Compact Representation for Answer Sets of n-ary Regular Queries

Kazuhiro Inaba[1] and Haruo Hosoya[1]

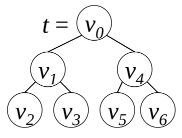
The University of Tokyo, {kinaba,hahosoya}@is.s.u-tokyo.ac.jp

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 to store the set using only $O(3^n|t|)$ space. Several basic operations on the sets are shown to be efficiently executable on the representation.

1 Introduction

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. And, 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 the 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 the figure. 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 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 the expression of length $2n + 1$, which consumes only linear space with respect to the size of the input tree.

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, the expression can be represented by a dag of size $O(3^n|t|)$. 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! The factor 3^n is sufficiently low for queries with small n such as binary or ternary queries, which are the most cases occur in practice (after all, it is quite rare to run, say, a 100-ary query).

Furthermore, the dag representation is extended to a data structure named *SRED (Set Representation by Expression Dags)*, which enjoys good time complexity as well as the size-efficiency. The SRED representation of the answer set can always be computed from the input tree t in time $O(3^n|t|)$, 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 $O(3^n a)$, 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 two important operations *without decompressing* it: (1) SELECTION: for an answer set s, the SRED representation of the set $s_{[i:u]} = \{(v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n) \mid (v_1, \dots, v_{i-1}, u, v_{i+1}, \dots, v_n) \in s\}$ can be computed in time $O(3^n h)$ where h is the height of the input tree for binary trees and is the height times $\log |t|$ for unranked trees, and (2) PROJECTION: the set $s_{@i} = \{v_i \mid (v_1, \dots, v_n) \in s\}$ can be computed in time $O(6^n h |s_{@i}|)$. The 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 to locate 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, Schulz, and Bry [7] as a compact representation of answer sets of queries. The size of a CAA is $O(nh|t|)$ which is competitive to our $O(3^n|t|)$. 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 $O(n|t|^{d_\phi})$ for some constant d_ϕ depending on the query, which grows to n in the worst case. Besides, precise complexity of operations like 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, Frick, and Grohe [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 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 is future work whether our compact representation of the answer sets can be extended to more general class of queries.

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 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 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 [10]. 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 are those 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 $\psi_L(t) = \{(v_1, \ldots, v_n) \mid mark(t, v_1, \ldots, v_n) \in L\}$ where $mark(t, v_1, \ldots, v_n)$ 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 $(v_1, \ldots, v_n)$ if and only if L contains a tree obtained by marking each selected node v_i with 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 naive 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 $(v_1, \ldots, v_n) \in V_t^n$ we generate the marked tree $mark(t, v_1, \ldots, v_n)$ and test whether it is accepted by $\mathcal{A}$. If it is, $(v_1, \ldots, v_n)$ 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.

```
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$_\mathcal{A}(lt(v))$;   $r_2 \leftarrow$ QUERY-RUN$_\mathcal{A}(rt(v))$
7:      for each $((label(v), b_0), q_1, q_2 \mapsto q_0) \in \delta_\mathcal{A}$ do
8:          for each disjoint $b_0, b_1, b_2$ in $00 \ldots 00$ to $11 \ldots 11$ do
9:              $r[q_0, b_0|b_1|b_2] \leftarrow r[q_0, b_0|b_1|b_2] \ \uplus \ singleton(v, b_0) * r_1[q_1, b_1] * r_2[q_2, b_2]$
10: 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 $u_i = \bot$ if $\beta_i = 0$. Here, $\bot$ is a special symbol not contained in V_t. In line 7, **for each disjoint** iterates over pairs of form $(b_1 = \beta_{11} \cdots \beta_{1n}, b_2 = \beta_{21} \cdots \beta_{2n}) \in (\mathcal{B}^n)^2$ such that for all $1 \leq i \leq n$, at most one of $\{\beta_{0i}, \beta_{1i}, \beta_{2i}\}$ is 1, with $\beta_{01} \cdots \beta_{0n} = b_0$. 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: $S{*}T = \{(u_1, \cdots, u_n) \mid (s_1, \cdots, s_n) \in S, (t_1, \ldots, t_n) \in T, \forall i : (u_i = s_i \wedge \bot = t_i) \vee (\bot = s_i \wedge u_i = t_i)\}$. For example, $\{(v_1, \bot, \bot), (v_2, \bot, \bot)\} * \{(\bot, \bot, v_3), (\bot, \bot, v_4)\}$ is equal to $\{(v_1, \bot, v_3), (v_1, \bot, v_4), (v_2, \bot, v_3), (v_2, \bot, v_4)\}$. Let us remark that we never take $*$-product of sets that have tuples with non-$\bot$ nodes on the same position, as will be shown in Lemma 1.

Let us explain how the algorithm works. Let $r = \text{QUERY-RUN}_{\mathcal{A}}(v)$ 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 $r = \text{QUERY-RUN}_{\mathcal{A}}(v)$ 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. Formally, the following lemma can be shown by induction on the structure of the tree rooted at v.

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

The lemma ensures the two disjointness in the procedure $\text{QUERY-RUN}_{\mathcal{A}}$. 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 $\perp$. For such a case, we have $|S * T| = |S| \cdot |T|$. Second, $\uplus$ is always taken between disjoint sets, because the operands of $\uplus$ are constructed by $*$-product over different flags.

The answer set of the query can be calculated from the result of $\text{QUERY-RUN}_{\mathcal{A}}$ applied to the root node, namely, $r = \text{QUERY-RUN}_{\mathcal{A}}(root_t)$. 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. $\psi_{\mathcal{L}(\mathcal{A})}(t) = \bigcup_{q \in F_{\mathcal{A}}} \text{QUERY-RUN}_{\mathcal{A}}(root_t)[q, 11 \cdots 11]$.

Proof (sketch; for more detail, consult Claim 1 of [9]). Let $v_1, \ldots, v_n \in V_t$ to be fixed and ρ be the unique run on the tree *mark*$(t, v_1, \ldots, v_n)$ by $\mathcal{A}$. Let $v \in V_t$. For each i, if $v \leq_t v_i$ then let $u_i = v_i$ and $\beta_i = 1$. Otherwise let $u_i = \perp$ and $\beta_i = 0$. We can prove by induction on the structure of v the following claim: if $\rho(v) \neq q$ the set $\text{QUERY-RUN}_{\mathcal{A}}(v)[q, b]$ is empty for any $b \in \mathbb{B}^n$, and if $\rho(v) = q$ then $(u_1, \ldots, u_n) \in \text{QUERY-RUN}_{\mathcal{A}}(v)[q, b]$ if and only if $b = \beta_1 \cdots \beta_n$. By applying the claim to the root node $v = root_t$, we have $(v_1, \ldots, v_n) \in \text{QUERY-RUN}_{\mathcal{A}}(root_t)[q, 11 \cdots 11]$ if and only if $q = \rho(root_t)$, which, together with the definition of $\psi_{\mathcal{L}(\mathcal{A})}$, proves the desired result. $\square$

What is the 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_0, b_1, b_2 requires 3^n iterations, 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: $O(3^n|\delta_{\mathcal{A}}||t|^{n+1})$. In fact, the complexity can be reduced by a 2-pass preprocessing proposed in [9]. Their preprocessing detects, for each node, whether or not each entry $r[q, b]$ really needs to be computed. By omitting the computations that turned out not to be need, the complexity is reduced to $O(3^n|\delta_{\mathcal{A}}|(|t| + a))$ where a is the size of the answer set.

In the next section, we take a completely different approach for reducing the complexity. Rather than changing the structure of the algorithm (like adding preprocessing passes), we introduce a novel data structure for representing sets of tuples. Just by using the data structure to represent 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.

4 SRED: Set Representation by Expression Dags

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 $s_1 = [\![r_1]\!]$). 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-*dag*s 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 $\beta_1 \cdots \beta_n \in \mathbb{B}^n$:

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

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 the efficiency, we specially treat the empty set (represented by $\mathsf{emp}\langle\rangle$) and the *unit set* ($\{(\bot, \ldots, \bot)\}$, 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

```
EVAL $(r)$
1: if $r \equiv \mathsf{emp}\langle\rangle$ then return $\emptyset$
2: else if $r \equiv \mathsf{unit}\langle\rangle$ then return $\{(\bot, \cdots, \bot)\}$
3: else if $r \equiv \mathsf{ne}\langle r'\rangle$ then 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 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. 1. Basic Operations on SRED

operands of cup. 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 simple recursion shown in Fig. 1, in the data complexity proportional to the size of the answer set.

Lemma 3 (EVALUATION). *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* EVAL(r) *(*EVAL-NE(r)*, respectively) runs in time* $O\big(3^k n|\text{EVAL}(r)|\big)$ *(*$O\big(3^k n$ $|\text{EVAL-NE}(r)|\big)$*) where k is the maximum number of* star *nodes in every path from r to any leaf.*

Proof. The proof is by induction on the structure of r. For the case of emp, unit, sing, and cup nodes, it is trivial and hence omitted here. For the case $r \equiv \mathsf{star}\langle v, r_1, r_2\rangle$, by induction hypothesis, $s_1 = \text{EVAL-NE}(r_1)$ and $s_2 = \text{EVAL-NE}(r_1)$ can be computed in $3^{k-1}n(|s_1| + |s_2|)$ steps. Since neither s_1 nor s_2 is empty, their sizes are less than or equal to $|s_1 * s_2|$. Thus, $3^{k-1}n(|s_1| + |s_2|)$ is no more than $2 \cdot 3^{k-1}n|s_1 * s_2|$. By the assumption, their *-product can be computed in time $n|s_1 * s_2|$. Altogether, the total time consumption for EVAL(r) in this case is $3^k n|s_1 * s_2| = 3^k n|\text{EVAL-NE}(r)|$ as desired. □

The complexity assumption is satisfied by, for instance, representing the sets by a doubly-linked list of elements. Disjoint union can be implemented by the list concatenation, and the $*$-product is implemented by a double-loop over two operand sets. Note that, the number k of star node in a path is at most n, because the star operation strictly increases the number of non-$\bot$ coordinates in the element tuples.

The basic three operations used in the algorithm QUERY-RUN$_{\mathcal{A}}$ are defined on SRED as in Figure 1. Note that, to avoid $\mathsf{emp}\langle\rangle$ and $\mathsf{unit}\langle\rangle$ to occur 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 is easily verified by induction on the structure of SRED, and we have the following results:

Lemma 4 (Correctness). EVAL(UNION-AT(v, r_1, r_2)) $=$ EVAL(r_1) $\uplus$ EVAL(r_2), EVAL(PRODUCT-AT(v, r_1, r_2)) $=$ EVAL$(r_1)*$EVAL(r_2), *and* EVAL(SINGLETON-AT(v, b)) $= singleton(v, b)$.

```
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 $\{v\}$
```

```
SELECT $(i, u, r)$
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 $v \leq u$ 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 $v \leq u$ 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$(v, \beta_1 \cdots \beta_{i-1} 0 \beta_{i+1} \cdots \beta_n)$
8: else return $\mathsf{emp}\langle\rangle$
```

Fig. 2. Projection and Selection on SRED

Theorem 5. *Let* S-QUERY-RUN$_{\mathcal{A}}$ *be a procedure obtained by replacing* $\emptyset$ *in the procedure* QUERY-RUN$_{\mathcal{A}}$ *with* $\mathsf{emp}\langle\rangle$, $x \cup\!\!\!\cdot\, y$ *with* UNION-AT(v, x, y), $x * y$ *with* PRODUCT-AT(v, x, y), *and singleton*(v, b) *with* SINGLETON-AT(v, b). *Then,* S-QUERY-RUN$_{\mathcal{A}}(t)$ *runs in time* $O(3^n |\delta_{\mathcal{A}}||t|)$ *and outputs a SRED* r *with at most* $3^n |\delta_{\mathcal{A}}||t|$ *nodes, such that* EVAL$(r) =$ QUERY-RUN$_{\mathcal{A}}(t)$.

Rather than enumerating the all elements of the answer set, we sometimes want to extract a sub-part of the answer set. Here, we give an implementation of two important operations on SRED, namely, PROJECTION and SELECTION. For a set s of n-tuples and $1 \leq i \leq n$, 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, 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. 2 is the implementation, which is straightforwardly obtained from the distributivity of projection and selection over disjoint union, etc.

Theorem 6 (PROJECTION**).** *By using memoization, the procedure* PROJ(i, r) *computes the set* EVAL$(r)_{@i}$ *in time* $O(6^n h |\delta_{\mathcal{A}}||\mathrm{EVAL}(r)_{@i}|)$ *where* h *is the height of the original input tree* t.

Proof. Correctness is proved by induction on the structure of r, which is omitted here due to the lack of the space. 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 hash table. Then the body of the procedure PROJ-NE is executed at most once per each node of r. In fact, it can be shown that PROJ-NE is applied only to the nodes that are an ancestor of a $\mathsf{sing}\langle v, \cdots \rangle$ node with $v \in \mathrm{EVAL}(r)_{@i}$. By the definition of the QUERY-RUN$_{\mathcal{A}}$ procedure, the number of such sing nodes is at most $2^n|\mathrm{EVAL}(r)_{@i}|$, and for each of them, the number of the ancestors is at most $3^n h|\delta_{\mathcal{A}}|$. By using list-concatenation for representing set-union[1], the body of PROJ-NE can be executed in constant time. Hence, we obtain the desired complexity. $\square$

Theorem 7 (SELECTION**).** *By using memoization, the procedure* $\mathrm{SEL}(i, u, r)$ *computes the set* $\mathrm{EVAL}(r)_{[i:u]}$ *in time* $O(3^n h|\delta_{\mathcal{A}}|)$.

Proof. Correctness is proved by induction on the structure of r. For the complexity, memoization ensures that the procedure SEL-NE is called at most once per each node of r. By Lemma 1, the test $v \leq u$ succeeds only at the node constructed at an ancestor (in the tree t) of u. Hence, SEL-NE is executed only on the nodes constructed at an ancestor of u, or their direct child. Since the number of the ancestor nodes is at most h and on each of such nodes at most $3^n|\delta_{\mathcal{A}}|$ SRED-node is created, SEL-NE is executed only $O(3^n h|\delta_{\mathcal{A}}|)$ times, which proves the desired complexity. $\square$

As a corollary, given a tuple $(u_1, \ldots, u_n)$, we can test whether a SRED contains the tuple in time $O(3^n nh|\delta_{\mathcal{A}}|)$ by applying SELECTION n times.

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. 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 to use 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 $O(h_0 \log |t|)$.

Application SRED is developed for the XML transformation language MTran [11]. Let us illustrate the benefits of SRED by the following pseudo code for XML translation:

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

```
{gather x | x:<person> do
 <row><col>{gather y | (x//<name>/y) do y}</col>
    {gather z | 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 [12], 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 (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 feasible to run the transformation over larger inputs, which could not be done without SRED due to memory shortage.

Thanks This work was supported by the Japan Society for the Promotion of Science.

References

1. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. Journal of Functional Programming **13** (2003) 961–1004
2. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathematical Systems Theory **2** (1968) 57–811
3. Niwinski, D.: Fixed points vs. infinite generation. In: LICS. (1988) 402–409
4. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Transactions on Database Systems **30** (2005) 444–491
5. Gottlob, G., Koch, C.: Monadic datalog and the expressive power of languages for Web information extraction. Journal of the ACM **51** (2004) 74–113
6. Neven, F., Bussche, J.V.D.: Expressiveness of structured document query languages based on attribute grammars. Journal of the ACM **49** (2002) 56–100
7. Meuss, H., Schulz, K.U., Bry, F.: Towards aggregated answers for semistructured data. In: International Conference on Database Theory (ICDT). (2001) 346–360
8. Filiot, E., Tison, S.: Regular n-ary queries in trees and variable independence. In: International Conference on Theoretical Computer Science (TCS). (2008) 429–443
9. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. Journal of the ACM **49** (2002) 716–752
10. Dietz, P.F.: Maintaining order in a linked list. In: STOC. (1982) 122–127
11. Inaba, K., Hosoya, H.: XML transformation language based on monadic second order logic. In: Programming Language Technologies for XML (PLAN-X). (2007) 49–60
12. Berlea, A., Seidl, H.: Binary queries for document trees. Nordic Journal of Computing **11** (2004) 41–71