Towards Aggregated Answers for Semistructured Data

Holger Meuss[1] Klaus U. Schulz[1] François Bry[2]
[1] Center for Information and Language Processing (CIS) [2] Institute for Computer Science
University of Munich, Oettingenstr. 67, 80538 Munich, Germany
meuss@cis.uni-muenchen.de

1 Introduction

Semistructured data [3, 12, 45, 22, 33, 42, 6] are used to model data transfered on the Web for applications such as e-commerce [1], astronomy [8], biomolecular biology [15], document management [9, 2], linguistics [43], thesauri and ontologies [26]. They are formalized as trees or more generally as graphs [33, 28, 6]. Query languages for semistructured data have been proposed [17, 30, 47] that, like SQL, can be seen as involving a number of variables [46, 4], but, in contrast to SQL, give rise to arrange the variables in trees or graphs reflecting the structure of the semistructured data to be retrieved. Leaving aside the "construct" parts of queries, answers can be formalized as mappings represented as tuples, hence called *answer tuples,* that assign database nodes to query variables. These answer tuples underly the semistructured data delivered as answers.

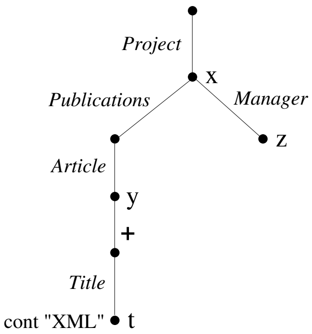
A simple enumeration of answer tuples following the old relational approach is problematic for several reasons. First, the number of answer tuples for a query may grow exponentially in the size of both, the query and the database. Second, even if the number of answer tuples is manageable, the frequent sharing of common data between distinct answer tuples is no more apparent in their enumeration.

In this article, it is first argued that, in the context of semistructured data, enumerating answer tuples is often not appropriate and that *aggregated answers* are preferable. Then, a notion of aggregated answers called <u>*C*</u>*omplete* <u>*A*</u>*nswer* <u>*A*</u>*ggregate (CAA)* generalizing [36, 35] is introduced and algorithms for computing CAAs are given. Finally, it is shown that CAAs enjoy nice complexity properties: (1) While the number of answer tuples may be exponential in the size of the query, the size of the CAA is at most linear in the size of the query and quadratic in the size of the database; (2) the complexity of computing the CAA of a query depends on the query's structural complexity (i.e. whether it is a sequence, tree, graph, etc.) but is independent of the structural complexity of the database. For tree queries, efficient polynomial algorithms are given. Besides, CAAs seem to be particularly appropriate for *answer searching* and *answer browsing.*

This article is organized as follows. The need for aggregated answers and the basics of CAAs are illustrated with a motivating example in Section 2. Section 3 introduces a few preliminary notions. Section 4 gives the formal definition of CAAs. In Section 5, a hierarchy of queries of increasing complexity is defined. In Section 6 algorithms for computing the CAAs of queries of various structural kinds are described and their complexity is analyzed using the query hierarchy. Query answering using CAAs is discussed in Section 8. Section 9 discusses related work. Section 10 is a conclusion. The proofs are given in Appendix.

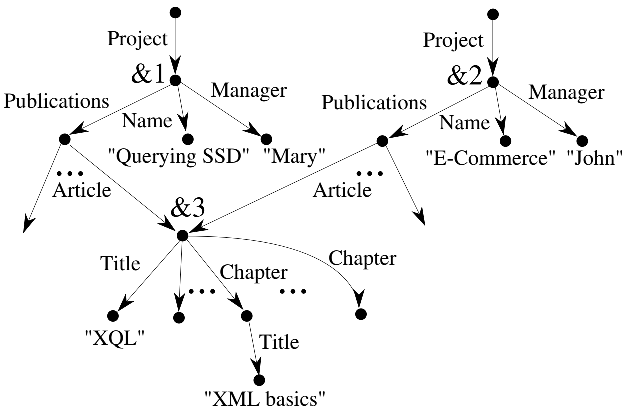
2 Motivating Example

Consider a database on research projects offering information on project managers, members and publications. Assume that the database is organized as a graph with labeled edges and nodes according to a model for semistructured data [7, 16, 27, 6]. Projects x and their managers z such that the string "XML" occurs in a title element t (at any depth) of some articles y of projects x are retrieved by the following query Q:



The restructuring facilities of full-fledge query languages for semistructured data [17, 30, 47] are not considered in this paper. Thus, only the "select" parts of queries are mentioned, possible "construct" parts are left implicit.

Assume that some of the articles retrieved by the query Q are common to several projects, as e.g. with the following database $\mathcal{D}$:



Evaluated against the database $\mathcal{D}$, the query Q returns the following answer tuples:

x	y	z	t
&1	&3	"Mary"	"XML basics"
&2	&3	"John"	"XML basics"

More generally, if the string "XML" occurs in k titles of article &3, then Q admits $2k$ answer tuples all referring to &3. Furthermore, if an article is shared by n projects, then Q has $n \cdot k$ answer tuples. In case of more complex queries and/or of database items with more complex interconnections, as often arise in Web and e-commerce servers, an enumeration of answer tuples à la Prolog or à la SQL results in a combinatorial explosion. If e.g. the functional dependency *Project* $\rightarrow$ *Manager* does not hold and if each of the two projects has m managers, then Q admits $2 \cdot k \cdot m$ answer tuples. In general, such a product giving the number of answers of a query like Q can have any number of factors, i.e. the number of answers is exponential.

Arguably, for many applications such an enumeration of answer tuples is not appropriate. Instead, a data structure stressing the common subelements shared between (parts of) answer tuples as well as their graph relationships would often be more convenient. Let us call *aggregated answer* such a hypothetical data structure. Aggregated answers would give rise to recognize "bottlenecks" in the "answer space". Furthermore, aggregated answers make advanced query answering forms possible, cf. Section 7.

In this paper, *Complete Answer Aggregates (CAAs)* are proposed as a formalization of such a notion of aggregated answer and CAAs are shown to be efficiently computable. A CAA reflects the graph structure of the query it is computed from. The CAA computed for Q over an example database (larger than $\mathcal{D}$) has a "slot", represented in Fig. 1 by a rectangle, for each variable x, y, z, and t. A slot for variable v contains possible binding elements for v: E.g. the slot for x contains project identifiers and the slot for z contains manager identifiers. The edges in Fig. 1 are *CAA links.* They represent (sequences of) database edges. Note that a presentation of a CAA such as Fig. 1 is not intended for end users.

Admittedly, it is possible to generate in some cases simple kinds of aggregated answers with usual query languages like [17, 30, 47], but this requires nested queries that might be complex [6] "4.1 Path Expressions", p. 55. In contrast, with CAAs, no complex queries are not needed and aggregated answers are obtained in all cases.

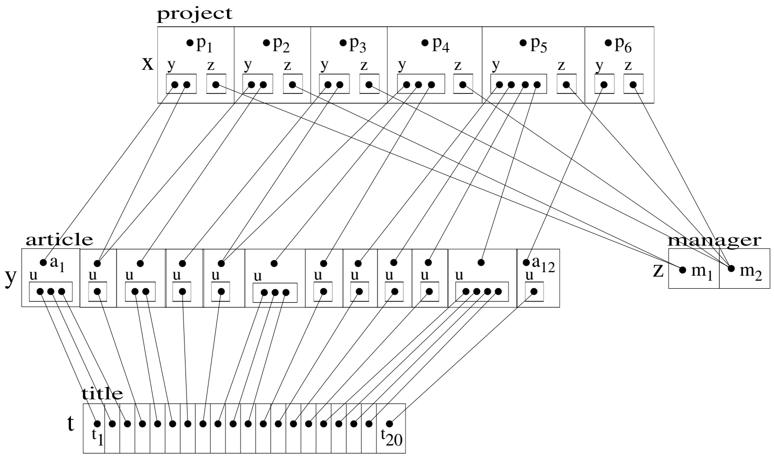


Figure 1: A complete answer aggregate (internal form)

CAAs are nothing else than semistructured data items of a certain kind. Thus queries can be posed to CAAs. Provided that the implementation of the query language is "CAA aware", such a querying can be performed without requiring from the user or application issuing the query to be aware of the internal structure of the CAAs constructed during query evaluation. Thus, CAAs are a convenient basis for an iterative, or cascade style query-answering: The evaluation of a query yields a CAA which can be stored and in turn queried using the same query language. Such a cascade style query-answering is often sought for in e-commerce applications [1]. Cf. Section 7 for more details.

3 Preliminary Notions

Definition 3.1 A *database* is a tuple $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ where
- N is a (finite) set of *nodes*,
- $E \subseteq N \times N$ is a set of directed *edges*,
- L_N is a (finite) set of *node labels*,
- L_E is a (finite) set of *edge labels* or *features*,
- $\Lambda_N : L_N \to 2^N$ is a (total) function that assigns to each node label a set of nodes,
- $\Lambda_E : E \to L_E$ is a (total) function assigning labels to edges.

$\mathcal{D}$ is a *sequence* (resp. *tree*, *DAG*, *graph*) *database* if the structure imposed by E upon N defines a (finite) set of sequences (resp. trees, DAGs, graphs).

Note that sequence, tree, and DAG databases are acyclic and that nodes in sequence (resp. tree) databases have at most one parent and one child (resp. one parent). Node labels are aimed at modeling attributes and attribute values. Multiple node labeling is allowed, so as to model textual content: A label w of node n might express that a string w occurs in the textual content of n. In the sequel, regular expressions α over the alphabet L_E of edge labels will be considered.

Definition 3.2 Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and α a regular expression over L_E. A node $d \in N$ is an *α-ancestor* of $e \in N$, if there exists a path from d to e such that the sequence of labels along the path belongs to the regular language $\mathcal{L}(\alpha)$ induced by α.

Definition 3.3 Let X be an enumerable set of variables and $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ a database. An *atomic path constraint* is an expression of the form
- $A(x)$ called a labeling constraint,
- $x \to_1 y$ called a child constraint,
- $x \to_f y$ called a f-child constraint,
- $x \to_+ y$ called a descendant constraint,
- $x \to_\alpha y$ called an α-descendant constraint,

where $x, y \in X$, $A \in L_N$, $f \in L_E$, and α is a regular expression over L_E. Atomic path constraints of the latter four types are called *edge constraints.*

Edge constraints of the form $x \rightarrow_1 y$, $x \rightarrow_f y$, and $x \rightarrow_+ y$ can be seen as special cases of α-descendant constraints. Without loss of generality, the (non-atomic) path constraints considered in this paper are conjunctions of atomic path constraints containing at most one atomic edge constraint $x \rightarrow_? y$ for each (ordered) pair (x, y) of query variables. Depending on their nature, edge constraints might impose sequence, tree, DAG, or graph structures on the variables.

Definition 3.4 A *sequence* (resp. *tree*, *DAG*, *graph*) *query* is a conjunction of atomic path constraints the atomic edge constraints of which impose a sequence (resp. tree, DAG, graph) structure on its variables. If D is a database and Q a query with set of variables X_Q, then $(Q, X_Q, \mathcal{D})$ is an *evaluation problem.*

Note that queries of either type can be represented as graphs, like e.g. Q in the previous section.

Definition 3.5 Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and Q a query. An *answer* to Q in $\mathcal{D}$ is a mapping μ that assigns a node in D to each variable in Q in such a way that
• $\mu(x) \in \Lambda_N(A)$ whenever Q contains the labeling constraint $A(x)$,
• $(\mu(x), \mu(y)) \in E$ whenever Q contains the child constraint $x \rightarrow_1 y$,
• $(\mu(x), \mu(y)) \in E$ and $\Lambda_E(\mu(x), \mu(y)) = f$ whenever Q contains the f-child constraint $x \rightarrow_f y$,
• $\mu(x)$ is an α-ancestor of $\mu(y)$ whenever Q contains the α-descendant constraint $x \rightarrow_\alpha y$.
If $(Q, X_Q, \mathcal{D})$ is an evaluation problem and μ an answer to Q in $\mathcal{D}$, then μ is called a *solution* of $(Q, X_Q, \mathcal{D})$.

Note that according to Definition 3.5, cyclic, i.e. proper graph queries have no answers in acyclic, i.e. sequence, tree, or DAG databases.

In the following queries are considered some variable of which are existentially quantified. These variables denote as usual query variables the values of which do not have to be returned with the answers. Extending the previous definition to such queries is straightforward. For space reasons, this extension is omitted here.

4 Complete Answer Aggregates

Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_N, \Lambda_E)$ be a database and Q a query with set of variables X_Q.

Definition 4.1 An *answer aggregate* for the evaluation problem $(Q, X_Q, \mathcal{D})$ is a pair (Dom, Π) such that
• $Dom : X_Q \rightarrow 2^N$ assigns to each variable of Q a set of nodes of $\mathcal{D}$ such that each $d \in Dom(x)$ has label A if the labeling constraint $A(x)$ of Q,
• Π maps each edge constraint $x \rightarrow_1 y$ (resp. $x \rightarrow_f y$, $x \rightarrow_+ y$, $x \rightarrow_\alpha y$) of Q to a set $\Pi(x, y) \subseteq Dom(x) \times Dom(y)$ such that for all $(d, e) \in \Pi(x, y)$ e is a child (resp. f-child, descendant, α-descendant) of d. A node $d \in Dom(x)$ is called a *target candidate* for x. A pair $(d, e) \in \Pi(x, y)$ is called a *link* between d and e.

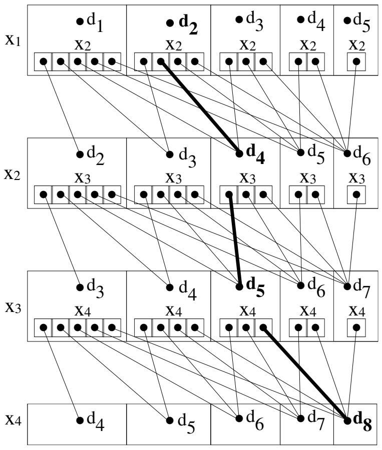
Definition 4.2 An *instantiation* of an answer aggregate (Dom, Π) is a mapping μ that maps each $x \in X_Q$ to a node $\mu(x) \in Dom(x)$ such that $(\mu(x), \mu(y)) \in \Pi(x, y)$ whenever Q contains an edge constraint $x \rightarrow_1 y$, $x \rightarrow_f y$, $x \rightarrow_+ y$, or $x \rightarrow_\alpha y$. Each node of the form $\mu(x)$, as well as every link of the form $(\mu(x), \mu(y))$ (for $x, y \in X_Q$) is said to *contribute* to instantiation μ.

Note that each instantiation of an answer aggregate for $(Q, X_Q, \mathcal{D})$ defines an answer to Q in $\mathcal{D}$.

Definition 4.3 (Dom, Π) is a *complete answer aggregate (CAA)* for $(Q, X_Q, \mathcal{D})$ if every answer to Q in $\mathcal{D}$ is an instantiation of (Dom, Π) and if every target candidate and every link of (Dom, Π) contributes to at least one instantiation.

Example 4.4 Assume that $Q = x_1 \rightarrow_+ x_2 \wedge x_2 \rightarrow_+ x_3 \wedge \cdots \wedge x_{q-1} \rightarrow_+ x_q$ and assume that D consists of $n > q$ nodes $d_1, \ldots, d_n$ sequentially ordered (i.e. $(d_i, d_{i+1}) \in E$ for all $1 \leq i \leq n-1$). Q has $\binom{n}{q}$ answers in D. For $q = 4$ and $n = 8$

the CAA representing the 70 answers (each mapping the four query variables to database nodes) has the form depicted below For one possible instantiation, contributing nodes and links are highlighted.



Lemma 4.5 *There exists a unique CAA for each evaluation problem.*

Size of CAAs

Let $(Q, X_Q, \mathcal{D})$ be an evaluation problem. In the following, q will denote the number of variables plus labeling constraints of Q, n the number of nodes of $\mathcal{D}$, and a the maximal number of ancestors of a database node plus the number of links between these nodes. Note that, if $\mathcal{D}$ is a sequence or a tree database, then a is bounded by the so-called height of $\mathcal{D}$.

As a measure for the size of a CAA (Dom, Π) for $(Q, X_Q, \mathcal{D})$ it makes sense to retain the total number of target candidates and links in (Dom, Π), i.e. $\Sigma_{x \in X_Q}|Dom(x)| + \Sigma_{x,y \in X_Q}|\Pi(x, y)|$ (where $\Pi(x, y) := \emptyset$ if x and y are not related by an edge constraint in Q). Although the number of answers to a query Q may be exponential, the size of a CAA is at most linear in the size of the query and quadratic in the size of the database:

Theorem 4.6 *The size of the CAA for an evaluation problem $(Q, X_Q, \mathcal{D})$ is $O(q \cdot n \cdot a)$.*

In some situations a better bound can be obtained.

Definition 4.7 A query Q is *rigid* if Q does not have any edge constraint of the form $x \to_+ y$ or $x \to_\alpha y$. A database $\mathcal{D}$ is *non-recursive* if in no paths in $\mathcal{D}$ a given label occurs at two distinct nodes. A query Q is *completely labeled* if for every variable x in Q, Q has a labeling constraint $A(x)$.

Let i be the maximal incoming degree of nodes in the database $\mathcal{D}$ (i.e. $i = 1$ for sequence and tree databases).

Lemma 4.8
- *If Q is a rigid query, then the CAA for $(Q, X_Q, \mathcal{D})$ has size $O(q \cdot n \cdot i)$.*
- *If Q is a completely labeled tree query and if $\mathcal{D}$ is a non-recursive database, then the CAA for $(Q, X_Q, \mathcal{D})$ has size $O(q \cdot n)$.*

CAAs as Semistructured data: According to Definition 4.1, the links of the CAA for an evaluation problem (Q, X_Q, D) are not labeled. If $x \to_f y$ is an f-child constraint in Q, then the label f can obviously be attached to each link in $\Pi(x, y)$. If $x \to_1 y$ is a child constraint in Q, then each link of $\Pi(x, y)$ clearly represents a labeled edge of $\mathcal{D}$, i.e. can be labeled like this database edge. Thus, if all edge constraints of the query are child- or f-child constraints, the CAA trivially extends to a semistructured data item. If the query Q contains descendant constraints of the form $x \to_+ y$ or $x \to_\alpha y$ and if $\mathcal{D}$ is a sequence or tree database, then for each $(d, e) \in \Pi(x, y)$ there exists in D a unique path π from d to e. The corresponding sequence of labels can be attached to the link (d, e) of the CAA yielding a semistructured data item in this case, too. If $\mathcal{D}$ is a proper graph database, a link $(d, e) \in \Pi(x, y)$ in general stands for a regular expression. Although this is less immediate than in the previous cases of sequence and tree databases, this expression can serve as a label of a CAA link. For space reasons, details are left out.

CAAs for extended query formalisms: Many query languages for XML data and semistructured data [5, 18, 13] allow for arbitrary value comparisons (joins), i.e. database leaves are assumed to carry values, and queries may contain value comparisons. A CAA does not suffice to completely represent all answers to a query with value comparisons. Nonetheless, the CAA for the join-free part of the query can be built up representing a coarsening of the set of answers. Note that such a generalization of a query might speed up its evaluation and be appropriate for some applications such as e-commerce [1]. Extensions to CAA as defined here can be thought of for a faithful representation of the answers of a query with value comparisons.

Example 4.9 Consider a database consisting of a root node with 5 children such that 2 children carry the value 1, 2 the value 2, and 1 the value 3. The query $x \rightarrow_1 y_1 \wedge x \rightarrow_1 y_2 \wedge \text{value}(y_1) = \text{value}(y_2)$ has 5 answers while the CAA for $x \rightarrow_1 y_1 \wedge x \rightarrow_1 y_2$ has 5 target candidates for each of y_1 and y_2 representing 25 answers.

These complications are not accidental. With semistructured data like with relational data, a high price has to be paid for value comparisons, as Theorem 5.3 (cf. next section) shows.

5 A Query Hierarchy

Evaluation problems can be classified according to the structure of both, queries and databases.

Definition 5.1 Let $\mathcal{EP} = (Q, X_Q, \mathcal{D})$ be an evaluation problem. $\mathcal{EP}$ is of type *S-S (Sequence-Sequence)* if Q is a sequence query and D a sequence database. Evaluation problem of types S-T, S-D, S-G, T-S, T-T, T-D, T-G, D-S, D-T, D-D, D-G, and G-G are similarly defined: The first letter (S: sequence, T: tree, D: DAG, and G: graph) denotes the type of the query, the second, that of the database.

The thirteen classes of evaluation problems of Definition 5.1 form a hierarchy of increasing structural complexity. This hierarchy does not include the types G-S, G-T, G-D, because they would correspond to evaluation problems with cyclic (i.e. type G) queries: According to Definition 3.5, evaluation problems with cyclic queries have no solutions in acyclic (i.e. type S, T, and D) databases. Note that all the evaluation problems of the types specified in Definition 5.1 are non-trivial in the sense that they might have solutions.

Definition 5.2 A query is called *simple* if all its edge constraints are child, f-child, or descendant constraints. An evaluation problem $(Q, X_Q, \mathcal{D})$ is *simple* if Q is simple.

According to Definitions 3.4 and 3.3, a query is simple if it does not involve regular expressions.

For simple queries, the algorithms described in the next section have optimal complexity. Even non-simple T-G evaluation problems can be solved in polynomial time. However, in presence of value comparisons, e.g. joins, even T-T evaluation problems become NP-complete:

Theorem 5.3 *Solvability of simple T-T evaluation problems with value comparison is NP-complete with respect to combined complexity.*

6 Computation of CAAs

In this Section an algorithm for computing the CAA of a simple sequence query is first given. Then, its adaptation to simple tree queries is outlined. For space reasons, further adaptations, e.g. to tree queries involving regular path expressions, are not explained in this paper. A polynomial algorithm for this case is given in Appendix.

Simple sequence queries

The algorithm of Fig. 2 takes a simple sequence query Q and a database $\mathcal{D}$ as arguments and computes the CAA (Dom_Q, Π_Q) for the evalation problem induced by Q and $\mathcal{D}$. Starting from an empty array (line 3) and an empty set of edges (line 4) the algorithm adds target candidates to the domains (slots) of the query variables (line 15) and adds appropriate links to the set of edges (lines 31, 38). A pair (x, d) is said to be "added" to express that target candidate d is added to the domain (slot) of x. Possibly, pairs (x, d) are added that are "illegal" in the sense that $d \notin Dom_Q(x)$.

```
 1 procedure Aggregate comp_agg(Q,db)
 2 begin
 3    Dom:=empty DomainSet;
 4    Π:=empty EdgeSet;
 5    q_l:=leaf of Q;
 6    for all nodes d in db do
 7       map(Dom,Π,q_l,d);
 8    Agg:=Aggregate(Dom,Π);
 9    clean(Agg);
10    return Agg;
11 end;
12
13 procedure boolean map(Dom,Π,x,dx)
14 begin
15    add dx to Dom(x);
16    if dx satisfies the labeling
17       constraints of x then
18    begin
19       if x=root then return true
20       else
21       begin
22          y:=parent(x);
23          Anc:=appr_ancestors(dx,x,y);
24          map_found:=false;
25          for all dy_i ∈ Anc do
26             if dy_i ∉ Dom(y) then
27                if map(Dom,Π,y,dy_i) then
28                begin
29                   map_found:=true;
30                   add (y,x,dy_i,dx) to Π;
31                end
32             else
33             begin
34                if not is_red(Dom,y,dy_i) then
35                begin
36                   map_found:=true;
37                   add (y,x,dy_i,dx) to Π;
38                end;
39             end;
40          if map_found=false then
41             color_red(Dom,x,dx);
42          return map_found;
43       end;
44    else /*labeling constraints not satisfied*/
45    begin
46       color_red(Dom,x,dx);
47       return false;
48    end;
49 end;
```

Figure 2

Illegal pairs are detectecd and marked "red" (lines 40, 46). Only "legal" links, i.e., links in Π_Q, are introduced. As a last step, for each red pair (x, d) node d is deleted from the slot of x. After this slot "cleaning" (line 9), the complete answer aggregate (Dom_Q, Π_Q) is obtained.

Let $x_1, \ldots, x_p$ be the ordered sequence of query variables such that $x_i \rightarrow_? x_{i+1}$ $(1 \leq i < p)$ occurs in the simple sequence query. The algorithm starts with the (unique) query leaf x_p (line 5). The outermost loop (line 6) calls for each database node d_p the recursive function `map`. This function returns a boolean value indicating whether the pair (x_p, d_p) is legal. First, the pair is added (line 15). Assume the pair (x_i, d_i) has been added. If $i = 1$, then the pair is legal (line 19). Otherwise (x_i, d_i) is legal if and only if there exists a legal pair (x_{i-1}, d_{i-1}) such that (d_{i-1}, d_i) satisfies the edge constraint $x_{i-1} \rightarrow_? x_i$ in Q. Hence, for each "appropriate" ancestor d_{i-1}, which depends on the kind of the edge constraint, it is checked whether (x_{i-1}, d_{i-1}) is a legal pair (lines 27, 34). Two cases are distinguished:
1. If (x_{i-1}, d_{i-1}) has not been previously added, then the legality of (x_{i-1}, d_{i-1}) is checked by a recursive call of the function `map`.
2. Otherwise (x_{i-1}, d_{i-1}) is illegal if it is marked red. In both cases, if (x_{i-1}, d_{i-1}) is legal then a link between (x_{i-1}, d_{i-1}) and (x_i, d_i) is added (lines 30, 37). After inspection of all appropriate ancestors of d_i, the pair (x_i, d_i) is marked red if no legal pair (x_{i-1}, d_{i-1}) is found (line 40). Similarly (x_{i-1}, d_{i-1}) is marked red if d_i does not satisfy all labeling constraints of x_i (line 46).

Complexity: Clearly, for each pair (x, d), `map` is called at most once. Hence, the total number of calls to this function is bounded by the maximal number of pairs $q \cdot n$. Under the assumption that each database node has links pointing to parent nodes, computing the set of appropriate ancestors of a target candidate x_i takes time $O(a)$. Whether a node x_i satisfies a given labeling constraint $A(x_i)$ can be checked in constant time. Since each labeling constraint refers to a unique query variable, the total time needed for all tests related to labeling constraints is $O(q \cdot n)$. Cleaning takes time $O(q \cdot n)$. Therefore, the overall complexity is $O(q \cdot n \cdot a)$.

Call *adapter point* the bottom-most common query node of two query paths in the tree query. The query paths are processed consecutively. For each query path, the algorithm of Fig. 2 is modified as follows: When reaching an adapter point x_{i-1} that has already been visited during processing of another path, no new target candidates for slot x_{i-1} is introduced. Furthermore, only the already collected non-red target candidates (x_{i-1}, d_{i-1}) are used for links between target candidates in x_{i-1} and x_i. If, after a query path has been fully processed, a target candidate (x_{i-1}, d_{i-1}) for an adapter point x_{i-1} has no links to a target candidate in x_i, then it is marked red.

Simple tree queries

With simple tree queries, cleaning is more complicated. Call *downwards isolated* target candidates (x_{i-1}, d_{i-1}) such that for some child x_i of x_{i-1} there are no links from (x_{i-1}, d_{i-1}) to a target candidate within the slot x_i. After entering all target candidates, downwards isolated target candidates are detected. Since they are illegal, they are marked red. Removal of red nodes may result in new downwards isolated target candidates. The removal of red target candidates is based upon a variant of (the second part of) the well-known AC-4 arc-consistency algorithm [39].

Complexity: Since the recursive calls do not fill slots of adapter points twice, in this case as well no target candidates are processed twice in the first part of the algorithm. This gives a time complexity of $O(q \cdot n \cdot a)$ for this part. For cleaning an adaption of arguments from [39] yields time complexity $O(q \cdot n \cdot a)$. Space complexity is still $O(q \cdot n \cdot a)$.

Adding regular path expressions

A simple modification suffices to adapt the algorithms for sequence and tree queries described above can also be applied to queries involving regular path expressions. At line 23, if the query contains an α-descendant constraint $y \to_\alpha x$, then the set of appropriate ancestors of the current node dx is now the set of all α-ancestors of the database node dx.

The computation of the sets of α-ancestors needs some extra time. Using standard techniques from automata and graph theory, it is shown in Appendix that the resulting time complexity is $O(q^2 \cdot n \cdot a)$.

Simple DAG and graph queries

Four evaluation problems of the hierarchy turn out to be NP-complete with respect to combined complexity.

Define the *weight* of a node (resp. variable) of a database (resp. query) as 1 plus the number of labels attached to the node (resp. variable). Define the *size* of a database (resp. query) as the sum of the weights of its nodes (resp. variables) and its the number of edges. Define the *size* of an evaluation problem $(Q, X_Q, \mathcal{D})$ as the sum of the size of Q and the size of $\mathcal{D}$. It is shown in Appendix that so-called 1-in-3 problems over positive literals [21] can be encoded as D-T evaluation problems, or as T-T evaluation problems with value comparisons, using a polynomial translation. The following theorem is a simple consequence.

Theorem 6.1 *Whether a simple D-T (resp. D-D, D-G, G-G) evaluation problem $(Q, X_Q, \mathcal{D})$ has a solution is NP-complete with respect to the size of $(Q, X_Q, \mathcal{D})$.*

Table 1 summarizes the complexity results for the computation of CAAs (resp. for deciding solvability) given in this section. The case D-S in the upper table is established in Appendix. The parameter e denotes the number of edges in the query.

Due to the results on the size of CAAs (cf. Section 4), the bounds given at lines S and T of the upper table are optimal.

In practice the worst case time complexity for computing a CAA can be exponential with D-T, D-D, D-G and G-G evaluation problems. This could be faced by a polynomial-time computation of an "upper approximation" to the CAA, i.e. an answer aggregate yielding not only all answers, but also possibly containing target candidates or links not contributing to any answer. Such an upper approximation to a CAA can be obtained by

		Database			
		S	T	D	G
Queries	S	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$
without	T	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$	$O(q \cdot n \cdot a)$
reg. path	D	$O(q \cdot e \cdot n^3 \cdot a)$	(*) NP-compl.	(*) NP-compl.	(*) NP-compl.
expressions	G	-	-	-	(*) NP-compl.
Queries with	S	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$
reg. path expr.	T	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$	$O(q^2 \cdot n \cdot a)$

Table 1: Complexity for computing the CAA resp. (*) for deciding solvability

first selecting a spanning tree T of the considered query, then compute the CAA for the subquery induced by this spanning tree. Links representing possible interpretations of the query edges that have been omitted from Q might then be added. Arc consistency techniques [39] can be used to erase nodes and edges that do not contribute to any instantiation.

7 Advanced Query Answering Using CAAs

Once the CAA for an evaluation problem has been computed, it can be exploited for advanced query answering techniques we call *answer searching* and for *answer browsing*. These notions are explained referring to the example of Section 2: A query Q to a research project database $\mathcal{D}$ retrieves projects x and the managers z of these projects such that the string "XML" occurs in a title element u (at any depth) of some article y of a project x.

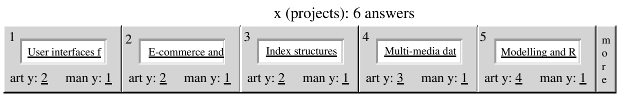
Answer Searching: In essence, the CAA of a query is a data structure making explicit the interdependencies between the answers to a query. Comparing queries and investigating the interrelationships between the various answers to a query is needed in many applications. CAAs can be used for scanning, comparing, filtering, ordering in the style of search engines as well as for analyzing in any other manner the answers to a query. Particularly promising are search primitives for detecting commonalities and differences between answers, computation of aggregate values like averages, maxima and minima. Note that the nodes of database items stored in a CAA potentially give access to the subelements rooted at these nodes, thus giving rise to a semantically rich "answer searching".

The set of answers for the above-mentioned query Q can be searched for:
- managers leading the highest number of project,
- for managers leading at least 2 projects,
- for projects with at least 10 XML articles.

To answer such queries, aggregate values have to be computed from the CAA of query Q. Note that such semantically related aggregate values are often computed in the same query. Many database applications like molecular biology sequence analysis and e-commerce require to perform such advanced comparisons from large answer sets computed from some "base query" [1].

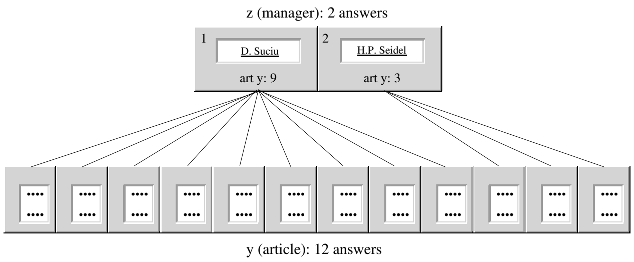
In many cases, the querying of the CAA of a base query can be carried out automatically. In some cases however, such an automatic "search" is not sufficient or not possible, an interactive "browsing of the answer space" is desirable.

Answer Browsing: A visualization of a CAA for a query can be convenient a basis for browsing the answers to that query. One can easily identify nodes of the CAA of query Q, like project 5, that deserve special attention by looking at the number of departing article links:

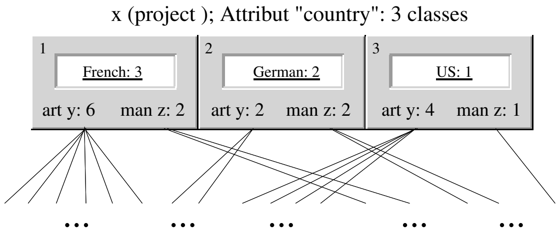


More elaborate visualization facilities would make it possible to directly browse between e.g. projects, articles, titles by following the links of the CAA.

If the CAA has a large number of nodes, then the user might get "lost in answer space". In such cases, it might be beneficial to restrict the visualization to a *view of the CAA*. In case of query Q, one might wish to restrict, say, the CAA to information associated with projects with at least three XML related articles. Also, *slot hiding* can provide with a better overview. Note that slot hiding corresponds to the projection operator of relational databases. If slot hiding is applied, then CAA links are inherited. Applying slot hiding to the running example yields:



A further visualization technique based upon CAA is *clustered aggregation.* Assuming that nodes have further attributes, CAA nodes that have identical attribute values can be merged. In case of query Q, if projects have a "country" attribute, if projects p_1, p_3, p_4 are French, projects p_2 and p_6 are German, and if p_5 is a US-project, then applying clustered aggregation might yield:



This presentation might be used to show which countries have projects of interest and how many.

General picture: cascade style query-answering: The possibilities for automated search as well as interactive browsing of answer sets the CAAs offer suggests that, for many applications, query answering can be processed in two or more successive phases, the first of which resulting in the construction and storing of the CAA, the following phases consisting in an inspection of this CAA or in the construction of further, more specialized CAAs. CAA inspection can consist both in an automated search or in an interactive browsing. In addition, to help analyzing and/or browsing answers, such a query answering based upon CAA can help to react rapidly to user query requests.

Cascade style query-answering query answering is possible with any query language, indeed. The contribution of CAAs lies in an intermediate data structure supporting this form of query answering which can be efficiently computed. Note that for many novel applications such as e-commerce such a cascade style query-answering is needed [1].

8 Related Work

Tree Databases: For "tree databases", a simplified form of CAA has already been introduced in [36, 35]. The present paper significantly extends over this early work.

Query formalisms for semi-structures data: Several query models for XML and semistructured data have been designed and/or implemented and used [49, 29, 10, 44, 23, 41, 42], cf. [11, 19] for surveys. These query models are more ambitious than the model presented in this paper, however, in contrast to this paper they are not devoted to *aggregating answers.* Thus, the contribution of this paper is widely orthogonal and complementary.

Conjunctive queries: The queries considered in this paper are related to conjunctive queries over relational databases as investigated in [14, 50, 24, 25]. However, it must be stressed that the distinction between tree queries and DAG or graph queries does *not* correspond to the conventional distinction between acyclic and cyclic conjunctive queries in database theory.

The distinction of database theory between acyclic and cyclic conjunctive queries refers to the hypergraph of the query, which is an undirected graph. In contrast, the query atoms considered in this paper are unary (labeling constraints) or binary (edge constraints). A binary atom $r(x, y)$ imposes a fixed orientation $x \to y$ on $\{x, y\}$ which reflects the direction of edges in the database. The conjunctions considered in this paper are such that the set of their binary atoms induces a sequence, tree, DAG, or graph structure on the query. Hence, the NP-hardness result for DAG-queries given in Section 6 is not in conflict with general results on polynomial tractability of acyclic conjunctive queries.

Dynamic programming and constraint reasoning: The algorithms described in Section 6 are closely related to methods of dynamic programming and to the arc-consistency techniques for constraint networks, cf. Appendix for details.

Index structures: Index structures as discussed in [33, 9, 22, 38, 37] can be used to improve the practical efficiency of the computation of CAAs. Details can be found in Appendix.

9 Conclusion

The paper motivated and introduced "complete answer aggregates (CAAs)" as a model for aggregating the answers to a sequence, tree, DAG, or graph query in a semistructured database. Algorithms for the computation of CAAs for queries of various structural kinds have been presented. A hierarchy of evaluation problems the CAAs of which can be computed in polynomial time (with respect to combined complexity) has been given. Cases have been characterized where computation of CAAs is NP-complete.

The query model presented in this paper has been implemented for the special case of tree queries and of tree databases. The implementation is being tested with large collections of complex structured documents. The prototype currently available does not handle regular path expressions, but can cope with left-to-right order constraints between the children of a query node, as needed in document management. The necessary adaptation of the notion of a CAA as well as the mathematical and algorithmic background is given in [36, 35]. For this expanded signature, the time complexity of the algorithm for computing the CAA is $O(q \cdot n \cdot a \cdot log(n))$. The additional logarithmic factor comes from the fact that order information is not being taken into account and handled during query evaluation. An "answer browser" based on CAAs is currently being developed.

Future work will concentrate on extensions of the concepts and algorithms of Section 6 to more expressive query languages following e.g. [41, 42]. Improved index structures, as well as other optimization techniques, will also be investigated in more detail. A promising issue for further research is to adapt CAAs and their computation algorithms to queries with "construct" parts. Furthermore, it would be worth investigating whether standard query answeriung can be optimized using CAA as an internal, temporary data structure used e.g. between the processing of the "select" and "construct" parts of a query so as to avoid the generation of exponentially many "object assignments" [33].

References

[1] A. Gupta. Some Database Issues in E-Commerce. Invited talk at the Int. Conf. on Extending Database Technology, http://www.edbt2000.uni-konstanz.de/invited/talks. html, 2000.

[2] A. Loeffen. Text Databases: A Survey of Text Models and Systems. *SIGMOD Record*, 23(1):97–106, March 1994.

[3] S. Abiteboul. Querying Semi-Structured Data. In *Int. Conf. on Database Theory*, 1997.

[4] S. Abiteboul et al. *Foundations of Databases.* Addison Wesley, 1995.

[5] S. Abiteboul et al. The Lorel Query Language for Semistructured Data. *Int. Jour. on Digital Libraries*, 1(1):68–88, 1997.

[6] S. Abiteboul et al. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[7] B. Amann and M. Scholl. Gram: A Graph Data Model and Query Language. In *ACM Conf. on Hypertext*, 1992.

[8] AML (Astronomical Markup Language). http://strule.cs.qub.ac.uk/damieng/these/, 1999.

[9] R. Baeza-Yates and G. Navarro. Integrating Contents and Structure in Text Retrieval. *SIGMOD Record*, 25(1):67–79, 1996.

[10] The BBQ Page. http://www.npaci.edu/DICE/MIX/BBQ/, 2000.

[11] A. Bonifati and S. Ceri. A Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, March 2000.

[12] P. Buneman. Semistructured Data. In *ACM Symp. on Princ. of Database Systems*, 1997.

[13] S. Ceri et al. XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks*, 31(11–16):1171–1187, May 1999.

[14] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symp. on Theory of Computing*, 1977.

[15] XML/CML – Chemical Markup Language. http://www.nottingham.ac.uk/pazpmr/README, 1999.

[16] M. Consens and A. Mendelzon. GraphLog: a Visual Formalism of Real Life Recursion. In *ACM Symp. on Princ. of Database Systems*, 1990.

[17] D. Maier. Database Desiderata for an XML Query Language. In *The Query Languages Workshop*, 1998.

[18] A. Deutsch et al. XML-QL: A Query Language for XML. Submission to W3C, http://www.w3.org/TR/NOTE-xml-ql/, 1998.

[19] A. Deutsch et al. Querying XML Data. *IEEE Data Bulletin*, 22(3):10–18, 1999.

[20] E. C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, Jan. 1982.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[22] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Int. Conf. on Very Large Data Bases*, 1997.

[23] R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases. In *Int. Workshop on the Web and Databases*, 1998.

[24] G. Gottlob et al. The Complexity of Acyclic Conjunctive Querie. In *Annual Symp. on Foundations of Computer Science*, 1998.

[25] G. Gottlob et al. Hypertree Decompositions and Tractable Queries. In *ACM Symp. on Princ. of Database Systems*, 1999.

[26] N. Guarino, editor. *International Conference on Formal Ontology in Information Systems*. IOS Press, 1998.

[27] M. Gyssens et al. A Graph-Oriented Object Database Model. *IEEE Trans. on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[28] J. Robie. The Tree Structure of XML Queries. http://www.w3.org/1999/10/xquery-tree.html, 1999.

[29] The Lore Page. http://www-db.stanford.edu/lore/demo/, 2000.

[30] M. Fernandez and others. XML Query Languages: Experiences and Exemplars. http://www.w3.org/1999/09/ql/docs/xquery.html, 1999.

[31] A. K. Mackworth. Consistency in Networks of Constraints. *Artificial Intelligence*, 8, 1977.

[32] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[33] J. McHugh et al. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), 1997.

[34] K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.

[35] H. Meuss. *Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents*. PhD thesis, Dept. of Computer Science, University of Munich, 2000.

[36] H. Meuss and K. U. Schulz. Complete Answer Aggregates for Structured Document Retrieval. Technical Report 98-112, CIS, University of Munich, 1998. Submitted.

[37] H. Meuss and C. Strohmaier. Improving Index Structures for Structured Document Retrieval. In *Annual Colloquium on IR Research*, 1999.

[38] T. Milo and D. Suciu. Index structures for path expressions. In *Int. Conf. on Database Theory*, 1999.

[39] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[40] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[41] F. Neven and T. Schwentick. Query Automata. In *ACM Symp. on Princ. of Database Systems*, 1999.

[42] F. Neven and T. Schwentick. Expressive and Efficient Pattern Languages for Tree-Structured Data. In *ACM Symp. on Princ. of Database Systems*, 2000.

[43] J. Oesterle and P. Maier-Meyer. The GNoP (German Noun Phrase) Treebank. In *Int. Conf. on Language Resources and Evaluation*, pages 699–703, 1998.

[44] The Strudel Page. http://www.research.att.com/sw/tools/strudel/, May 2000.

[45] D. Suciu. An Overview of Semistructured Data. *SIGACT News*, 29(4), 1998.

[46] J. D. Ullman. *Database and Knowledge-Base Systems, Vol. I and II.* Computer Science Press, 1989.

[47] XML Query Requirements. http://www.w3.org/TR/2000/WD-xmlquery-req-20000131, 2000.

[48] I. Wegener. *Theoretische Informatik.* Teubner, 1993. (in German).

[49] The XML-QL Page. http://db.cis.upenn.edu/XML-QL/, May 2000.

[50] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Int. Conf. on Very Large Data Bases*, 1981.

10 Appendix: Algorithms and Complexity

In this Appendix we add those parts of the full paper that are necessary to verify the claims of the previous sections. If not said otherwise we always refer to simple queries and simple evaluation problems, respectively.

10.1 The algorithm for simple sequence queries

We give a correctness proof for the algorithm given in Fig. 3, first for queries without regular path expressions.

Correctness. Obviously the output (`Dom`,Π) of `comp_agg` is an answer aggregate for the given query since label and edge constraints are checked. It is simple to see that the algorithm introduces an edge `(y,x,dy,dx)` only once it is clear that `(x,dx)` and `(y,dy)` remain uncolored. It is also simple to see that each uncoloured pair `(x,dx)` and each link `(y,x,dy,dx)` contributes to an instantiation of the answer aggregate. It remains to show that each answer to the query `Q` is an instantiation of (`Dom`,Π). Let x_1 (root), $x_1, \ldots, x_p$ (leaf) be the sequence of variables of q. Let $\nu : x_i \mapsto d_i$ $(1 \leq i \leq p)$ be an answer. Since `map` is called with argument (x_p, d_p) in `comp_agg` it follows that for each pair (x_i, d_i) $(1 \leq i \leq p)$ there is a call to `map`. A simple induction, starting from root variable x_1, shows that each pair (x_1, d_1),…,(x_p, d_p) remains uncoloured and for $1 \leq i < p$ we have links $(x_i, x_{i+1}, d_i, d_{i+1})$ in Π. Since `clean` only removes red nodes it follows that ν can be obtained as an instantiation of the output (`Dom`,Π) of `comp_agg`.

10.2 The algorithm for simple tree queries

For tree queries (we first consider queries without regular path expressions) we need the help of various supporting data structures, since the procedure `clean` becomes more complicated. `Dep_Set` stores for every target candidate `(x,dx)` the set of dependent target candidates, i.e. target candidates with a link to `(x,dx)` or from `(x,dx)`. In addition, we need counters `link_count` for every triple `(x,dx,y)` that store the number of links connecting target candidate `(x,dx)` with target candidates in `Dom(y)`. Nodes that are colored red are in addition stored in a data structure `Del_Stack` that guides in the procedure `clean` the removal of red target candidates from the answer aggregate. The last new data structure is a variable `Adaptors` that stores all pairs of query nodes `(x,y)`, where `x` is a parent of `y`, and where `x` was treated by the algorithm as node in a previously treated path (not containing `y`).

The function `map` is triggered by all query leaves. For every query leaf `map` recursively visits all nodes in the corresponding query path up to the root. Lines 24a-24h treat the case that a query node y was already treated by another path of the query. In this case, only those ancestors of `dx` are included in `Valid` that already occur in `Dom(y)`. For the valid nodes approriate links are added to Π and the pair `(y,x)` is stored in `Adaptors`, but no recursive call to `map` is made.
Instead of simply adding a link to Π like in the sequence case, a function `add` is called in lines 24f, 30 and 37. This function adds the link to Π and updates the supporting data structures `Dep_Set` and `link_count`.
The following algorithm is an adaption of the algorithm for sequence queries. In order to keep line numbering consistent, lines that had to be inserted for the tree case are marked with alphabetical numbering, e.g. line 24a.

```
procedure Aggregate comp_agg(Q,db)
begin
   Dom=empty DomainSet;
   Π=empty EdgeSet;
   Del_Stack=empty Stack;
   Dep_Set=empty DependencySet;
   link_count(x,dx,y)=0 for all x,y ∈ Q and dx ∈ db;
   Adaptors=empty AdaptorSet;
   for all leaves q_l in Q do
      for all nodes d in db do
         map(Dom,Π,q_l,d,Del_Stack,Dep_Set,link_count,Adaptors);
   check_adaptors(Adaptors,Dom,link_count,Del_Stack);
   clean(Del_Stack,Dom,Π,Dep_Set,link_count);
   Agg=Aggregate(Dom,Π);
   return Agg;
end;

procedure boolean map(Dom,Π,x,dx,Del_Stack,Dep_Set,link_count,Adaptors)
begin
   add dx to Dom(x);
   if dx satisfies labeling
      constraints of x then
   begin
      if x=root then return true
      else
      begin
         y=parent(x);
         Anc=ancestors(dx,y,x);
         map_found=false;
         if y was already treated then
         begin
            add (y,x) to Adaptors;
            Valid=Anc ∩ non_red(Dom(y));
            for all nodes dy_i ∈ Valid do
               add(Π,y,x,dy_i,dx,Dep_Set,link_count);
            map_found=not(empty(Valid));
         end
         else
         begin
            for all dy_i ∈ Anc do
               if dy_i ∉ Dom(y) then
                  if map(Dom,Π,y,dy_i) then
                  begin
                     map_found=true;
                     add(Π,y,x,dy_i,dx,Dep_Set,link_count);
                  end
               else
               begin
                  if not is_red(Dom,y,dy_i) then
                  begin
                     map_found=true;
                     add(Π,y,x,dy_i,dx,Dep_Set,link_count);
                  end;
               end;
            if map_found=false then
            begin
```

```
41b                 color_red(Dom,x,dx);
41c                 push(Del_Stack,(x,dx));
41d              end;
42               return map_found;
43            end;
43a        end;
44      else      /*   labeling constraints not satisfied    */
45      begin
46         color_red(Dom,x,dx);
46a        push(Del_Stack,(x,dx));
47         return false;
48      end;
49   end;
50
51
52   procedure void  add(Π,y,x,dy,dx,Dep_Set,link_count)
53   begin
54      add (y,x,dy,dx) to Π;
55      add (x,dx) Dep_Set(y,dy);
56      add (y,dy) Dep_Set(x,dx);
57      link_count(y,dy,x)=link_count(y,dy,x)+1;
58      link_count(x,dx,y)=link_count(x,dx,y)+1;
59   end;
```

Due to the special treatment of adaptor points in lines 24a-24h it is possible that a slot `x`, where `(x,y)` is an adaptor point, contains target candidates having no link to a target candidate in `Dom(y)`. These target candidates are colored red by the procedure `check_adaptors`.

In the case of tree queries, the procedure `clean` gets more complicated, since the removal of a red target candidate `(x,dx)` can trigger removal of other target candidates that depend on `(x,dx)` and became downwards isolated. The sets `Dep_Set` and the counters `link_count` are used in order to make these dependencies explicit. If a target candidate `(x,dx)` is removed in `clean` we test for each dependent target candidate if it still has a further link to another target candidate in `Dom(x)`. In the negative case the target candidate is downwards isolated and can not contribute to an answer. It is colored red for later removal (lines 83,84). The algorithm for cleaning the answer aggregate is a special instance of the second part of the algorithm AC-4 given in [39] for pruning a constraint network. In our special case, we can improve on the parameters for the time complexity.

```
60   procedure void check_adaptors (Adaptors,Dom,link_count,Del_Stack)
61   begin
62      for all (x,y) ∈ Adaptors do
63         for all dx ∈ Dom(x) do
64            if link_count(x,dx,y)=0 then
65            begin
66               push(Del_Stack,(x,dx));
67               color_red(Dom,x,dx);
68            end;
69   end;
70
71
72   procedure void clean(Del_Stack,Dom,Π,Dep_Set,link_count)
73   begin
74      while not empty(Del_Stack) do
75      begin
76         (q_del,d_del)=pop(Del_Stack);
```

```
77          for all (y,dy) ∈ Dep_Set(q_del,d_del) do
78          begin
79             remove (y,q_del,dy,d_del) from Π;
80             link_count(y,dy,q_del)=link_count(y,dy,q_del)-1;
81             if not(is_red(Dom,y,dy)) and link_count(y,dy,q_del)=0 then
82             begin
83                color_red(Dom,y,dy);
84                push(Del_Stack,(y,dy));
85             end;
86          end;
87          remove d_del from Dom(q_del);
88       end;
89    end;
```

Correctness. As in the sequence case, all edges and non-red target candidates added to the answer aggregate contribute to a solution. Now let ν be an answer to the query and database, let x, y be two query variables connected with an edge constraint $y \rightarrow_f x$, $y \rightarrow x$ or $y \rightarrow_+ x$. We show that $\nu(x)$ ($\nu(y)$) appears as a non-red target candidate in slot x (y) and $(y, x, \nu(y), \nu(x))$ is in Π. In the sequence case, we already showed that `map` is called for the pairs $(x, \nu(x))$ and $(y, \nu(y))$. Labeling constraints are satisfied. Consider a call to `map` with the pair $(x, \nu(x))$. If y was already treated as part of another path of the query (lines 24a-24h), we can show by induction that $\nu(y)$ is contained as a non-red target candidate in slot y. Therefore a link is added (line 24f) and $\nu(x)$ remains uncolored, because `map_found` is set to true. If y was not treated before, the argumentation is the same as in the sequence case. Now consider the procedure `check_adaptors`. If the pair (y, x) corresponds to a previously treated adaptor point, we are in the case discussed above, where y was treated as path of another query path. $\nu(y)$ is not colored red here, since its `link_count` to target candidates in slot x is at least 1 (for the link $(y, x, \nu(y), \nu(x))$). Now consider `clean`. Neither $(x, \nu(x))$ nor $(y, \nu(y))$ are colored red so far. By induction we can infer that the `link_count` values of $(x, \nu(x))$ and $(y, \nu(y))$ to other query nodes z cannot be 0 (for query nodes z connected with an edge to x or y, respectively). Therefore they are not removed and $\nu(x)$, $\nu(y)$ and $(y, x, \nu(y), \nu(x)$ appear in the output answer aggregate.

Complexity. As in the sequence case we can see that `map` is called at most once for each pair `(x,dx)`. The time complexity of the new case (line 24a-24h) is of order $O(a)$ for the intersection and the loop, since there are maximally a elements in `Anc` and for every element a lookup in array `Dom(y)` is of constant time. The other cases remain essentially unchanged, yielding a total time complexity of $O(q \cdot n \cdot a)$ for `map`. Procedure `check_adaptors` is of time complexity $O(q \cdot n)$.

For the procedure `clean` we follow the arguments given in [39] for AC-4: Since each link in the answer aggregate increments the sum of all `link_count` values by two (in procedure `add`), the sum of all `link_count` values is of order $O(q \cdot n \cdot a)$. No `link_count` value can become negative, since for every element of a `Dep_Set` there is one decrementation in `clean`, as there has been one incrementation in `add`. Therefore the total number of times the inner loop of `clean` is executed is bounded by the total sum of `link_count` values. This gives a time complexity of $O(q \cdot n \cdot a)$ for `clean`.

10.2.1 Use of index structures

We briefly discuss how index structures developed for graph databases and structured document retrieval can be integrated in the above algorithms.

Local index structures Local index structures implement a mapping from node properties (like attribute values, labels, text containment) to sets of nodes, or from nodes to sets of nodes constituting e.g. the ancestors or parents of the input node. As an illustrating example we use index structures such as defined in the Lore system ([33]). Similar mechanisms appear in other semistructured databases or structured document retrieval systems. Lore uses three index structures that implement these mappings, namely the V-Index, T-Index and L-Index. The V-Index and T-Index are used to retrieve leaves with a specified content (in the case of the T-Index textual content), whereas the L-Index maps nodes to parents with taking edge labels into account. We can use the V-Index and T-Index in line 6 of the algorithm and call the function `map` only for those database nodes that meet the requirements imposed by the query node `q_1`. If $y \rightarrow_f x \in E$ or $y \rightarrow x \in E$, we can call in line 23 the L-Index that computes the set `Anc` efficiently and omits nodes that do not fit with query node `y`. A special index for computing the ancestors of a database node can be used in line 23 of the algorithm, if $y \rightarrow_+ x \in E$.

Context index structures Context index structures take into account a whole part of the query (e.g. a path) instead of single nodes only. We will describe two kinds of path index structures.

Exact path index structures return a set of paths matching a query path exactly. Sophisticated path index structures have been developed in the field of semistructured databases ([38, 22]). They return only those sets of nodes that are reachable via a given path starting from the root of the database. The path expressions can even contain regular expressions over the labels to be traversed. Path index structures can be used[1] in line 6 of the algorithm in order to return only those query nodes that are reachable via the actual query path. This reduces the task of the rest of the algorithm. For sequence queries it is only necessary to enter the target candidates into the appropriate slot of the answer aggregate.

Path filter structures ([37]) are used on top of local index structures like the V-Index of Lore. The local index structure, e.g. the V-index, is assumed to encode additional contextual information as a bit string. These bit strings are compared with the contextual information imposed by the actual query path. This comparison can be efficiently computed and filters out nodes not conforming to the contextual requirements imposed by the query. Due to the efficient encoding as bit strings, the path filter structure is not exact, i.e. possibly not all irrelevant nodes are filtered out. But correctness is maintained since the query evaluation algorithm ensures that nodes not matching the query nodes are not entered into the answer aggregate. A path filter can be used in line 6 of the algorithm in the same way as an exact path index.

10.3 Algorithm for queries with regular path expressions

When adding regular path expressions to (sequence or tree) queries, the correctness proofs given above are not affected. This follows easily from the fact that algorithmic modifications only concern line 23 of the above algorithms.

In order to give complexity results for queries with regular path expressions we modify the definition of the size q of a query $\mathcal{Q}$ in the following way: q is the number of variables plus the number of labeling constraints in $\mathcal{Q}$ plus the sum of the notational lengths r_i of all regular expressions α_i in $\mathcal{Q}$.

For a complexity analysis of the enhanced algorithms we imagine (for simplicity) that the computation is organized in two steps. In the first preparatory step, we compute for each database node `dx` and for each regular expression α occurring in the query the set of α-ancestors of `dx`. Assuming that this preprocessing step is done, execution of line 23 of the algorithm takes constant time. Hence, ignoring the time for the preprocessing step, the complexity of the remaining algorithm is then $O(q \cdot n \cdot a)$, as we have seen above. In reality, the set of α-ancestors of a database node is of course computed only on demand.

Let us now determine the time-complexity for the preprocessing step. We proceed in the following way. For each regular expression α occurring in Q, say, of notational length r, using standard constructions (see, e.g., [48]) we compute in time

[1] Since [38] does assume the existence of a unique root node in the database, some simple adaptions are necessary.

$O(r)$ a non-deterministic finite automaton $A^{\epsilon}_{\alpha^R}$ for the language L that represents the reverse language of $L(\alpha)$. This automaton may contain ϵ-transitions. Using well-known techniques from automata theory (cf., e.g. [48], Chapter 4, Proposition 4.4.8) a non-deterministic automaton A_{α^R} for the same language L without ϵ-transitions can be computed in time $O(r^2)$. The overall complexity for the computation of the ϵ-free non-deterministic automata A_{α^R} for all regular expressions α occurring in the query Q can be estimated by $O(q^2)$.

Given a database node d, let A_d denote the automaton where the set of states Q_d is given by the set of ancestors of d including d, where d is the start state (we do not fix final states), and where the transitions are obtained by reverting each edge between two nodes of Q_d. Recall that the size (number of nodes and edges) of A_d is bounded by the parameter a. Again using algorithms for reachability, A_d can be computed in time $O(a)$.

Consider one of the regular expressions α occurring in Q, say, of notational length r. Let $\mathcal{A}$ denote the product automaton of A_{α^R} with A_d. The set of α-ancestors of d is obtained by taking all the second components e of states (f, e) of $\mathcal{A}$ that can be reached from the start state and where f is a final state of A_{α^R}. Since the size of $\mathcal{A}$ is of order $O(a \cdot r^2)$ this reachability problem can be solved in time $O(a \cdot r^2)$ (cf., e.g., [34] Sec. IV.4). For fixed node d all the sets of α-ancestors for arbitrary regular expressions α in the query can be computed in time $O(a \cdot q^2)$. The total time needed for arbitrary database nodes, which yields the time needed for the preprocessing step, is $O(a \cdot q^2 \cdot n)$.

Since the time for the preprocessing step dominates the time needed for the second phase, the total time-complexity for computing the CAA is of order $O(a \cdot q^2 \cdot n)$.

10.4 NP-complete classes of evaluation problems

Lemma 10.1 *The problem if a D-T evaluation problem* $(Q, X_Q, \mathcal{D})$ *has solution is NP-complete w.r.t. the size of* $(Q, X_Q, \mathcal{D})$.

Proof. Obviously the problem is in NP: we may guess a variable assignment that maps the set of variables of Q to the set of nodes of $\mathcal{D}$ and then check in polynomial-time if the mapping satisfies all formulae of Q. In order to prove NP-hardness we show that solvability of 1-in-3 problems over positive literals[2] can be encoded as DAG-tree evaluation problems using a polynomial translation. Since the problem of solvability of 1-in-3 problems over positive literals is NP-complete (cf. [21]) NP-hardness follows.

Example. Before we treat the general case we illustrate the encoding technique with an example. Consider the problem with the clauses $C := \{p, q, r\}$, $D := \{p, q, s\}$, and $E := \{q, r, s\}$. There is exactly one solution ν, mapping q to 1 (true) and p, r, s to 0 (false). The DAG query Q (tree database $\mathcal{D}$) assigned to $\{C, D, E\}$ is depicted on the left-hand (right-hand) side of Fig. 3. We use expressions $x \rightarrow_n y$ ($n \geq 1$) as a short-hand for $\exists x_1 \cdots x_{n-1} : x \rightarrow x_1 \rightarrow_1 \cdots \rightarrow x_{n-1} \rightarrow y$.
Let μ denote any solution of this problem. Because of the labeling constraint $C(c)$ query variable c is mapped to a node in $\{p^C, q^C, r^C\}$ under μ. If c is mapped to p^C (resp. q^C or r^C) we read this as "literal p (resp. q or r) is selected from C". Variable d must be mapped to a node in $\{p^D, q^D, s^D\}$, and variable e must be mapped to a node in $\{q^E, r^E, s^E\}$. This shows that μ defines a selection function ν that maps each of the clauses C, D, E to one of its literals. It remains to see that ν defines a truth value assignment on the common set of literals $\{p, q, r, s\}$.

Consider query variables p^c and p^d, corresponding to the two occurrences of p in C and D respectively. Since under μ these two variables must be mapped to database nodes with a *common* descendant with label F (the image of y^p) they have to be mapped to nodes of the same subbranch of the subtree with root 1 of $\mathcal{D}$. Similarly all variables labeled with Q (or R, S respectively) have to be mapped to nodes of the same subbranch

[2] A *1-in-3 problems over positive literals* is given by a finite set of clauses $C_1, \ldots, C_n$, each containing exactly three positive literals. A *solution* is a truth value assignment, defined on the joint set of literals, that evaluates exactly one literal of each clause to 1 (true).

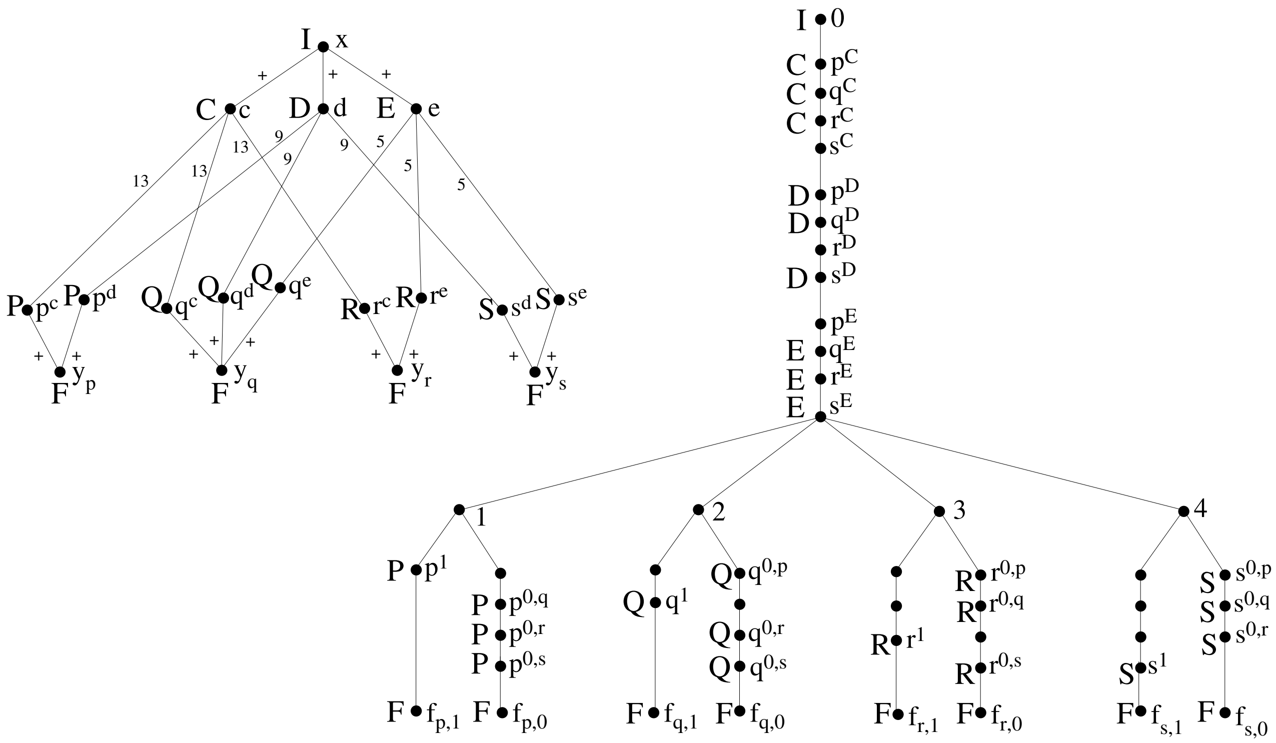


Figure 3: Encoding 1-in-3 problems as D-T evaluation problems

of the subtree with root 2 (or 3, 4 respectively) under μ.

Assume now that $\mu(c) = p^C$ (read: "we select literal p from clause C"). This implies that $\mu(p^c)$ is a descendant of p^C with label P in distance 13, which implies $\mu(p^c) = p^1$ (the exponent 1 of p^1 is meant to indicate that p is a selected literal.) The above remarks show that also $\mu(p^d) = p^1$. The distance between p^1 and $\mu(d)$ has to be 9, hence $\mu(d) = p^D$, which means that ν selects literal p in clause D, too. In the same way it follows that whenever ν selects a literal in one clause, it selects the same literal in all other clauses where the literal appears. This shows that ν in fact defines a solution of $\{C, D, E\}$.

Now let ν be a solution of $\{C, D, E\}$ (in our example, there is just one solution which selects q in each clause). In order to find a solution μ of $(Q, \mathcal{D})$ we map each of the variables c, d, e to the node that represents the literal selected under ν in C, D, E respectively. If literal p (q, etc.) evaluates to 1 under ν, we map all variables with label P (Q, etc.) to node p^1 (q^1, etc.). In the other case, if literal p is not selected, then we map variable p^c to $p^{0,x}$ (index 0 indicates that p is not selected) where x denotes the unique literal of C selected under ν, and we map variable p^d to $p^{0,y}$ where y denotes the unique literal of D selected under ν. In the same way, mutatis mutandis, we map all other query variables of the form v^z where v is a literal not selected under ν. It is easy to see that all distance requirements and labeling constraints are satisfied under this mapping, which means that we receive a solution of the evaluation problem.

In the concrete example, the mapping that we obtain from the unique solution is represented in the CAA depicted in Fig. 4 (simplified visual presentation).

General encoding. Assume we are given a 1-in-3 problem over positive literals $p_1, \ldots, p_m$ with clauses $C_1, \ldots, C_n$. The database $\mathcal{D}$ associated with this problem contains the following nodes.

1. a node 0 with label I representing the root

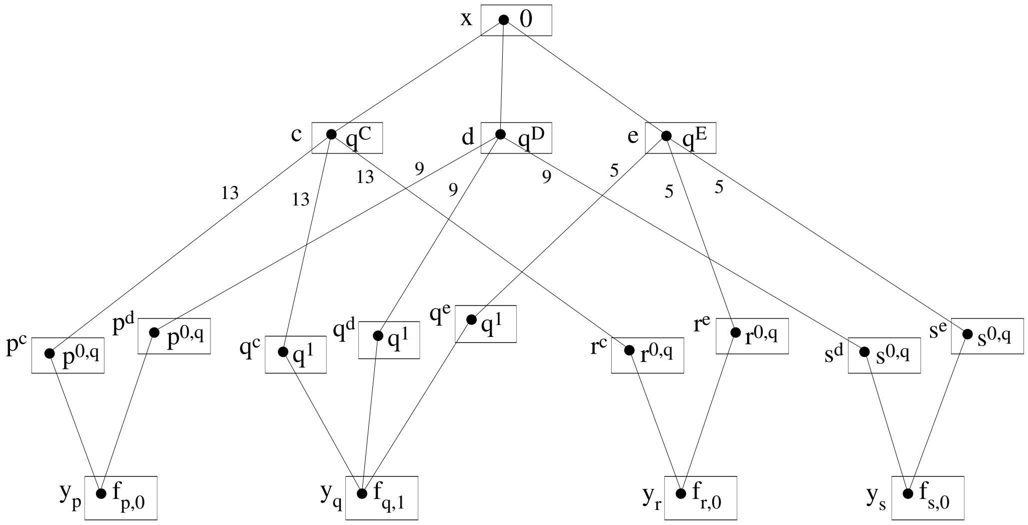


Figure 4: CAA (simplified visual presentation) for the D-T evaluation problem of Fig. 3

of $\mathcal{D}$,

2. a node named ${p_i}^{C_j}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. Node ${p_i}^{C_j}$ has label C_i iff $p_i \in C_j$,

3. nodes $1, \ldots, m$,

4. nodes p_i^1 for $1 \leq i \leq m$, and nodes p_i^{0,p_k} for $1 \leq i \neq k \leq m$. Each node p_i^1 or p_i^{0,p_k} has label P_i,

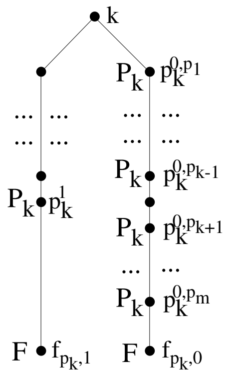
5. for each literal p_i, i additional unlabeled nodes (the names are irrelevant),

6. nodes $f_{p_i,1}$ and $f_{p_i,0}$ with label F, for all $1 \leq i \leq m$.

There are no additional labels. Root 0 has ${p_1}^{C_1}$ as single child. Starting from the latter node, there is a linear sequence

$$
{p_1}^{C_1}, \ldots, {p_1}^{C_n}, \ldots, {p_m}^{C_1}, \ldots, {p_m}^{C_n}
$$

where each successor represents the unique child of its predecessor. The last node of this sequence, ${p_m}^{C_n}$, has nodes $1, \ldots, m$ as its children. Each child k is the root of a subtree of the following form (the unlabeled nodes below k are the k additional nodes mentioned above):



The query Q associated with $C_1, \ldots, C_n$ has root variable x labeled I. For each clause C_j, the root has one child c_j. For each occurrence of a literal p_i in a clause C_j the query has a variable ${p_i}^{c_j}$ with label P_i, the single parent of ${p_i}^{c_j}$ is variable c_j. For each i, $1 \leq i \leq m$, all variables of the form ${p_i}^*$ have a common child y_{p_i} labeled F. This defines the set of variables, the set of label constraints and the general DAG structure. The edge constraints are the following:

1. constraints $x \rightarrow_+ c_j$ for $1 \leq j \leq n$,

2. a constraint ${p_i}^{c_j} \rightarrow_+ y_{p_i}$ for each variable of the form ${p_i}^{c_j}$,

3. a constraint $c_j \rightarrow_{m(n-j+1)+1} {p_i}^{c_j}$ for each literal p_i occurring in C_j.

It is easy to see that $(Q, \mathcal{D})$ has size $O(n \cdot m + m \cdot m)$ and the encoding is polynomial. It remains to show that $(Q, \mathcal{D})$ has a solution if $C_1, \ldots, C_n$ has a solution.

First assume that $C_1, \ldots, C_n$ has a solution ν. Consider the following mapping μ from the set of variables of Q to the set of nodes of $\mathcal{D}$:

1. x is mapped to 0,
2. each variable c_i is mapped to the node ${p_i}^{C_j}$ where p_i is the unique literal of C_j which evaluates to 1 under ν,
3. each variable ${p_i}^{c_j}$ is mapped to p_i^1 iff $\nu(p^i) = 1$ and to p_i^{0,p_k} if $i \neq k$ and $p_k \in C_j$ evaluates to true under ν otherwise,
4. each variable y_{p_i} is mapped to $f_{p_i,1}$ iff $\nu(p_i) = 1$ and to $f_{p_i,0}$ if $\nu(p_i) = 0$.

It is trivial to see that all labeling constraints of Q are satisfied under μ. Constraints $x \rightarrow_+ c_i$ are obviously satisfied.

Consider a constraint ${p_i}^{c_j} \rightarrow_+ y_{p_i}$. If $\nu(p_i) = 1$, then $\mu({p_i}^{c_j}) = p_i^1$, $\mu(y_{p_i}) = f_{p_i,1}$ and the constraint ${p_i}^{c_j} \rightarrow_+ y_{p_i}$ is satisfied (cf. above figure). If $\nu(p_i) = 0$, then $\mu({p_i}^{c_j})$ has the form p_j^{0,p_k} for some k, $\mu(y_{p_i}) = f_{p_i,0}$ and the constraint ${p_i}^{c_j} \rightarrow_+ y_{p_i}$ is satisfied, too.

Consider a constraint $c_j \rightarrow_{m(n-j+1)+1} {p_i}^{c_j}$. First assume that $p_i \in C_j$ is the unique literal in C_j that evaluates to 1 under ν. In this situation, $\mu(c_j) = {p_i}^{C_j}$ and $\mu({p_i}^{c_j}) = p_i^1$. It is easy to check that in fact p_i^1 is a descendant of ${p_i}^{C_j}$ in distance $m(n-j+1)+1$. Second assume that for $k \neq i$ literal p_k is the unique literal of C_j which evaluates to 1 under ν. In this case $\mu(c_j) = {p_k}^{C_j}$ and $\mu({p_i}^{c_j}) = p_i^{0,p_k}$. Again it is easy to check that p_i^{0,p_k} is a descendant of ${p_k}^{C_j}$ in distance $m(n-j+1)+1$.

To sum up, we have seen that μ defines a solution of $(Q, \mathcal{D})$.

2. Assume that $(Q, \mathcal{D})$ has a solution μ. We first define a selection function ν that assigns to each clause one of its literals. We then show that any literal that is selected in one clause is also selected in any other clause where it appears. Let $\nu(C_j)$ be the unique literal p_i where $\mu(c_j) = {p_i}^{C_j}$ (since μ respects labeling constraints $\mu(c_j)$ has this form). Because of the constraint $c_j \rightarrow_{m(n-j+1)+1} {p_i}^{c_j}$ it follows that $\mu({p_i}^{c_j}) = p_1^1$.

Now assume that p_i also occurs in clause $C_{j'}$. The form of Q enforces that the images of ${p_i}^{c_j}$ and ${p_i}^{c_{j'}}$ have a common descendant. Hence we have also $\mu({p_i}^{c_{j'}}) = p_1^1$. Because of the constraint $c_{j'} \rightarrow_{m(n-j'+1)+1} {p_i}^{c_{j'}}$ it follows easily that $\mu(c_{j'}) = {p_i}^{C_{j'}}$. We have seen that ν defines a truth value assignment on $\{p_1, \ldots, p_m\}$ that selects exactly one literal from each clause. Hence ν solves $\{C_1, \ldots, C_n\}$. □

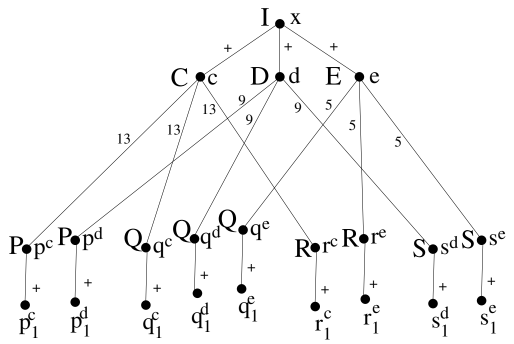
Corollary 10.2 *The problem if a D-D (resp. D-G, G-G) evaluation problem has a solution is NP-complete.*

Proof. Clearly the problem is in NP since we may guess a variable assignment and check in polynomial-time it the mapping is a solution. NP-hardness is an obvious consequence of Lemma 10.1. □

Corollary 10.3 *The problem if a T-T evaluation problem with value comparison has a solution is NP-complete.*

Proof. Reconsider the DAG-query and the tree database depicted in Fig. 3. In the proof of Lemma 10.1, the DAG-structure of the query was just used to guarantee that the two query nodes with label P (resp. R, S) are mapped to the same branch of the subtree of the database starting at node 1 (resp. 3, 4), and the three query nodes with label Q are mapped to the same branch of the subtree of the database starting at node 2. This observation, then, was used to show that the selection of an atom from each clause that comes with each solution defines a truth value assignment (atoms do not receive contradictory values).

When we allow for value comparisons in queries, the same effect can be achieved without introducing common descendants for query nodes. Assuming that all database leaves $f_{*,1}$ (resp. $f_{*,0}$) have value 1 (resp. 0) we may use the tree query



with value comparisons $V(p_1^c) = V(p_1^d)$, $V(q_1^c) = V(q_1^d) = V(q_1^e)$, $V(r_1^c) = V(r_1^e)$, and $V(s_1^d) = V(s_1^e)$. The value comparisons obviously guarantee that each solution of the evaluation problem defines a truth value assignment. An obvious generalization of this argument to the general encoding technique for 1-in-3 problems over positive literals given above shows that these problems can be encoded as T-T evaluation problems with value comparison using a polynomial translation. This shows that solvability of T-T evaluation problems with value comparisons is NP-hard. Clearly the problem is in NP. □

11 CAAs and Constraint Networks

In this section we investigate the close relationship between query evaluation problems and constraint satisfaction problems as discussed, e.g., in [31, 40, 32, 39]. We first have to recall some basic definitions.

Definition 11.1 Let $\mathcal{D}$ be a relational structure with finite domain D. A *binary constraint* has the form $r(x, y)$ where x and y are distinct variables and r is a relation symbol with a fixed interpretation in $\mathcal{D}$. A *constraint satisfaction problem* (over $\mathcal{D}$) is a tuple (Y, Dom, BC) where Y is a finite set of variables, Dom is a function that assigns to each $x \in Y$ a finite subset $Dom(x)$ of D, and BC is a set of binary constraints with variables in Y. A constraint satisfaction problem (Y, Dom, BC) is *non-trivial* if $Dom(x) \neq \emptyset$ for all $x \in Y$. A *solution* of a constraint satisfaction problem (Y, Dom, BC) is an assignment μ that maps each variable $x \in Y$ to an element $\mu(x) \in Dom(x)$ such that all formulae of BC become true in $\mathcal{D}$ under assignment μ. A constraint satisfaction problem (BC, Dom) is *consistent* if it has at least one solution.

Definition 11.2 A constraint satisfaction problem (Y, Dom, BC) is *arc-consistent* if for each $r(x, y) \in BC$ the following conditions hold:

1. for each $d \in Dom(x)$ there exists $e \in Dom(y)$ such that $r(d, e)$ holds in $\mathcal{D}$, and
2. for each $e \in Dom(y)$ there exists $d \in Dom(x)$ such that $r(d, e)$ holds in $\mathcal{D}$.

It is well-known that in general arc-consistency does not imply consistency. A constraint satisfaction problem (Y, Dom', BC) is a *refinement* of (Y, Dom, BC) if $Dom'(x) \subseteq Dom(x)$ for all $x \in Y$. For each constraint satisfaction problem (Y, Dom, BC) there exists a unique refinement (Y, Dom', BC) that is arc-consistent and maximal in the sense that every other arc-consistent refinement of (Y, Dom, BC) is a refinement of (Y, Dom', BC). This problem will be called the *maximal arc-consistent refinement* of (Y, Dom, BC). There exist various "arc-consistency algorithms" that compute the maximal arc-consistent refinement of a given constraint satisfaction problem (Y, Dom, BC) ([39, 32]). The algorithm in [39] needs time $O(e \cdot n^2)$ where e is the number of constraints in BC and n is the maximal cardinality of a domain $Dom(x)$.

11.1 Using arc-consistency algorithms for computing complete answer aggregates

We now give a straightforward translation of evaluation problems into constraint satisfaction problems. Subsequently we characterize the cases where arc-consistency algorithms can be used for computing complete answer aggregates. In the sequel, for a database $\mathcal{D}$ and a node label A, let D_A denote the set of all nodes of $\mathcal{D}$ with label A.

Definition 11.3 To each evaluation problem $(Q, X_Q, \mathcal{D})$ we assign the constraint satisfaction problem (X_Q, Dom, BC) where BC contains all binary constraints of Q (interpreted in $\mathcal{D}$) and where $Dom(x)$ is the intersection of the sets D_A where Q has a labeling constraint $A(x)$ if there is any such constraint, and where $Dom(x) := D$ otherwise. The maximal arc-consistent refinement of (X, Dom, BC) is called the *maximal arc-consistent problem defined by* $(Q, X_Q, \mathcal{D})$.

Lemma 11.4 *Let* (X_Q, Dom, BC) *be the maximal arc-consistent problem defined by* $(Q, X_Q, \mathcal{D})$. *Each solution of* (X_Q, Dom, BC) *is a solution of* $(Q, X_Q, \mathcal{D})$ *and vice versa.*

Proof (Sketch). Arc-consistency algorithms only remove a node d from the domain of a variable x if x is not mapped to d under any solution. It easily follows that each answer to Q in $\mathcal{D}$ is a solution of (X_Q, Dom, BC). Since each solution of (X_Q, Dom, BC) satisfies all labeling constraints and all edge constraints of Q it always represents an answer to Q in $\mathcal{D}$. $\square$

In order to characterize situations where arc-consistency algorithms can be used to compute the complete answer aggregate we need a translation of arc-consistent constraint problems into answer aggregates.

Definition 11.5 Let (Y, Dom, BC) be an arc-consistent constraint satisfaction problem over the database $\mathcal{D}$. The *realization* of (Y, Dom, BC) is the answer aggregate (Dom, Π) where the mapping Π assigns to each binary constraint $r(x, y) \in BC$ the set of links $\{(d, e) \in Dom(x) \times Dom(y) \mid r(d, e) \text{ holds in } \mathcal{D}\}$.

The following lemma is straightforward.

Lemma 11.6 *In the situation of Definition 11.5, let* (Dom, Π) *denote the realization of* (Y, Dom, BC). *Each instantiation of* (Dom, Π) *is a solution of* (Y, Dom, BC) *and vice versa.*

We can now show that for evaluation problems with tree queries arc-consistency algorithms in fact yield the complete answer aggregate.

Lemma 11.7 *Let* $(Q, X_Q, \mathcal{D})$ *be an evaluation problem with a tree query* Q. *Let* (X_Q, Dom, BC) *be the maximal arc-consistent problem defined by* $(Q, X_Q, \mathcal{D})$. *Then the realization* (Dom, Π) *of* (X_Q, Dom, BC) *is the complete answer aggregate for* $(Q, X_Q, \mathcal{D})$.

Proof. Obviously (Dom, Π) is an answer aggregate for $(Q, X_Q, \mathcal{D})$. Lemma 11.4 and 11.6 show that each answer to Q is an instantiation of (Dom, Π) and vice versa. Hence it remains to show that (Dom, Π) does not have a redundant node or a redundant link. We first show that (Dom, Π) does not have a redundant link. Assume there exists a link between occurrences of nodes d_x and d_y in $Dom(x)$ and $Dom(y)$ respectively. We may assume that y is a child of x in the tree associated with query Q. If $r(x, y)$ denotes the unique binary relation between x and y in Q, the relation $r(d_x, d_y)$ holds in $\mathcal{D}$. If x is not the root of Q and if x_1 denotes the parent of x, then arc-consistency of (X_Q, Dom, BC) guarantees that $Dom(x_1)$ contains a node instance d_{x_1} such that $r'(d_{x_1}, d_x)$ holds in $\mathcal{D}$, where $r'(x_1, x)$ denotes the unique binary relation between x_1 and x in Q. Proceeding in this way we successively obtain an instantiation value for all ancestors of x in Q. The given partial instantiation of the answer aggregate can be completed in top-down manner to a full instantiation of (Dom, Π): given a node u with instantiation d_u and a child v that does not have an instantiation value yet, arc consistency guarantees that there exists an occurrence $d_v \in Dom(v)$ in $Dom(v)$ such that d_u and d_v satisfy the unique binary constraint imposed on u and v in Q. Obviously, the given link participates to the given instantiation. We have seen that (Dom, Π) does not have a redundant link. The proof that (Dom, Π) does not have a redundant node is a trivial variant. $\square$

Let us remark that the proof shows that Lemma 11.7 is closely related to general results on backtrack-free search in "tree-structured constraint graphs" described in [20]. The following example shows that the result of Lemma 11.7 cannot be generalized to more general classes of queries.

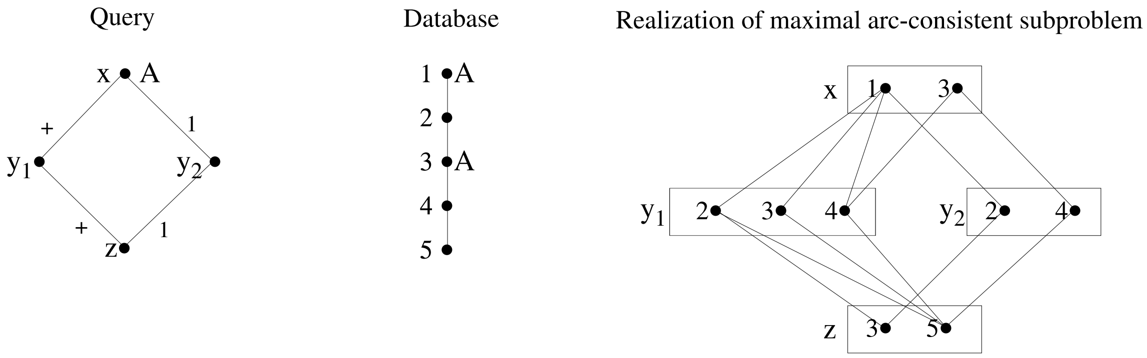


Figure 5: Arc-consistent constraint network with redundant nodes

Example 11.8 The realization of the maximal arc-consistent problem defined by a D-S evaluation problem can have redundant nodes. Given the query and database in Fig. 5, since x has to be mapped to a node with label A, only nodes 1 and 3 are possible images for x. Then, as a consequence of arc-consistency, only nodes 2 and 4 (resp. 3 and 5) are possible images for y_2 (resp. z). By arc-consistency, values 1 and 5 can be excluded for y_1. The subproblem reached now is arc-consistent. Its realization contains the redundant node 3 in the domain of y_1. In other examples, links of the realization may be redundant even if all nodes are non-redundant.

We summarize the previous results. Lines S and T follow from Lemma 11.7. Entry D-S follows from Example 11.8. All other entries are consequences.

		Database			
		S	T	D	G
	S	yes	yes	yes	yes
Query	T	yes	yes	yes	yes
	D	no	no	no	no
	G	-	-	-	no

Remark 11.9 Even in the situation where the maximal arc-consistent constraint satisfaction problems defined by an evaluation problem may have redundant nodes, there are cases where arc-consistency algorithms can be used in an indirect way for computing complete answer aggregates. Let $\mathcal{C}$ be a class of constraint satisfaction problems closed under refinement such that arc-consistency plus non-triviality implies consistency, for all problems $(Y, Dom, BC) \in \mathcal{C}$. Let $\mathcal{E}$ denote a class of query evaluation problems where the translation of an evaluation problem always yields a constraint satisfaction problem in $\mathcal{C}$. Given an arc-consistency algorithm we can first compute the maximal arc-consistent problem (X_Q, Dom, BC) defined by $(Q, X_Q, \mathcal{D})$. We may then check if the realization (Dom, Π) contains redundant nodes in the following way: for each $x \in X_Q$ and each node $d \in Dom(x)$ we consider the refined problem (X_Q, Dom', BC) where $Dom'(x) := \{d\}$ and $Dom'(y) := Dom(y)$ for $y \neq x$. We use the given arc-consistency algorithm to compute the maximal arc-consistent refinement (X_Q, Dom'', BC) of (X_Q, Dom', BC). Obviously, if (X_Q, Dom'', BC) is trivial, then node d cannot contribute to a solution of (X_Q, Dom', BC), which means that node d is redundant in (Dom, Π). Conversely, since arc-consistency and non-triviality of (X_Q, Dom'', BC) guarantee consistency, there exists a solution of (X_Q, Dom', BC) that assigns d to x, which shows that d is non-redundant in (Dom, Π). In this way we may check for each $d \in Dom(x)$ redundancy. Redundant nodes are eliminated (note that this does not lead to new redundant nodes). The process is iterated until we reach an arc-consistent subproblem (X_Q, Dom_1, BC) without redundant nodes. We may now check for each link of its realization (Dom_1, Π_1) if it contributes to a solution. Here we again use the arc-consistency algorithm,

restricting now the domains of two variables to singleton sets.

Let (Dom_1, Π_2) be the variant of (Dom_1, Π_1) where we erase redundant links. Now (Dom_1, Π_2) is the complete answer aggregate for $(Q, X_Q \mathcal{D})$. Assuming that we have an arc-consistency algorithm running in time $O(e \cdot n^2)$ (with e being the number of edges in the query Q) the total time complexity of the above algorithm is bounded by the number of edges of (X_Q, Dom, BC) times $O(e \cdot n^2)$. This leads to bound $O(q \cdot e \cdot n^3 \cdot a)$.

Summing up, we have seen that an arc-consistency algorithm can be used for computing complete answer aggregates in time $O(q \cdot e \cdot n^3 \cdot a)$, provided that for the constraint satisfaction problems obtained via translation arc-consistency and non-triviality implies consistency. We shall now discuss for which evaluation problems this strategy can be used.

11.2 When does arc-consistency and non-triviality imply consistency?

Let $(Q, X_Q, \mathcal{D})$ be an evaluation problem, let (X_Q, Dom, BC) be the maximal arc-consistent problem defined by $(Q, X_Q, \mathcal{D})$. We ask for which type of evaluation problems non-triviality of (X_Q, Dom, BC) guarantees that Q has at least one answer in $\mathcal{D}$. First we treat the case of sequence databases.

Definition 11.10 A relation R on a sequence database $\mathcal{D} = (N, E, L_N, L_E, \Lambda_L, \Lambda_E)$ is *directed* iff for all $R(d, e)$ the pair (d, e) is in the transitive closure $<$ of E. A directed relation R is *tolerant* iff for all $d \leq d' < e' \leq e$ such that $R(d, e)$ and $R(d', e')$ we have $R(d, e')$.

Note that children relationship $\rightarrow_1$, descendant relationship $\rightarrow_+$, as well as labeled children relationship $\rightarrow_f$ are tolerant.

Lemma 11.11 *Let $\mathcal{D} = (N, E, L_N, L_E, \Lambda_L, \Lambda_E)$ be a sequence database. Assume that the constraint satisfaction problem (Y, Dom, BC) over $\mathcal{D}$ is arc-consistent. Assume that all binary relations occurring in BC are tolerant. Then (Y, Dom, BC) is consistent iff it is non-trivial.*

Proof. Clearly triviality of (Y, Dom, BC) implies inconsistency. Assume now that $Dom(y) \neq \emptyset$ for all $x \in Y$. For each $x \in Y$, let d_x denote the minimal element of $Dom(x)$ with respect to the transitive closure $<$ of E. We show that $\mu : x \mapsto d_x$ defines a solution of (Y, Dom, BC).

Let $R(x, y)$ be any binary constraint in BC. Since (Y, Dom, BC) is arc-consistent, there exists some $d'_y \in D_y$ such that $R(d_x, d'_y)$. Since R is directed we have $d_x < d'_y$. Similarly there exists $d'_x \in D_x$ such that $d'_x < d_y$ and $R(d'_x, d_y)$. By choice of d_x and d_y we have $d_x \leq d'_x$ as well as $d_y \leq d'_y$. Since R is tolerant we have $R(d_x, d_y)$. We have seen that μ satisfies all relations in BC. Hence μ is a solution of (Y, Dom, BC). $\square$

Corollary 11.12 *There exists an algorithm that computes the complete answer aggregate for evaluation problems over sequence databases in time $O(q \cdot e \cdot n^3 \cdot a)$ (with e being the number of edges in Q).*

Proof. This follows from Remark 11.9 and Lemma 11.11 since children relationship $\rightarrow_1$, descendant relationship $\rightarrow_+$, labeled children relationship $\rightarrow_f$ are tolerant. $\square$

From Lemma 6.1 and Remark 11.9 it follows already that the result of Lemma 11.11 cannot be extended to the case of tree databases. We add explicit counterexamples.

Example 11.13 The maximal arc-consistent constraint problem defined by a D-T evaluation problem can be non-trivial but inconsistent. For the query and database in Fig. 6, label constraints show that only nodes 2 and 3 are possible images for y_1. By arc-consistency, only nodes 1 and 2 are possible images for variable x. The only node with label C in depth 5 below 2 (resp. 3) is 7 (resp. 8). Each of the nodes 7 and 8 has exactly one ancestor with label B.

The realization of the resulting arc-consistent problem on the right-hand side in Fig. 6 shows that there is no solution.

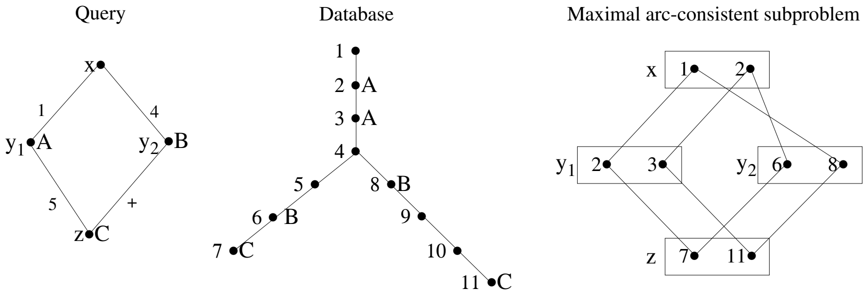


Figure 6: Arc-consistent constraint network having no solution

The following table summarizes the results. Entries in the first column follow from Lemma 11.11. Entry D-T follows from Example 11.13. The remaining entries in columns D and G are consequences. Entry T-G follows from Lemma 11.7. The remaining entries in lines S and T are consequences.

		Database			
		S	T	D	G
Query	S	yes	yes	yes	yes
	T	yes	yes	yes	yes
	D	yes	no	no	no
	G	-	-	-	no