SHARP

A Smart Hypertree-Decomposition-based Algorithm FRamework for Parameterized Problems

STUDENT PROJECT DOCUMENTATION

Michael Morak

0627699

Institute for Information Systems E184

Database and Artificial Intelligence Group

Supervisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran

Contents

Introduction

1 Motivation

1.1 Parameterized Complexity Theory

The field of parameterized complexity has seen lots of activity in recent years (a good overview can be found in e.g. [10]). In order to find efficient algorithms for computationally hard problems in terms of the size of the input, one tries to find certain parameterized versions of the problem where an instance consists of the input (of size n) and a parameter of some kind (usually an integer, say k). By doing this one hopes to find algorithms that exhibit a runtime that can be upper bounded by the following expression:

$$O(f(k)\cdot p(n))$$

where $f(k)$ is a function depending only on the value of the parameter k and $p(n)$ is some polynomial of n. Courcelle's theorem lays the foundations of this kind of approach in [2].

When such an algorithm is found, one can easily see that, by considering the parameter as fixed and therefore constant, the upper bound for the overall runtime reduces to $O(p(n))$ which makes the problem tractable. The parameterized problem is therefore called "Fixed-Parameter Tractable" or, in short, FPT. As $f(k)$ usually is a highly exponential function when considering computationally hard problems, the thus obtained algorithm is only efficient as long as the parameter (that is, k) remains small. Many such algorithms have recently been published (see e.g. [3], [5], [8], [9]).

1.2 Generalized Approach

Our motivation to develop **SHARP** essentially lies in the fact that to obtain algorithms for problems in this way one usually follows a uniform approach:

1. From the input problem, obtain the parameterized representation of the problem.
2. Using the fixed-parameter algorithm, obtain the solution.

This can be narrowed down even further when considering only a specific parameterization: Hypertreewidth (or just treewidth for trees). The algorithm then basically follows the following steps:

1. From the input problem, obtain a hypertree decomposition with a specific hypertreewidth, representing the parameterized version of the problem.
2. On the thus obtained hypertree decomposition, do a traversal of the tree in order to check for a solution.
3. Depending on what type solution is sought (e.g. enumeration of answer-sets of an answer-set program), do a second traversal of the tree, generating only the relevant solutions in the process.

SHARP provides a framework to easily implement algorithms of this type for various problems building on hypertree decompositions. For someone seeking to turn a theoretical fixed-parameter algorithm based on treewidth into an actual runnable program, the framework provides a set of base classes that provide the means to focus on implementing the actual algorithm without having to deal with the problems of data management, flow control or the like, which usually is a time-consuming and tedious task.

Framework Description

2 Parsing

2.1 Responsibilities of the Parser

Parsing the input usually is the first thing that needs to be done. The framework allows much freedom at this point as different problems usually tend to have different input formats therefore not much common ground could be found to provide base functionality out of the box.

The usual method (which is currently used in all existing algorithm implementation) is to write a lexer and parser with lex[1] and yacc[2] set of tools.

Also, if possible one might run certain preprocessing and pre-optimization subroutines at this point (e.g. redundancy reduction/elimination, consolidation, etc.).

2.2 Provided Parameters

The parser should be able to start work with two parameters: A stream which yields the entire input when read and a pointer to the `Problem` class (see section 5.1).

In order to work with the framework, the parser has to store the input in such a way that later on the hypergraph representation of the problem can be constructed from it. This can either be in form of a hypergraph itself or in the form of an intermediate data structure more suited for working with later on in the algorithm. Usually this is done by implementing the necessary methods in the problem class and calling them from the parser class, whereby all the input data is stored in the `Problem` class.

[1] `http://dinosaur.compilertools.net/lex/index.html`
[2] `http://dinosaur.compilertools.net/yacc/index.html`

3 Hypertree Decomposition

Once the input is read, the goal is to obtain the parameterized version of the problem or, in other words, to generate a hypertree decomposition of the hypergraph representation of the input. This is done in three steps:

1. Obtain the hypergraph representation of the input.
2. Decompose the thus created hypergraph into a hypertree.
3. Normalize the hypertree for easier use later on in the algorithm.

Each of these steps is discussed in detail in the following sections.

3.1 Hypergraph Representation

In order to use the **SHARP** framework, it must be possible to represent the problem (or at least certain aspects thereof) by a graph or hypergraph. This usually is the case when the problem consists of entities and relations between entities. The following is a short list of examples:

- Answer-set programs: Here, the incidence graph servers as a graph representation of the problem (i.e. the vertices of the graph are the rules and variables in the program and for each occurrence of a variable in a rule an edge between the two is added to the graph).
- Argumentation problems: Here the vertices are the arguments and the attack relation directly corresponds to the edges in the graph.
- Multi-Cut problems: Here the input is already a graph and therefore can be directly used for the hypertree decomposition step.

The hypergraph representation in the framework is represented by the `Hypergraph` class. This is a straight-forward approach as this class consists of two collections, one for the vertices and one for the hyperedges. In order to fill the `Hypergraph` class so that the framework can work with it, one has to instantiate the `Node` class for each vertex and the `Hyperedge` class for each hyperedge between the two. The following is an example, instantiating a graph that has two vertices and one edge that connects them.

Listing 3.1: Example of a graph with two nodes and one edge.

```
Hypergraph *hg = new Hypergraph();
Node *a = new Node(1, 1), *b = new Node(2, 2);
Hyperedge *e = new Hyperedge(1, 1);
e->insNode(a); e->insNode(b); a->insEdge(e); b->insEdge(e);
a->updateNeighbourhood(); b->updateNeighbourhood();
e->updateNeighbourhood();
hg->iMyMaxNrOfNodes = 2;
hg->iMyMaxNrOfEdges = 1;
```

However, in order to avoid having to implement the tedious subroutine of moving vertices and edges into the `Hypergraph` class, the framework provides a ready-to-use method doing exactly that for the case that the hypergraph representation of the problem is indeed a graph (i.e. each edge connects exactly two vertices). This (static) method can be found in the `Problem` class and has the following signature:

Listing 3.2: The createHypergraphFromSets method signature.

```
Hypergraph *Problem::createHypergraphFromSets
                (VertexSet, EdgeSet);
```

The `VertexSet` and `EdgeSet` data types are defined as an STL set of integers (i.e. `std::set<int>`) and an STL map from integers to integers (`std::map<int, int>`) whereby each integer represents the internal number of the corresponding vertex in the graph. This internal number should be defined during the parsing process as it is the only way to map the labels in the hypertree decomposition back to the corresponding vertices.

3.2 Decomposition

The hypertree decomposition step turns the hypergraph representation of the problem (as provided in the `Hypergraph` class) into a hypertree, trying to minimize the treewidth in the process. As finding a hypertree decomposition with minimal width is in itself an NP-hard problem, this is done by a heuristic to speed things up. Therefore the generated hypertree will not necessarily have minimal width with a relative performance guarantee of at most 2. The specific heuristics currently implemented can be found in [4]. Standalone implementations can be obtained from `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`.

The hypertree decomposition routine yields an instance of the `Hypertree` class which represents one of the possible hypertrees corresponding to the given hypergraph. Each of the nodes in this hypertree consists of a set of vertices of the hypergraph (also called a “bag”). The treewidth of the thus obtained hypertree is then defined as the maximum bag size in the hypertree, minus one.

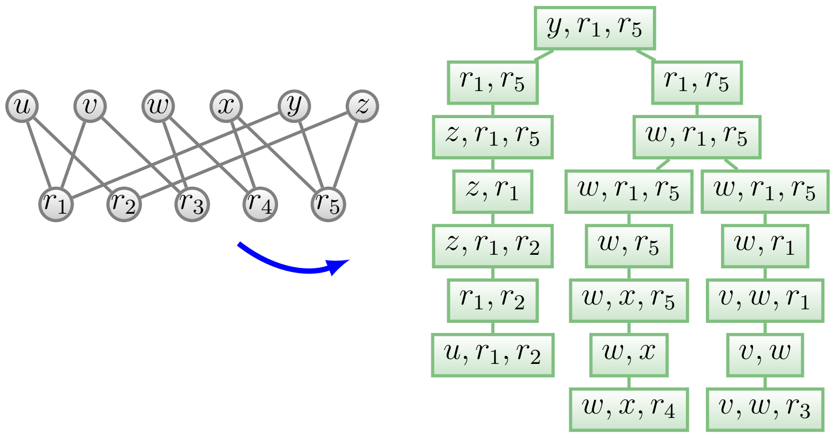

Figure 3.1: Tree decomposition of a graph.

A tree decomposition example can be seen in figure 3.1. It shows an already normalized tree decomposition of a graph. A tree decomposition of a graph is a labeled tree whose labels (bags) consist of the vertices of the graph and for which the following conditions hold:

- Two vertices connected by an edge in the graph must occur together in at least one bag.
- All the bags containing a specific vertex in the tree decomposition must be connected.

Both these conditions are obviously satisfied in the tree in figure 3.1, therefore it is a valid hypertree decomposition of the graph on the left. Observe that if a node in the hypertree has two children, these children will have exactly the same bags as their parent.

3.3 Normalization

In the normalization step the previously obtained `Hypertree` class is transformed into an instance of the `ExtendedHypertree` class which provides additional methods for easily accessing the bags in the hypertree. During this conversion the hypertree is normalized. This means that the difference between each node in the hypertree and its child is reduced to at most one by introducing intermediate nodes that each introduce or remove exactly one vertex.

4 Algorithm

4.1 Walking the Tree

This is where the “action” takes place. The `AbstractAlgorithm` class represents a hypertree-based algorithm. There are only four types of nodes in the hypertree decomposition:

- Leaf nodes: These are the leafs of the hypertree decomposition.
- Branch nodes: Here, the hypertree splits, i.e. this node has two children. As discussed earlier, the children of a branch node have exactly the same bag as the branch node itself.
- Vertex introduction node: This node, when compared to its child, introduces an additional vertex. Observe that, as the hypertree decomposition is normalized, only exactly one vertex is introduced.
- Vertex removal node: This node, when compared to its child, removes one vertex. Observe that, as the hypertree decomposition is normalized, only exactly one vertex is removed.

4.2 The Algorithm Class

Therefore the algorithm has to specify actions taking place when each of these node types is encountered. Listing 4.1 shows the declaration of the algorithm class as seen in the corresponding C++ header file. In order to implement an algorithm in the **SHARP** framework, one has to implement the five purely virtual (i.e. abstract) methods as seen in the listing.

Listing 4.1: The `AbstractAlgorithm` class header.

```
class AbstractAlgorithm
{
public:
        AbstractAlgorithm(Problem *problem);
        virtual ~AbstractAlgorithm();

protected:
        Instantiator *instantiator;
        Problem *problem;

public:
        void setInstantiator(Instantiator *instantiator);
        Solution *evaluate(const ExtendedHypertree *root);

protected:
        virtual Solution *selectSolution(
                TupleSet *tuples,
                const ExtendedHypertree *root) = 0;

        virtual TupleSet *evaluateLeafNode
                (const ExtendedHypertree *node) = 0;
        virtual TupleSet *evaluateBranchNode
                (const ExtendedHypertree *node) = 0;
        virtual TupleSet *evaluateIntroductionNode
                (const ExtendedHypertree *node) = 0;
        virtual TupleSet *evaluateRemovalNode
                (const ExtendedHypertree *node) = 0;

        TupleSet *evaluateNode
                (const ExtendedHypertree *node);
};
```

The `evaluate*Node` methods represent the actions the algorithm takes when encountering one of the four node types. The framework automatically calls the correct method when the `evaluateNode` method is called for a node.

4.3 Tuples and Possible Worlds

In order to pass data calculated in one node on to the next one, the concept of tuples is introduced: One `Tuple` instance represents a possible world for a particular node. Basically a bottom-up hypertree-based algorithm calculates at each node all the possible worlds for that node by applying incremental operations to all the possible worlds of its child node(s). All "impossible" worlds are simply omitted or eliminated (note that this results in a dynamic programming algorithm).

As the `Tuple` data structure varies strongly from algorithm to algorithm, one needs to derive the `Tuple` class and define the data structures needed for each tuple (i.e. each possible world). As an example, one might consider an algorithm for the **SATISFIABILITY** problem. Here each tuple just contains the set of positive variables, the set of negative variables and the set of clauses, as these three sets suffice to represent a possible world in a node (the bags of the nodes would then contain as vertices a subset of the variables and a subset of the clauses of the original problem). Note that the set of negative variables could also be stored implicitly as the difference of the variables in the Node and the positive variables.

Usually the information that is stored in the `Tuple` class is enough to calculate an answer for the decision problem (i.e. answer “yes” or “no”). However, usually one also wants to actually find one or more solutions to the problem (i.e. in case of the **SATISFIABILITY** problem, one wants to not only know that the input formula is satisfiable, but also what all the satisfying truth assignments look like). In order to do just that, the concept of a `Solution` is introduced.

4.4 Solution and SolutionContent classes

Each `Tuple` in a node represents a possible world from which one ore more partial–partial in the sense that one node only represents a part of the whole problem–solution to the problem can be generated. However calculating all (partial) solutions during the bottom-up traversal would ultimately lead to an exponential running time for hard problems which is what we want to avoid. Also many of these partial solutions would never be needed again as the corresponding `Tuple` may at some point be eliminated. Therefore the **SHARP** framework provides the `Solution` and `SolutionContent` classes. Again the structure of a solution to the problem depends on the problem, therefore one has to define the data structure for the solutions. This is done by deriving the `SolutionContent` class. One instance of the `SolutionContent` class represents one or more partial solutions associated with a `Tuple` instance (i.e. with a possible world).

The framework already provides ready-made derived `SolutionContent` classes for some common solution types (namely Enumeration, Counting and Boolean solution types). However, new derived `SolutionContent` classes may be implemented as needed. Each `SolutionContent` must provide implementations for the three (idempotent) merging operations:

- **Union**: When by modifying the `Tuple` instances from a child node (i.e. in an introduction or removal node) after modification two `Tuple` instances coincide, their associated `SolutionContent` instances are merged using the `calculateUnion` method.
- **CrossJoin**: When in a branch node a `Tuple` instance of the left child and right child are joined, their associated `SolutionContent` instances are merged using the `calculateCrossJoin` method.
- **AddDifference**: When in an introduction node a `Tuple` instance is modified by incorporating the introduced vertex, its associated `SolutionContent` instance incorporates the same vertex using the `calculateAddDifference` method.

All these operations can be triggered by calling the appropriate method of the provided `Instantiator` class instance associated with the algorithm. This class handles `SolutionContent` instantiation and also ensures lazy evaluation of solutions if needed.

4.5 Skeleton Implementation

Once the `Tuple` class is derived from and the algorithm-specific `Tuple` is defined, one can start implementing the respective node evaluation methods. For a bottom-up traversal, the implementation usually takes the form of listing 4.2 (here the case of an introduction node is taken as an example).

Listing 4.2: Generic skeleton for implementing the algorithm methods.

```
TupleSet *SomeAlgorithm::
evaluateIntroductionNode(const ExtendedHypertree *node)
{
// call this method first to do a bottom-up traversal
TupleSet *base = this->evaluateNode(node->firstChild());

// instantiate the new Tuple set for this node
TupleSet *ts = new TupleSet();

for(TupleSet::iterator it = base->begin();
        it != base->end(); ++it)
{
        SomeTuple &told = *(SomeTuple *)it->first;

        // calculate the new Tuple based on the
        // old one and the node type/difference
        // NOTE: this is where the actual code goes
        SomeTuple &tnew = modify(x, node);

        // incorporate the change into the solution
        // by calling the appropriate method of the
        // Instantiator instance
        // NOTE: This is only an example, your code
        //       may call other Instantiator methods
        Solution *snew = this->instantiator->
                addDifference(it->second,
                        node->getDifference());

        // try to insert the new Tuple into the set
        pair<TupleSet::iterator, bool> result =
                ts->insert(TupleSet::value_type(
                        &tnew, snew));
        // if the very same Tuple is already contained
        // in the Tuple set, merge the solutions using
        // the Union operation, then insert it instead
        // of the old one
        if(!result.second)
        {
                Solution *sold = result.first->second;
                ts->erase(result.first);
                ts->insert(TupleSet::value_type(&tnew,
                        this->instantiator->combine
                               (Union, sold, snew)));
        }
}

// free up some memory, old Tuples not needed anymore
delete base;

// return the new Tuple set
return ts;
}
```

After evaluating the last node (i.e. the root node of the decomposition), the method `selectSolutions` is called. This method should simply check all TupleSets and return the Union (using the `Instantiator`) of all solutions that belong to valid tuples.

5 Pulling it all together

Now that we have all the principal components to run our algorithm, the only thing that still lacks is a class that provides the necessary program flow control (i.e. that determines when to do what and in which order). In order to do this, the framework provides the `Problem` class:

5.1 The Problem Class

The `Problem` class is provided as a base class by the **SHARP** framework and handles the earlier mentioned task of flow control. The `Problem` class also acts as the interface to the "outside world" such that it provides simple methods to read a problem instance and generate a solution for it.

For each new problem the framework should handle, the `Problem` class needs to be derived from (i.e. one may have an `AnswerSetProblem` class for answer set programs and a `SatisfiabilityProblem` class for the **SATISFIABILITY** problem). Each one of these derived classes then may use multiple different parsers (i.e. for different input formats of the same problem) and (in the future, this is not implemented yet) different hypertree decomposition algorithms (i.e. special optimized versions if the input is known to be a graph instead of a hypergraph etc.).

In order to implement a `Problem` class, one needs to implement three methods:

1. The `parse` method: This method should call the parser (if there is one) and store the problem in an internal data format, which can be defined as needed (i.e. private fields in the Problem class, etc.).
2. The `preprocess` method: This method should perform preprocessing tasks on the problem stored in the internal data format.
3. The `buildHypergraphRepresentation` method: This method should convert the internal representation of the problem to an instance of the `Hypergraph` class, as discussed in section 3.1.

The `Problem` class must be instantiated by passing as a parameter in the constructor an instance of the `AbstractAlgorithm` class that should be used to solve the problem. Once this is done and the methods discussed above are implemented, the

`Problem` class is ready for use and will use the specified algorithm to solve the problem read by the `parse` method. This is done by calling the `calculateSolution` method of the `Problem` class. This method handles the program flow. Its (for the sake of readability simplified) definition is provided in listing 5.1.

Listing 5.1: Definition of the `parse` method of the `Problem` class (simplified).

```
Solution *Problem::calculateSolution(Instantiator *inst)
{
        this->parse();
        this->preprocess();

        Hypergraph *hg =
                this->buildHypergraphRepresentation();

        H_BucketElim be;
        Hypertree *ht =
                be.buildHypertree(hg, BE_MIW_ORDER);

        ht = new ExtendedHypertree(ht);
        ((ExtendedHypertree *)ht)->normalize();

        this->algorithm->setInstantiator(inst);
        return this->algorithm->
                evaluate((ExtendedHypertree *)ht);
}
```

Obviously this method does what is expected in order to run the whole algorithm from parsing to solution generation. First, the parser is started, then, after preprocessing, the hypergraph representation is built, which is then decomposed into a hypertree and subsequently normalized, after which the actual algorithm is run using the provided `Instantiator` instance.

As discussed in section 4.4, the `Instantiator` instance that is used by the algorithm determines which `SolutionContent` class will get instantiated. Therefore, by just calling the `calculateSolution` method of the `Problem` class with different `Instantiator` instances, one can easily specify what the solution should look like (i.e. a call with an `Instantiator` that creates boolean `SolutionContent` instances, only a yes/no answer is provided, whereas when the `Instantiator` creates `CountingSolutionContent` instances, the solution yields a number–counting for example the number of answer sets).

Implementation by Example

6 The dynASP Algorithm

The “dynASP” algorithm is a dynamic programming-based algorithm for solving ground answer-set programs (that is, determine the stable models of such a program). Based on a tree decomposition of the incidence graph of the program, the algorithm does a bottom-up traversal of said tree checking whether a solution to the program exists and if it does, output either just “yes”, or count the number of stable models, or enumerate the stable models. A first prototype of this implementation was presented at JELIA’10 (see [1]). Generally, answer-set problems are known to be hard (that is, Σ_P^2-complete, see e.g. [6]). However it is known to be FPT and a first FPT-algorithm was published by Michael Jakl et. al. in [7] in 2009.

As an example, take the following program:

$$\begin{array}{llllll} r_1 = u \leftarrow v, y; & r_2 = z \leftarrow u; & r_3 = v \leftarrow w; \\ r_4 = w \leftarrow x; & r_5 = x \leftarrow \neg y, \neg z. \end{array}$$

This program has an incidence graph equal to the one in figure 3.1, which means that the tree decomposition in that figure is also a valid tree decomposition for the program. The (simplified) implementation of the algorithm is described in the following sections.

6.1 Implementing the Problem Class

Listing 6.1: Definition of the `DatalogProblem` class (simplified).

```
typedef Vertex Rule;
typedef Vertex Variable;
typedef VertexSet RuleSet;
typedef VertexSet VariableSet;
typedef std::map<Rule, std::map<Variable, bool> > SignMap;
typedef std::vector<VariableSet> HeadMap;

class DatalogParser;
class DatalogProblem : public Problem
{
public:
        DatalogProblem(std::istream *stream);
        Rule addNewRule();
        Variable addVariable(std::string name);
        void addEdge(Rule rule, Variable variable,
                  bool positive, bool head);
protected: ...    // declaration of parse, preprocess
                  // and buildHG methods
private:
        SignMap signs;
        HeadMap heads;
        TypeMap types;
        DatalogParser *parser;
};
```

Listing 6.1 shows the (simplified) declaration of the derived `Problem` class. The main points are the following:

- Declaration of the internal data structures to store the answer-set problem. The `SignMap` stores the rules in **SAT** representation and the `HeadMap` stores the rule heads.
- Declaration of methods called by the parser (`addNewRule`, `addVariable`, `addEdge`) when the respective event is encountered during parsing. These methods then store the data in the `HeadMap` and `SignMap`.
- Stream that supplies the input is used as parameter of the constructor and a pointer to the parser instance (`DatalogParser`) is kept.

The implementation of the various functions is fairly self-explanatory so we omit details here but only try to make the main points clear. The constructor of the `Problem` class above simply initializes the corresponding parser class (i.e. creates an instance of the `DatalogParser`) and passes a pointer to itself and the input stream to the parser.

6.2 Implementing the Parser

The parser in this particular program is written with the lex/yacc combination as already discussed in chapter 2. The parser is written in such a way that it accepts as constructor arguments the stream that yields the input and a pointer to the `Problem` class mentioned in section 6.1. Whenever the parser encounters a new rule during parsing, the `addRule` method of the `Problem` class is called. Analogously, for each variable the `addVariable` method is called and for each occurrence of a variable in a rule the `addEdge` method is called, including information whether the variable occurs negated or in the head of the rule.

6.3 Implementing the Algorithm

Implementing the algorithm is fairly straight-forward given the structure that is already provided by the framework. Doing a bottom-up traversal of the tree using the method described in section 4.2 and 4.5 immediately results in an easy way to implement the algorithm in the framework. There are however a few points worth noting when looking at the declarations in listing 6.2:

Listing 6.2: Definition of the `AnswerSetAlgorithm` class and corresponding tuple (simplified).

```
class AnswerSetTuple : public Tuple
{
public:
        set<Variable> variables;
        set<Rule> rules;
        set<Atom> guards;
};
class AnswerSetAlgorithm : public AbstractAlgorithm
{
public:
        AnswerSetAlgorithm(Problem *problem);
protected:
        virtual Solution *selectSolution
                (TupleSet *tuples, const ExtendedHypertree *node);
        virtual TupleSet *evaluateLeafNode
                (const ExtendedHypertree *node);
        ... // declaration of all other evaluation methods
private:
        DatalogProblem *problem;
};
```

- In the constructor a pointer to the `Problem` instance is taken. This is usually needed, as all the information about the problem (i.e. in this case `HeadMap`, `SignMap`) is stored in the `Problem` instance.
- For the sake of clarity, equality and less-than operators have been omitted in the `Tuple` declaration. These operators are however needed in order to store `Tuple` instances efficiently in sets.
- In the `evaluateLeafNode` method, the `Solution` instances are created using the `Instantiator`'s `createLeafSolutions` method for every partition of the variables in the node.

6.4 Implementing the Instantiator

No separate implementation for the `Instantiator` class is needed, as the framework already provides a `GenericInstantiator` class which can be used for all `SolutionContent`-derived classes, as long as they implement the same constructors as the prototype. Also, default-implementations for enumerating solutions, counting solutions and boolean solutions are provided by the framework. The `Instantiator` instance can thus simply be obtained by using the code in listing 6.3.

Listing 6.3: Using the generic `Instantiator` class.

```
Instantiator *inst =
        new GenericInstantiator<EnumerationSolution>(true);
// or...
inst =
        new GenericInstantiator<CountingSolution>(false);
```

The parameter passed to the `GenericInstantiator` constructor is a boolean determining whether to enable lazy evaluation or not. Enable this (i.e. call with “true”) when exponential blowup is otherwise to be expected. For example when enumerating stable models, at each step in the tree enumerating all partial solutions would be infeasible. Therefore in this case we enable lazy evaluation so that only the partial solutions that actually contribute to the full solution are enumerated. On the other hand when counting is the objective, we may (almost) safely disable lazy evaluation as in each step only a number needs to be updated.

Appendices

A Class List

- `Problem`-the class providing flow control and acting as the interface to the "outside world"
- `AbstractAlgorithm`-the algorithm skeleton for hypertree-based algorithms
- `Tuple`-skeleton class, represents the data the algorithm needs to represent one possible world in a hypertree node
- `SolutionContent`-skeleton class, represents the solution data associated with one possible world (i.e. `Tuple`)
- `Solution`-framework helper class that provides lazy solution evaluation support
- `Instantiator`-framework class for instantiating the `Solution` class and the `SolutionContent` classes
- `GenericInstantiator`–framework class that provides a generic instantiator supporting lazy evaluation and supports all derived `SolutionContent` classes that implement the skeleton constructors of the `SolutionContent` class
- `EnumerationSolutionContent`–`SolutionContent` class that stores sets of sets of vertices (i.e. multiple (partial) solutions per `Tuple` may be possible here as one solution is represented by a set of vertices)
- `ConsistencySolutionContent`–`SolutionContent` class that stores a simple boolean value representing the answers "yes" or "no"
- `CountingSolutionContent`–`SolutionContent` class that stores an arbitrarily large number for counting purposes
- `Hypertree`-represents a hypertree
- `ExtendedHypertree`-represents a hypertree with extended functionality (i.e. vertex-aware, normalization capabilities, etc.) for easier use in the algorithm
- `Hypergraph`-represents a hypergraph

- `Hyperedge`-represents a hyperedge in a hypergraph
- `Node`-represents a vertex in a hypergraph

Bibliography

[1] M. Morak, R. Pichler, S. Rümmele, S. Woltran. A Dynamic-Programming Based ASP-Solver. In *Proc. JELIA'10*, volume 6341 of *LNCS*, pages 369–372. Springer, 2010.

[2] B. Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B*, pages 193–242. Elsevier Science Publishers, 1990.

[3] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The second answer set programming competition. In *Proc. LPNMR'09*, volume 5753 of *LNCS*, pages 637–654. Springer, 2009.

[4] A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI'08*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[5] W. Dvořák, R. Pichler, and S. Woltran. Towards fixed-parameter tractable algorithms for argumentation. In *Proc. KR'10*, pages 112–122. AAAI Press, 2010.

[6] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[7] M. Jakl, R. Pichler, and S. Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI'09*, pages 816–822. AAAI Press, 2009.

[8] R. Pichler, S. Rümmele, and S. Woltran. Belief revision with bounded treewidth. In *Proc. LPNMR'09*, volume 5753 of *LNCS*, pages 250–263. Springer, 2009.

[9] R. Pichler, S. Rümmele, and S. Woltran. Multicut algorithms via tree decompositions. In *Proc. CIAC'10*, volume 6078 of *LNCS*, pages 167–179. Springer, 2010.

[10] R. Niedermeier. Invitation to Fixed-Parameter Algorithms. *Oxford Lecture Series in Mathematics and its Applications*, volume 31. Oxford University Press, 2006.