Bachelor Thesis

Visualizing Dynamic Programming on Tree Decompositions

Martin Röbke

born: 04.03.1995 in Dresden, Germany

matriculation number: 3949819

martin.roebke@tu-dresden.de

Technische Universität Dresden
Faculty of Computer Science
International Center For Computational Logic

Supervisor: Dr. Johannes Fichte

Second evaluator: Prof. Dr. rer. nat. Stefan Gumhold

Dresden, June 17, 2020

Abstract

This thesis introduces a practical and lightweight implementation for visualizing dynamic programming on tree decompositions. The provided source code is located in the python-package tdvisu for the purpose of visualizing, teaching and analyzing the dynamic solving process.
It defines a JSON-format specification for portability and customization of the visualization combined in one human-readable file and two reference implementations in actual 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 that can be compressed and modified very easily without loss of quality. The images are split up into different views on the current state of the tree decomposition for consecutive user-defined time steps showing the progress of the dynamic programming. As illustrations of the possibilities for an application smaller examples from the problem-types "#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.

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

Contents

1. Introduction

Graphs are increasingly interesting in scientific work, as the applications of interconnected datasets grow. The use cases for example outlined by Neo4j in [1], a provider of open source database tools for *labeled property graphs* does include fields of interest like

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

While many problems can be expressed using the language of graphs, there is still plenty of room to utilize the graph structure of problems. 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 [2]) problems using highly parallelized architectures like graphics processing units or state of the art databases. One early implementation is published in [3] where for different real world examples the results looked promising These projects are very competitive ????REF????? for solving even large instances of those problems. The source code for TDVisu is available under GPL3 license.

Graphviz is open source graph visualization software that provides customizable visualization for directed and undirected graphs. The information processed by graphviz

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

Visualization Pipeline

Stand Umsetzung, Tools: Slack, Trello, GitHub, Presentations

2. Background

This chapter provides the reader with a brief background for this work.

We begin with a description on SAT and #SAT as examples for a very general problem that can be described with monadic second order logic (MSOL). Furthermore the general case of MSOL will be described, as well as the *DIMACS*-file-format used in the projects. The following section describes Tree Decompositions (TDs) which are the basis for our visualization. Finally we shortly discuss Courcelle's Theorem [2] as a related method of solving these problems.

2.1. Boolean satisfiability problem

`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

SAT was the first known NP-complete problem, shown by Stephen Cook at the University of Toronto in 1971 [4]

literal ≡ boolean variable v or its negation
clause ≡ finite set of literals, interpreted as the disjunction
unit ≡ clause with |c|=1
CNF formula ≡ set of clauses, interpreted as their conjunction
var ≡ set of variables contained in the clause or clause set C
assignment ≡ α:var(C) $\rightarrow$ {0,1}
satisfiedclause ≡ if $\exists$v $\in$ var(c), v$\in$c and α(v)=1 or $\neg$v$\in$c and α(v)= 0. Otherwise falsified
satisfiedform ≡ each clause in the formula is satisfied by assignment

Connection to graphs with [5]

SAT Handbook: Even finding a single solution can be a challenge for such problems; counting the number of solutions is much harder.Not surprisingly, the largest formulas we can solve for the model counting problem with state - of - the - art model counters are orders of magnitude smaller than the formulas we can solve with the best SAT solvers. Generally speaking, current exact counting methods can tackle problems with a couple of hundred variables, while approximate counting methods push this to around 1, 000 variables.

2.2. Monadic Second Order Logic

See also figure 2.

Explain graphs?Node, Edge

MSO graph properties are "fixed-parameter-tractable" with respect to clique-width and tree-width. `https://www.youtube.com/watch?v=hZI-wANHO1w` 5th workshop on Graph Classes, Optimization, and Width Parameters (GROW 2011) 2011-10-28. MSO

counting (k-colorings)and optimizing (distance between two vertices...) functions. Interested in MSO logic over graphs.

Two types of MSO formulas *or logical graph representations.*

- MSO formulas
- MSO_2 formulas with edge quantification ≡ MSO formulas over incidence graphs
- G=(vertices, edges as binary relation)
- INC(G) = (vertices and edges, Inc) for G undirected: Inc(e,v) ¡-¿ v is a vertex of edge e
- FPT for clique width
- FPT for tree-width

This can also be done for directed graphs!

Typical MSO_2 graph properties:

has a perfect matching has a Hamiltonian circuit spanning tree of degree ≤ 3

The expressions have the form: "There exists a **set of edges** that is..." can not be transferred into "set of vertices"

`https://youtu.be/Wyn3djrYg7c?t=1385` Bruno Courcelle: Recognizable sets of graphs: algebraic and logical aspects `https://library.cirm-math.fr/Record.htm?idlist=2&record=19276851124910940339` Recording during the thematic meeting: "Frontiers of reconnaissability" the April 29, 2014 at the Centre International de Rencontres Mathématiques (Marseille, France)

FPT for model checking: An algorithm is FPT if it takes time $f(k) \cdot n^c$ for some fixed function f and constant c. The size of the input is n. The value k is a parameter of the input. This algorithm is then usable for small values of k. usually tree-width and clique width.

2.3. DIMACS Format

Developed in 1993 at Rutgers University. DIMACS (the Center for Discrete Mathematics and Theoretical Computer Science)

Wolfram Language fully supports the DIMACS format for storing a single undirected graph. `https://reference.wolfram.com/language/ref/format/DIMACS.html`

DIMACS CNF: This format is used to define a Boolean expression, written in conjunctive normal form. `https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html`

Other formats for WMC, different graphs...

Supported also in Maple `https://www.maplesoft.com/support/help/maple/view.aspx?path=Formats/CNF`

Examples in appendix

2.4. Tree Decomposition

[5]chapter 2.2 Tree decompositions were originally introduced by Robertson and Seymour [6] in 1984. 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 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.

!!!Example (can take one from the visualizations) also in [7] page 169

2.5. Courcelle's Theorem

> Every graph property definable in monadic second-order logic (MSO) is decidable in linear time on graphs of bounded tree-width.
> Courcelle, Bruno (1990)[1]

For all $k \in \mathbb{N}$ and MSO-sentences 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.

- *drawback:* still expensive ($2^{p(twG)}$, $2^{2^{(\#Q)}}$, large constants)

The workflow then looks like we see in figure 1.

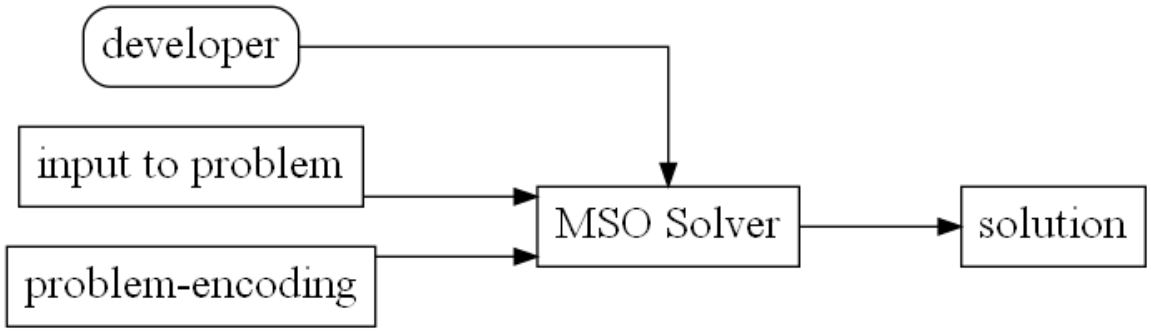

Figure 1: Implementation of the theorem

[1]Courcelle, Bruno "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs", Information and Computation, 85 (1990) no. 1: 12-75

3. Concept

We did setup a board with `https://trello.com` to track the progress and gather information connected to the project. Discussion and problems were mostly discussed with a dedicated https://slack.com/intl/en-de/ in the *Collaborations Parameterized Algorithms and Complexity* team.

Research: language (python - explain) graph-construction (graphviz vs networkX), examples (diploma at first).

My previous experience connected to this work mainly stemmed from these two very good 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. [8]

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.

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

The first stages 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). Files / Classes / Methods Current perspective Checking with www.deepcode.ai

4.1. 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 [1], 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.

Usage: ./gpusat [OPTIONS]

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

4.1.1. Class Graphoutput

First steps in automatically visualizing the solving process with its tree decomposition. Outputs a .dot graph !!!EXAMPLE!!! with the bags and their solution nodes.

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"
- a graph representing the tree decomposition of the primal graph with it's bags containing variables. output as tdFile = "cypherTreedec.txt"

4.1.2. 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.

4.2. 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 [9] based on the implementation by NetworkX [10] in bidirectional_dijkstra. in our 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* 1 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. At this time the solver handles the problems of *satisfiability* (Sat), *count solutions to a Boolean formula* (SharpSat) as well as *minimum vertex cover* (VertexCover).

Beispiel

5. Application and Images

6. Summary and Outline

What is achieved? What worked good, what bad?

References

[1] "Graph database use cases and solutions." `http://neo4j.com/use-cases`, Aug. 2016. [Online; accessed 11-June-2020].

[2] B. Courcelle and J. Engelfriet, *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach*. Cambridge: Cambridge University Press, 1 ed., 2012.

[3] A. Langer, F. Reidl, P. Rossmanith, and S. Sikdar, "Evaluation of an mso-solver," *Proc. of ALENEX 2012*, Jan. 2012.

[4] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), p. 151–158, Association for Computing Machinery, 1971.

[5] M. Zisser, *Solving the #SAT problem on the GPU with dynamic programming and OpenCL*. 2018.

[6] N. Robertson and P. Seymour, "Graph minors. iii. planar tree-width," *Journal of Combinatorial Theory, Series B*, vol. 36, no. 1, pp. 49 – 64, 1984.

[7] J. K. Fichte, "Parameterized complexity and its applications in practice." Summer 2019 (May 6th – May 16th).

[8] H. Voigt and A. 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/`, Feb. 2019. [Online; accessed 11-June-2020].

[9] A. V. Goldberg, C. Harrelson, H. Kaplan, and R. F. Werneck, "Efficient point-to-point shortest path algorithms." `https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf`, 2006. [Online; accessed 14-June-2020].

[10] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

A. Images

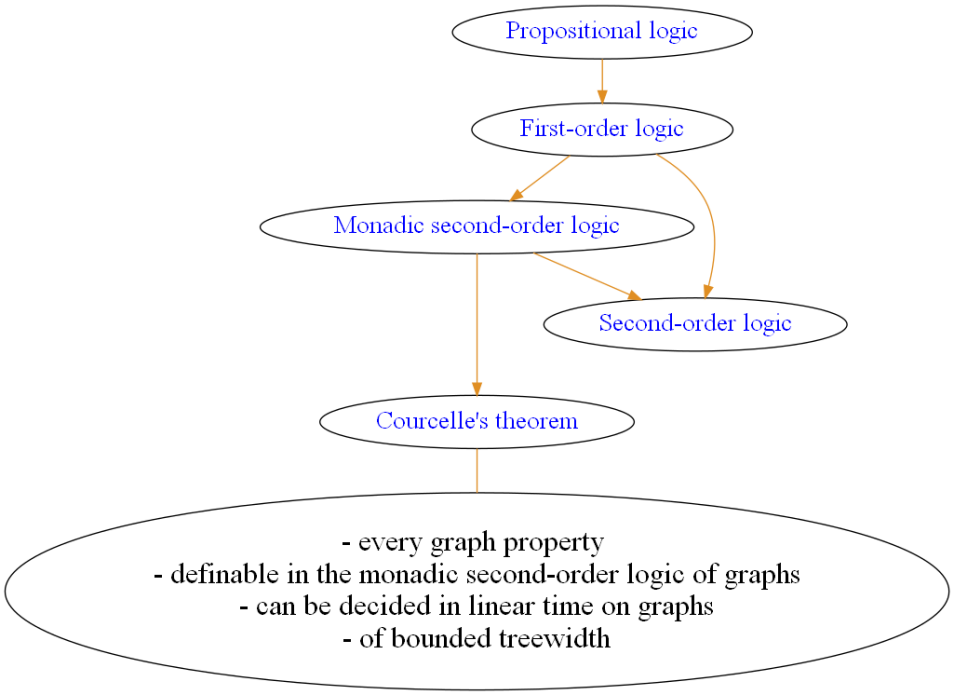

Figure 2: From propositional logic to monadic second order logic and Courcelle's Theorem

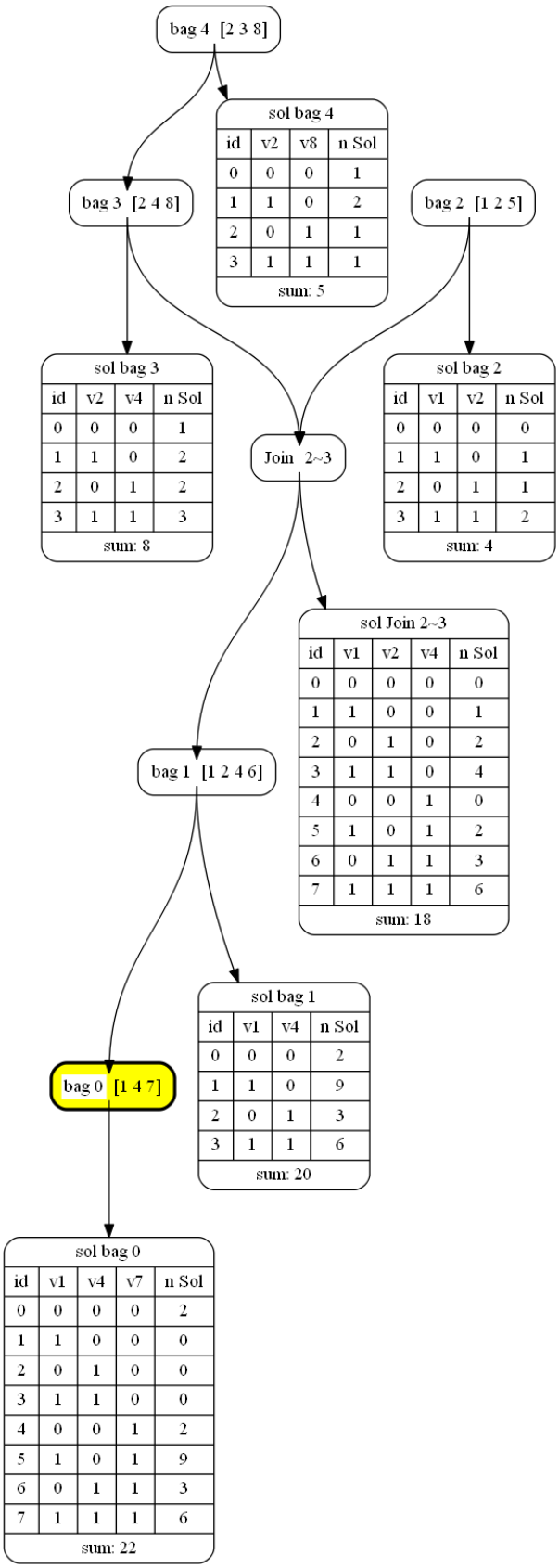

Figure 3: Created scalable-vector-graphic directly from 5

B. Code Snippets

Listing 1: 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 {}
```

C. Input Examples

Listing 2: Edge encoding of example graph with 16 vertices

```
p tw 16 36
1 2
2 1
2 3
3 2
3 4
4 3
3 5
5 3
4 5
5 4
4 6
```

```
6  4
6  7
7  6
7  8
8  7
8  9
9  8
9  10
10  9
9  11
11  9
11  12
12  11
12  13
13  12
12  14
14  12
11  14
14  11
14  7
7  14
6  15
15  6
15  16
16  15
```

Listing 3: CNF clauses from example 4.1 on page 27 [5]

```
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 4: 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 5: 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]
}
```