Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)

ASP for Anytime Dynamic Programming on Tree Decompositions

Bernhard Bliem[1] and **Benjamin Kaufmann**[2] and **Torsten Schaub**[2,3] and **Stefan Woltran**[1]
[1] TU Wien, Vienna, Austria [2] University of Potsdam, Germany [3] INRIA Rennes, France

Abstract

Answer Set Programming (ASP) has recently been employed to specify and run dynamic programming (DP) algorithms on tree decompositions, a central approach in the field of parameterized complexity, which aims at solving hard problems efficiently for instances of certain structure. This ASP-based method followed the standard DP approach where tables are computed in a bottom-up fashion, yielding good results for several counting or enumeration problems. However, for optimization problems this approach lacks the possibility to report solutions before the optimum is found, and for search problems it often computes a lot of unnecessary rows. In this paper, we present a novel ASP-based system allowing for "lazy" DP, which utilizes recent multi-shot ASP technology. Preliminary experimental results show that this approach not only yields better performance for search problems, but also outperforms some state-of-the-art ASP encodings for optimization problems in terms of anytime computation, i.e., measuring the quality of the best solution after a certain timeout.

1 Introduction

Answer Set Programming (ASP) [Brewka *et al.*, 2011] is a vibrant area of AI providing a declarative formalism for solving hard computational problems. Thanks to the power of modern ASP technology [Gebser *et al.*, 2012], ASP was successfully used in many application areas, including product configuration [Soininen and Niemelä, 1998], bioinformatics [Guziolowski *et al.*, 2013], and many more.

Recently, ASP has been proposed as a vehicle to specify and execute dynamic programming (DP) algorithms on tree decompositions (TDs). TDs [Robertson and Seymour, 1984] are a central method in the field of parameterized complexity [Downey and Fellows, 1999], offering a natural parameter (called treewidth) in order to identify instances that can be solved efficiently due to inherent structural features. Indeed, many real-world networks enjoy small treewidth; problems like STEINER TREE can then be solved in linear time in the size of the network (see, e.g., [Chimani *et al.*, 2012]). Such efficient algorithms process a TD in a bottom-up manner, storing in each node of the TD a table containing partial solutions; see, e.g., [Niedermeier, 2006; Bodlaender, 1993]. The size of these tables is bounded by the treewidth, which, roughly speaking, guarantees the aforementioned running times. Abseher *et al.* [2014] proposed a system that calls an ASP solver in each node of the TD on a user-provided specification (in terms of an ASP program) of the DP algorithm, such that the answer sets characterize the current table. Its contents are then handed over to the next call which materializes the table of the parent node and so on.

Albeit this method proved useful for rapid prototyping of DP algorithms and performed well on certain instances, there is one major drawback: A table is always computed in its entirety before the next table is processed. Hence, the system cannot report anything before it has finished the table of the TD's root node (and thus computed all tables entirely). Moreover, this final table implicitly contains information on all solutions. This leads to situations where unnecessarily many rows are computed, in particular if we only need one solution. Even worse, in optimization problems the system cannot give any solution until all solutions have been obtained.

In this paper, we present an alternative approach to using ASP for DP on TDs that overcomes these shortcomings. Our method is based on a "lazy" table computation scheme. In particular, for optimization problems this allows us to interrupt the run and deliver the best solution found so far.

We first describe our general framework independently of ASP. Then we show how we use ASP in the core of our algorithm for computing the DP tables. In contrast to the standard approach from [Abseher *et al.*, 2014], we now require multiple coexisting ASP solvers that communicate with each other. Achieving this in an efficient way poses a challenge, however: A naive way would be to restart a solver every time new information comes in. Alternatively, the recent *multi-shot ASP solving* approach [Gebser *et al.*, 2014] might be useful, as it allows us to add information to a solver while it is running.

We implemented both alternatives and provide an experimental evaluation on several problem encodings using real-world graphs. The multi-shot approach turns out to have clear advantages. We also demonstrate that the performance of our new "lazy" algorithm is typically superior to the traditional "eager" approach. Finally, we compare our algorithm to the state-of-the-art ASP system clingo and show that on some problems our system performs better in an anytime setting.

Related Work Anytime algorithms for certain ASP problem have been investigated in the literature. Alviano *et al.* [2014] presented such an algorithm for computing atoms that occur in all answer sets. Nieuwenborgh *et al.* [2006] proposed an approximation theory for standard answer sets. Also related to our approach, Gebser *et al.* [2015] propose a method for improving the quality of solutions for optimization problems within a certain time bound. They customize the heuristics to find the first solutions faster, with the drawback that checking for optimality becomes more expensive. In contrast to that, our lazy evaluation method does not have such undesirable side-effects compared to the eager approach.

2 Background

Answer Set Programming ASP [Brewka *et al.*, 2011] is a popular tool for declarative problem solving due to its attractive combination of a high-level modeling language with high-performance search engines. In ASP, problems are described as logic programs, which are sets of rules of the form

$$\mathtt{a}_0 \leftarrow \mathtt{a}_1\mathtt{,}\ldots\mathtt{,a}_m\mathtt{,}\textbf{not}\ \mathtt{a}_{m+1}\mathtt{,}\ldots\mathtt{,}\textbf{not}\ \mathtt{a}_n$$

where each $\mathtt{a}_i$ is a propositional atom and **not** stands for *default negation*. We call a rule a *fact* if $n = 0$, and an *integrity constraint* if we omit $\mathtt{a}_0$. Semantically, a logic program induces a collection of so-called *answer sets*, which are distinguished models of the program determined by answer sets semantics; see [Gelfond and Lifschitz, 1991] for details.

To facilitate the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthand for the set of their ground instances. Further language constructs include *conditional literals* and *cardinality constraints* [Simons *et al.*, 2002]. The former are of the form $\mathtt{a:b}_1\mathtt{,}\ldots\mathtt{,b}_m$, the latter can be written as $\mathtt{s\{c}_1\mathtt{,}\ldots\mathtt{,c}_n\mathtt{\}t}$, where a and $\mathtt{b}_i$ are possibly default-negated literals and each $\mathtt{c}_j$ is a conditional literal; s and t provide lower and upper bounds on the number of satisfied literals in the cardinality constraint. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's antecedent expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2{a(X):b(X)}4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true.

We use the input language of the ASP system *clingo* [Gebser *et al.*, 2014], which allows us to set the truth values of certain *external atoms* in the ground program "from the outside". We can thus assume certain truth values for these atoms, compute answer sets, and repeat this under different assumptions, without having to re-ground the program. Atoms can be declared as external by directives of the form `#external` $\mathtt{a:b}_1\mathtt{,}\ldots\mathtt{,b}_m$ where $\mathtt{a:b}_1\mathtt{,}\ldots\mathtt{,b}_m$ is a conditional literal.

Tree decompositions Tree decompositions, originally introduced in [Robertson and Seymour, 1984], are tree-shaped representations of (potentially cyclic) graphs. The intuition is that multiple vertices of a graph are subsumed under one TD node, thus isolating the parts responsible for cyclicity.

Definition 1 *A* tree decomposition *(TD) of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ where $T = (N, E')$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*) as follows:* (1) *For each vertex $v \in V$, there is a node $n \in N$ such that $v \in \chi(n)$.* (2) *For each edge $e \in E$, there is a node $n \in N$ such that $e \subseteq \chi(n)$.* (3) *For each $v \in V$, the subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected. We call $\max_{n \in N} |\chi(n)| - 1$ the* width *of $\mathcal{T}$. We call a node $n \in N$ a* join node *if it has two children with equal bags, and we call $\mathcal{T}$* semi-normalized *if all nodes with more than one child are join nodes.*

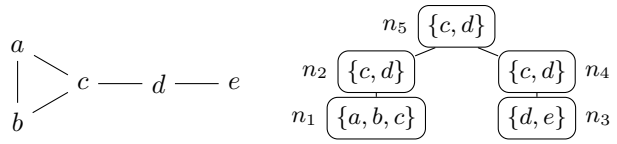

Figure 1: A graph G and a (semi-normalized) TD $\mathcal{T}$ of G

In general, constructing a minimum-width TD is intractable [Arnborg *et al.*, 1987]. However, there are heuristics that give "good" TDs in polynomial time [Dechter, 2003; Dermaku *et al.*, 2008; Bodlaender and Koster, 2010], and anytime algorithms that allow for a trade-off between time and width [Gogate and Dechter, 2004]. We can transform any TD into a semi-normalized one in linear time without increasing the width [Kloks, 1994].

Many computationally hard problems become tractable if the instances admit TDs whose width can be bounded by a constant. This is commonly achieved by DP algorithms that traverse a TD in post-order (cf. [Niedermeier, 2006]). We use the following framework for such computations: At each TD node n, partial solutions for the subgraph induced by the vertices encountered so far are computed and stored in a *table* T_n. When clear from the context, we equate nodes with their tables (e.g., "child tables" are tables at child nodes). Each table is a set of *rows* r consisting of (a) a set $\mathrm{items}(r)$ that stores problem-specific data to be handled by the DP algorithm, (b) a nonempty set $\mathrm{extend}(r)$ of tuples $(e_1, \ldots, e_k)$, where k is the number of child tables and e_i is a row in the i-th child table, and (c) an integer $\mathrm{cost}(r)$ whose intended purpose is to indicate the cost of each (partial) solution obtainable by recursively combining predecessor rows from $\mathrm{extend}(r)$. If the root table is nonempty in the end, we can obtain complete solutions by recursively combining rows with their predecessors. To achieve tractability if the width w is bounded, the size of each table should be bounded by some function $f(w)$.

Example 1 *Consider the graph G and TD $\mathcal{T}$ in Figure 1. To compute a minimum vertex cover of G via DP on $\mathcal{T}$, we start at n_1. For each vertex cover X of the subgraph induced by the current bag $\chi(n_1) = \{a, b, c\}$, we insert a row r into T_{n_1} with $\mathrm{items}(r) = X$, $\mathrm{cost}(r) = |X|$ and $\mathrm{extend}(r) = \emptyset$ (as n_1 is a leaf, r has no predecessors). This gives us one row for each of $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ and $\{a, b, c\}$.*

Now we proceed to n_2. Here a and b have been removed *($\{a, b\} \subseteq \chi(n_1) \setminus \chi(n_2)$), and d has been* introduced *($d \in \chi(n_2) \setminus \chi(n_1)$). We go through each row $r_1 \in T_{n_1}$ and try to extend it as follows. We try each subset $X \subseteq \chi(n_2)$ with $\mathrm{items}(r_1) \cap \chi(n_2) \subseteq X$. If X is a vertex cover of the subgraph induced by $\chi(n_2)$, X extends r_1 to a vertex cover of the*

subgraph induced by $\chi(n_1) \cup \chi(n_2)$, so we insert a row r into T_{n_2} with $\text{items}(r) = X$, $\text{extend}(r) = \{r_1\}$ and $\text{cost}(r) = \text{cost}(r_1) + |X \setminus \chi(n_1)|$. When we have tried out all r_1 and X, we compress T_{n_2} by discarding each row r such that there is a row s with $\text{items}(r) = \text{items}(s)$ and $\text{cost}(r) > \text{cost}(s)$ (no solution involving r will be optimal). Then we exhaustively replace all $r, s \in T_{n_2}$ such that $\text{items}(r) = \text{items}(s)$ with a single row t having $\text{items}(t) = \text{items}(r)$, $\text{extend}(t) = \text{extend}(r) \cup \text{extend}(s)$ and $\text{cost}(t) = \text{cost}(r)$.

For instance, let r_1 be the row in T_{n_1} with $\text{items}(r_1) = \{a, c\}$. We try to extend this row at n_2 by trying out subsets of $\chi(n_2) = \{c, d\}$ that contain c (as $c \in \text{items}(r_1)$ and c is still present in the bag of n_2). Both subsets $\{c\}$ and $\{c, d\}$ are vertex covers of the subgraph induced by $\chi(n_2)$, so we add a row into T_{n_2} with items $\{c\}$ and a row with items $\{c, d\}$. These rows extend r_1 to the partial solution $\{a, c\}$ and $\{a, c, d\}$, respectively. Next, to extend the row in T_{n_1} with items $\{a, b\}$, we go through all subsets of $\chi(n_2)$. The subset $\emptyset$ is not a vertex cover of the subgraph induced by $\chi(n_2)$, so we skip it. For the subset $\{c\}$, for example, we again add a row. Note that T_{n_2} now contains two rows whose items are $\{c\}$: One has cost 2 and one has cost 3. Due to the compression outlined above, we will eventually delete the latter. Finally, we merge all remaining rows that have the same items into one. For instance, we end up with a row having items $\{c\}$ and cost 2 that extends two rows in T_{n_1}: The one with items $\{a, c\}$ and the one with items $\{b, c\}$.

We handle n_3 and n_4 in an analogous way. For the join node n_5, we try out each pair of predecessors (r_2, r_4) from T_{n_2} and T_{n_4}, respectively, such that they are "compatible", i.e., $\text{items}(r_2) = \text{items}(r_4)$. From such a pair we obtain a row r at T_{n_5} with $\text{items}(r) = \text{items}(r_2)$, $\text{extend}(r) = \{(r_2, r_4)\}$ and $\text{cost}(r) = \text{cost}(r_2) + \text{cost}(r_4) - |\text{items}(r)|$ (due to the inclusion-exclusion principle). Finally we compress T_{n_5} as before. Any row at T_{n_5} whose cost is minimal among all rows in T_{n_5} can then be recursively extended to obtain a minimum vertex cover of the original graph.

Traditional TD-based DP algorithms like this follow an "eager evaluation" approach: At each decomposition node, they compute a table in its entirety based on the (entire) child tables. While this is theoretically efficient in many cases (as long as the width is bounded), it has the property that we cannot give any solution until all tables are computed and we can construct an optimal solution. In many practical applications, though, we would like to report the best solution found after a certain amount of time even if this solution is not optimal. Traditional DP on TDs lacks such an anytime feature.

3 ASP for Anytime DP on TDs

We now present our main contribution: An algorithm that performs DP on TDs via "lazy evaluation" in contrast to the traditional "eager" approach. The basic idea is that whenever a new row r is inserted into a table T, we try to extend r to a new row in the parent table, and we only compute a new row at T if all attempts of extending r have failed. In the best case, we thus only need to compute a single row at each table to reach a solution, whereas the eager approach would compute potentially huge tables. In the worst case, we compute as many rows as the eager approach, with some additional overhead due to the constant "jumping around" between tables.

3.1 Algorithm Description

Algorithm 1: Main program

```
   Input: table T_root of the root of a tree decomposition
1  lowestCost ← ∞
2  r ← nextRow(T_root)
3  while r ≠ fail do
4  |  construct a solution from r and print it
5  |  lowestCost ← cost(r)
6  |  r ← nextRow(T_root)
```

We first compute a TD and run the main program (Algorithm 1) with the (still empty) root table T_{root} as input. There we initialize the global variable lowestCost, which will keep track of the cost of the best complete solution found so far. Then we try to compute a row r at T_{root} by calling `nextRow`. If this succeeds, then r witnesses a solution whose cost is denoted by `cost(r)`. As we will see, `nextRow` only produces rows that lead to solutions cheaper than lowestCost. Hence, whenever we reach Line 4, we can use r to obtain a solution that is the best so far. Then we update lowestCost and repeat these steps to compute even better rows until this is no longer possible and we have reached the optimum.

Algorithm 2: `nextRow`

```
   Input: table T whose children we call T_1, ..., T_n
   Result: adds a row r to T s.t. cost(r) < lowestCost;
           returns r, or "fail" if there is no such row
7  if T is empty then
8  |  initialize solver_T
9  |  foreach T' in {T_1, ..., T_n} do
10 |  |  r' ← nextRow(T')
11 |  |  if r' = fail then return fail
12 |  |  active_T' ← r'
13 |  childCalledLast_T ← n
14 r ← fail
15 while r = fail do
16 |  while solver_T cannot find a new row candidate do
17 |  |  if activateNext(T) = false then
18 |  |  |  if callChild(T) = false then return fail
19 |  s ← new row candidate from solver_T
20 |  if cost(s) < lowestCost and s is cheaper than any
   |    row in T that has the same items as s then
21 |  |  insert s into T
22 |  |  r ← s
23 return r
```

In the beginning, the subroutine `nextRow` (Algorithm 2) initializes all tables (Lines 7–13): For each table T we start an instance of an external solver, denoted by solver_T, which for now we consider as a black box responsible for computing rows at T when given a combination of child rows.

Let T be any table and $T_1, \ldots, T_n$ be its children (if there are any). By the time we reach Line 13 during a call to `nextRow(`T`)`, there is exactly one row r' in each $T' \in \{T_1, \ldots, T_n\}$ that has been marked as the *active* row of T' (using the global variable active$_{T'}$ in Line 12). The meaning of active rows is the following: At any point of the execution, the tuple $(\text{active}_{T_1}, \ldots, \text{active}_{T_n})$ indicates the combination of child rows that we are currently trying to extend to a new row at T. Whenever solver$_T$ is invoked, it uses this tuple to construct row candidates that extend the active rows.

Once we have set the active rows, we try to find a row candidate s extending them. Suppose there is one (Line 19). If its cost exceeds our bound (lowestCost), or if there is a row in T having the same items as s and at most the cost of s, we continue with the next row candidate. Otherwise we insert s into T (replacing a row having the same items if possible) and return s; execution will then proceed with the parent of T making use of s (or, if T is the root, with the main program registering a new solution).

Once there are no more row candidates at T for the current active row combination (Line 16), we try the next combination of existing child rows (Line 17). If we have already tried all of those, we need to compute a new child row (Line 18). If this also fails, we know that there is no further row at T since we have processed all combinations of child rows.

Algorithm 3 sets the active rows to a combination of existing child rows that has not been considered yet. For each table T we use the global variable childCalledLast$_T$ to store which child table has most recently produced a row (initialized in Line 13). The reason is the following: Once a row r has been added to a child table T_i, we need to check whether r allows us to produce a new row at T by trying out all combinations of r with existing rows from the other child tables. In the course of this, we will keep active$_{T_i}$ fixed to r and vary all active$_{T_j}$ with $j \neq i$ until we have handled all combinations of r with existing rows. Only then will we proceed to compute a new row at a child table in Line 18.

Algorithm 3: `activateNext`

Input: table T whose children we call $T_1, \ldots, T_n$; integer i (1 if not specified)
Result: sets $(\text{active}_{T_i}, \ldots, \text{active}_{T_n})$ to next combination of existing rows; returns true iff successful

24 **if** $i > n$ **then return** false
25 **if** $i = \text{childCalledLast}_T$ **then**
26 **return** `activateNext(`$T, i+1$`)`
27 **if** active$_{T_i}$ *is the last row of* T_i **then**
28 active$_{T_i} \leftarrow$ first row of T_i
29 **return** `activateNext(`$T, i+1$`)`
30 active$_{T_i} \leftarrow$ row in T_i that follows active$_{T_i}$
31 **return** true

Repeated calls of `activateNext(`T`)` allow us to iterate over all existing child rows similar to the way we can produce all n-digit numbers by repeatedly incrementing a variable that is initially 0. We can adapt this idea for our purpose by imagining that each of these n digits can have a different base: For $i \neq \text{childCalledLast}_T$, the i-th digit identifies a row in T_i, while the digit at position childCalledLast$_T$ can only take one symbol (representing r) as if its base was 1.

Algorithm 4: `callChild`

Input: table T whose children we call $T_1, \ldots, T_n$
Result: produces new row at a child of T; resets active rows of other children; returns true iff successful

32 **if** T *has no children* **then return** false
33 $i \leftarrow$ `nextChildToCall(`T`)`
34 $r \leftarrow$ `nextRow(`T_i`)`
35 **while** $r =$ fail **do**
36 mark the table T_i as exhausted
37 **if** *all of* $T_1, \ldots, T_n$ *are exhausted* **then return** false
38 $i \leftarrow$ `nextChildToCall(`T`)`
39 $r \leftarrow$ `nextRow(`T_i`)`
40 childCalledLast$_T \leftarrow i$
41 **foreach** T' **in** $\{T_1, \ldots, T_n\}$ **do**
42 **if** $T' = T_i$ **then** active$_{T'} \leftarrow r$
43 **else** active$_{T'} \leftarrow$ first row of T'
44 **return** true

The subroutine `callChild` (Algorithm 4) is invoked with T as input when all combinations of existing child rows have been active. Its purpose is to compute a new row at a child of T or return false if this is not possible. Let the children of T be called $T_1, \ldots, T_n$. For each T_j we maintain a flag that indicates whether T_j is exhausted, i.e., whether no new rows can be added to T_j. Algorithm 4 uses a subroutine `nextChildToCall` (whose code we omit) to decide which child table should produce the next row. The idea is that `nextChildToCall(`T`)` takes the tuple $(1, \ldots, n)$, rotates it such that the element equal to childCalledLast is at the back, and then removes any j such that T_j is exhausted. Then `nextChildToCall(`T`)` returns the first element of the resulting tuple and thus identifies the next non-exhausted child table following the child table called most recently (possibly with "wrapping around"). In this way, we call child tables in a round-robin fashion to ensure a certain fairness in the hope of obtaining solutions more quickly than by filling one table completely before proceeding to the next one. This hope is based on the assumption that the likelihood of one row from, say, T_2 being "bad" (i.e., not leading to a solution) is higher than the likelihood of a lot of rows from T_1 being "bad".

3.2 Further Optimizations

We implemented two further optimizations that we omitted in the pseudocode for the sake of a clearer presentation. First of all, especially TD nodes with more than one child incur a big computational effort. For this reason, in addition to our general algorithm that works on every TD, we implemented a special treatment for semi-normalized TDs that does not call an external solver at join nodes. Rather, it uses the fact that many DP algorithms (e.g., the one in Example 1) produce a row at a join node if and only if there is a combination of child rows that "fits together", which means that they all contain the same "join-relevant" items. Which items are "join-relevant" depends on the problem. In our implementation, the user can mark the desired items with a special flag.

This join behavior allows us to greatly restrict the active row combinations that must be considered. In fact, we can ignore all combinations where some active$_i$ and active$_j$ differ on their "join-relevant" items. Whenever a new row r is added to a child table, we can quickly identify which rows from the other child tables can be combined with r. To find such matches, we keep tables sorted and use binary search.

The second optimization regards the cost bound that is used to discard rows that are too expensive. In our pseudocode, we follow the naive approach of eliminating rows whose cost exceeds the cost of the best solution found so far (Line 20). However, we can do better if the DP algorithm exhibits "monotone costs" (i.e., the cost of a row is at least the sum of the costs of its origins): Suppose, for example, that T is a table with children T_1, T_2, and that T_1 is exhausted, which means that all tables below it are exhausted, too. Then, in a way, we "know everything" about the subgraph G' induced by the vertices that occur somewhere below T_1 but have been forgotten. In particular, we know that G' contributes *at least* a certain amount c to the cost of every solution, since each solution in part originates from some row in T_1. If the currently best solution has cost k and we now try to compute a new row at T_2, we can restrict ourselves to rows whose cost is below $k - c$: If a row r at T_2 had `cost(r)` $\geq k - c$, any row in T that originates from r would have a cost of at least k (due to monotonicity) and would thus not lead to a better solution. This argument can easily be extended to the case where not T_1 but some table below T_1 is exhausted, and to the case of more than two child tables. In this way, our implementation can tighten the cost bound for new rows in many cases, which often results in significantly fewer rows being computed.

3.3 Practical Realization

We implemented our algorithm by modifying the publicly available *D-FLAT* system [Abseher *et al.*, 2014]. This system so far only allowed for an eager evaluation technique that works as follows. The user provides an ASP specification Π of a DP algorithm for her problem. At each TD node, D-FLAT runs the ASP system *clingo* on Π, with the input facts consisting of (a) the bags of the current node and its children, (b) the subgraph of the problem instance induced by the vertices in these bags, and (c) the (complete) child tables. From each answer set D-FLAT then extracts a table row.

Our lazy evaluation algorithm on the other hand also calls clingo on Π with the relevant bags and part of the instance as input, but this time we do not provide the entire child tables. Instead, we only supply the currently *active* child rows. In Line 8 of Algorithm 2, we set up clingo by grounding Π together with its input. Note that we perform grounding only in this initialization step and use the resulting propositional program for all of the remaining execution.

For example, the following encoding allows us to compute vertex covers of a graph with our lazy evaluation algorithm:

```
item(X) ← childItem(X), current(X).
{ item(X) : introduced(X) }.
← edge(X,Y), current(X), current(Y)
   not item(X), not item(Y).
#external childItem(X) :
          childNode(N), bag(N,X).
```

At each TD node, the system grounds this program together with the bag information (provided as facts over predicates `childNode/1`, `bag/2`, `current/1` and `introduced/1`) and the induced subinstance (facts over `edge/2`). The last line declares all atoms of the form `childItem(X)` (where `X` is a vertex in a child bag) as external, so the grounder does not assume these atoms to be false even though no rule can derive them. Assumption-based solving allows us to temporarily freeze the truth value of these atoms according to the currently *active rows*, compute answer sets, then possibly change the active rows and repeat the process.

Whenever we call the ASP solver (Line 16 of Algorithm 2), one row in each child table is active and we set those atoms `childItem(X)` to true where `X` is in an active child row (and the others to false). Each resulting answer set specifies a table row whose content is given by the terms within true atoms over the `item/1` predicate.

For optimization problems, a cost is associated with each row. Our prototype implementation just uses the size of the set of (join-relevant) items for this. This suffices for many problems (in particular the ones we investigate in Section 4). In the future, we will allow costs to be specified in ASP.

3.4 Applicability

It is a natural question for which problems our approach is suitable. If a problem is expressible in monadic second-order (MSO) logic, bounded treewidth leads to tractability by [Courcelle, 1990]. The eager approach followed by the D-FLAT systems works for all such problems [Bliem *et al.*, 2016]. In principle this also holds for our lazy algorithm, but our implementation is currently restricted to problems in NP (which simplifies algorithm descriptions and implementation). For MSO-definable problems on sufficiently large instances of small treewidth, our algorithm is expected to outperform systems that do not exploit bounded treewidth.

4 Experiments

We pursued three research questions in our experiments: 1. Does our lazy evaluation approach pay off compared to the traditional "eager" one, which is conceptually simpler but does not allow for anytime solving? 2. How does our system compare to the ASP system clingo with respect to running time, or the solution quality in case of timeouts? 3. Does assumption-based solving perform better than the straightforward way of re-grounding for each active row combination?

We encoded two search problems (3-COL, INDEPENDENT DOMINATING SET) and three optimization problems (VERTEX COVER, DOMINATING SET, STEINER TREE with unit costs) using 11 real-world graphs (public transport networks of different cities) as input. All instances and problem encodings are available at http://dbai.tuwien.ac.at/proj/dflat/lazy-experiments.tar.gz. We chose real-world instances because DP on TDs is reasonable only when the width is small, and the observation that real-world graphs often admit TDs of small width is the main motivation for this approach. The median width of our considered TDs is 5 (minimum: 3; maximum: 8).

We used the D-FLAT system with eager evaluation as well as our lazy evaluation algorithm (denoted by *eager* and *lazy*,

	Beijing	Berlin	London	Munich	Sant.	Shangh.	Sing.	Timis.	Tokyo	V. (M)	V. (MT)
	STEINER TREE (times in seconds)										
clingo B	∞	154	∞	(569) **20**	103	∞	∞	∞	∞	**22**	∞
clingo U	253	176	341	(206) **20**	134	313	136	141	136	(205) **22**	150
lazy	(476) **65**	(4) **65**	136	(1) **20**	(4) **44**	(172) **73**	(18) **44**	74	(5) **35**	(1) **22**	(42) **33**
eager	∞	(4) **65**	∞	(1) **20**	(3) **44**	∞	(59) **44**	∞	(38) **35**	(1) **22**	(439) **33**
	DOMINATING SET (times in seconds)										
clingo B	113	**85**	**132**	(589) **48**	**61**	142	**63**	86	**69**	**45**	44
clingo U	(0) **112**	(0) **85**	(0) **132**	(0) **48**	(0) **61**	(0) **139**	(0) **63**	(0) **82**	(0) **69**	(0) **45**	(0) **43**
lazy	(16) **112**	(3) **85**	(1) **132**	(0) **48**	(0) **61**	(8) **139**	(1) **63**	(132) **82**	(1) **69**	(0) **45**	(4) **43**
eager	(23) **112**	(3) **85**	(1) **132**	(0) **48**	(0) **61**	(4) **139**	(1) **63**	(229) **82**	(1) **69**	(0) **45**	(5) **43**
	VERTEX COVER (times in centiseconds)										
clingo B	115	85	153	**48**	**63**	146	**62**	98	70	**45**	66
clingo U	(1) **111**	(1) **83**	(1) **150**	(0) **48**	(0) **63**	(1) **141**	(0) **62**	(1) **96**	(0) **69**	(0) **45**	(1) **64**
lazy	(55) **111**	(24) **83**	(55) **150**	(13) **48**	(19) **63**	(52) **141**	(20) **62**	(46) **96**	(22) **69**	(12) **45**	(26) **64**
eager	(65) **111**	(32) **82**	(68) **150**	(17) **48**	(24) **63**	(63) **141**	(26) **62**	(68) **96**	(28) **69**	(15) **45**	(36) **64**

Table 1: Cost of best found solution for optimization problems after ten minutes (median over all runs; bold if optimal). Cells without parentheses: timeout. In parentheses: running time. Some *eager* runs on STEINER TREE ran out of memory; we treated this as timeout. For clingo, "B" and "U" denote branch-and-bound-based and USC-based optimization, respectively.

	Beijing	Berlin	London	Munich	Santiago	Shanghai	Singap.	Timisoara	Tokyo	Vienna (M)	Vienna (MT)
	3-COL										
clingo	0.02	0.01	0.02	0.01	0.01	0.02	0.01	0.02	0.01	0.01	0.01
lazy	2.14	0.36	0.90	0.18	0.27	0.95	0.30	1.34	0.32	0.16	0.49
eager	16.28	1.28	3.96	0.36	0.75	6.22	0.97	30.93	1.07	0.30	2.85
	INDEPENDENT DOMINATING SET										
clingo	0.01	0.01	0.01	0.00	0.00	0.01	0.00	0.01	0.00	0.00	0.01
lazy	0.47	0.31	0.61	0.17	0.24	0.55	0.24	0.43	0.26	0.16	0.30
eager	0.62	0.36	0.73	0.20	0.29	0.68	0.24	0.64	0.32	0.18	0.65

Table 2: Time (in seconds; median over all runs) for finding a solution or determining that none exists.

respectively), and for *lazy* we tested both the re-grounding and the assumption-based approach. Moreover, we compared *lazy* against the ASP system clingo 4.5.4 with the default branch-and-bound optimization strategy as well as one based on unsatisfiable cores (with disjoint-core preprocessing). Our encodings for clingo do not make use of decomposition.

The benchmarks ran on a Debian GNU/Linux system (kernel 3.16.0.4) on an Intel Xeon E5345 CPU at 2.33 GHz, using only one core without hyperthreading. We limited each run to 1 GB of main memory and 10 minutes of CPU time. D-FLAT running times include heuristic generation of a TD. As performance varies greatly depending on the TD, we used 30 different random seeds (causing different TDs) per instance.

4.1 Impact of Lazy Evaluation

Here we mainly inspect the ratio of the running time of *lazy* to *eager* to find out if *lazy* brings improvements. The results are summarized in Table 1 and Table 2.

Eager terminated within the resource limits on all runs except for some on STEINER TREE: On Beijing, Shanghai and Timisoara all of the respective 30 runs were aborted, and 25 on London. For *lazy*, the situation is better: Only STEINER TREE on Beijing, Timisoara and London had aborted runs (6, 25 and 2, respectively). Thanks to lazy evaluation all of the aborted runs were able to report a solution. The median cost was even optimal (65) on Beijing, whereas on London it was 136 (optimum: 75) and on Timisoara 74 (optimum: 29).

In the following we compare the times for each pair of completed runs. On all 3-COL instances, *lazy* is significantly faster (based on a Wilcoxon signed-rank test) with an overall improvement of 72 % (1 minus median of the ratio of *lazy*'s time to *eager*'s time; lower quartile $Q_1 = 64$ %, upper quartile $Q_3 = 84$ %). The improvement was particularly high on Timisoara with 95 % ($Q_1 = 94$ %, $Q_3 = 96$ %).

Similarly, for INDEPENDENT DOMINATING SET, *lazy* is significantly faster on all instances except Singapore, where there is no significant difference. In fact, this is the only case where there is no solution, so *lazy* is forced to compute as many rows as *eager*. On the other instances, *lazy* overall gives an 18 % improvement ($Q_1 = 15$ %, $Q_3 = 22$ %).

For DOMINATING SET, *lazy* is faster on most instances, but significantly slower on Shanghai and Tokyo (median "im-

provement" -82 % and -11 %, respectively). On the other hand, *lazy* is particularly fast on Singapore (58 % improvement, $Q_1 = 56$ %, $Q_3 = 61$ %). Overall, *lazy* gives a 22 % improvement ($Q_1 = 2$ %, $Q_3 = 41$ %) for this problem.

STEINER TREE displays interesting behavior since the improvement ratios vary a lot between instances. On three instances *lazy* is significantly faster and on four it is significantly slower (which is not visible in Table 1 due to rounding). When looking at all instances together, its median "improvement" is -4 % ($Q_1 = -10$%, $Q_3 = 80$ %), but the mean is 20 %. The reason is that on those instances where *lazy* is better it offers a huge improvement (up to 90 %).

On all VERTEX COVER instances, *lazy* is significantly faster (overall improvement 22 %, $Q_1 = 20$ %, $Q_3 = 25$ %).

4.2 Comparison to Clingo

As expected, clingo is always faster than *lazy* on the search problems (cf. Table 2). For the optimization problems, we compare the quality of the best solution found before timeout in Table 1. It makes a huge difference which of clingo's optimization strategies is used: By default, clingo employs branch-and-bound search (BB), which aims at successively producing models of decreasing costs until an optimal model is found. Alternatively, clingo supports unsatisfiable-core-based (USC) optimization, which relies on successively identifying and relaxing unsatisfiable cores until eventually an optimal model is obtained. USC often has an edge over BB when the optimum can be established. However, while it can be combined with disjoint core preprocessing, aiming at an approximation of the optimal solution, it lacks the anytime behavior of BB. Hence, when the optimum is out of reach, BB may dominate USC in terms of solution quality. More detailed explanations are given in [Gebser *et al.*, 2015].

On DOMINATING SET and VERTEX COVER, USC solves each instance optimally in at most 0.012 seconds, whereas BB mostly times out. The corresponding median of the running times of *lazy* is 0.49 seconds ($Q_1 = 0.2$, $Q_3 = 1.33$), with many outliers (maximum: 255.9 seconds).

The situation is different on STEINER TREE, however. Both USC and BB time out on most instances. On a few instances BB finds better solutions than USC before timing out, but generally USC is better. We can observe that *lazy* clearly outperforms both clingo configurations in this benchmark.

4.3 Influence of Assumption-based Solving

We compared our assumption-based algorithm to a naive approach, which, instead of using assumptions, simply grounds the program anew whenever the active rows change. For this comparison, we used all problems except STEINER TREE due to technical limitations that we will eliminate in the future. As all of the 3960 runs completed except for 17 timeouts, we only considered completed runs. Overall the re-grounding approach is significantly slower on each instance of every problem. It is 7.93 times slower overall ($Q_1 = 4.34$, $Q_3 = 14.92$), and 1.79 times slower in its best run.

4.4 Discussion

Usually *lazy* is more efficient than *eager*. Especially on search problems it often manages to avoid computation of many rows and thus yields a solution much quicker. On optimization problems, *lazy* is forced to inspect a larger search space, which leads to less predictable behavior in terms of running time compared to *eager*. *Lazy* mostly still offers better performance, but it depends heavily on the problem and instance. For example, on one DOMINATING SET instance *lazy* required 82 % more time for finding the optimum, but on a STEINER TREE instance *eager* was more than ten times slower. Even though *lazy* may sometimes find the optimum slower, this might be a price worth paying due to the possibility of giving suboptimal solutions along the way.

Clingo is generally very quick in finding some solution, so on the search problems *lazy* could not compete with it. For some optimization problems, clingo's USC-based optimization strategy works remarkably well and outperforms *lazy*. On other problems, it works less well: On STEINER TREE *lazy* often terminates in a short amount of time where clingo times out, and even in cases where *lazy* times out it produces much better solutions. We conclude that *lazy* offers a clear advantage compared to the traditional eager approach and can in some cases outperform state-of-the-art ASP systems.

As a by-product, initial experiments yielded the insight that for some search problems *lazy* is much faster when given a *path decomposition*, i.e., a TD where every node has at most one child. However, when using path decompositions for optimization problems, the memory requirements are usually prohibitive. Furthermore, when comparing *lazy* and *eager* on semi-normalized TDs, our experiments did not indicate a clear winner in terms of memory. Which variant requires less space seems to depend on the problem, instance and TD. We plan to investigate this in future work.

5 Conclusion

We presented a generic algorithm for performing DP on TDs by means of lazy evaluation, and we implemented a system that allows the user to specify the DP steps for a particular problem in a declarative language. In contrast to existing solutions like [Abseher *et al.*, 2014], this allows us to print solutions before the optimum is found. Our experiments demonstrated that this is typically more efficient, also for search problems without optimization, and on some problems it outperforms state-of-the-art ASP systems. We verified that assumption-based solving, a recent advance in ASP solving technology, is indispensable for good performance.

In the future, we intend to improve efficiency by integrating the ASP solver tighter and tuning its parameters. Alternatively, it might be interesting to incorporate different formalisms instead of ASP. Moreover, for deciding at which table we should compute a new row, we proposed a round-robin strategy, and we plan to investigate different strategies. As described in Section 3.2, we implemented a branch-and-bound technique by discarding table rows that are more expensive than the best known (global) solution so far. Currently, we just ignore models that would induce such rows. We plan to compare this to a version that adds constraints in ASP instead. Finally, our experiments indicate that the actual shape of the TD has a high impact on running times, so techniques from [Abseher *et al.*, 2015] could also be beneficial.

Acknowledgments This work was funded by DFG grant SCHA 550/9 and by the Austrian Science Fund (FWF): Y698, P25607.

References

[Abseher *et al.*, 2014] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proc. JELIA*, volume 8761 of *LNCS*, pages 558–572, 2014.

[Abseher *et al.*, 2015] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. In *Proc. IJCAI*, pages 275–282, 2015.

[Alviano *et al.*, 2014] Mario Alviano, Carmine Dodaro, and Francesco Ricca. Anytime computation of cautious consequences in answer set programming. *TPLP*, 14(4-5):755–770, 2014.

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[Bliem *et al.*, 2016] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Implementing Courcelle's Theorem in a declarative framework for dynamic programming. *Journal of Logic and Computation*, 2016. DOI: 10.1093/logcom/exv089.

[Bodlaender and Koster, 2010] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[Bodlaender, 1993] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[Chimani *et al.*, 2012] Markus Chimani, Petra Mutzel, and Bernd Zey. Improved Steiner tree algorithms for bounded treewidth. *J. Discrete Algorithms*, 16:67–78, 2012.

[Courcelle, 1990] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Dermaku *et al.*, 2008] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[Downey and Fellows, 1999] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[Gebser *et al.*, 2014] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[Gebser *et al.*, 2015] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In *Proc. LPNMR*, pages 368–383, 2015.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[Gogate and Dechter, 2004] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In David Maxwell Chickering and Joseph Y. Halpern, editors, *Proc. UAI '04*, pages 201–208. AUAI Press, 2004.

[Guziolowski *et al.*, 2013] Carito Guziolowski, Santiago Videla, Federica Eduati, Sven Thiele, Thomas Cokelaer, Anne Siegel, and Julio Saez-Rodriguez. Exhaustively characterizing feasible logic models of a signaling network using answer set programming. *Bioinformatics*, 29(18):2320–2326, 2013. Erratum see Bioinformatics 30, 13, 1942.

[Kloks, 1994] Ton Kloks. *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.

[Nieuwenborgh *et al.*, 2006] Davy Van Nieuwenborgh, Stijn Heymans, and Dirk Vermeir. Approximating extended answer sets. In *Proc. ECAI*, pages 462–466, 2006.

[Robertson and Seymour, 1984] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[Soininen and Niemelä, 1998] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proc. PADL*, volume 1551 of *LNCS*, pages 305–319. Springer, 1998.