Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Compact representation for answer sets of *n*-ary regular queries

Kazuhiro Inaba [a,*], Haruo Hosoya [b]

[a] *National Institute of Informatics, 2-1-1, Hitotsubashi, Chiyoda-ku, Tokyo, Japan*

[b] *The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan*

ARTICLE INFO

Keywords:
Tree automata
Data structure
XML

ABSTRACT

An n-ary query over trees takes an input tree t and returns a set of n-tuples of the nodes of t. In this paper, a compact data structure is introduced for representing the answer sets of n-ary queries defined by tree automata. Despite that the number of the elements of the answer set can be as large as $|t|^n$, our representation allows storing the set using only $O(|t|)$ space. Several basic operations on the sets are shown to be efficiently executable on the representation.

1. Introduction

The finite state automaton is a well-known model for representing properties for trees and strings. The class of queries definable by finite state automata is called *regular* and is widely used both in theory and in practice. A number of query formalisms are shown to be equivalent or subsumed by regular queries. Examples of such formalisms include, regular expression pattern [1], monadic second-order logic [2], μ-calculus [3], Core XPath [4], monadic Datalog [5], Boolean attribute grammar [6], etc.

In this paper, we are interested in the space complexity of the n-ary queries defined by tree automata. An n-ary query over trees takes an input tree t and returns a set of n-tuples of the nodes of t. The number of elements in the answer set of an n-ary query may be as large as $|t|^n$ where $|t|$ is the number of the nodes of t. Also, usually, storing a set of $|t|^n$ elements requires at least $c|t|^n$ space, where c is the space required to store a single element (in this case, one n-tuple of nodes). The $O(|t|^n)$ space consumption is unavoidable if the elements are chosen in a perfectly random manner; it is a well-known consequence from information theory. Note, however, we are interested in more practical, less random queries. Queries defined by tree automata have much more structure than random ones. By exploiting the structural characteristics of regular queries, we can represent the answer sets in some *compressed* form.

Let us explain the idea by an example. Consider the regular query "select all pair of nodes (x, y) such that x is in the left subtree of the root node and y is in the right subtree of the root node" with the input tree t as in Fig. 1. Then the answer set consists of nine elements: $\{(v_1, v_4), (v_2, v_4), (v_3, v_4), (v_1, v_5), (v_2, v_5), (v_3, v_5), (v_1, v_6), (v_2, v_6), (v_3, v_6)\}$. Obviously, if an input tree has n nodes both in the left and the right subtrees, the size of the answer set will be n^2, which is quadratic in the number $2n + 1$ of the nodes. Our approach for avoiding the quadratic blow-up is to represent the answer set by a symbolic *expression*, instead of computing the concrete list of elements. For this example, we can represent the answer set by the expression $\{v_1, v_2, v_3\} \times \{v_4, v_5, v_6\}$ where $\times$ denotes the product of two sets. Counting the number of variables v_i and the operator, the length of the expression is 7 instead of 9. Analogously, for the general case with n nodes in both the left and the right subtrees, the answer set can be represented by an expression of length $2n + 1$, which consumes only linear space with respect to the size of the input tree.

* Corresponding author. Tel.: +81 90 5822 4094.
E-mail addresses: kinaba@nii.ac.jp (K. Inaba), hahosoya@is.s.u-tokyo.ac.jp (H. Hosoya).

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.05.026

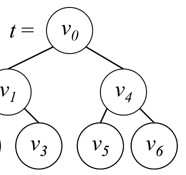

Fig. 1. Sample input.

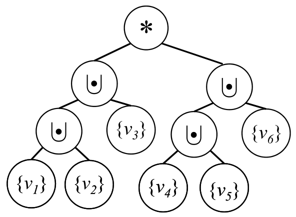

Fig. 2. Query result in expression-based representation.

The contribution of our work is in establishing the expression-based compact representation as illustrated above. In fact, only two operators – $\uplus$ (*disjoint union*) and $*$ (a slight variant of *product*) – are necessary for achieving the linear-size representation of the answer sets of regular queries. We show that for any fixed n-ary regular query and an input tree t, the answer set can always be represented by an expression on $\uplus$ and $*$ with every leaf expression being a singleton set of an input node. By sharing common sub-expressions, such an expression can always be represented by a dag of size at most $4 \cdot 3^n|\delta_{\mathcal{A}}||t|$, where $|\delta_{\mathcal{A}}|$ is the size of the deterministic automaton representing the query. That is, regardless of the arity n of the query, the data complexity with respect to the size $|t|$ of the input is always linear! As an instance, Fig. 2 shows the expression-dag (just a tree in this case) representation of the query result of the previous example. The factor complexity 3^n is sufficiently low for queries with small n, such as binary or ternary queries, which are the most common cases in practice; after all, it is quite rare to run, say, a 100-ary query.

Our dag-based representation is named *SRED (Set Representation by Expression Dags)*, which enjoys good time complexity as well as size-efficiency. The SRED representation of the answer set can always be computed from the input tree t in time $O(3^n|\delta_{\mathcal{A}}||t|)$, regardless how large the actual answer set is. Also, evaluation (or we could say, *decompression*) of a SRED to yield the concrete list of answer tuples can be done in time $O(n^2|a|)$, where $|a|$ is the number of the answers. By combining these two steps, we obtain an algorithm for regular queries in the optimal data complexity $O(|t| + |a|)$. More than that, on SRED, we can carry out the following important operations *without decompressing* it: (1) Selection: for an answer set s, the SRED representation of the set $s_{[i:u]} = \{(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \mid (v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n) \in s\}$ can be computed in time $O(3^n h|\delta_{\mathcal{A}}|)$, where h is the height of the input tree for binary trees and is the height times $\log|t|$ for unranked trees, (2) Projection: the set $s_{@i} = \{v_i \mid (v_1, \ldots, v_n) \in s\}$ can be computed in time $O(3^n h|\delta_{\mathcal{A}}||s_{@i}|)$. Besides the expression-based representation, another key idea of SRED is to remember for every sub-expression the least common ancestor of the nodes contained in the set represented by the sub-expression. The information allows locating the leaf expressions containing each input node in time proportional only to the height of the expression-dag.

Related work. SRED has much similarity to the Complete Answer Aggregate (CAA) introduced by Meuss et al. [7] as a compact representation of answer sets of queries. The size of a CAA is $O(h|t|)$ which is competitive to our $O(|t|)$. CAA is also suitable for applying several operations such as membership testing. The main advantage of our work is that it supports arbitrary regular queries, which is strictly more expressive than the query language used in [7]. Though an attempt to represent the answer sets of regular queries with CAA is given by Filiot and Tison [8] through a decomposition of queries, the space complexity is $O(|t|^{d_\phi})$ for some constant d_ϕ depending on the query, which grows to n in the worst case. Furthermore, precise complexity of operations such as selection or projection for CAA was not estimated.

An algorithm (FFG algorithm) for answering regular n-ary queries in the optimal time complexity $O(|t|+|a|)$ is shown by Flum et al. [9]. Since no compact data structure was used in their work, the FFG algorithm requires $O(|a|)$ space to be carried out. In fact, our algorithm can be regarded as a space-efficient variant of the FFG algorithm. The expression dag generated in our algorithm precisely corresponds to the set operations executed in the FFG algorithm. On the other hand, the class of queries that the FFG algorithm can be applied to is more general than our algorithm. The FFG algorithm can also be used for querying n-tuples of *sets of* nodes of *graphs* that have a tree decomposition, while our algorithm only supports queries for n-tuples of nodes of trees. It will be future work to determine whether our compact representation of the answer sets can be extended to more general classes of query.

Another related area of research is becoming hot recently, namely, *linear-delay enumeration* of MSO query results [10,11]. In their algorithms, the input tree of a query is first converted to an intermediate data structure that allows linear-delay enumeration of the query results. Since the intermediate data structure is of linear size with respect to the input tree, it can also be used as a compact representation of the answer set. Compared to our SRED representation, however, their data structures are concentrated only for linear-delay enumeration and other operations such as selection or projection are not supported.

Outline. The paper is organized as follows. In Section 2, we introduce basic notations on trees and tree languages. Section 3 presents a simple but inefficient algorithm for executing n-ary queries on binary trees, as the basis of our main algorithm. Then, in Section 4, we give our main results. We introduce the SRED data structure as a compact representation of set of tuples, and show that just by using SRED, the previous naïve algorithm can be turned into one that efficiently produces a compact answer-set representation. Section 5 shows an application to XML processing. Section 6 concludes.

2. Preliminaries

In this paper, we mainly consider *binary trees*, in which every node has either zero or two children. Generalization to the trees with other arity is briefly mentioned in the end of Section 4. Let Σ be a finite alphabet that is a disjoint union of two alphabets $\Sigma^{(0)}$ and $\Sigma^{(2)}$. A *binary tree* (or simply, a *tree*) over Σ is a tuple $t = (V_t, label_t, lt_t, rt_t, root_t)$, where V_t is the disjoint union $V_t^{(0)} \uplus V_t^{(2)}$ of finite sets of *nodes*, $label_t : V_t^{(0)} \to \Sigma^{(0)} \uplus V_t^{(2)} \to \Sigma^{(2)}$ is the *label* function, $lt_t, rt_t : V_t^{(2)} \to V_t$ is the *left*- and *right-child* function respectively, and $root_t \in V_t$ is the *root* node. We call the nodes in $V_t^{(0)}$ *leaf nodes*, and the nodes in $V_t^{(2)}$ *branching nodes*. We require a tree to satisfy the following conditions: (1) rooted: there is no node $v \in V_t$ such that $lt_t(v) = root_t$ or $rt_t(v) = root_t$, (2) acyclic: there is no node $v \in V_t$ that is reachable from itself by finite applications of lt_t and rt_t, and (3) tree-formed: for any non-root node $v \in V_t \setminus \{root_t\}$, there exists a unique node u called the *parent* of v such that $lt_t(u) = v \vee rt_t(u) = v$. A structure only satisfying (1) and (2) is called a *dag*. For $v_1, v_2 \in V_t$, the binary order relation $v_1 \leq_t v_2$ is defined to hold if and only if v_2 is reachable from v_1 by zero or finitely many applications of lt_t and rt_t. We usually omit the subscript $_t$ if clear from the context. By $|t|$ we denote the number $|V_t|$ of the nodes. We use the notation $\mathsf{a}\langle v_1, v_2\rangle$ to denote a node v such that $label_t(v) = \mathsf{a}$, $lt_t(v) = v_1$, and $rt_t(v) = v_2$.

For a tree t, we assume that each node $v \in V_t$ can be stored on memory in constant space independent from $|t|$. In practice, this implies the assumption that the tree t fits in the address space of the computer and each node can be represented by a single pointer. We also assume that the operations *label*, *lt*, *rt*, and $\leq$ can be executed in constant time. In particular, we can test the relation $\leq$ in constant time by, e.g., the preorder/postorder numbering [12]. Again by the assumption that $|t|$ fits in the address space, preorder and postorder numbers can be stored in constant space.

A *tree language* over Σ is a set of trees over Σ. By T_Σ, we denote the set of all trees over Σ. An important class of tree languages is defined in terms of tree automata. A *(bottom-up deterministic) tree automaton* over Σ is a tuple $\mathcal{A} = (Q_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A})$, where $Q_\mathcal{A}$ is the set of states, $\delta_\mathcal{A} : (\Sigma^{(0)} \cup (\Sigma^{(2)} \times Q_\mathcal{A} \times Q_\mathcal{A})) \to Q_\mathcal{A}$ is the transition function, and $F_\mathcal{A} \subseteq Q_\mathcal{A}$ is the set of accepting states. The subscript $_\mathcal{A}$ is omitted if clear from the context. A *run* of a tree automaton $\mathcal{A}$ on the input tree t is the unique function $\rho : V_t \to Q_\mathcal{A}$ such that $\rho(v) = \delta_\mathcal{A}(label_t(v))$ if $label_t(v) \in \Sigma^{(0)}$ and $\rho(v) = \delta_\mathcal{A}\big(label_t(v), \rho(lt_t(v)), \rho(rt_t(v))\big)$ if $label_t(v) \in \Sigma^{(2)}$. The automaton *accepts* t if and only if $\rho(root_t) \in F_\mathcal{A}$. By $\mathcal{L}(\mathcal{A})$, we denote the set of trees accepted by $\mathcal{A}$. A tree language is said to be *regular* if it is equal to $\mathcal{L}(\mathcal{A})$ for some tree automaton $\mathcal{A}$.

3. N-ary regular tree queries

As a basis of our algorithm for computing the compact representation of answer sets, we first explain a basic bottom-up algorithm for regular queries with $O(|t|^{n+1})$ time complexity, which has already been known in the literature. Our new algorithm is obtained by changing the data structure used in the algorithm, as explained later in Section 4.

An n-ary *query* for trees over Σ is a function ψ that maps each tree $t \in T_\Sigma$ to a set of n-tuples of its nodes. Let $\mathbb{B} = \{0, 1\}$, $\Sigma_n^{(0)} = \Sigma^{(0)} \times \mathbb{B}^n$, $\Sigma_n^{(2)} = \Sigma^{(2)} \times \mathbb{B}^n$, and $\Sigma_n = \Sigma_n^{(0)} \uplus \Sigma_n^{(2)}$. For a tree language $L \subseteq \Sigma_n$, an n-ary *query defined by* L is the function $\psi_L(t) = \{(v_1, \dots, v_n) \mid mark(t, v_1, \dots, v_n) \in L\}$ where $mark(t, v_1, \dots, v_n)$ is a tree $m = (V_t, label_m, lt_t, rt_t, root_t)$ with $label_m(v) = (label_t(v), b_1 \cdots b_n)$, where $b_i = 1$ if $v = v_i$ and 0 otherwise. Intuitively, a query defined by a language L selects a tuple $(v_1, \dots, v_n)$ if and only if L contains a tree obtained by marking each selected node v_i with 1.[1] A query defined by a regular language L is called a *regular query*. In the rest of the paper, we assume the regular language L to be given as a tree automaton $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$. Nevertheless, our algorithm can be applied, without changing the data complexity, to many other query formalisms as long as they define regular languages by first compiling them into tree automata and then running the algorithm.

The most naïve algorithm for a regular n-ary query is, to try all possible markings. Given an automaton $\mathcal{A}$ over Σ_n and a tree t, for all $(v_1, \dots, v_n) \in V_t^n$ we generate the marked tree $mark(t, v_1, \dots, v_n)$ and test whether it is accepted by $\mathcal{A}$. If it is, $(v_1, \dots, v_n)$ is an answer and hence we output it. This algorithm takes $O(|t|^{n+1})$ time, because computing each run of $\mathcal{A}$ takes $O(|t|)$ time and we try $|t|^n$ runs in total.

Another approach is to try all marking parallelly by a single bottom-up run. The following recursive procedure QUERY-RUN$_\mathcal{A}$ takes a node v of t and computes a table containing the result of the parallel marking run.

[1] In general, L may contain "ill-marked" trees that have two or more nodes marked as $b_i = 1$ for the same i and hence can never be in the range of *mark*. Such trees, however, are simply ignored and have no effect on the definition of ψ_L.

```
QUERY-RUN$_{\mathcal{A}}(v)$
1:  $r \leftarrow$ new 2-dimensional array of size $|Q_{\mathcal{A}}| \times 2^n$ with each element initialized to $\emptyset$
2:  if $label(v) \in \Sigma^{(0)}$ then
3:      for each $((label(v), b_0) \mapsto q_0) \in \delta_{\mathcal{A}}$ do
4:          $r[q_0, b_0] \leftarrow singleton(v, b_0)$
5:  else if $label(v) \in \Sigma^{(2)}$ then
6:      $r_1 \leftarrow$ QUERY-RUN$_{\mathcal{A}}(lt(v))$
7:      $r_2 \leftarrow$ QUERY-RUN$_{\mathcal{A}}(rt(v))$
8:      for each $((label(v), b_0), q_1, q_2 \mapsto q_0) \in \delta_{\mathcal{A}}$ do
9:          for each disjoint $b_0, b_1, b_2$ in $00\ldots00$ to $11\ldots11$ do
10:             $r[q_0, b_0|b_1|b_2] \leftarrow r[q_0, b_0|b_1|b_2] \uplus singleton(v, b_0) * r_1[q_1, b_1] * r_2[q_2, b_2]$
11: return $r$
```

By $singleton(v, \beta_1 \cdots \beta_n)$ we denote the singleton set $\{(u_1, \ldots, u_n)\}$, where $u_i = v$ if $\beta_i = 1$ and $u_i = \perp$ if $\beta_i = 0$. Here, $\perp$ is a special symbol not contained in V_t. In line 9, **for each disjoint** iterates over pairs of the form $(b_1 = \beta_{11} \cdots \beta_{1n}, b_2 = \beta_{21} \cdots \beta_{2n}) \in (\mathcal{B}^n)^2$ such that for all $1 \le i \le n$, at most one of $\{\beta_{0i}, \beta_{1i}, \beta_{2i}\}$ is 1, with $\beta_{01} \cdots \beta_{0n} = b_0$. Note that b_0 is determined by the outer $\delta_{\mathcal{A}}$ loop and fixed during each single inner loop. The operator $|$ is for bitwise-or and $\uplus$ is disjoint union of sets (the operands are indeed disjoint, as explained later). The operator $*$ is a kind of "product" operation that combines two sets of tuples, defined as follows: $S * T = \{(u_1, \ldots, u_n) \mid (s_1, \ldots, s_n) \in S, (t_1, \ldots, t_n) \in T, \forall i : (u_i = s_i \wedge \perp = t_i) \vee (\perp = s_i \wedge u_i = t_i)\}$. For example, $\{(v_1, \perp, \perp), (v_2, \perp, \perp)\} * \{(\perp, \perp, v_3), (\perp, \perp, v_4)\}$ is equal to $\{(v_1, \perp, v_3), (v_1, \perp, v_4), (v_2, \perp, v_3), (v_2, \perp, v_4)\}$. Let us remark that we never take $*$-product of sets that have tuples with non-$\perp$ nodes on the same position, as will be shown in Lemma 1.

Let us explain how the algorithm works. Let $r =$ QUERY-RUN$_{\mathcal{A}}(v)$ for a node $v \in V_t$. For each $q \in Q_{\mathcal{A}}$ and $b = \beta_1 \cdots \beta_n \in \mathbb{B}^n$, $r[q, b]$ is a set of n-tuples over the set $V_t \cup \{\perp\}$. A tuple in $(V_t \cup \{\perp\})^n$ is called a *partial answer* to the query. For example, $(v_1, \perp)$ is a partial answer that selects the node v_1 as the first coordinate and leaves the second coordinate to be selected later. Intuitively, $r[q, b]$ is the set of partial answers α such that, if a tree is marked according to α, then at the node v, the run of the automaton $\mathcal{A}$ reaches the state q. For example, if $(v_1, \perp) \in r[q, b]$, it means that "if the node v_1 is marked as the first component of the answer and no node in the subtree under v is marked as the second component, $\mathcal{A}$ reaches the state q at node v". As an example, let us assume v to be a leaf node labeled $\sigma \in \Sigma^{(0)}$ and $\mathcal{A}$ to define a binary query. Suppose $\delta_{\mathcal{A}}$ has the following four rules: $\delta_{\mathcal{A}}((\sigma, 00)) = q_1$, $\delta_{\mathcal{A}}((\sigma, 01)) = q_2$, $\delta_{\mathcal{A}}((\sigma, 10)) = q_1$, and $\delta_{\mathcal{A}}((\sigma, 11)) = q_2$. Then, the table $r =$ QUERY-RUN$_{\mathcal{A}}(v)$ is:

$$
\begin{array}{llll}
r[q_1, 00] = \{(\perp, \perp)\} & r[q_1, 01] = \emptyset & r[q_1, 10] = \{(v, \perp)\} & r[q_1, 11] = \emptyset \\
r[q_2, 00] = \emptyset & r[q_2, 01] = \{(\perp, v)\} & r[q_2, 10] = \emptyset & r[q_2, 11] = \{(v, v)\}.
\end{array}
$$

The set $r[q_1, 00]$ contains $(\perp, \perp)$ because if we do not select any node below v, the automaton reaches the state q_1. On the other hand, the set $r[q_2, 00]$ is empty, because we cannot reach the state q_2 at node v if we do not select any node. Similarly, $r[q_1, 01]$ is empty, because we cannot reach the state q_1 if we select the second coordinate of the answer. On the other hand, we have $r[q_2, 01] = \{(\perp, v)\}$, because if we choose v as the second coordinate, the automaton reaches the state q_2.

The index b of r called *flag* denotes the already selected coordinates; the i-th coordinate of the elements of $r[q, b]$ is non-$\perp$ if and only if the i-th bit of b is 1. Thus we have the following lemma.

Lemma 1. *Let $r =$ QUERY-RUN$_{\mathcal{A}}(v)$ for some v and $(u_1, \ldots, u_n) \in r[q, \beta_1 \cdots \beta_n]$. For all $1 \le i \le n$, we have ($u_i \in V_t$ and $v \le_t u_i$) if $\beta_i = 1$, and $u_i = \perp$ if $\beta_i = 0$.*

Proof. The proof is by induction on the structure of the tree rooted at v. If v is a leaf node, $r[q, b]$ is either empty (the case $((label(v), b) \mapsto q) \notin \delta_{\mathcal{A}}$) or a singleton set $singleton(v, b)$. The lemma obviously holds for the empty case, and the latter case is also immediate from the definition of $singleton(v, b)$.

If v is a branching node, by the construction of the set $r[q, b]$, the condition $(u_1, \ldots, u_n) \in r[q, b]$ implies that we have $(u_1, \ldots, u_n) \in singleton(v, b_0) * r_1[q_1, b_1] * r_1[q_2, b_2]$ for some $q_1, q_2 \in Q_{\mathcal{A}}$ and disjoint $b_0, b_1, b_2 \in \mathbb{B}^n$ with $b_0|b_1|b_2 = b$. Now, assume that the i-th bit (β_i) of b is 0, which at the same time means that the i-th bits of b_0, b_1, and b_2 is 0. By the definition of *singleton* and the induction hypothesis, the i-th coordinate of each element of $singleton(v, b_0)$, $r_1[q_1, b_1]$, and $r_2[q_1, b_2]$ is $\perp$. Hence, from the definition of $*$, u_i also has to be $\perp$ in this case. Contrarily assume $\beta_i = 1$, which means that exactly one of the i-th bits of b_0, b_1, and b_2 is 1. Then, if we take any three tuples $(s_1, \ldots, s_n) \in singleton(v, b_0)$, $(t_1, \ldots, t_n) \in r_1[q_1, b_1]$, and $(w_1, \ldots, w_n) \in r_2[q_2, b_2]$, exactly one of s_i, t_i, and w_i is non-$\perp$ due to the induction hypothesis. Let us call x the non-$\perp$ node. We have $v \le_t x$, because $s_i = v$, and by induction hypothesis $lt(v) \le_t t_i$ and $rt(v) \le_t w_i$ if they are not $\perp$. The definition of $*$ tells us that u_i is one of such-chosen x, which is non-$\perp$, and $v \le_t x$ as desired. □

The lemma ensures two disjointness properties in the procedure QUERY-RUN$_{\mathcal{A}}$. First, the $*$-product is always taken between the sets with disjoint selected-coordinates. That is, we need to compute $S * T$ only for the sets S, T such that $(\ldots, v_i, \ldots) \in S$ and $(\ldots, u_i, \ldots) \in T$ implies either v_i or u_i is $\perp$. This holds because in line 10 ($*$ occurs only here) of the QUERY-RUN$_{\mathcal{A}}$ algorithm, the flags b_0, b_1, and b_2 are disjoint. Note that, for such a case, we have $|S * T| = |S| \cdot |T|$.

Second, $\uplus$ is indeed taken between disjoint sets. This is because the operands of $\uplus$ (whose only one occurrence is in line 10) are constructed by $*$-product either over different flags or over different states, i.e., the union is of the form $singleton(v, b_0) * r_1[q_1, b_1] * r_2[q_2, b_2] \uplus singleton(v, b'_0) * r_1[q'_1, b'_1] * r_2[q'_2, b'_2]$ where either $(b_0, b_1, b_2) \neq (b'_0, b'_1, b'_2)$ or $(q_1, q_2) \neq (q'_1, q'_2)$. Disjointness in the former case follows from Lemma 1, and in the latter case it follows from the determinism of the automaton $\mathcal{A}$.

The answer set of the query can be calculated from the result of QUERY-RUN$_{\mathcal{A}}$ applied to the root node, namely, $r =$ QUERY-RUN$_{\mathcal{A}}(root_t)$. For each $q \in F_{\mathcal{A}}$, recall that the set $r[q, 1 \cdots 1]$ is the set of tuples such that "if the tree is marked according to the tuple, $\mathcal{A}$ reaches the state q at the root node", which is by definition the answer set.

Theorem 2. $\psi_{\mathcal{L}(\mathcal{A})}(t) = \bigcup_{q \in F_{\mathcal{A}}}$ QUERY-RUN$_{\mathcal{A}}(root_t)[q, 11 \cdots 11]$.

Proof. Let $v_1, \ldots, v_n \in V_t$ be fixed and ρ be the unique run on the tree $mark(t, v_1, \ldots, v_n)$ by $\mathcal{A}$. Let $v \in V_t$. Let $partial(v) = (u_1, \ldots, u_n)$ with $u_i = v_i$ if $v \leq_t v_i$ and otherwise $u_i = \perp$. Let $flags(v) = \beta_1 \cdots \beta_n$ with $\beta_i = 1$ if $v \leq_t v_i$ and otherwise $\beta_i = 0$. We can prove for all v in V_t the following claim:

$$\text{for all } q \in Q_{\mathcal{A}}, partial(v) \in \text{QUERY-RUN}_{\mathcal{A}}(v)[q, flags(v)] \quad \text{if and only if} \quad q = \rho(v).$$

We have $(v_1, \ldots, v_n) \in$ QUERY-RUN$_{\mathcal{A}}(root_t)[q, 11 \cdots 11]$ if and only if $q = \rho(root_t)$, by applying the claim to the root node $v = root_t$. It, together with the definition of $\psi_{\mathcal{L}(\mathcal{A})}$, proves the desired result.

Proof of the claim is done by induction on the structure of the tree rooted at v. Consider the case when v is a leaf. From the leaf-node case of the QUERY-RUN$_{\mathcal{A}}$ procedure, we have QUERY-RUN$_{\mathcal{A}}(v)[q, flags(v)] = singleton(v, flags(v)) = \{partial(v)\}$ when $((label(v), flags(v)) \mapsto q)$ is in $\delta_{\mathcal{A}}$, and otherwise it is empty. This already shows the claim for the leaf case, because the discriminating condition is equivalent to $q = \rho(v)$.

Consider the case when v is a branch node. Let $r_1 =$ QUERY-RUN$_{\mathcal{A}}(lt(v))$ and r_2 be that of $rt(v)$. We first show the "if" direction; assume $q = \rho(v)$. Let $q_1 = \rho(lt(v))$, $q_2 = \rho(rt(v))$, $b_0 = \beta_1 \cdots \beta_n$ where $\beta_i = 1$ iff $v = v_i$, $b_1 = flags(lt(v))$, and $b_2 = flags(rt(v))$. Note that $flags(v) = b_0|b_1|b_2$, and by the assumption $q = \rho(v)$, it must be the case $q = \delta_{\mathcal{A}}((label(v), b_0), q_1, q_2)$; the line 10 of the procedure QUERY-RUN$_{\mathcal{A}}$ is executed in this variable binding. That is, the set QUERY-RUN$_{\mathcal{A}}(v)[q, flags(v)]$ is a superset of $singleton(v, b_0) * r_1[q_1, b_1] * r_2[q_2, b_2]$. By the induction hypothesis, the latter set contains the unique element of the product $singleton(v, b_0) * \{partial(lt(v))\} * \{partial(rt(v))\}$, which is $partial(v)$ as desired. For the "only if" direction, assume $partial(v) \in$ QUERY-RUN$_{\mathcal{A}}(v)[q, flags(v)]$. From the construction of this set in QUERY-RUN$_{\mathcal{A}}$, it implies that for some disjoint $b_0|b_1|b_2 = flags(v)$ and $q_1, q_2 \in Q_{\mathcal{A}}$ with $\delta_{\mathcal{A}}((label(v), b_0), q_1, q_2) = q$, it must be the case $partial(v) \in singleton(v, b_0) * r_1[q_1, b_1] * r_2[q_2, b_2]$. But by Lemma 1, it can only happen when $b_1 = flags(lt(v))$, $b_2 = flags(rt(v))$, $partial(lt(v)) \in r_1[q_1, b_1]$, and $partial(rt(v)) \in r_2[q_2, b_2]$; other entries of r_1 and r_2 cannot generate $partial(v)$ by $*$-product. Now, from the induction hypothesis we obtain $q_1 = \rho(lt(v))$ and $q_2 = \rho(rt(v))$, and therefore, $\rho(v) = q$. $\square$

What is the time complexity of this algorithm? For each node $v \in V_t$, the procedure QUERY-RUN$_{\mathcal{A}}$ is applied exactly once. In other words, the procedure is called $|t|$ times. In the body of the procedure, the case for $\Sigma^{(2)}$ labels is computationally harder; the outer loop requires $|\delta_{\mathcal{A}}|$ iterations, the inner loop for b_1, b_2 requires at most 3^n iterations (for each of n bits we have 3 choices–the bit belongs to either b_1, b_2, or none of the two), and inside the loop, one $\uplus$ operation and two $*$ operations are required. Note that the result of those set operations can be as large as $O(|t|^n)$ in the worst case. As long as we represent such sets as a concrete collection of tuples, the operation $*$ need to enumerate all its output elements. Hence it takes at least $O(|t|^n)$ time. Altogether, the total time complexity is still high: $O(3^n|\delta_{\mathcal{A}}||t|^{n+1})$.

One approach for reducing the complexity is to do some preprocessing before running the algorithm, as proposed by Flum et al. [9]. Their algorithm consists of 3-passes over the input tree; the first two passes detect, for each node, whether or not each entry $r[q, b]$ really needs to be computed. The last pass is essentially the same as QUERY-RUN$_{\mathcal{A}}$, but skipping the computation for "unneeded" entries $r[q, b]$. This optimization leads to the complexity $O(3^n|\delta_{\mathcal{A}}|(|t| + |a|))$, where $|a|$ is the size of the answer set. The complexity of this strategy with respect to the data size seems optimal in some sense; if the size of the input is $|t|$ and the size of the output is $|a|$, even just reading and writing those data already takes $O(|t| + |a|)$ time, doesn't it?

Yes, it is optimal—as long as you write down all the elements of the answer set as the output. In the next section, to avoid the issue, we propose here to use a *compressed representation* of the answer set, whose size can be bounded by $O(|t|)$.

4. SRED: set representation by expression dags

In this section, we propose a novel data structure named SRED for representing the answer sets of n-ary regular queries. The size of SRED is always bounded by the input size $O(|t|)$, regardless how large the actual set it represents is. Just by using the data structure instead of normal sets in the QUERY-RUN$_{\mathcal{A}}$ procedure, we obtain linear running time with respect to $|t|$, as well as a compact representation of the answer set. We first give the formal definition of SRED, then show how easily and efficiently it can be adapted to the QUERY-RUN$_{\mathcal{A}}$ algorithm, and finally, show several important set-operations can be directly applied to SRED.

```
EVAL ($r$)
1: if $r \equiv \mathsf{emp}\langle\rangle$ then
2:     return $\emptyset$
3: else if $r \equiv \mathsf{unit}\langle\rangle$ then
4:     return $\{(\bot, \dots, \bot)\}$
5: else if $r \equiv \mathsf{ne}\langle r'\rangle$ then
6:     return EVAL-NE($r'$)

UNION-AT ($v, r_1, r_2$)
1: if $r_1 \equiv \mathsf{emp}\langle\rangle$ then
2:     return $r_2$
3: else if $r_2 \equiv \mathsf{emp}\langle\rangle$ then
4:     return $r_1$
5: else if $r_1 \equiv \mathsf{ne}\langle r_1'\rangle$ and $r_2 \equiv \mathsf{ne}\langle r_2'\rangle$ then
6:     return $\mathsf{ne}\langle \mathsf{cup}\langle v, r_1', r_2'\rangle\rangle$

SINGLETON-AT ($v, \beta_1 \cdots \beta_n$)
1: if $\beta_1 \cdots \beta_n = 0 \cdots 0$ then
2:     return $\mathsf{unit}\langle\rangle$
3: else
4:     return $\mathsf{ne}\langle \mathsf{sing}\langle v, \beta_1 \cdots \beta_n\rangle\rangle$

EVAL-NE ($r$)
1: if $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$ then
2:     return EVAL-NE($r_1$) $\uplus$ EVAL-NE($r_2$)
3: else if $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$ then
4:     return EVAL-NE($r_1$) $*$ EVAL-NE($r_2$)
5: else if $r \equiv \mathsf{sing}\langle v, b\rangle$ then
6:     return $singleton(v, b)$

PRODUCT-AT ($v, r_1, r_2$)
1: if $r_1 \equiv \mathsf{emp}\langle\rangle$ or $r_2 \equiv \mathsf{emp}\langle\rangle$ then
2:     return $\mathsf{emp}\langle\rangle$
3: else if $r_1 \equiv \mathsf{unit}\langle\rangle$ then
4:     return $r_2$
5: else if $r_2 \equiv \mathsf{unit}\langle\rangle$ then
6:     return $r_1$
7: else if $r_1 \equiv \mathsf{ne}\langle r_1'\rangle$ and $r_2 \equiv \mathsf{ne}\langle r_2'\rangle$ then
8:     return $\mathsf{ne}\langle \mathsf{star}\langle v, r_1', r_2'\rangle\rangle$
```

Fig. 3. Basic operations on SRED.

4.1. Definition

The idea of our compact representation is quite simple. To represent a set s, we use a syntax tree r of an expression that evaluates to s. For example, let r_1 and r_2 be the root nodes of the syntax-tree representations of sets s_1 and s_2 (we write $s_1 = [\![r_1]\!]$ and $s_2 = [\![r_2]\!]$, respectively). Then we denote the set $s_1 \uplus s_2$ by the tree $r = \mathsf{cup}\langle r_1, r_2\rangle$. To denote the set $[\![r_1]\!] \uplus ([\![r_2]\!] * [\![r_3]\!])$, we use $\mathsf{cup}\langle r_1, \mathsf{star}\langle r_2, r_3\rangle\rangle$. Note that, by allowing sharing of subtrees (i.e., using syntax-*dags* instead of syntax-trees, which allows a node like $\mathsf{cup}\langle r_1, r_1\rangle$), each operation can be executed in constant time, because it is just a creation of one new node. Since the algorithm QUERY-RUN$_{\mathcal{A}}$ carries out set operations at most $O(3^n|\delta_{\mathcal{A}}||t|)$ times, under this representation of sets, the running time of QUERY-RUN$_{\mathcal{A}}$ is in $O(3^n|\delta_{\mathcal{A}}||t|)$, and so is the size of the output dag representing the answer set.

Let us formally explain the syntax-dag-based representation, which we call *SRED (Set Representation by Expression Dags)*. An answer set of an n-ary query over a tree t is represented by a dag of the following BNF, for $\beta_1 \cdots \beta_n \in \mathbb{B}^n \setminus \{0 \cdots 0\}$:

$$
\begin{aligned}
ST_{\beta_1\cdots\beta_n} &::= \mathsf{emp}\langle\rangle \mid \mathsf{ne}\langle NST_{\beta_1\cdots\beta_n}\rangle \\
ST_{0\cdots0} &::= \mathsf{emp}\langle\rangle \mid \mathsf{unit}\langle\rangle \\
NST_{\beta_1\cdots\beta_n} &::= \mathsf{cup}\langle v, NST_{\beta_1\cdots\beta_n}, NST_{\beta_1\cdots\beta_n}\rangle \text{ with } v \in V_t \\
&\quad\mid \mathsf{star}\langle v, NST_{\alpha_1\cdots\alpha_n}, NST_{\gamma_1\cdots\gamma_n}\rangle \text{ with } v \in V_t \text{ and } \alpha_i \oplus \gamma_i = \beta_i \\
&\quad\mid \mathsf{sing}\langle v, \beta_1 \cdots \beta_n\rangle \text{ with } v \in V_t
\end{aligned}
$$

where $a \oplus c = b$ if and only if $a \neq c$ and $b = 1$ or $a = b = c = 0$. Note that, for enabling fast navigation, as will be explained later, we record the node $v \in V_t$ at each operator. Also for efficiency, we specially treat the empty set (represented by $\mathsf{emp}\langle\rangle$) and the *unit set* ($\{(\bot, \dots, \bot)\}$, represented by $\mathsf{unit}\langle\rangle$), so that they do not occur at operand positions. For example, $\mathsf{cup}\langle v, \mathsf{emp}\langle\rangle, \mathsf{emp}\langle\rangle\rangle$ is ill-formed because $\mathsf{emp}\langle\rangle$ occurs as operands of cup. We call a node labeled emp, unit, or ne a *set-node*, and a node labeled cup, star, or sing a *neset-node* (*ne* stands for non-empty). For a neset-node $r \in NST_{\beta_1\cdots\beta_n}$, we denote by $dim(r)$ the number of 1s in $\beta_1 \cdots \beta_n$. Note that we have $dim(\mathsf{sing}\langle v, \beta_1 \cdots \beta_n\rangle) \geq 1$, $dim(\mathsf{cup}\langle v, r_1, r_2\rangle) = dim(r_1) = dim(r_2)$, and $dim(\mathsf{star}\langle v, r_1, r_2\rangle) = dim(r_1) + dim(r_2)$.

By avoiding $\mathsf{emp}\langle\rangle$ and $\mathsf{unit}\langle\rangle$ to occur at non-root position, we can evaluate the syntax-dag by a straightforward recursion shown in Fig. 3, in a time complexity proportional to the size of the answer set.

Lemma 3. *Assume the disjoint union $s_1 \uplus s_2$ can be computed in constant time and the product $s_1 * s_2$ can be computed in time $O(n|s_1 * s_2|)$ for $s_1, s_2 \neq \emptyset$. Then for a neset-node r,* EVAL-NE(r) *runs in time $O\big((k+1)n|\text{EVAL-NE}(r)|\big)$ where $k = dim(r) - 1$.*

Proof. Without loss of generality, we assume the disjoint union $s_1 \uplus s_2$ to take one unit computation step, product $s_1 * s_2$ to take $n|s_1 * s_2|$ steps, and $singleton(v, b)$ to take n steps. Under the assumption, we prove by induction that the computation of EVAL-NE(r) takes at most $T(k, r) = 2((2k + 1)n|\text{EVAL-NE}(r)|) - 1$ steps.

If r is a node labeled sing, we have $k \geq 0$ and thus $T(k, r) \geq 2n - 1 \geq n$.

If $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$, by induction hypothesis, $s_1 = \text{EVAL-NE}(r_1)$ and $s_2 = \text{EVAL-NE}(r_2)$ can be computed in time $T(k, r_1) + T(k, r_2) = 2((2k + 1)n|\text{EVAL-NE}(r)|) - 2$ steps (note that $|\text{EVAL-NE}(r)| = |\text{EVAL-NE}(r_1)| + |\text{EVAL-NE}(r_2)|$, because it is disjoint union). Adding one unit computation step for the $\uplus$, the obtained computation steps is equal to $T(k, r)$ as desired.

```
EVAL-NE-1BY1 ($r$, $callback$)
1: if $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$ then
2:     EVAL-NE-1BY1($r_1$, $callback$)
3:     EVAL-NE-1BY1($r_2$, $callback$)
4: else if $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$ then
5:     EVAL-NE-1BY1($r_1$, $\lambda p$.EVAL-NE-1BY1($r_2$, $\lambda q.callback(p * q)$))
6: else if $r \equiv \mathsf{sing}\langle v, b\rangle$ then
7:     $callback(singleton(v, b))$
```

Fig. 4. One-by-one generation of the element tuples of a SRED.

If $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$, by induction hypothesis, $s_1 = \text{EVAL-NE}(r_1)$ and $s_2 = \text{EVAL-NE}(r_2)$ can be computed in $T(k_1, r_1)+T(k_2, r_2)$ steps for some $k_1+k_2+1 = k$. Note that neither s_1 nor s_2 is empty, because return values of EVAL-NE are built up only from *singleton*, $*$, and $\uplus$. Thus, their sizes $|s_1|, |s_2|$ are less than or equal to $|s_1| \cdot |s_2| = |s_1 * s_2| = |\text{EVAL-NE}(r)|$. The total number of steps can be estimated as follows:

$$\begin{aligned} T(k_1, r_1) + T(k_2, r_2) + n|s_1 * s_2| &= 2((2k_1 + 1)n|s_1|) - 1 + 2((2k_2 + 1)n|s_2|) - 1 + n|s_1 * s_2| \\ &\le 2((2k_1 + 2k_2 + 2)n|s_1 * s_2|) - 2 + n|s_1 * s_2| \\ &= 2(2kn|s_1 * s_2|) - 2 + n|s_1 * s_2| \\ &\le 2((2k + 1)n|\text{EVAL-NE}(r)|) - 1 = T(k, r). \quad \square \end{aligned}$$

Theorem 4 (*Evaluation*)**.** *Under the same complexity assumption on $\uplus$ and $*$ as in Lemma 3, for a set-node r, the set* EVAL(r) *can be computed in time $O(n^2|\text{EVAL}(r)|)$.*

Proof. Immediately follows from Lemma 3, because by definition of *dim*, the number k is at most $n - 1$. $\square$

The complexity assumption is satisfied by, for instance, representing the concrete sets by a doubly-linked list of tuples. Disjoint union can be implemented by the list concatenation, and the $*$-product is implemented by a double-loop over two operand sets. Purely functional catenable lists [13] might be an option, in particular when it is desirable to avoid destructive updates. Another interesting implementation is shown in Fig. 4. Instead of constructing the whole set of tuples, it generates each element tuple *one-by-one*; it takes a procedure *callback* and calls it back for each element tuple. It also has the same time complexity as the normal EVAL-NE.

The reader may notice that the evaluation EVAL(r) visits every node below r at least once. Hence, from Theorem 4, we can conclude that the number of nodes below r is $O(n^2|\text{EVAL}(r)|)$. In fact, we are able to give a tighter upper-bound.

Theorem 5 (*Out-Size-Bound*)**.** *For a set-node r, the number of nodes of a dag rooted at r is at most $2n|\text{EVAL}(r)|$.*

Proof. Proof is by induction on structure of a neset-node r, showing that the procedure EVAL-NE is called at most $S(k, r) = 2(k+1)|s| - 1$ times during the computation of $s = \text{EVAL-NE}(r)$, where $k = dim(r) - 1$. If this is proved, the desired bound $2n|\text{EVAL}(r)|$ immediately follows from the fact $k \le n - 1$.

If r is a node labeled sing, the number of procedure calls is 1, which is bounded by $S(k, r) \ge S(0, r) = 1$. If $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$, the number of procedure calls is $S(k, r_1) + S(k, r_2) + 1 = 2(k + 1)|\text{EVAL-NE}(r)| - 2 + 1 = S(k, r)$. If $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$, the number of procedure calls is $S(k_1, r_1) + S(k_2, r_2) + 1$ for some $k_1 + k_2 + 1 = k$. Using the fact that $|\text{EVAL-NE}(r_1)|$ and $|\text{EVAL-NE}(r_2)|$ is no more than $|\text{EVAL-NE}(r)|$, this is bounded by $S(k, r)$. $\square$

Note that this is the worst case estimation. In many cases, particularly when $|\text{EVAL}(r)|$ is large compared to the original input tree of the query, the number of nodes is much smaller than the bound, as will be shown in the next subsection. What we can tell from Theorem 5 is that, *even in the worst case*, we are not losing much. Since it is a set of n-tuples, Representation of the same set in an uncompressed form at least requires $n|\text{EVAL}(r)|$ space, which only differs by a constant-factor from ours.

4.2. N-ary query algorithm using SRED

The basic three operations used in the algorithm QUERY-RUN$_{\mathcal{A}}$ are defined on SRED as in Fig. 3. Note that, to avoid $\mathsf{emp}\langle\rangle$ and $\mathsf{unit}\langle\rangle$ to occurring in operand positions, we deal with the nodes specially. For example, since $\emptyset \cup s = s$ for any set s, when either one of the operands of the UNION-AT operation is an $\mathsf{emp}\langle\rangle$ node, it returns the other operand rather than constructing a new cup node. The correctness of those short-cuts are based on easy set-theoretic equations, and summarized in the following two lemmas.

Lemma 6. *The following four properties hold.*

1. $\text{EVAL}(\mathsf{emp}\langle\rangle) = \emptyset$,
2. $\text{EVAL}(\text{SINGLETON-AT}(v, b)) = singleton(v, b)$,
3. $\text{EVAL}(\text{UNION-AT}(v, r_1, r_2)) = \text{EVAL}(r_1) \uplus \text{EVAL}(r_2)$, *and*
4. $\text{EVAL}(\text{PRODUCT-AT}(v, r_1, r_2)) = \text{EVAL}(r_1) * \text{EVAL}(r_2)$.

Proof. The property 1 and 2 hold by the definition of EVAL. The property 3 follows from $\emptyset \uplus s = s \uplus \emptyset = s$. Note that in the implementation of UNION-AT we have not explicitly considered the case when r_1 or r_2 is $\mathtt{unit}\langle\rangle$, because it is covered by the $\mathtt{emp}\langle\rangle$ cases; disjointness implies that $\mathtt{unit}\langle\rangle$ can be added only to $\mathtt{emp}\langle\rangle$. The property 4 is from the equations $\emptyset * s = s * \emptyset = \emptyset$ and $\{(\bot, \ldots, \bot)\} * s = s * \{(\bot, \ldots, \bot)\} = s$. □

Lemma 7. *Let* S-QUERY-RUN$_{\mathcal{A}}$ *be a procedure obtained by replacing* (1) $\emptyset$ *in the procedure* QUERY-RUN$_{\mathcal{A}}$ *with* $\mathtt{emp}\langle\rangle$, (2) $x \uplus y$ *with* UNION-AT(v, x, y), (3) $x * y$ *with* PRODUCT-AT(v, x, y), *and* (4) *singleton*(v, b) *with* SINGLETON-AT(v, b). *Then,* EVAL(S-QUERY-RUN$_{\mathcal{A}}(t)[q, b]$) = QUERY-RUN$_{\mathcal{A}}(t)[q, b]$ *for any* $t \in T_{\Sigma_{\mathcal{A}}}$, $q \in Q_{\mathcal{A}}$, *and* $b \in \mathbb{B}^n$.

Proof. Clear from Lemma 6, by induction on the structure of r. □

Now, we have the following two main theorems of this paper: the answer set of an n-ary regular query can efficiently be computed as a SRED in linear time with respect to the size of the input, and it is also compact; its size is linear, no matter how large the actual answer set is.

Theorem 8 (*Querying*)**.** *For any n-ary regular query* $\psi_{\mathcal{L}(\mathcal{A})}$ *and a tree t, we can compute a SRED r that represents the answer set (i.e.,* EVAL$(r) = \psi_{\mathcal{L}(\mathcal{A})}(t)$*) in time* $O(3^n|\delta_{\mathcal{A}}||t|)$.

Proof. Let r' = S-QUERY-RUN$_{\mathcal{A}}(t)$. We can compute the desired SRED r by combining all $r'[q, 1\cdots 1]$'s with $q \in F_{\mathcal{A}}$ by UNION-AT. From Theorems 2 and 7, this satisfies the equation EVAL$(r) = \psi_{\mathcal{L}(\mathcal{A})}(t)$ (here, representing the $\cup$ operation in Theorem 2 by UNION-AT is justified because it is indeed a disjoint union, due to the premise that $\mathcal{A}$ is deterministic). The complexity analysis goes similar to the case of QUERY-RUN$_{\mathcal{A}}$. The procedure S-QUERY-RUN$_{\mathcal{A}}$ is applied once for each node in t (that is, the procedure is invoked at most $|t|$ times), and at each node, the innermost loop body (line 10) is executed at most $3^n|\delta_{\mathcal{A}}|$ times. Different from the case of QUERY-RUN$_{\mathcal{A}}$, this time, set operations UNION-AT and PRODUCT-AT in the loop body run in constant time. Hence, the total time complexity of S-QUERY-RUN$_{\mathcal{A}}$ is $O(3^n|\delta_{\mathcal{A}}||t|)$. The last union-phase requires at most $|F_{\mathcal{A}}| - 1$ execution of UNION-AT, whose time consumption can asymptotically be ignored. □

Theorem 9 (*In-Size-Bound*)**.** *The number of nodes of the SRED r in Theorem 8 is at most* $4 \cdot 3^n|\delta_{\mathcal{A}}||t| + |F_{\mathcal{A}}| - 1$.

Proof. Clear from the proof of Theorem 8 (note that in each loop body, up to 4 nodes are created). □

Before developing further algorithms on SRED, it is worth remarking here that Theorem 8 combined with Theorem 4 can be used to derive the "optimal" data complexity for regular queries.

Corollary 10 (*It follows also from Corollary 4.5 of [9]*)**.** *The time complexity of n-ary regular query with respect to the data size is* $O(|t| + |a|)$*, where* $|t|$ *is the size of the input node, and* $|a|$ *is the size of the output answer set.*

This way of using SRED just as an intermediate structure can be regarded as a different presentation of essentially the same algorithm as that of [9]. As mentioned before, in [9], the complexity was achieved by running two pre-processing phases that determine whether each entry $r[q, b]$ (in their notation, $Sat_{t,q}$) at each node contributes to the final query answer, and skipping the computation of the unneeded part. Two cases are considered to be unneeded: the case that we can never reach states in $F_{\mathcal{A}}$ at the root node starting from the state q, and the case that the set $r[q, b]$ is taken a product with the empty set afterward in the computation. In our algorithm, the former case is dealt with by splitting the construction of a SRED structure and the evaluation of it; the construction has low complexity, and the evaluation is only done on the states that reach $F_{\mathcal{A}}$ states. The latter case is detected by the special treatment of $\mathtt{emp}\langle\rangle$ node in the PRODUCT-AT procedure; a SRED that is taken product with an $\mathtt{emp}\langle\rangle$ set is discarded and thus is never evaluated. Despite the similarity, we believe that our presentation is much simpler and easier to understand. In our algorithm, structure of the first naïve algorithm QUERY-RUN$_{\mathcal{A}}$ is kept unchanged, and only just a few set-operations are replaced with (almost trivially correct) SRED-based operations in Fig. 3.

4.3. Direct manipulation of SRED

SRED is not only useful as an intermediate data structure for generating the concrete result of answer tuples. In fact, it allows manipulation of the represented set directly on SRED, without evaluation. Here, we give an implementation of two important operations on SRED, namely, PROJECTION and SELECTION. For a set s of n-tuples and $1 \le i \le n$, PROJECTION $s_{@i} = \{v_i \mid (v_1, \ldots, v_n) \in s\}$ is the set of i-th coordinates of s. Given an element u, SELECTION $s_{[i:u]} = \{(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \mid (v_1, \ldots, v_{i-1}, u, v_{i+1}, \ldots, v_n)\}$ is the set of tuples in s such that the i-th coordinate is u. As an example of a use-case of the two operations, consider the following scenario: first we apply PROJECTION $_{@1}$ to an answer set, sort the result in some preferable order, and with each element u of the projected set, apply SELECTION $_{[1:u]}$ to get the remaining coordinates. In this way, we can enumerate the answers of queries in a user-specified order on the first coordinate, rather than in the default order of EVALUATION procedure.

On SRED representation of the answer sets, those two operations can be carried out in time proportional to the *height* of the input tree. That is, we do not need to traverse the whole structure of SRED, nor to re-traverse the original input tree. Fig. 5 is the implementation, which is straightforwardly obtained from the distributivity of projection and selection over disjoint union, etc.

```
PROJ $(i, r)$
1: **if** $r \equiv \mathsf{emp}\langle\rangle$ **then**
2:     **return** $\emptyset$
3: **else if** $r \equiv \mathsf{ne}\langle r'\rangle$ **then**
4:     **return** PROJ-NE$(i, r')$

PROJ-NE $(i, r)$
1: **if** $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$ **then**
2:     **return** PROJ-NE$(i, r_1) \cup$ PROJ-NE$(i, r_2)$
3: **else if** $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$ **(with** $r_1 \in NST_{\beta_1\cdots\beta_n}$**) then**
4:     **if** $\beta_i = 1$ **then return** PROJ-NE$(i, r_1)$ **else return** PROJ-NE$(i, r_2)$
5: **else if** $r \equiv \mathsf{sing}\langle v, \beta_1\cdots\beta_n\rangle$ **then**
6:     **return** $\{v\}$

SEL $(i, u, r)$
1: **if** $r \equiv \mathsf{emp}\langle\rangle$ **then**
2:     **return** $\mathsf{emp}\langle\rangle$
3: **else if** $r \equiv \mathsf{ne}\langle r'\rangle$ **then**
4:     **return** SEL-NE$(i, u, r')$

SEL-NE $(i, u, r)$
1: **if** $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$ **and** $v \le u$ **then**
2:     **return** UNION-AT$(v,$ SEL-NE$(i, u, r_1),$ SEL-NE$(i, u, r_2))$
3: **else if** $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$ **(with** $r_1 \in NST_{\beta_1\cdots\beta_n}$**) and** $v \le u$ **then**
4:     **if** $\beta_i = 1$ **then return** PRODUCT-AT$(v,$ SEL-NE$(i, u, r_1), r_2)$
5:     **else return** PRODUCT-AT$(v, r_1,$ SEL-NE$(i, u, r_2))$
6: **else if** $r \equiv \mathsf{sing}\langle v, \beta_1\cdots\beta_n\rangle$ **and** $v = u$ **then**
7:     **return** SINGLETON-AT$(v, \beta_1\cdots\beta_{i-1}\beta_{i+1}\cdots\beta_n)$
8: **else return** $\mathsf{emp}\langle\rangle$
```

Fig. 5. Projection and selection on SRED.

Theorem 11 (*Projection*). *By using memoization, the procedure* PROJ(i, r) *computes the set* $\mathrm{EVAL}(r)_{@i}$ *in time* $O(\min(m, 3^n h|\delta_{\mathcal{A}}|\,|\mathrm{EVAL}(r)_{@i}|))$ *where h is the height of the original input tree t, and m is the number of nodes of r.*

Proof. Correctness immediately follows from the following set-theoretic properties of projection: $\phi_{@i} = \phi$, $(s_1 \uplus s_2)_{@i} = (s_1)_{@i} \cup (s_2)_{@i}$, $(s_1 * s_2)_{@i} = (s_1)_{@i}$ if the i-th coordinates of s_1 is non-$\perp$ and $(s_1 * s_2)_{@i} = (s_2)_{@i}$ otherwise, and $\{(u_1, \ldots, u_n)\}_{@i} = \{u_i\}$ for $u_i \ne \perp$.

For the complexity, we assume the procedure PROJ-NE to be memoized, i.e., if it is applied to the same arguments second time, it immediately returns the previous result in constant time. We can implement such memoization by using a hash table. Then the body of the procedure PROJ-NE is executed at most once per each node of r. Actually, the procedure PROJ-NE is applied only to the nodes in $NST_{\beta_1\cdots\beta_n}$ with $\beta_i = 1$. The number of such nodes is at most $4 \cdot 3^n|\delta_{\mathcal{A}}|h|\mathrm{EVAL}(r)_{@i}|$, because to have $\beta_i = 1$, it must have a descendant node of the form $\mathsf{sing}\langle v, \ldots\rangle$ with $v \in \mathrm{EVAL}(r)_{@i}$. Since such a SRED node is created only at the ancestor nodes of v in the original input tree (whose number is at most $h|\mathrm{EVAL}(r)_{@i}|$) and at each of such ancestors at most $4 \cdot 3^n|\delta_{\mathcal{A}}|$ SRED nodes are created, we obtain the bound on the number of the nodes. By using list-concatenation for representing set-union,[2] the body of PROJ-NE can be executed in constant time. Hence, we obtain the desired complexity. □

Theorem 12 (*Selection*). *By using memoization, the procedure* SEL(i, u, r) *computes the SRED representation of the set* $\mathrm{EVAL}(r)_{[i:u]}$ *in time* $O(\min(m, 3^n h|\delta_{\mathcal{A}}|))$*, where m is the number of nodes of r.*

Proof. Correctness immediately follows from the following set-theoretic properties of selection: $\phi_{[i:u]} = \phi$, $(s_1 \uplus s_2)_{[i:u]} = (s_1)_{[i:u]} \uplus (s_2)_{[i:u]}$, $(s_1 * s_2)_{[i:u]} = (s_1)_{[i:u]} * s_2$ if the i-th coordinates of s_1 is non-$\perp$ and $(s_1 * s_2)_{[i:u]} = s_1 * (s_2)_{[i:u]}$ otherwise, and $\{(u_1, \ldots, u_n)\}_{[i:u]} = \{(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n)\}$ for $u_i = u$. The side condition $v \le u$ in lines 1 and 3 is justified by Lemma 1; if the comparison does not hold, EVAL-NE(r) cannot contain u.

For the complexity, memoization ensures that the procedure SEL-NE is called at most once per each node of r. Since the test $v \le u$ succeeds only at the node constructed at an ancestor (in the tree t) of u, the procedure SEL-NE is executed only on the nodes constructed at an ancestor of u, or their direct child. Note that the number of the ancestor nodes in the input tree is at most h, and on each of such nodes at most $4 \cdot 3^n|\delta_{\mathcal{A}}|$ SRED-node is created. By, multiplying them, we obtain the desired complexity. □

Corollary 13 (*Membership*). *Given a SRED r and a tuple* $(u_1, \ldots, u_n)$ *of nodes, we can test whether* $(u_1, \ldots, u_n)$ *is in* EVAL(r) *or not in time* $O(n \min(m, 3^n h|\delta_{\mathcal{A}}|))$.

Proof. Repeat SELECTION n times. □

Another interesting operation that can easily be executed on SRED without evaluation is, counting of the size of the represented set.

Theorem 14 (*Size*). *By using memoization, given a SRED r, the size of the represented set* $|\mathrm{EVAL}(r)|$ *can be computed in time* $O(m)$ *where m is the number of nodes of r.*

Proof. Fig. 6 shows the implementation. By memoization, the procedure SIZE-NE is called m times, and the body of the procedure runs in constant time. Correctness follows from the following facts: $|s_1 \uplus s_2| = |s_1| + |s_2|$, $|s_1 * s_2| = |s_1| \times |s_2|$, and $|\{(u_1, \ldots, u_n)\}| = 1$. □

[2] Precisely speaking, since it is not a *disjoint* union this time, list-concatenation based implementation may cause duplication. It, however, can be removed by a linear time ‘uniq’ algorithm.

```
SIZE $(r)$
1: if $r \equiv \mathsf{emp}\langle\rangle$ then
2:     return 0
3: else if $r \equiv \mathsf{ne}\langle r'\rangle$ then
4:     return SIZE-NE$(r')$
```

```
SIZE-NE $(r)$
1: if $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$ then
2:     return SIZE-NE$(r_1)$ + SIZE-NE$(r_2)$
3: else if $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$ then
4:     return SIZE-NE$(r_1)$ $\times$ SIZE-NE$(r_2)$
5: else if $r \equiv \mathsf{sing}\langle v, \beta_1 \cdots \beta_n\rangle$ then
6:     return 1
```

Fig. 6. Computing the size of the set represented by a SRED.

Note that, the procedure SIZE-NE is computing the size of the represented set for all nodes in r. By using the size information, the one-by-one enumeration procedure shown in Fig. 4 is improved to a *log-delay* enumerator.

Corollary 15 (*Log-Delay-Enum*)**.** *For a SRED r, we can enumerate the elements of* EVAL(r) *in log-delay after $O(m)$ time preprocessing. That is, in the enumeration process, the time required to output adjacent two elements are $O(\log_2 |\text{EVAL}(r)|)$ for any adjacent pairs, and also the first element is generated in $O(\log_2 |\text{EVAL}(r)|)$ time.*

Proof. By using the size information obtained by SIZE-NE, without loss of generality we can assume $|\text{EVAL-NE}(r_1)| \leq |\text{EVAL-NE}(r_2)|$ in line 1 to 3 of the EVAL-NE-1BY1 procedure, (otherwise swap r_1 and r_2, which only changes the order but not the enumerated set). Note that this implies $2 \cdot |\text{EVAL-NE}(r_1)| \leq |\text{EVAL-NE}(r)|$.

Then, we can show that during the computation of EVAL-NE-1BY1(r, f), we enter the procedure EVAL-NE-1BY1 at most $\log_2(|\text{EVAL-NE}(r)|) + k + 1$ times between any two successive calls for f (and between the beginning of the computation and the first call to f), where $k = dim(r) - 1$. The same estimation $\log_2(|\text{EVAL-NE}(r)|) + k + 1$ applies also to the number of times we leave the procedure between two successive calls for f (and between the last call to f and the end of the computation, under the assumption that tail-calls are optimized away). This proves the corollary.

The proof of the above statement is by induction on structure of r. When $r \equiv \mathsf{sing}\langle v, \beta_1, \ldots, \beta_n\rangle$, only one call to f is made and between the call and the start of the computation of EVAL-NE-1BY1(r, f), exactly one call to EVAL-NE-1BY1 is made. Since $\log_2(|\text{EVAL-NE}(r)|) + k + 1 \geq 1$, we have proved the inductive statement for this case. When $r \equiv \mathsf{cup}\langle v, r_1, r_2\rangle$, interval of two successive calls to f is at most $\max(\log_2(|\text{EVAL-NE}(r_1)|) + k + 1, \log_2(|\text{EVAL-NE}(r_2)|) + k + 1) \leq \log_2(|\text{EVAL-NE}(r)|) + k + 1$ by induction hypothesis. The delay to the first call to f is $1 + \log_2(|\text{EVAL-NE}(r_1)|) + k + 1$, which is less than or equal to $\log_2(|\text{EVAL-NE}(r)|) + k + 1$, because of our assumption on the size of r_1. Between the last call to f and the end of the computation, the number of times we leave the procedure can be made at most $\log_2(|\text{EVAL-NE}(r_2)|) + k + 1 \leq \log_2(|\text{EVAL-NE}(r)|) + k + 1$; since the second call to EVAL-NE-1BY1 is a tail-call, return from the call can directly leave the whole computation.[3] When $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$, the delay is at most $1 + (\log_2(|\text{EVAL-NE}(r_1)|) + k_1 + 1) + (\log_2(|\text{EVAL-NE}(r_2)|) + k_2 + 1)$ for some $1 + k_1 + k_2 = k$. Since we have $(\log_2(|\text{EVAL-NE}(r_1)|) + \log_2(|\text{EVAL-NE}(r_2)|)) = \log_2(|\text{EVAL-NE}(r)|)$, the induction statement is now proved. □

4.4. Generalizations to unranked trees

So far, we have considered only binary trees. In many applications, however, we are interested in *unranked* trees with varying number of child nodes. For example, in XML trees [14] such as XHTML documents, the number of children may not be two, or even, may differ even between two nodes with the same label (e.g., an `<ol>` (ordered list) node can have an arbitrary number of `<li>` (list item) child nodes).

To deal with unranked trees, we encode such trees to binary trees. A widely used encoding is *fc-ns encoding*. In a binary tree obtained as the fc-ns encoding of an unranked tree, the first child of each node is mapped to the *first child* of the corresponding node in the original unranked tree, and the second child of each node is mapped to the *next sibling* in the unranked tree. It is a folklore result that the encoding preserves the regularity of queries, i.e., any regular query for unranked trees can be converted to a regular query on the encoded trees. Hence, by first encoding the unranked input trees and the queries to the binary-tree form and then running S-QUERY-RUN$_{\mathcal{A}}$, we can compute the linear-size representation of the answer sets of regular queries. One problem of fc-ns encoding is the time complexity of operations on SRED that depends on the factor h, the height of the tree. Suppose an original unranked tree has small height h_0 and nodes with large number $w_0(\simeq|t|)$ of children (which is often the case for most XML documents). The problem is that the height of the fc-ns encoded tree is $O(h_0 w_0)$. To deal with such trees, we recommend using another encoding, namely, the *bb encoding*, to reduce the complexity to $O(h_0 \log w_0)$. In bb encoding, the list of children of each node is encoded to a *balanced binary tree* whose left-to-right sequence of leaf nodes corresponds to the child sequence in the original tree. Such an encoding also preserves regularity, because the 'first-child' and the 'next-sibling' relations remain regular. Moreover, since the height of a balanced binary tree is in the logarithmic order of the number of the leaves, the height of the bb-encoded tree reduces to $O(h_0 \log |t|)$.

[3] Such tail-call optimization is performed in almost all practical compilers for popular programming languages. Even if it does not, the tail-recursion can easily be rewritten to an iteration by while-loop manually.

5. Application

SRED is developed for the XML transformation language MTran [15]. Let us illustrate the benefits of SRED by the following pseudo code for XML translation:

```
{gather x where x:<person> do
   <row>
      <col>{gather y where x//<name>/y do y}</col>
      {gather z where z:<person> & document-order(z,x) do
         <col>···</col>}
   </row>}
```

The program takes a document containing a list of <person> elements and generates some triangular matrix table. The first query "x:<person>" lists up all the <person> elements, and for each of them, the second query "x//<name>/y" selects a descendant y of x labeled <name> (for simplicity, we assume that such y uniquely exists). If we really run for each x the second query, which takes in general $O(|t|)$ time where $|t|$ is the size of the tree, total running time of the query becomes quadratic, because there may be linearly many <person> nodes. Rather, as pointed out in [16], it is better to regard the second query as a *binary query* for selecting pairs (x, y). By using SRED, the answer set of such a binary query can be computed in linear time. Furthermore, by the SELECTION operation followed by the EVALUATION operation, for each x we can obtain the corresponding y in time $O(h_0 \log |t|)$. Total running time reduces to $O(h_0|t| \log |t|)$. So far, we could have used the FFG algorithm [9] (or equivalently, query with SRED directly followed by EVALUATION) for the same purpose, because its running time is linear under the assumption that y uniquely exists for each x. Consider, then, the third query that selects all <person> elements z preceding x in the document order (preorder). Similarly, we run the query as a binary query for selecting pairs (x, z). In this case, the size of the answer set is quadratic. If we use the FFG algorithm, we need $O(|t|^2)$ working space for carrying out the binary-query based approach. While, with SRED, it requires only $O(|t|)$ working space. This makes it feasible to run the transformation over larger inputs, which could not be done without SRED due to memory shortage.

6. Conclusion and future work

The paper introduced a data structure named SRED (Set Representation by Expression Dags), which allows representing answer sets of regular tree queries compactly. Here is the summary of its performance for the n-ary query defined by an automaton with transition function $\delta_{\mathcal{A}}$, with an input tree t and an output set a:

QUERYING	EVALUATION	Size (number of nodes) of SRED ($= m$)
$O(3^n|\delta_{\mathcal{A}}||t|)$	$O(n^2|a|)$	at most $\min(2n|a|,\ 4\cdot 3^n|\delta_{\mathcal{A}}||t|)$

Regardless how large the output answer set is, the time for computing its SRED representation is independent of it; it is always linear with the size of the input. Evaluation (or decompression) of SRED only depends on the size of the answer set and is independent from the input size. The size of the SRED representation stays at the minimum of them. Thus, for a large answer set (e.g., $|a| \simeq |t|^n$), SRED works as a concise representation of the set, and even for a small ($|a| \ll |t|$) answer set that could not benefit from the compression, it works no worse than non-compressed representations. Furthermore, SRED allow several kinds of direct manipulations on the represented sets, without decompression:

PROJECTION	SELECTION	SIZE
$O(\min(m, 3^n h|\delta_{\mathcal{A}}||p|))$ ($|p|$ is the size of the projected set)	$O(\min(m, 3^n h|\delta_{\mathcal{A}}|))$	$O(m)$

In the paper, we have used the *total deterministic* tree automaton as a representative of regular queries. One possible direction for future work is to extend the SRED representation to support other query formalisms directly, rather than through a conversion to a deterministic automaton. In fact, the algorithms given in this paper works without any change for *partial* deterministic automata, and, as long as it is *unambiguous*, for non-deterministic ones. It seems an interesting question whether there is a possibility to support arbitrary non-deterministic tree automata.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. This work was partly supported by the Japan Society for the Promotion of Science.

References

[1] H. Hosoya, B.C. Pierce, Regular expression pattern matching for XML, Journal of Functional Programming 13 (2003) 961–1004. doi:10.1017/S0956796802004410.

[2] J.W. Thatcher, J.B. Wright, Generalized finite automata theory with an application to a decision problem of second-order logic, Mathematical Systems Theory 2 (1968) 57–811. doi:10.1007/BF01691346.

[3] D. Niwinski, Fixed points vs. infinite generation, in: Logic in Computer Science, LICS, 1988, pp. 402–409. doi:10.1109/LICS.1988.5137.
[4] G. Gottlob, C. Koch, R. Pichler, Efficient algorithms for processing XPath queries, ACM Transactions on Database Systems 30 (2005) 444–491. doi:10.1145/1071610.1071614.
[5] G. Gottlob, C. Koch, Monadic datalog and the expressive power of languages for Web information extraction, Journal of the ACM 51 (2004) 74–113. doi:10.1145/962446.962450.
[6] F. Neven, J.V.D. Bussche, Expressiveness of structured document query languages based on attribute grammars, Journal of the ACM 49 (2002) 56–100. doi:10.1145/505241.505245.
[7] H. Meuss, K.U. Schulz, F. Bry, Towards aggregated answers for semistructured data, in: International Conference on Database Theory, ICDT, 2001, pp. 346–360. doi:10.1007/3-540-44503-X_22.
[8] E. Filiot, S. Tison, Regular *n*-ary queries in trees and variable independence, in: International Conference on Theoretical Computer Science, IFIP TCS, 2008, pp. 429–443. doi:10.1007/978-0-387-09680-3_29.
[9] J. Flum, M. Frick, M. Grohe, Query evaluation via tree-decompositions, Journal of the ACM 49 (2002) 716–752. doi:10.1145/602220.602222.
[10] G. Bagan, MSO queries on tree decomposable structures are computable with linear delay, in: Computer Science Logic, CSL, 2006, pp. 167–181. doi:10.1007/11874683_11.
[11] B. Courcelle, Linear delay enumeration and monadic second-order logic, Discrete Applied Mathematics 157 (2009) 2675–2700. doi:10.1016/j.dam.2008.08.021.
[12] P.F. Dietz, Maintaining order in a linked list, in: ACM Symposium on Theory of Computing, STOC, 1982, pp. 122–127. doi:10.1145/800070.802184.
[13] C. Okasaki, Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking, in: Foundations of Computer Science, FOCS, 1995, pp. 646–654. doi:10.1109/SFCS.1995.492666.
[14] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, Extensible markup language (XML^{TM}), 2000. http://www.w3.org/XML/.
[15] K. Inaba, H. Hosoya, XML transformation language based on monadic second order logic, in: Programming Language Technologies for XML, PLAN-X, 2007, pp. 49–60.
[16] A. Berlea, H. Seidl, Binary queries for document trees, Nordic Journal of Computing 11 (2004) 41–71.