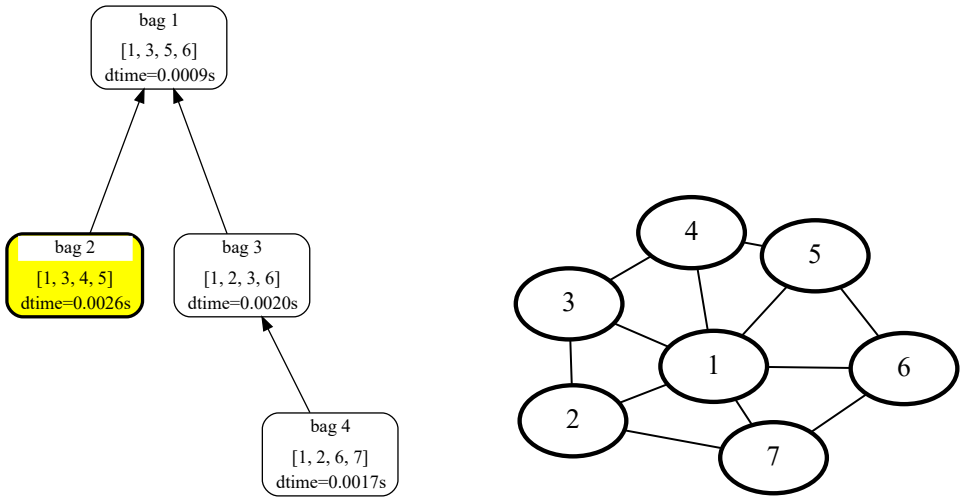


Figure 35: Joining results from section 6.3 at the final step. Also shifting the second graphic to to 60% of the height of the first image and scaling with scale2 = 1.5 .

B. Code Snippets

Listing 6: 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 7: 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:
      query = """SELECT name,type,num_bags
      FROM public.problem WHERE id=%s"""
      cur.execute(query, (problem,))
```

```
      (name, ptype, num_bags) = cur.fetchone()

    constructor: IDpdbVisuConstruct
    if ptype == 'SharpSat':
      constructor = DpdbSharpSatVisu(
            connection, problem, intermed_nodes)
    elif ptype == 'VertexCover':
      constructor = DpdbMinVcVisu(
            connection, problem, intermed_nodes, tw_file)
    return constructor.construct()
  return {}
```

Listing 8: 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):     # Create the positions
    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 9: 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 10: 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. Input Examples

Listing 11: CNF clauses from example 4.1 on page 27 [Zis18]

```
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 12: 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 13: 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 14: stdout of program gpusat with call *./gpusat -f ../examples/test_da4_1.cnf -v -p -d ../examples/td4p1.txt -g ../examples/graphfileda41.txt --visufile ../examples/visufileda41.json*

```
Seed: 1592417295
Platform - 1
  1.1 CL_PLATFORM_NAME: AMD Accelerated Parallel Processing
  1.2 CL_PLATFORM_VENDOR: Advanced Micro Devices, Inc.
  1.3 CL_PLATFORM_VERSION: OpenCL 2.1 AMD-APP (2671.3)
  1.4 CL_PLATFORM_PROFILE: FULL_PROFILE
  1.5 CL_PLATFORM_EXTENSIONS: cl_khr_icd cl_amd_event_callback
            cl_amd_offline_devices
  Device - 1:
    CL_DEVICE_NAME: Ellesmere
    CL_DEVICE_VENDOR: Advanced Micro Devices, Inc.
    CL_DRIVER_VERSION: 2671.3
    CL_DEVICE_VERSION: OpenCL 1.2 AMD-APP (2671.3)
    CL_DEVICE_MAX_COMPUTE_UNITS: 32
    CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE : 3422266572
    CL_DEVICE_MAX_CONSTANT_ARGS : 8

-- treeDecomp Before --
bags: 5
0 : [1 4 7 ]-(1 )-
1 : [1 2 4 6 ]-(2 3 )-
2 : [1 2 5 ]()
3 : [2 4 8 ]-(4 )-
4 : [2 3 8 ]()

-- treeDecomp after preprocessFacts--
bags: 5
0 : [1 4 7 ]-(1 )-
1 : [1 2 4 6 ]-(2 3 )-
```

```
2 : [1 2 5 ]()
3 : [2 4 8 ]-(4 )-
4 : [2 3 8 ]()
---Determining datastructure---
input:
SOLUTIONTYPE : TREE
treeDecomp.width : 4
------
Opened visualization with file ../examples/visufileda41.json true
----------
==> Entering solveProblem on id 0 <==
==> Entering solveProblem on id 1 <==
==> Entering solveProblem on id 2 <==
solveIF 1 + 0 => 2
  bag(2): bags= 0 , exp= 0 , correction= 0
  var= [1, 2, 5, ]
Solved IF-0 on node 2
  bag(2): bags= 1 , exp= 1 , correction= 0
  var= [1, 2, ]
==> Entering solveProblem on id 3 <==
==> Entering solveProblem on id 4 <==
solveIF 3 + 0 => 4
  bag(4): bags= 0 , exp= 0 , correction= 0
  var= [2, 3, 8, ]
Solved IF-0 on node 4
  bag(4): bags= 1 , exp= 1 , correction= 0
  var= [2, 8, ]
solveIF 1 + 4 => 3
  bag(3): bags= 0 , exp= 0 , correction= 0
  var= [2, 4, 8, ]
  edges to [4, ]
Solved IF-1 on node 3
  bag(3): bags= 1 , exp= 0 , correction= 1
  var= [2, 4, ]
  edges to [4, ]
Solved JOIN-1 on nodes 2~3
  bag(0): bags= 1 , exp= 0 , correction= 2
  var= [1, 2, 4, ]
solveIF 0 + 0 => 1
  bag(1): bags= 0 , exp= 0 , correction= 0
  var= [1, 2, 4, 6, ]
  edges to [0, 3, ]
Solved JOIN-IF on node 1
  bag(1): bags= 1 , exp= 1 , correction= 2
  var= [1, 4, ]
  edges to [0, 3, ]
solveIF 0 + 1 => 0
  bag(0): bags= 0 , exp= 0 , correction= 0
  var= [1, 4, 7, ]
```

```
  edges to [1, ]
Solved IF-1 on node 0
  bag(0): bags= 1 , exp= 0 , correction= 3
  var= [1, 4, 7, ]
  edges to [1, ]

==== GRAPH END ====
Entering writeJsonFile, enabled 1

--- Solutions: ---
bag 0   (from 0 to 7)
id: 0 count: 0.25
id: 1 count: 0
id: 2 count: 0
id: 3 count: 0
id: 4 count: 0.25
id: 5 count: 1.125
id: 6 count: 0.375
id: 7 count: 0.75
...

{
    "Num Join": 1
    ,"Num Introduce Forget": 5
    ,"max Table Size": 13
    ,"Model Count": 22
    ,"Time":{
        "Decomposing": 0
        ,"Solving": 0.006
        ,"Total": 0.383
    }
}
```

Listing 15: Edges for example 14

```
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: Tree decomposition for example 14

```
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
```

References

[11] *On the constructive power of monadic second-order logic*. https://www.youtube.com/watch?v=hZI-wANHO1w. [Online; accessed 11-June-2020]. Daejeon, Korea, Oct. 2011.

[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.

[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.

[Bro+15] I.N. Bronshtein et al. *Handbook of Mathematics*. English. 6th ed. Berlin: Springer Verlag Berlin Heidelberg, 2015. Chap. 5.8, pp. 401–412. 1207 pp. ISBN: 978-3-662-46220-1. DOI: 10.1007/978-3-662-46221-8. eprint: 978-3-662-46221-8.

[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.

[Fer05] Henning Fernau. "Parameterized algorithmics: A graph-theoretic approach". In: (May 2005), p. 310.

[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.

[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.

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

[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.

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

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

[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.

[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.

[VK19] Hannes Voigt and Alexander Krause. *Course: Graph Data Management and Analytics*. `https://wwwdb.inf.tu-dresden.de/study/teaching/teaching-archive/winter-term-2018-19/graph-data-management-and-analytics/`. [Online; accessed 11-June-2020]. Feb. 2019.

[Web20] MDN Web Docs. *SVG: Scalable Vector Graphics — MDN*. `https://developer.mozilla.org/en-US/docs/Web/SVG`. [Online; accessed 2-July-2020]. Apr. 2020.

[Wol] Inc. Wolfram Research. *Mathematica, Version 12.1*. Champaign, IL, 2020. URL: `https://www.wolfram.com/mathematica`.

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