A-FOLIO DPDB – SYSTEM DESCRIPTION FOR ICCMA 2021

Johannes K. Fichte
International Center for Computational Logic
TU Dresden
01062 Dresden, Germany
johannes.fichte@tu-dresden.de

Markus Hecher
Institute of Logic and Computation
TU Wien
Favoritenstraße 9-11, 1040 Wien, Austria
hecher@dbai.tuwien.ac.at

Piotr Gorczyca
TU Dresden
01062 Dresden, Germany
gorczycapj@gmail.com

Ridhwan Dewoprabowo
TU Dresden
01062 Dresden, Germany
ridhwan.dewoprabowo@gmail.com

ABSTRACT

A-Folio DPDB[1] is a *portfolio* system build on top of an existing solver DPDB[2]. DPDB is designed to solve #P problems such as #SAT or Minimal Vertex Cover using dynamic programming and a database management system. A-Folio DPDB extends its functionality by additionally supporting the tasks of counting complete and stable extensions given an argumentation framework.

1 Introduction

Bounded treewidth is one of the most cited combinatorial invariants, that originates from graph theory and is well studied in the area of parametrized complexity. Informally, it can be described as a measurement of graph's tree-likeness. For several problems hard for complexity class NP, there are algorithms running in polynomial time under the assumption that a given parameter (e.g., treewidth) is fixed, indicating *fixed-parameter tractability*[1] of said algorithms. Practical implementations exploiting treewidth are oftentimes based on *dynamic programming (DP)*, where a *tree decomposition (TD)* is traversed in post-order, i.e., for each node of a TD tables are computed. An algorithm that runs on a TD node (called *local algorighm*) usually does so in time exponential in the size of the node. Given that a graph's treewidth is defined as the size of the largest node (minus 1) in a TD among all TD's of the graph, for instances of high treewidth these DP approaches in practice become intractable. Nevertheless, in the area of Boolean satisfiability this approach proved to be successful for counting problems and it seems only rational to apply it to other areas, e.g. abstract argumentation.

2 DPDB

Due to the space reasons of this description only the gist of the underlying idea for DPDB is described below. For readers interested in detailed explanation, we refer to [2].

DPDB is a general framework designed for solving counting problems. It is written in Python3 and employs PostgreSQL as a *database management system (DBMS)*.

The initial task that DPDB was capable of solving was the #SAT problem, which asks for the number of satisfying assignments for a given propositional formula in *DIMACS CNF* format. The process is as follows: first, the propositional formula is translated into a so-called *primal graph* – a graph, where each node represents a variable and there is an edge between every pair of variables appearing together in a clause. Once a primal graph is obtained, a tool called htd[3]

[1] https://github.com/gorczyca/dp_on_dbs/tree/competition
[2] https://github.com/hmarkus/dp_on_dbs
[3] https://github.com/TU-Wien-DBAI/htd/

is used to calculate its tree decomposition. The obtained tree decomposition is then traversed in post-order and for every node a table is generated with a `BOOLEAN` column for every propositional variable, containing satisfying variable assignments restricted to those clauses whose variables are contained in the node. In order to calculate the satisfying assignments, *Structured Query Language (SQL)* `SELECT` queries are generated that correspond to the aforementioned clauses and the task itself is delegated to a DBMS. Solving this task with DBMS not only has the advantage of naturally describing and manipulating the tables that are obtained during DP, but also allows to benefit from sophisticated database technology query optimization methods, data-dependent execution plans, capability of dealing with huge tables using limited amount of main memory (RAM) or dedicated database joins.

`DPDB` can be extended to other counting problems by implementing generators of SQL `SELECT` queries corresponding to DP local algorithms for those problems.

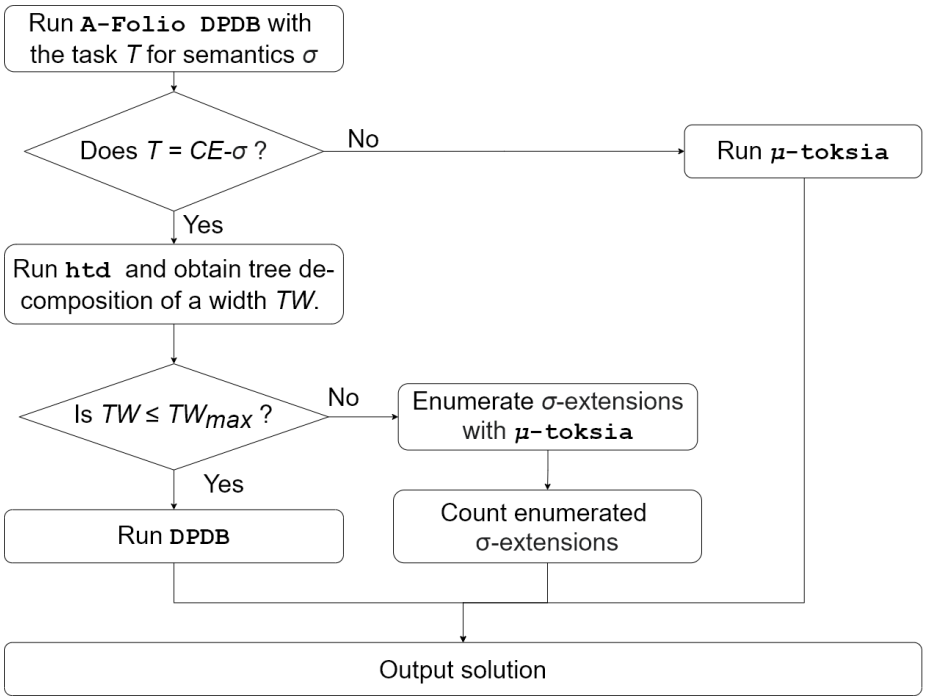

Figure 1: Flowchart of `A-Folio DPDB` system.

3 Extension of DPDB for counting extensions

Recall that the admissibility-based semantics can be defined not only with respect to the sets of arguments that a particular extension defends or attacks, but also using the notion of a *complete labelling*. A complete labelling is a total function $\mathcal{L} : Ar \mapsto \{in, out, undec\}$ where Ar is the set of frameworks' arguments and it holds that (for $a \in Ar$):

- $\mathcal{L}(a) = in$ if all a's attackers are labelled out or
- $\mathcal{L}(a) = out$ if there is an attacker of a labelled in or
- $\mathcal{L}(a) = undec$ if neither of the two above cases hold.

From that perspective an extension corresponds to the set of all arguments labelled in. For complete semantics, an extension corresponds to any complete labelling, whereas for stable semantics only those complete labelling that label no argument with $undec$ correspond to stable extensions.

Since abstract argumentation framework is a directed graph, an instance of the framework can be directly used to obtain the tree decomposition. Given the tree decomposition, each node corresponds to a subframework induced by arguments contained in a node. Again the tree decomposition is traversed in post-order with a SQL `SELECT` query being generated at each node that finds labellings of the subframework.

One of the differences between the #SAT problem and counting extensions is that in #SAT every variable is mapped to either $true$ or $false$. For counting extensions things get more complicated. Once again, below we present only the main ideas for local algorithms and refer to [3] for details regarding algorithms.

3.1 Stable semantics

As mentioned above, a complete labelling corresponding to a stable extension does not label any argument with $undec$. Equivalently, a stable extension attacks every argument it does not accept, thus requiring every argument to be labelled either in or out.

Because of the fact that each node of the tree decomposition contains only a subset $A \subseteq Ar$ of arguments and at the time of the node's processing not all attack relations have been considered, it is necessary to introduce another labelling $\mathcal{L}_{st} : A \mapsto \{in, out, out_e\}$ where $(a, b \in A)$:

- $\mathcal{L}(a) = in$ if a belongs to the extension or
- $\mathcal{L}(a) = out$ if a is attacked by an argument b, such that $\mathcal{L}(b) = in$ (we say that a has been defeated), or
- $\mathcal{L}(a) = out_e$ if a is not in the extension, but so far has not been defeated either.

All arguments labelled with out_e are expected to be defeated at some point while traversing the tree decomposition and therefore change their label to out. If that is not the case, such a labelling is not counted as a corresponding stable extension.

Since during the computation every argument has to be mapped to one of the three values $\{in, out, out_e\}$, this value is represented in a database table by either two `BOOLEAN` columns or with the addition of `null` as a `BOOLEAN` variable.

3.2 Complete semantics

Every complete labelling corresponds to a complete extension, i.e., every argument can be labelled either in, out or $undec$. For the same reasons as in case of stable semantics it is necessary to distinguish whether an argument is expected to be labelled out and $undec$, or is proven to be labelled as such, thus we distinguish new labelling $\mathcal{L}_{co} : A \mapsto \{in, out, out_e, undec, undec_e\}$ where $(a, b \in A)$:

- $\mathcal{L}(a) = in$ if a belongs to the extension or
- $\mathcal{L}(a) = out$ if a is attacked (defeated) by an argument b, such that $\mathcal{L}(b) = in$, or
- $\mathcal{L}(a) = out_e$ if a is not in the extension, but so far has not been defeated either, or
- $\mathcal{L}(a) = undec$ if a is attacked by an argument b, such that $\mathcal{L}(b) = undec$ and a does not attack nor is attacked by an argument labelled in, or
- $\mathcal{L}(a) = undec_e$ if a is expected to be labelled $undec$, but so far is not attacked by an argument b, such that $\mathcal{L}(b) = undec$ and a does not attack nor is attacked by an argument labelled in.

Note that arguments can be labelled $undec$ only if they appear in a cycle which all arguments of are labelled $undec$ too, thus in order to label an argument $undec$ it has to be attacked by an argument labelled $undec$.

Again, it is required that every argument labelled out_e or $undec_e$ has at some point its label changed to out or $undec$, otherwise the labelling does not count.

Since there are five possible values ($\{in, out, out_e, undec, undec_e\}$) of an argument labelling, the value is represented in a database table by either two `BOOLEAN` columns, one of which uses the `null` value or by `BOOLEAN` and a `SMALLINT` columns.

4 A-Folio DPDB's architecture

By the term of a *portfolio* solver we understand a solver that internally executes calls to other solvers. This is the case with our solver: in order to support the stable and complete sub-tracks of the static abstract argumentation track, `μ-toksia`[4][4] is used for tasks other than counting, i.e., the tasks DS-σ, DC-σ and SE-σ for $\sigma \in \{CO, ST\}$. For some input instances where the treewidth exceeds some constant treshold, `DPDB` cannot be used either. In those rare cases the counting is also performed with the help of `μ-toksia` – by naive counting of all enumerated extensions. The chart of Figure 1 presents the flow of our solver.

[4] `https://bitbucket.org/andreasniskanen/mu-toksia/`

5 Supported tasks

`A-Folio DPDB` supports two sub-tracks of the static abstract argumentation track corresponding to stable and complete semantics, i.e., the tasks:

- CE-ST, CE-CO – directly (assuming small treewidth, otherwise by a call to μ-`toksia`),
- DS-ST, DS-CO, DC-ST, DC-CO, SE-ST, SE-CO – by a call to μ-`toksia`.

6 Future work

We expect our solver to support the counting task for all semantics listed in competition rules, that is also for preferred, semi-stable and stage semantics. Furthermore, we plan on modifying the current implementation to utilize the nested dynamic programming paradigm[5] in order to increase the maximal treewidth acceptable by our solver.

References

[1] Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[2] Johannes K. Fichte, Markus Hecher, Patrick Thier, and Stefan Woltran. Exploiting database management systems and treewidth for counting. 2020.

[3] Günter Charwat. Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, TU Wien, 2012.

[4] Andreas Niskanen and Matti Järvisalo. μ-toksia: An efficient abstract argumentation reasoner. In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 800–804, 2020.

[5] Markus Hecher, Patrick Thier, and Stefan Woltran. *Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology*, pages 343–360. 2020.

[6] Johannes K. Fichte, Markus Hecher, and Arne Meier. Counting complexity for reasoning in abstract argumentation. In *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, pages 2827–2834, 2019.