5-17-2013

Search and Result Presentation in Scientific Workflow Repositories

Susan Davidson
University of Pennsylvania, susan@cis.upenn.edu

Xiaocheng Huang
Nankai University, huangx@seas.upenn.edu

Julia Stoyanovich
Drexel University, stoyanovich@drexel.edu

Xiaojie Yuan
Nankai University, yuanxj@nankai.edu.cn

Search and Result Presentation in Scientific Workflow Repositories

Abstract

We study the problem of searching a repository of complex hierarchical workflows whose component modules, both composite and atomic, have been annotated with keywords. Since keyword search does not use the graph structure of a workflow, we develop a model of workflows using context-free bag grammars. We then give efficient polynomial-time algorithms that, given a workflow and a keyword query, determine whether some execution of the workflow matches the query. Based on these algorithms we develop a search and ranking solution that efficiently retrieves the top-k grammars from a repository. Finally, we propose a novel result presentation method for grammars matching a keyword query, based on representative parse-trees. The effectiveness of our

Keywords

scientific workflows, keyword search, result presentation

Disciplines

Other Computer Engineering

Search and Result Presentation in Scientific Workflow Repositories

Susan B. Davidson[1] Xiaocheng Huang[2] Julia Stoyanovich[3] Xiaojie Yuan[2]

[1]University of Pennsylvania
Philadelphia, USA
susan@cis.upenn.edu

[2]Nankai University
Tianjin, China
huangx@seas.upenn.edu
yuanxj@nankai.edu.cn

[3]Drexel University
Philadelphia, USA
stoyanovich@drexel.edu

ABSTRACT

We study the problem of searching a repository of complex hierarchical workflows whose component modules, both composite and atomic, have been annotated with keywords. Since keyword search does not use the graph structure of a workflow, we develop a model of workflows using *context-free bag grammars*. We then give efficient polynomial-time algorithms that, given a workflow and a keyword query, determine whether some execution of the workflow *matches* the query. Based on these algorithms we develop a *search and ranking* solution that efficiently retrieves the top-k grammars from a repository. Finally, we propose a novel result presentation method for grammars matching a keyword query, based on *representative parse-trees*. The effectiveness of our approach is validated through an extensive experimental evaluation.

1. INTRODUCTION

Data-intensive workflows are gaining popularity in the scientific community. Workflow repositories are emerging in support of sharing and reuse, either as part of a particular workflow system (e.g., VisTrails [4] or Taverna [28]) or independently within a particular community (e.g., *myExperiment.org* [29]). As workflows become more widely used, workflow repositories grow in size, making information discovery an interesting challenge.

Current workflow repositories, e.g., *myExperiment.org*, support tagging of workflows with keywords. Notably, because workflows are modular, users may wish to share and reuse *components* of a workflow [31]. It is thus important to support tagging, and to enable search, not just at the level of a workflow, but also at the level of modules and subworkflows.

Recent work considered search in workflow repositories [10, 25, 30], and also argued that, because workflows can be large and complex, it is important to provide usable result presentation mechanisms. In this paper we propose a novel search and result presentation approach for complex hierarchical workflows. We now illustrate our approach with an example.

Consider a workflow in Figure 1 that computes susceptibility of an individual to genetic disorders, and is based closely on [33]. This workflow takes a person's genetic information in the form of single nucleotide polymorphisms (SNPs) as input, and produces an assessment of genetic disorder risk. The workflow is implemented by module M_0, which is *composite*, and, when invoked, executes modules M_1 and M_2 in sequence. Module M_1 expands the set of

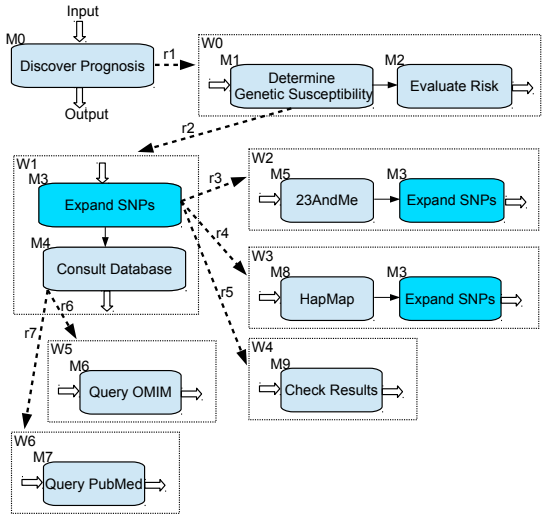


Figure 1: Disease succeptibility workflow.

$r_1 : M_0 \Rightarrow \{M_1, M_2\}$ $r_2 : M_1 \Rightarrow \{M_3, M4\}$
$r_3 : M_3 \Rightarrow \{M_5, M_3\}$ $r_4 : M_3 \Rightarrow \{M_8, M_3\}$ $r_5 : M_3 \Rightarrow \{M_9\}$
$r_6 : M_4 \Rightarrow \{\text{lookup}, M_6\}$ $r_7 : M_4 \Rightarrow \{\text{lookup}, M_7\}$
$r_8 : M_2 \Rightarrow \{\text{evaluate}\}$ $r_9 : M_5 \Rightarrow \{\text{23andMe}\}$
$r_{10} : M_9 \Rightarrow \{\text{check}\}$ $r_{11} : M_6 \Rightarrow \{\text{OMIM}\}$
$r_{12} : M_7 \Rightarrow \{\text{PubMed}\}$ $r_{13} : M_8 \Rightarrow \{\text{HapMap}\}$

Figure 2: Disease succeptibility workflow as a bag grammar.

SNPs by considering known associations between SNP pairs and triplets. This module is composite, and has three *alternative* executions. In the first, the SNP set is expanded using a proprietary database of associations, e.g., 23andMe (module M_5), followed by a *recursive* call to M_3. In the second alternative, a public association database, e.g., HapMap (module M_8) is used, followed by a recursive call to M_3. The final alternative involves checking the retrieved results and terminating the recursion (module M_9). Having computed an expanded SNP set, the workflow goes on to look up any genetic disorders associated with the SNPs. This is implemented by module M_4, which has two alternative executions: it may issue a query to OMIM (module M_6) or to PubMed (module M_7). Having retrieved results from OMIM or from PubMed, the workflow terminates.

Suppose that this workflow exists in a repository, and that some of its modules are tagged. Let us assume the

following assignment of keywords to workflow modules: M_2 (evaluate), M_4(lookup), M_5 (23andMe), M_6 (OMIM), M_7 (PubMed), M_8 (HapMap), and M_9 (check). It is not required that all modules be tagged, e.g., there is no keyword assigned to M_1 in our example. It is also possible, and even likely, that multiple keywords are assigned per module, and that keywords are reused across modules, and across workflows [31]. However, we do not use multiple or repeating keywords here, to simplify our example.

Suppose now that a user wants to know whether the workflow in Figure 1 matches a particular keyword query. Assuming "and" query semantics, answering this question amounts to determining if *there exists some execution of the workflow in which all query keywords are present.* For example, query $\{23andMe, HapMap\}$ matches an execution in which module M_3 is run twice, evaluating $M_5 \rightarrow M_3$ (23andMe) on the first invocation, and $M_8 \rightarrow M_3$ (HapMap) on the second invocation. Intuitively, this execution exists because of the combination of alternation and recursion at M_3.

On the other hand, there is no execution that matches $\{OMIM, PubMed\}$ because, once an alternative for expanding M_4 is chosen, then M_6 (OMIM) or M_7 (PubMed) is executed, and there is no recursion that allows M_4 to repeat, possibly choosing another branch.

It has recently been shown that complex hierarchical workflows can be naturally represented as context-free graph grammars involving recursion and alternation [1, 3]. We build on this work and adapt it to keyword search in workflows with tagged modules. Because of our proposed query semantics, observe that, while hierarchical workflow structure, alternation and recursion are important for determining whether a workflow matches a query, the graph structure within each composite module is unimportant for our purposes. This observation leads us to model scientific workflows as *context-free bag grammars* (also called *commutative grammars* [13]).

Figure 2 represents a bag grammar corresponding to the workflow in Figure 1. The bag grammar captures the hierarchical structure of the workflow (expansion of composite modules), alternation and recursion. Importantly, the grammar makes assignment of keywords to modules explicit, by including keywords as terminals. Note that keywords may annotate both atomic and *composite modules*, appearing in the corresponding grammar productions. So, M_4 is tagged with *lookup*, which is captured in productions r_6 and r_7.

Query $\{23andMe, HapMap\}$ matches the workflow in Figure 1, and we would now like to explain to the user how the match occurs. Let us now return to our example, and focus on *result presentation.* Providing a usable result presentation mechanism is important, because workflow specifications can be large, and each workflow can match a query in multiple ways, due in large part to recursion and alternation. We propose here a result presentation mechanism based on a novel notion of *representative parse trees* (*rpTree* for short). Figure 3 shows a particular *rpTree* for query $\{23andMe, HapMap\}$, with nodes representing bag grammar productions and terminals (see Figure 2), and edges corresponding to a firing of a production. Keyword matches occur at the leaves.

Intuitively, an *rpTree* represents a class of parse trees of a bag grammar that derive a particular set of terminals. An *rpTree* is an *irredundant* representative of its class, in the sense that it does not fire recursive productions that do not derive additional query-relevant terminals. For example, the

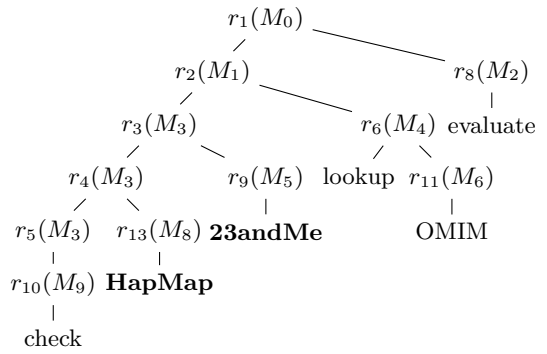


Figure 3: An *rpTree* for query $\{23andMe, HapMap\}$.

rpTree in Figure 3 represents also a tree in which production r_3 is fired recursively twice, both times followed by r_9, and thus generating the terminal *23andMe* twice. We will make the sense in which an *rpTree* is an irredundant representative of its class more precise in Section 5, and will show how *rpTree*s can be derived efficiently.

Contributions. The contributions of our paper include:

- *A model of search and result presentation over keyword-annotated context-free bag grammars* (Section 2): Although the motivation for our model is derived from the problem of searching workflow repositories, it is applicable to any other scenario involving search over context-free bag grammars, e.g., business processes, call structure of programs, or other hierarchical graph applications.

- *Search and ranking algorithms* (Sections 3 and 4): We give a bottom-up *match* algorithm, and develop an optimization, which borrows ideas from semi-naive datalog evaluation to avoid unproductive calculations. Next, translating *probabilistic context-free grammars* to our setting, we develop efficient *search* and *ranking* algorithms, and use them to identify top-k grammars.

- *Novel result presentation methods* (Section 5): Since a workflow may match a query in many different ways, we develop a presentation mechanism to help the user understand how the keyword matches are most likely to occur. The mechanism is based on a novel notion of *representative parse trees*, which are the most probable parse trees that are structurally irredundant.

- *Extensive experimental evaluations* (Section 6): We use synthetic datasets to demonstrate the effectiveness of our approach. Our synthetic data is generated in a way that resembles characteristics of workflows in *myExperiment.org*, in terms of keyword assignment and workflow size. However, since current scientific workflow management systems do not yet allow that workflows be expressed as grammars, we are unable to use the *myExperiment.org* dataset directly in our experiments.

Related Work. Much effort [7, 17, 32] has been made recently to annotate scientific workflows to enable keyword search. As observed in [25], since scientific workflows are usually modeled as a three-dimensional graph structure when considering the expansions of composite modules (dashed

edges in Figure 1), results on searching relational and XML data [5, 24, 35] or graph data [9, 18, 20, 34] can not be easily extended. [10, 25, 30] consider the scenario when alternation or recursion is not present in workflows. [23, 27] consider the scenario where nodes of XML documents exist with a probability (analogous to choice), however there is no recursion.

There are also extensive results in (context-sensitive) *commutative grammars* [14, 13], which have been cast as vector addition systems [21] and Petri nets [8]. Decidability issues of Petri nets are surveyed in [14], and are shown to be $\mathcal{NP}$*-complete*, directing focus towards more specific problems. We focus on a novel sub-problem,i.e., on whether there exists a bag that contains a keyword set and is accepted by a commutative grammar.

Most related to our work on matching and ranking is [12], which considers the more general problem of querying the structure of a specification using graph patterns. The paper gives a query evaluation algorithm of polynomial data complexity; the authors also consider the probability of the match in [11]. By considering each permutation of the keyword query Q as a simple graph pattern, where each node represents a keyword and nodes form a chain connected by transitive edges, and taking the union of matching specifications for each permutation, these results could be used in our setting. However, our matching algorithm is optimized for queries that are sets of keywords, and is therefore simpler and considerably more efficient than [12] for this setting (see results in Section 6). Importantly, unlike [12], our solution does not require that the input grammar be transformed for each query. For this reason, our solution can be tailored to present results using representative parse trees (Section 5), while the solution of [12] cannot.

Also closely related to our match algorithm is [2], which gives a polynomial time algorithm for checking if the intersection of a context-free string grammar (which could represent the workflow specification) and a finite automaton (which could represent the query) results in an empty grammar; however, our match algorithm has a much better average case performance since it can terminate early if a match is found.

2. MODEL

In this section, we give background and definitions that will be used throughout the paper. We start with the definition of a context-free bag grammar, its language, and what it means for a search query with "and" semantics to match a grammar. We then introduce parse trees and derivation sequences. Finally, we define the notion of a repository of context-free bag grammars.

Definition 2.1. (Context-free Bag Grammar) A *context-free bag grammar* is a grammar $G = (\Sigma, \Delta, S, R)$ where Σ is the set of symbols (variables and terminals), $\Delta \subset \Sigma$ is the set of terminals, S is the start variable and R is the set of production rules. For production $r \in R$, we denote by $head(r) \in \Sigma/\Delta$ the head of the production and $body(r)$ the bag of symbols in the body of the production. The language of the grammar, $L(G)$, is a set of bags whose elements are in Δ: $L(G) = \{w \in \Delta^* | S \stackrel{*}{\Longrightarrow} w\}$

We use context-free bag grammars to represent keyword-annotated scientific workflows, of the kind described in Figure 1, and with the corresponding grammar given in Figure 2. This grammar was derived by replacing each composite module (variable) with production rules that emit their keywords (as in rules r_6 and r_7 for module M_4), and adding a production rule for each atomic module (terminal) that emits its keyword (as in rules r_8 through r_{13}).

In the remainder of the paper, we will refer to a context-free bag grammar simply as a grammar, and will use $productions(M) \subseteq R$ to denote the set of productions with M in the head. For clarity, we also label productions.

Definition 2.2. (Match) Given a grammar $G = (\Sigma, \Delta, S, R)$ and a keyword query $Q \subseteq \Delta$, we say that G *matches* Q iff there exists some $X \in L(G)$ such that $Q \subseteq X$.

Example 2.1. *Consider the grammar* $G = (\{S, A, B, C, s_1, s_2, b, c\}, \{s_1, s_2, b, c\}, S, R)$*, where* R *is:*

$$r_1 : S \Longrightarrow \{A, S\} \qquad r_5 : B \Longrightarrow \{b\}$$
$$r_2 : S \Longrightarrow \{s_1\} \qquad r_6 : C \Longrightarrow \{B\}$$
$$r_3 : S \Longrightarrow \{s_2\} \qquad r_7 : C \Longrightarrow \{c\}$$
$$r_4 : A \Longrightarrow \{B, C\}$$

The language of G *is* $L(G) = \{(s_1 + s_2)(bc + bb)^n | n \in \mathbb{N}_{\geq 0}\}$*.* G *matches query* $Q_1 = \{s_1, b\}$ *since* $Q_1 \subseteq \{s_1, b, b\} \in L(G)$*. However,* G *does not match* $Q_2 = \{s_1, s_2\}$*.*

Since the language of a grammar can be infinite, we will need to focus our attention on a small sample of its elements in which a match can be found. For this, we will use the notion of parse trees and derivation sequences.

Definition 2.3. (Parse Tree) A parse tree T associated with a grammar $G = (\Sigma, \Delta, S, R)$ is a finite unordered tree where each interior node represents a *production* $r \in R$ and whose children represent $body(r)$, i.e. each child is either a terminal in $body(r)$ (in which case it is a leaf) or is a production whose head is a variable in $body(r)$. If T consists of a single node, then it represents a terminal in Δ. We use *root(T)*, and *leaves(T)* to denote the root production and leaves of T, respectively.

For our purposes, a parse tree can be rooted at *any* terminal or production rather than just those whose head is S. Given a parse tree T of a grammar, we denote by $paths(T)$ the bag of all root-to-leaf paths in T, $productions(T)$ the bag of productions applied in T, and $symbols(T)$ the set of symbols that appear in $productions(T)$.

We now define what portions of a keyword query a parse tree T matches in terms of the query-relevant keywords generated by T, and adapt the notion of *derivation sequence* to our setting.

Definition 2.4. (Generates) Given a grammar $G = (\Sigma, \Delta, S, R)$ and a query $Q \subseteq \Delta$, we say a parse tree T *generates* the set of matching keywords $leaves(T) \cap Q$. A symbol $M \in \Sigma$ *generates* a set $X \subseteq Q$ iff one of its parse trees generates X.

Definition 2.5. (Derivation Sequence) Given a grammar $G = (\Sigma, \Delta, S, R)$, we say a variable $A \in \Sigma/\Delta$ *derives* a symbol $B \in \Sigma$ iff there exists a parse tree T where $root(T) \in productions(A)$, and $B \in symbols(T)$. Each path from A to B in T is a sequence of productions called a *derivation sequence*. sequence $seq(A \mapsto B)$, we denote by $Set(seq(A \mapsto B))$ the bag of all productions applied in it.

If a variable A derives a symbol B, we say A is an ancestor of B and B is a descendant of A.

Definition 2.6. (Simple Derivation Sequence) A derivation sequence is *simple* iff there is no production that appears in it more than once.

Intuitively, a simple derivation sequence is a derivation sequence where a recursion is fired at most once. It disregards unnecessary recursions and provides an upperbound for the complexity results in Section 3.1.

Definition 2.7. (Distance) Given a grammar $G = (\Sigma, \Delta, S, R)$, the *distance* from a variable $A \in \Sigma/\Delta$ to symbol $B \in \Sigma$ is the length of the longest simple derivation sequence for A deriving B, denoted $d(A \mapsto B)$.The *distance* of a grammar is defined as the longest distance between any two symbols, denoted $d(G)$. It is easy to see $d(A \mapsto B) \leq d(G) \leq |R|$.

Example 2.2. *For the grammar in Example 2.1 there is only one simple derivation sequence for S to derive C, i.e. $r_1 r_4$. However, there are two simple derivation sequences to derive B, i.e. $r_1 r_4$ and $r_1 r_4 r_6$. Hence $d(S \mapsto C) = 2$ and $d(S \mapsto B) = 3$. It is also easy to check that $d(G) = 4$.*

We end this section by discussing the notion of a *repository* of bag grammars. A repository of grammars is essentially a set of grammars in which symbols (modules) may be *shared*, but must be done so consistently. We assume that all grammars are *proper*, i.e. have no underivable symbols or unproductive variables.

Definition 2.8. (Bag Grammar Repository) A *bag grammar repository* is a set of bag grammars such that for any two grammars $G_1 = (\Sigma_1, \Delta_1, S_1, R_1)$ and $G_2 = (\Sigma_2, \Delta_2, S_2, R_2)$, $\forall M \in (\Sigma_1 \cap \Sigma_2)$, $\bigcup_{\substack{r \in R_1 \\ head(r)=M}} r = \bigcup_{\substack{r \in R_2 \\ head(r)=M}} r$

3. MATCHING AND SEARCHING

In this section we give an efficient algorithm that, given a grammar G and keyword query Q, determines whether G matches Q. We start by presenting the basic algorithm, *Match*, which computes a fixpoint of subsets of matching keywords using a bottom-up approach over the hierarchy of nonterminals (Section 3.1). We then give an optimized match algorithm (Section 3.2), and extend the results to searching a *repository* of grammars.

Note that the matching problem is NP-complete with respect to combined (data and query) complexity, as shown in [12]. However, since query size is typically small (6 or fewer keywords), we focus on the *data complexity* and give a matching algorithm that is polynomial in the size of the grammar. As observed in the introduction, the algorithm in [12] could also be used here, since it considers a (more general) graph pattern query. However, our algorithm is optimized for keyword queries and is therefore simpler and considerably more efficient in our setting (see results in Section 6).

3.1 Keyword query match

We now give an algorithm, *Match* (Algorithm 1), which determines whether or not a grammar G matches a keyword query Q. To do this, *Match* builds a parse tree bottom-up until some symbol generates Q, in which case G matches Q, or until a fixpoint of query-relevant keywords is reached for each symbol, in which case G does not match Q. It can be shown that the fixpoint will be reached at or before height $O(d(G))$, and that therefore the algorithm is polynomial in the size of the grammar (although exponential in the query size due to the cost of generating sets of query-relevant keyword sets).

Match generates for each symbol $M \in \Sigma$ the set $F(M)$ of sets of query-relevant keywords for M. It does so by considering parse trees of increasing height i, and calculating for each $M \in \Sigma$ the set $F_i(M)$ of sets of query-relevant keywords for M that are derived by parse trees of height $\leq i$. For terminals $M \in \Sigma$ (which are the leaves in a parse tree), $F_0(M)$ can be calculated directly (line 2, ignore for now line 3). For variables $M \in \Sigma/\Delta$, $F_0(M) = \emptyset$ (line 4). It then calculates $F_i(M)$ by initializing it to $F_{i-1}(M)$ (line 7) or $\emptyset$, then considering each production r with M as head, taking the "product-union" of $F_{i-1}(\alpha_j)$ for each α_j in $body(r)$, and adding the resulting set of query-relevant keywords to $F_i(M)$ (lines 9-11). This continues until the query is matched by some M (line 12), or until a fixpoint is reached (for each symbol M, $F_i(M) = F_{i-1}(M)$, line 13).

Algorithm 1: *Match*

Input: a grammar $G = (\Sigma, \Delta, S, R)$ and a query $Q \subseteq \Delta$
Output: boolean
`/*` $F(M)$: `set of sets of query-relevant keywords for` M `*/`
`/*` $F_i(M) = \{leaves(T) \cap Q | root(T) \in productions(M), height(T) \leq i\}$ `*/`
begin
1 **foreach** $M \in \Delta$ **do**
2 **if** $F(M) = \emptyset$ **then** $F_0(M) \leftarrow \{\{M\} \cap Q\}$
`/* for OptMatch (Algorithm 2) */`
3 **else** $F_0(M) \leftarrow F(M)$
4 **foreach** $M \in \Sigma/\Delta$ **do** $F_0(M) \leftarrow \emptyset$
5 $i \leftarrow 0$ `// a counter for iteration times`
6 **L:** $i \leftarrow i + 1$
7 **foreach** $M \in \Delta$ **do** $F_i(M) \leftarrow F_{i-1}(M)$
8 **foreach** $M \in \Sigma/\Delta$ **do** $F_i(M) \leftarrow \emptyset$
9 **foreach** *production* $r : M \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n \in R$ **do**
`/* Construct parse trees rooted at` r `*/`
10 **foreach** $X_1 \in F_{i-1}(\alpha_1), \ldots X_n \in F_{i-1}(\alpha_n)$ **do**
11 $F_i(M).addElement(\bigcup_{j=1}^{n} X_j)$
12 **if** $\exists M \in \Sigma, Q \in F_i(M)$ **then return** *true*
13 **if** $\exists M \in \Sigma/\Delta,\ F_i(M) \neq F_{i-1}(M)$ **then** goto L
14 **return** *false*

Example 3.1. *Consider the grammar in Example 2.1 and query $Q = \{b, c\}$. Initially, $F_0(s_1) = F_0(s_2) = \{\emptyset\}$, $F_0(b) = \{\{b\}\}$, $F_0(c) = \{\{c\}\}$, and $F_0(S) = F_0(A) = F_0(B) = F_0(C) = \emptyset$ (lines 1-4). In the first iteration of* **L** *($i = 1$), we add to $F_1(S)$ the set $\emptyset$ (after processing rule r_1, r_2, r_3). After processing all other productions, we have $F_1(A) = \emptyset$, $F_1(B) = \{\{b\}\}$, and $F_1(C) = \{\{c\}\}$. We then proceed to the second iteration ($i = 2$). During this iteration, when rule r_4 is processed, we add to $F_2(A)$ (which was initialized to $F_2(A) = \emptyset$) the product of $F_1(B)$ and $F_1(C)$, at each step taking the union of the two elements (e.g. $\{b\} \cup \{c\} = \{b, c\}$), resulting in $F_2(A) = \{\{b, c\}\}$. Since $Q \in F_2(A)$,* Match *terminates. If this early termination condition were omitted, the fixpoint would have been reached in the fifth iteration, since $F_5(S) = F_4(S)$, $F_5(A) = F_4(A)$, $F_5(B) = F_4(B)$, $F_5(C) = F_4(C)$.*

We now show that the data complexity of *Match* is $O(|G| *$

$d(G)$). We start by showing that it will reach a fixpoint in $d(G) * (|Q| + 1) + |Q|$ iterations by proving that if a symbol $M \in \Sigma$ generates a set $X \subseteq Q$, then there exists a parse tree T rooted at $r \in productions(M)$ of height at most $d(G) * (|Q| + 1) + |Q|$ such that $leaves(T) \cap Q = X$.

Lemma 3.1. Given a grammar $G = (\Sigma, \Delta, S, R)$ and a query $Q \subseteq \Delta$, $\forall M \in \Sigma$, $\forall X \subseteq Q$, $height(M, X) \leq d(G) * (|Q| + 1) + |Q|$.

Proof. Note that if $M \in \Sigma$ generates $X \subseteq Q$, then the parse trees that generate X fall in one of two classes: (1) each child subtree generates a **subset** of X, in which case we say the parse tree *produces* the set X; (2) some child subtree generates X **by itself**, in which case we say the parse tree *broadcasts* X.

We denote by $height(M, X)$ ($M \in \Sigma/\Delta, X \subseteq Q$) the minimum height of parse trees of M that generate X; $height(M, X) = -1$ if M cannot generate X. Similarly, we denote by $height_p(M, X)$ ($height_b(M, X)$) the minimum height of parse trees of M that produce (broadcast) X. For $height_p(M, X)$ we have the following:

$$height_p(M, X) = \begin{cases} 0 & \text{if } M \in \Delta, X = \{M\} \cap Q \\ -1 & \text{if } M \in \Delta, X \neq \{M\} \cap Q \\ -1 & \text{if } M \in \Sigma/\Delta, |X| \leq 1 \end{cases} \quad (1)$$

$$height_p(M, X) \leq 1 + \max_{M' \in \Sigma, X' \subset X} height(M', X') \quad (2)$$

Turning to $height_b(M, X)$, we have:

$$height_b(M, X) \leq d(G) + \max_{M' \in \Sigma} height_p(M', X) \quad (3)$$

$$height(M, X) \leq \max(height_p(M, X), height_p(M, X)) \quad (4)$$

So we have $height(M, X) \leq d(G) * (|Q| + 1) + |Q|$. □

Using Lemma 3.1 we can prove the following.

Theorem 1. *The data complexity of* Match *is* $O(|G| * d(G))$.

Proof. The size of a grammar $G = (\Sigma, \Delta, S, R)$ is defined as the sum of the sizes of its productions, $|G| = \sum_{r \in R} (1 + |body(r)|)$. By Lemma 3.1 we know that the number of iterations of Algorithm 1 is bounded by $d(G) * (|Q| + 1) + |Q|$. Each iteration (loop **L**) of Algorithm 1 processes all productions, each of which takes $O(|body(r)| * 2^{|Q|})$. Since the query size is considered as a constant, this yields a total time of $O(|G| * d(G))$. □

3.2 Optimized keyword query match

We now introduce two improvements to *Match*, one of which avoids unproductive calculations introduced by variables that are fixed early, and the other of which reduces the size of $F_i(M)$.

Optimization 1: Symbol Dependencies Borrowing ideas from semi-naive Datalog evaluation, we observe that a grammar yields symbol dependencies through the head and body structure of rules, e.g. that in *Match* the start variable S will not be fixed until A is fixed (a variable M is *fixed* in the i^{th} iteration, iff $\forall j > i, F_j(M) = F_i(M)$, $\forall j < i, F_j(M) \subset F_i(M)$). The first optimization is to avoid unproductive calculations introduced by variables which are fixed early.

Given a grammar $G = (\Sigma, \Delta, S, R)$, we create a *precedence graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of symbols as follows: $\mathcal{V} = \Sigma$, and a directed edge (M, M') is added to $\mathcal{E}$ iff $\exists r \in R, head(r) = M, body(r) \ni M'$. $\mathcal{G}$ has a directed cycle iff G is recursive. Two symbols are *mutually recursive* iff they participate in the same cycle of $\mathcal{G}$. Mutual recursion is an equivalence relation on Σ, where each equivalence class corresponds to a strongly connected component of $\mathcal{G}$. Denote by $[M]$ the symbol equivalence class of M, and perform a topological sort of $\mathcal{G}$ to construct a list $[M_1], [M_2] \ldots [M_n]$. Clearly, if there is a path from $[M_j]$ to $[M_i]$ ($i < j$), then symbols in $[M_i]$ are fixed earlier than symbols in $[M_j]$; symbols in the same equivalence class are fixed in the same iteration.

Optimization 2: Set Domination We note that an element in $F_i(M)$ is *useless* if it is a subset of (*dominated by*) another element in $F_i(M)$. For example, $\{b\}$ is useless in $F_3(A) = \{\{b\}, \{b, c\}\}$ of Example 3.1. Using set domination, $|F(M)|$ decreases from $2^{|Q|}$ to $\binom{|Q|}{\lfloor |Q|/2 \rfloor}$.

Algorithm 2: *OptMatch*

Input: a grammar $G = (\Sigma, \Delta, S, R)$ and a query $Q \subset \Delta$
Output: boolean
Initialization
Precompute the symbol equivalence classes $[M]$
Precompute the topological order of the symbol equivalence classes, T_order
begin
1 **foreach** $M \in \Sigma$ **do** $F(M) = \emptyset$
2 **foreach** $[M] \in T\_order$ **do**
// Find the relevant productions of $[M]$
3 $R' \leftarrow \{r | r \in R, head(r) \in [M]\}$
4 $\Sigma' \leftarrow \bigcup_{r \in R'} (head(r) \cup body(r))$
/* all descendants have been fixed and hence are treated as terminals */
5 $\Delta' \leftarrow \Sigma' / ([M]/\Delta)$
/* $F(\cdot)$ is a global variable visible for both $OptMatch()$ and $Match()$ */
6 **if** $Match(Q, \Sigma', \Delta', M, R')$ **then return** *true*
7 **return** *false*

Algorithm 2 gives the optimized algorithm, *OptMatch*. Note that the topological order of symbol equivalence classes is query-independent and can be precomputed. Line 6 of *OptMatch* calls *Match* for the current equivalence class. $F(M)$ is global, and is used in lines 2-3 of *Match*. Set domination is captured in the *addElement* method in *Match*.

Example 3.2. *Consider the grammar in Example 2.1 and query* $Q = \{b, c\}$. *One topological order of the symbol equivalence classes is* $[b], [c], [B], [C], [s_1], [s_2], [A], [S]$ *where each of the classes consists of the symbol itself. For* $[b]$, $R' = \emptyset$, $\Sigma' = \{b\}$, $\Delta' = \{b\}$ *(lines 3-5); after calling* Match *(line 6),* $F(b) = \{\{b\}\}$. *Similarly,* $F(c) = \{\{c\}\}$, $F(B) = \{\{b\}\}$. *Now we process* $[C]$ *where* $R' = \{r_6, r_7\}$, $\Sigma' = \{C, B, c\}$, $\Delta' = \{B, c\}$. *When calling* Match *for* $[C]$, *the initialization results in* $F_0(B) = F(B) = \{\{b\}\}$, $F_0(c) = F(c) = \{\{c\}\}$ *(line 3 of Algorithm 1). At the end of* Match, *we get* $F(C) = \{\{b\}, \{c\}\}$. *We can check that* OptMatch *terminates after processing* $[A]$ *since* $Q \in F(A) = \{\{b, c\}\}$.

3.3 Searching a bag grammar repository

Given a bag grammar repository and query Q, we must retrieve all grammars that match Q. A straightforward way to do this is to run $OptMatch$ over each grammar. This way a grammar that is reused by other grammars will be processed multiple times. One solution is to union productions of all grammars to form a universal grammar. However, this grammar would be too large to fit in memory.

We therefore process grammars one by one while recognizing grammar reuse; an individual grammar can be assumed to fit in main memory. We first build inverted indexes to help identify *candidate grammars*, i.e. grammars in which every keyword of the query appears (although they may not simultaneously occur in some bag in the language of the grammar). Each index maps a keyword to a list of grammars in which the keyword appears. Given a query, we find candidate grammars by intersecting the corresponding lists.

We then process candidate grammars (using $OptMatch$) so that a grammar is always processed earlier than the grammars that reuse it. Specifically, we create a precedence graph where nodes of the graph are grammars, and there is an edge from G_i to G_j iff G_j reuses G_i (i.e. the start module of G_i appears in G_j as a variable). The graph is a DAG, since there is a temporal dimension to reuse. A topological order of the DAG is the order in which grammars are processed.

We cache intermediate results ($F(S)$) for grammars that are reused and clean them from memory when there are no unprocessed grammars that reuse them. In this way, we balance memory size and overhead of redundant computations.

4. RANKING

For large repositories of grammars, more grammars may match a query than can be shown to a user, motivating ranking. In this section we describe a relevance measure based on probabilistic grammars (Section 4.1) and develop an algorithm for computing the relevance of a grammar to a query (Section 4.2). We also describe an efficient top-k algorithm (Section 4.3).

4.1 Semantics of relevance

Probabilistic context-free grammars (PCFG) have been used in applications such as natural language processing (NLP) to analyze the probability that a string is generated by a particular grammar. It is therefore natural to use them for ranking. Although our grammars generate bags rather than strings, the formalism applies literally in our setting.

Definition 4.1. ***(Probabilistic Context-Free Bag Grammar)*** *A* probabilistic context-free bag grammar $G = (\Sigma, \Delta, S, R, \rho)$ *is a context-free bag grammar in which each production is augmented with a probability* $\rho : R \to (0, 1]$ *such that* $\forall M \in \Sigma/\Delta, \sum_{r \in productions(M)} \rho(r) = 1$.

In a PCFG, which production $r \in productions(M)$ is chosen at a given composite module M is independent of the choices that lead to M. Thus, the probability of a parse tree T, which we will denote $\rho(T)$, is the product of the probabilities of productions used in the derivation, i.e., $\rho(T) = \prod_{r \in productions(T)} \rho(r)$.

Probabilities of productions in G may be given by an expert or mined from the corpus. In this paper, we do not consider how these probabilities are derived, but note that much work on the topic has been done in the NLP community [26], and the techniques are likely applicable here.

Given a grammar G and a query Q, our goal is to compute a relevance score, denoted $score(G, Q) \in [0, 1]$, representing the likelihood of G to generate a parse tree that matches Q. We want these scores to be comparable across grammars, which would enable us to say that, if $score(G, Q) > score(G', Q)$, then G is more query-relevant than G'. Generating scores that are comparable across grammars turns out to be tricky, because, as we show next, parse trees of probabilistic context-free bag grammars may not form a valid probabilistic space.

Consider the grammar G' in Figure 4 that consists of two productions, chosen with equal probability (probabilities are indicated in parentheses). Since G' is recursive, it generates an infinite number of parse trees that match query $Q = \{a\}$. Two of these are shown in Figure 4.

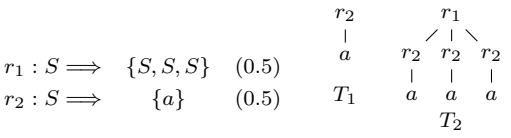


Figure 4: Recursive bag grammar

These trees have probabilities: $\rho(T_1) = 0.5$ and $\rho(T_2) = 0.5^4 = 0.0625$. Unfortunately, it is not possible to compute a normalization factor by summing the probabilities of the infinitely many parse trees, because this sum is irrational [6]. Generally, given a PCG $G = (\Sigma, \Delta, S, R, \rho)$, and a query $Q \subseteq \Delta$, it is customary to define a relevance score $score(G, Q)$ using *max* or *sum* semantics.

$$score_{sum}(G, Q) = \sum_{\substack{r(T) \in R(S) \\ Q \subseteq leaves(T)}} \rho(T) / \sum_{r(T) \in R(S)} \rho(T) \quad (5)$$

$$score_{max}(G, Q) = \max_{\substack{r(T) \in R(S) \\ Q \subseteq leaves(T)}} \rho(T) / \max_{r(T) \in R(S)} \rho(T) \quad (6)$$

Sum semantics is intuitive: we normalize the total probability of all parse trees matching the query by the total probability of all parse trees. Unfortunately, as we argued above, this value cannot be computed because probabilities may be irrational. It was shown in [15] that $score_{sum}$ may be approximated by solving a monotone system of polynomial equations. However, this approach is in PSPACE, and is very expensive in practice.

Motivated by these considerations, we opt for max scoring semantics, where the score of a grammar for a query is computed by dividing the probability of the most likely parse tree matching the query by the probability of the over-all most likely parse tree. This semantics is reasonable, and, as we will show next, $score_{max}(G, Q)$ can be computed efficiently. We refer to $score_{max}(G, Q)$ simply as $score(G, Q)$ in the remainder of the paper.

4.2 Computing the score of a workflow

We now present Algorithm 3, *Score*, which computes the relevance score of grammar G for query Q per Equation 6. This is a fixpoint algorithm that is similar in spirit to *Match*.

Note that the algorithm of [11] can also be used to compute the relevance score of grammar G for query Q. This algorithm uses a similar framework as [12], and so is very general, but is less efficient in our particular scenario. We will give experimental support to this claim in Section 6.3.

Algorithm 3: *Score*

Input: grammar $G = (\Sigma, \Delta, S, R)$ and query $Q \subseteq \Delta$
Output: double
begin
1 **foreach** $M \in \Delta$ **do**
2 **if** $F(M) = \emptyset$ **then**
$F_0(M) \leftarrow \{\{M\} \cap Q\}$; $\rho_0(M, \{M\} \cap Q) \leftarrow 1.0$
3 **else**
$F_0(M) \leftarrow F(M)$
$\forall X \in F_0(M), \rho_0(M, X) \leftarrow \rho(M, X)$
4 **foreach** $M \in \Sigma/\Delta$ **do** $F_0(M) \leftarrow \emptyset$
5 $i \leftarrow 0$ `// a counter for iteration times`
6 **L:** $i \leftarrow i + 1$
7 **foreach** $M \in \Delta$ **do**
$F_i(M) \leftarrow F_{i-1}(M)$
$\forall X \in F_i(M), \rho_i(M, X) \leftarrow \rho_{i-1}(M, X)$
8 **foreach** $M \in \Sigma/\Delta$ **do** $F_i(M) \leftarrow \emptyset$
9 **foreach** *production* $r : M \rightarrow \alpha_1\alpha_2 \ldots \alpha_n \in R$ **do**
10 **foreach** $X_1 \in F_{i-1}(\alpha_1), \ldots X_n \in F_{i-1}(\alpha_n)$ **do**
$X \leftarrow \bigcup_{j=1}^{n} X_j$ $prob \leftarrow \rho(r) \times \prod_{j=1}^{n} \rho_{i-1}(\alpha_j, X_j)$
if $X \notin F_i(M)$ **then** $\rho_i(M, X) \leftarrow prob$
else $\rho_i(M, X) \leftarrow \max(\rho_i(M, X), prob)$
11 $F_i(M).addElement(X)$
13 **if** $\exists M \in \Sigma/\Delta,\ F_i(M) \neq F_{i-1}(M)$ **then** goto L
if $Q \notin F(S)$ **then return** 0.0 `// not matching`
if $\exists M \in \Sigma, \exists X \in F_i(M),\ \rho_i(M, X) > \rho_{i-1}(M, X)$ **then** goto L
$F(M) = F_i(M)$; $\rho(M, X) = \rho_i(M, X)$
return $\rho(S, Q)/\rho_{max}(S)$

Recall from *Match* that $F_i(M)$ represents the set of sets of query-relevant keywords that module M matches, and that are derived by parse trees of height $\leq i$. *Score* uses $F_i(M)$ and a data structure $\rho_i(M, X)$, in which it stores, for each $X \in F_i(M)$, the score of the corresponding parse tree. Like *Match*, *Score* manipulates a global data structure $F(M)$; *Score* also maintains the corresponding global $\rho(M, X)$. These data structures will become important when we consider an optimization, called *OptScore*.

Algorithm *Score* starts by storing the set of query-relevant keywords annotating each terminal module M in $F_0(M)$, and by recording the probability score of 1.0 in ρ_0. Next, for non-terminal modules, $F_0(M)$ is initialized to an empty set. The bulk of the processing happens next, where, at each iteration i, we consider each production r rooted at M, and generate all sets of query terminals resulting from parse trees of height at most i rooted at M. We record the resulting subset of query keywords X if this subset has not been seen before, or it if is the highest-scoring parse tree for this subset at the current round. We compute the probability score of a parse tree resulting from production r as the product of the probabilities of its subtrees and the probability of r.

Score terminates when either no new subsets of the query are generated, or no better (more probable) derivations of existing subsets are found. *Score* returns the normalized probability of the most probable derivation tree matching Q. Note that normalization factor $\rho_{max}(S)$ is query-independent and can be computed by invoking $Score(G, \emptyset)$. We compute $\rho_{max}(S)$ offline and store it for future use.

Example 4.1. *Consider the grammar in Example 2.1 and assume that productions with the same head are equally likely. Given the query* $Q = \{b, c\}$, Score *calculates* $\rho_{max}(S) = 1/3$, $\rho(S, Q) = \frac{1}{3} * \frac{1}{3} * \frac{1}{2}$ *and returns* $\frac{1}{3} * \frac{1}{3} * \frac{1}{2}/\rho_{max}(S) = \frac{1}{6}$ *when it terminates at the end of the* 5^{th} *iteration.*

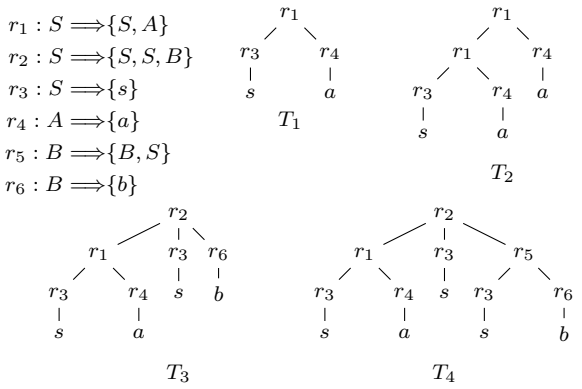


Figure 5: An example of a grammar.

The worst-case running time of *Score* is polynomial in the size of the grammar. This can be shown by a similar argument as for *Match* (Section 3.1), and is based on the observation that, for any symbol M, the height of the most probable parse tree generating any subset of Q is bounded by $\leq d(G) * (|Q| + 1) + |Q|$.

We also developed an optimized version of *Score*, which we call *OptScore*. We do not detail the *OptScore* algorithm here, but note that it is based on the two optimizations performed in *OptMatch*. The first optimization, **symbol dependencies**, identifies equivalence classes of modules of G based on their reuse of variables on the right-hand-side of productions. *OptScore* computes a topological ordering among equivalence classes, and runs algorithm *Score* for each class in reverse topological order, saving intermediate results in global data structures $F(M)$ and $\rho(M, X)$. The second optimization, **set domination**, includes probabilities in the notion of domination: a set $X_1 \in F_i(M)$ dominates $X_2 \in F_i(M)$ iff $X_1 \supset X_2$ and $\rho_i(M, X_1) \geq \rho_i(M, X_2)$.

4.3 Identifying the top-k workflows

We conclude our discussion of ranking by presenting an efficient way to retrieve, and compute the scores of, the top-k workflows in a repository. Given a repository, a query Q, and an integer k, a naive approach is to compute $score(G, Q)$ using, e.g., *OptScore*, sort grammars in decreasing order of score, and return the top-k. Here, *OptScore* may be executed on the entire repository, or only on its promising subset, leveraging an inverted index that maps each keyword to the set of workflows in which it appears. Even assuming that only the grammars matching Q are considered (i.e., grammars for which *Match(G,Q)* returns true), this naive approach will still require us to compute $score(G, Q)$ for many more than k grammars.

We use the *Threshold Algorithm* (TA) [16] to limit the number of score computations. Our use of TA is based on the observation that $score(G, Q) \leq \min_{X \subset Q} score(G, X)$. In particular, $score(G, Q)$ is at most as high as the score of G for any single-keyword subset of Q.

We leverage this observation and build inverted lists, one per keyword a, storing all grammars G that match a, in decreasing order of $score(G, \{a\})$. Then, given a multi-keyword query Q, we access the query-relevant lists sequentially in parallel, and compute $score(G, Q)$ for the first k

grammars. We refer to the current k^{th} highest score as θ, and we update θ as the algorithm proceeds.

We consider grammars in inverted list order, and, when an unseen grammar G is encountered, retrieve its entries from all inverted lists with random accesses, and compute the score upper-bound $ub(G,Q) = \min_{a \in Q} score(G, \{a\})$. If $ub(G,Q) > \theta$, we compute $score(G,Q)$ using an algorithm from Section 4.2 and update θ if necessary. TA terminates when the score upper-bound of unseen grammars, computed as the minimum of current scores in the relevant inverted lists, is lower than θ.

5. RESULT PRESENTATION

Since grammars may be large and complex objects, it is important to develop presentation mechanisms that help the user understand where keyword matches occur in the result grammars. Interestingly, a single grammar may match a query in many ways, more than can be shown to a user. In Section 4 we proposed to compute probabilities of parse trees, and to use these probabilities to rank grammars. In this section, we build on this idea and propose to choose the most probable parse trees that are structurally irredundant. We refer to such trees as *representative parse trees*, and describe them in Section 5.1. We then give an algorithm for finding the top-k representative parse trees for a given grammar in Section 5.2.

5.1 Representative parse trees

Recursion gives rise to structural redundancy. Consider the grammar in Figure 5, and its parse trees T_1 and T_2. These trees both match query $Q = \{s, a\}$. Both trees fire the same productions in the same order, and, while T_1 cuts through the chase, T_2 loops by firing r_1 twice in sequence, and by generating a along the path $S \rightarrow A \rightarrow a$ twice. The concept of a *representative parse tree* (*rpTree* for short), which we define next, models the intuition that, while both trees match the query in the same way (by firing the same productions in the same order), T_1 is more concise than T_2.

For convenience, we will sometimes represent parse trees as bags of paths, denoted $paths(T)$. Recall that a path is a sequence of productions that ends with a terminal, e.g., r_1r_3s is a path in T_1 in Figure 5. We can represent T_1 as $paths(T_1) = \{r_1r_3s,\ r_1r_4a\}$. Note that the paths representation may be ambiguous i.e. the same bag of paths may correspond to two different parse trees (See Figure 6).

Definition 5.1. (Path Subsumption) Path p subsumes path p', denoted $p \prec p'$ if p is a sub-sequence of p'.

For example, $r_1r_3s \prec r_1r_1r_3s$. Note that, since a path ends with a terminal, if $p \prec p'$ then the paths must end in the same terminal. We use path subsumption to define the main concept of this section, parse tree subsumption.

Definition 5.2. (Parse Tree Subsumption) Parse tree T *subsumes* parse tree T', denoted $T \prec T'$, iff $head(root(T)) = head(root(T'))$ and there exists an *onto* mapping from $paths(T')$ to $paths(T)$, in which, if $p' \in paths(T')$ is mapped to $p \in paths(T)$ then $p \prec p'$.

For example, consider the parse trees in Figure 5. Observe that $T_1 \prec T_2$ according to Definition 5.2. The onto mapping from $paths(T_2)$ to $paths(T_1)$ is: $r_1r_1r_3s \rightarrow r_1r_3s$, $r_1r_1r_4a \rightarrow r_1r_4a$ and $r_1r_4a \rightarrow r_1r_4a$. In contrast, no subsumption holds between parse trees T_2 and T_3.

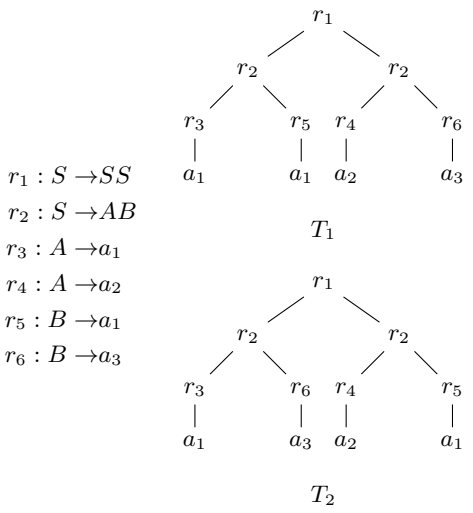


Figure 6: $paths(T_1) = paths(T_2)$ while $T_1 \neq T_2$

Definition 5.3. (Representative Parse Tree) A *representative parse tree* (*rpTree*) T of a grammar G is a parse tree s.t. there does not exist a parse tree T' of G that subsumes T.

We now list several important properties of *rpTree*s. Given a path p, we denote by $len(p)$ the length of p.

Theorem 2. *A grammar G matches a query Q iff there exists an* rpTree *T in G that generates Q.*

Proof. The forward direction is trivial. For the reverse, recall that if a grammar G matches Q, then Q must be contained in the leaves of some parse tree for G. Then it must also be contained in the leaves of an *rpTree*, since an *rpTree* generates the same terminal set as the trees that it subsumes. □

Lemma 5.1. If a parse tree is representative, then all its subtrees are also representative.

Proof. Proof is by contradiction. Let T be an *rpTree*. Suppose a subtree of T denoted by T_{sub} is subsumed by tree T'_{sub}. Let T' be the tree obtained from T by replacing T_{sub} with T'_{sub}. It is easy to see that $T' \prec T$, which contradicts that T is an *rpTree* tree. □

The converse is not true, see T_2 in Figure 5 for a counterexample. One can verify that the two child subtrees of T_2 are *rpTree*s. T_2 however is subsumed by T_1 and hence is not representative.

Lemma 5.2. Given parse trees T and T', if $T \prec T'$ then $height(T) \leq height(T')$.

Proof. Let p_{max} be one of the longest paths in $paths(T)$. Then $height(T) = len(p_{max})$. Since $T \prec T'$, for any path $p \in paths(T)$, there is a path $p' \in paths(T')$ s.t. $p \prec p'$. Let p' be a path in $paths(T')$ that p_{max} subsumes. Note that $len(p) \leq len(p')$ if $p \prec p'$. Then $height(T) = len(p_{max}) \leq len(p') \leq height(T')$. □

Consider trees T_3 and T_4 in Figure 5. These trees are of the same height, yet $T_3 \prec T_4$.

Lemma 5.3. If $T \prec T'$ and $height(T) = height(T')$, then T and T' must be rooted at the same production.

Proof. Let p_{max} be one of the longest paths in T. Since $T \prec T'$, there exists a path $p' \in paths(T')$ s.t. $p_{max} \prec p'$. Thus $height(T) = len(p_{max}) \leq len(p') \leq height(T')$. Since $height(T) = height(T')$, $len(p_{max}) = len(p')$. Recall that $p_{max} \prec p'$. Thus $p_{max} = p'$. Since p_{max} (p') starts with the root production of T (T'), T and T' are rooted at the same production. □

Theorem 3. *Given trees T and T', the time complexity of checking if $T \prec T'$ is polynomial in tree size.*

Proof. We prove the theorem by giving a polynomial algorithm of checking if $T \prec T'$.

The algorithm starts by computing a mapping $f : paths(T') \rightarrow paths(T)^*$ such that for any path $p' \in paths(T')$, $f(p') \ni p$ iff $p \prec p'$. Note that if $\exists p' \in paths(T'), f(p') = \emptyset$ then $T \not\prec T'$.

We then build a flow network s.t. the network has maximum flow $|paths(T)|$ iff there exists a **surjective** mapping $g : paths(T') \rightarrow paths(T)$ s.t. if $g(p') = p$, $p \prec p'$. The flow network is built as follows. Let $N = (V, E)$ be a network with s, t being the source and the sink of N, respectively. For each path $p \in paths(T)$ ($p' \in paths(T')$), there is a node p (p') $\in V$. For each $p \in paths(T)$, there is an edge from p to sink t in E. For each $p' \in paths(T')$, there is an edge from source s to p' in E and an edge from p' to p for any $p \in f(p')$. Every edge has a capacity of 1. It is easy to see that N has maximum flow $|paths(T)|$ iff such g exists.

It is easy to see that $T \prec T'$ iff $\forall p' \in paths(T'), f(p') \neq \emptyset$ and g exists.

Note that given two paths p, p', checking if $p \prec p'$ can be done in time $len(p) + len(p')$, i.e. $O(height(T'))$. It takes $O(|paths(T)| * |paths(T')| * height(T'))$ to compute f. In the worst case, the size of f could be $|paths(T)| * |paths(T')|$. It takes $O(|paths(T)| * |paths(T')|)$ to build the flow network. Using the Ford-Fulkerson algorithm [22], it takes $O(|paths(T)|^2 * |paths(T')|)$ to compute the maximum flow. In total, the algorithm has a time complexity $O(|paths(T)| * |paths(T')| * height(T') + |paths(T)|^2 * |paths(T')|)$. □

Lemma 5.4. Given a terminal **set** of a grammar, the height of *rpTree*s that generate it may be exponential in grammar size.

Proof. Consider the grammar:

S→ AS | s A→ B | C B→ D
C→ D D→ E | F E→ a F→ a

There are exponential (in grammar size) number of different paths for A to derive $\{a\}$, precisely $2^2 = 4$. By firing $S \rightarrow AS$ once, we get one instance of A and the height increases by 1. Consider a tree that is derived by firing $S \rightarrow AS$ 4 times where instances of A derive $\{a\}$ distinctly. One can verify that the tree is an *rpTree*. And height of the tree is exponential in grammar size. □

Lemma 5.5. All parse trees of non-recursive grammars are *rpTree*s.

Lemma 5.6. Given two parse trees T and T', if $T \prec T'$ then $\rho(T) > \rho(T')$.

Lemma 5.7. A tree may be subsumed by two different *rpTree*s.

Proof. Consider grammar:

$r_1 : S \rightarrow$ SS $r_2 : S \rightarrow$ AAA
$r_3 : A \rightarrow a_1$ $r_4 : A \rightarrow a_2$

There are three trees $T_1 = \{r_2 r_3 a_1, r_2 r_3 a_1, r_2 r_4 a_2\}$, $T_2 = \{r_2 r_3 a_1, r_2 r_4 a_2, r_2 r_4 a_2\}$, $T_3 = \{r_1 T_1, r_1 T_2\}$ where T_1, T_2 are representative, and $T_1 \prec T_3$, $T_2 \prec T_3$. □

5.2 Identifying top-k representative parse trees

We now describe an algorithm for identifying top-k *rpTree*s of grammar G that generate Q. This is a bottom-up algorithm that progressively builds *rpTree*s of height at most i by combining *rpTree*s of lower height. Correctness of such a procedure is based on Lemma 5.1.

Consider first a naive bottom-up algorithm, which first builds all possible parse trees of height i that generate Q, and then removes subsumed trees using pair-wise subsumption checks. The algorithm stops when no new *rpTree*s are found. This algorithm, while correct by Lemmas 5.1 and 5.2, will be very expensive, as there may be exponentially many *rpTree*s for a given grammar, which would all have to be retained until fixpoint, and against which all newly generated trees would need to be checked for subsumption. Yet, since our goal is to find only the top-k highest-scoring *rpTree*s, most of these would be discarded at the end of the run.

Thus, to control the running time and the space overhead, we designed an algorithm that keeps a bounded number of *rpTree*s in memory. As another naive approach, consider an algorithm that keeps up to a fixed number of highest-scoring *rpTree*s found so far in a buffer. When a new tree T is constructed, the algorithm checks whether any *rpTree* in the buffer subsumes it and, if not, assumes that the new tree is an *rpTree*. This algorithm is straight-forward, but it may return trees that are not representative. This will happen if there exists a tree $T' \prec T$, yet T' was not retained in the buffer from the previous round. Fortunately, we can use Lemma 5.6, stating that a tree can only be subsumed by a tree with a higher probability score, to devise an algorithm that is both correct and uses bounded buffers. This is Algorithm 4, which we now describe.

The algorithm uses the following data structures. Denote by $T = \langle r, T_1 \dots T_n \rangle$ a parse tree rooted at production r with $T_1 \dots T_n$ as subtrees. Also denote by $\mathcal{T}_i(M, X)$ the *rpTree*s of height $\leq i$ rooted at M and generating $X \subseteq Q$. $\mathcal{T}_i(M, X)$ is an ordered list of *rpTree*s sorted by decreasing score. The size of $\mathcal{T}_i(M, X)$ is bounded by some constant c, and we refer to this data structure as the bounded buffer.

We associate with $\mathcal{T}_i(M, X)$ a boolean function $isTrunc_i(M, X)$, indicating whether the list $\mathcal{T}_i(M, X)$ was truncated to accommodate bounded size. Importantly, we also associate with $\mathcal{T}_i(M, X)$ a score lower-bound, denoted $LB_i(M, X)$, set as follows:

$$LB_i(M, X) = \begin{cases} 0 & \neg isTrunc_i(M, X) \\ MIN_{T \in \mathcal{T}_i(M,X)} \rho(T) & isTrunc_i(M, X) \end{cases}$$

$LB_i(M, X)$ represents the lowest score of any parse tree rooted at M and generating X for which we can confidently state whether it is subsumed by any *rpTree* currently in $\mathcal{T}_i(M, X)$. Intuitively, if no truncation took place, then we can check all trees for subsumption ($LB_i(M, X) = 0$). If some *rpTree*s were not retained, then we can only check for subsumption of trees that have a higher score

Algorithm 4: *topKRep*

Input: a grammar $G = (\Sigma, \Delta, S, R)$, a query $Q \subseteq \Delta$
k, buffer size c
Output: a set of top-k *rpTrees*
begin

```
1   foreach M ∈ Δ, X ⊆ Q do
2       if X = {M} ∩ Q then T_0(M, X) ← {⟨M⟩}
3       else T_0(M, X) ← ∅
4   foreach M ∈ Σ/Δ, X ⊆ Q do T_0(M, X) ← ∅
5   i ← 0
6   L: i ← i + 1
7   foreach M ∈ Σ, X ⊆ Q do
8       T_i(M, X) ← T_{i−1}(M, X)
9   foreach M ∈ Σ/Δ do
10      findNewTrees(M, i, c)
11  if ∃M ∈ Σ, X ⊆ Q, T_i(M, X) ≠ T_{i−1}(M, X) then
12      goto L   // fixpoint
13  if |T_i(S, Q)| ≥ k then return T_i(S, Q).subList(0, k)
14  else if !isTrunc_i(S, Q) then return T_i(S, Q)
15  else topKRep(G, Q, k, 2 ∗ c)
```

than the lowest-scoring *rpTree* in the buffer ($LB_i(M, X) = MIN_{T \in \mathcal{T}_i(M,X)} \rho(T)$).

Algorithm 4 (*topKRep*) finds the top-k *rpTrees* for G matching Q. The most interesting part of the algorithm is in line 10, in the call to procedure $findNewTrees(M, i, C)$. We omit algorithmic details of this procedure due to lack of space, and describe it in text.

Procedure $findNewTrees(M, i, C)$ identifies new *rpTrees* of height up to i rooted at M generating $X \subseteq Q$. For a given X, we first construct candidate trees by considering all productions $r \in productions(M)$. A production can generate X in multiple ways, by combining different sets $X_1 \cup \ldots \cup X_n = X$. Each combination yields several parse trees, and we can find the top-scoring trees among them.

Consider for example production $r : M \rightarrow \{A, B\}$, and suppose that A can generate $\{a\}$, while B can generate $\{b\}$ or $\{a, b\}$. Then this production can generate $\{a, b\}$ in two ways: as $\{a\} \cup \{b\}$ or $\{a\} \cup \{a, b\}$.

Suppose now that, to generate $\{a, b\} = \{a\} \cup \{b\}$ we combine $T_1^A \in \mathcal{T}_{i-1}(A, \{a\})$ with $T_1^B \in \mathcal{T}_{i-1}(B, \{b\})$, deriving a tree $T_1^{AB} = \langle r, T_1^A, T_1^B \rangle$ with probability ρ_1. Alternatively, we may use the combination $\{a, b\} = \{a\} \cup \{a, b\}$, combining $T_1^A \in \mathcal{T}_{i-1}(A, \{a\})$ with $T_2^B \in \mathcal{T}_{i-1}(B, \{a, b\})$, deriving a tree $T_2^{AB} \langle r, T_1^A, T_2^B \rangle$ with probability ρ_2.

It is not guaranteed that T_1^{AB} and T_2^{AB} are the top-2 trees for M generating $\{a, b\}$. This is because $\mathcal{T}_{i-1}(A, \{a\})$ may have been truncated. For example, we may have removed $T_2^A \in \mathcal{T}_{i-1}(A, \{a\})$, which, if used to construct $T_3^{AB} = \langle r, T_2^A, T_1^B \rangle$, would have probability $\rho_3 > \rho_2$. We could be sure that no such tree T_3^{AB} exists if either $\mathcal{T}_{i-1}(A, \{a\})$, $\mathcal{T}_{i-1}(B, \{b\})$, $\mathcal{T}_{i-1}(B, \{a, b\})$ were not truncated, or if the new tree had a higher score than $\rho(r) * MAX(LB_{i-1}(A, \{a\}) * LB_{i-1}(B, \{b\}), (LB_{i-1}(A, \{a\}) * LB_{i-1}(B, \{a, b\}))$.

Similar reasoning is used when multiple productions are combined to generate $\mathcal{T}_i(M, X)$.

We note that that we implemented a more efficient version of *topKRep* for non-recursive grammars. Recall from Lemma **??** that all parse trees of a non-recursive grammar are representative. We can directly construct the highest-scoring parse trees by combining highest-scoring subtrees. No subsumption checks are required in the process. This algorithm is straight-forward, and its details are omitted.

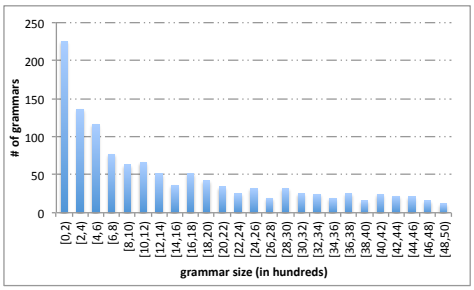


Figure 7: Distribution of grammar sizes in the repository.

An interesting point to note is that the top-k *rpTrees* in non-recursive grammars are made up of subtrees that are themselves top-k *rpTrees*. This is not necessarily the case for recursive grammars, which makes the *topKRep* algorithm less efficient in the recursive case.

6. EXPERIMENTAL EVALUATION

6.1 Experimental setup

All experiments were implemented in Java 6 and performed on a local PC with Intel Core i7 3.4GHz CPU and 4G memory running Linux. Experiments were executed against memory-resident data structures. All reported running times are averages of 5 executions per setting.

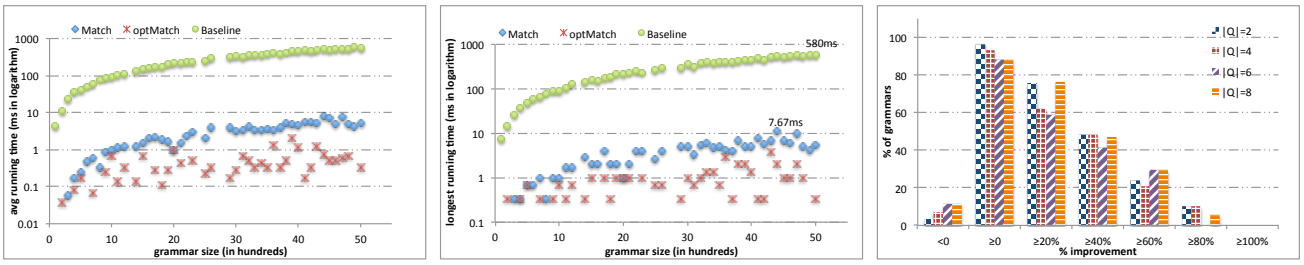
Dataset. We implemented a workflow generator that creates a repository containing a mix of recursive and non-recursive grammars, of which some are stand-alone, while others reuse existing workflows as modules. Repository size, workflow characteristics, and the amount of reuse are specified as generator parameters. All experiments in this section were executed with the following parameter settings. A simple workflow has at most 5 modules; a given module has a probability of 0.6 to be composite, and a probability of 0.4 to occur multiple times within the workflow. Each composite module has at most 3 productions, each simple workflow has a probability of 0.5 to be recursive, and each grammar reuses at most 5 other grammars.

Using these parameter settings, we generated a repository consisting of about 1,200 grammars. The distribution of grammar sizes in the repository is shown in Figure 7. The size of a grammar ranges from 10 to 5,000, and most grammars have size smaller than 1,000. (Note that grammar size is defined as the total number of symbols in its productions.)

Our choice of parameters is based on our analysis of *myExperiment.org*, the largest public repository of scientific workflows, and on [31], where it was observed that most current workflows are small.

Next, we generated keyword annotations for the workflows in the repository using results of keyword co-occurrence analysis of [32]. This analysis was based on *myExperiment.org*, where users tag workflows in support of keyword search. In [32] we used topic mining to extract 20 topics from the repository, with each topic defining a probability distribution over the tags. Here, we take 20 most frequent keywords per topic, and use their probabilities to achieve a realistic keyword assignment to workflow modules. Given a workflow, the repository generator first randomly chooses a topic, and then assigns at most 3 keywords to each module in accordance with the topic's probability distribution.

Queries. We experimented with many different queries,



(a) *Average* running time of $Match$ for Q_2. Point (x, y) represents running time over grammars of size in $[x - 100, x)$.

(b) *Longest* running time $Match$ for Q_2. Point (x, y) represents running time over grammars of size in $[x - 100, x)$.

(c) Improvement of $OptMatch$ over $Match$.

Figure 8: Performance of $Match$ and $OptMatch$.

generated by first randomly choosing a topic, and then drawing between 2 and 8 keywords according to the topic's probability distribution. Due to space constraints, we show only representative results. Unless otherwise noted, all experiments use three queries described below.

$Q_1 = \{text\ mining, e\text{-}lico, workflow\ component\}$ consists of 3 most frequent keywords from its topic, and retrieves workflows with text mining components, contributed by members of *e-lico* (an e-laboratory for collaborative research). $Q_2 = \{TerMine, text\ encoding, input\}$ looks for possible inputs to $TerMine$ (a web demonstration service). $Q_3 = \{input, xml\ invalid, read\ file\}$ is a more technical query. In experiments that focus on scalability, we work with 7 additional queries that contain up to 8 keywords, and where larger queries are supersets of smaller queries.

6.2 Keyword query match

We now evaluate the performance of algorithms described in Section 3. We show that $Match$ (Algorithm 1) runs in time polynomial in the size of the grammar, and that the $OptMatch$ optimization (Algorithm 2) is effective for the vast majority of queries. We then discuss the effectiveness of the sharing optimization (Section 3.3).

We first evaluate the performance of $Match$ compared with the general method that intersects a given grammar G with a query Q represented by a finite state automaton [2] or a graph chain pattern [12]. An adaption of the method to our scenario (called *Baseline* in Figure 8) works as follows. First, we transform grammar G to grammar G', where each production has at most two symbols on the right-hand side. Having the grammar in this form guarantees quadratic data complexity of the algorithm, and is done off-line. Next, we intersect G' with Q to construct a new grammar G'', where 1) for each production $M \rightarrow \alpha_1\alpha_2$ in G', add a production $(M, X) \rightarrow (\alpha_1, X_1)(\alpha_2, X_2)$ to G'', for each $X \subseteq Q$, $X_1 \cup X_2 = X$, making (M, X), (α_1, X_1), (α_2, X_2) symbols in G''; and 2) for each terminal M in G', mark the symbol $(M, \{M\} \cap Q)$ in G'' as a terminal. Having constructed G'', the algorithm checks whether its language is empty in time linear in grammar size [19]. G matches Q iff the language of G'' is not empty.

Figures 8a and 8b demonstrate the average and longest running time of $Match$ and *Baseline* for query Q_2 for grammars of different sizes. Some 154 grammars in our repository contain all keywords of Q_2, and we run the algorithms on these grammars. According to Figure 8a and 8b, $Match$ runs in time polynomial in grammar size, as expected. The running time of the algorithm is reasonable, and is below 10ms for all grammars. We observed similar trends for other queries. Although *Baseline* and $Match$ both run in time polynomial in grammar size, $Match$ significantly outperforms *Baseline* in all cases, because it terminates early if a variable other than the start symbol matches the query.

Figure 8c shows that $OptMatch$, which is an optimization of $Match$, is effective at reducing the running time for most queries. For example, $OptMatch$ outperforms $Match$ by at least 20% for 80% of 2-keyword queries. $OptMatch$ slightly increases running times for some queries. We also measured the total running time of $Match$ and $OptMatch$ for a variety of queries, and for all workflows in the repository. We found that $OptMatch$ brings an over-all gain of at least a factor of 2 for queries of size between 2 and 8. For example, the total running time of $Match$ for queries of size 2 is 77.67ms, compared to 43ms for $OptMatch$.

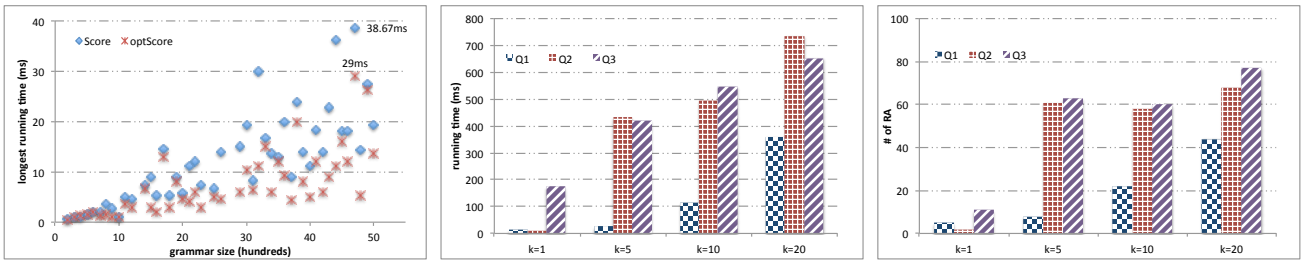
Finally, we measured the effectiveness of leveraging grammar reuse, when executing $Match$ and $OptMatch$ on all workflows in the repository (see Section 3.3). We found that this optimization, to which we refer as *sharing*, is extremely effective, bringing the total running time of $Match$ to between 130ms and 150ms for queries of size 2 to 8. $Match$ and $OptMatch$ have comparable performance with this optimization. Details are omitted due to lack of space.

In summary, $Match$ and $OptMatch$ are efficient algorithms. $OptMatch$ outperforms $Match$ for most grammars, and should be used when individual grammars are tested. Either $Match$ or $OptMatch$ with the sharing optimization may be used when all workflows in the repository are tested.

6.3 Ranking of grammars

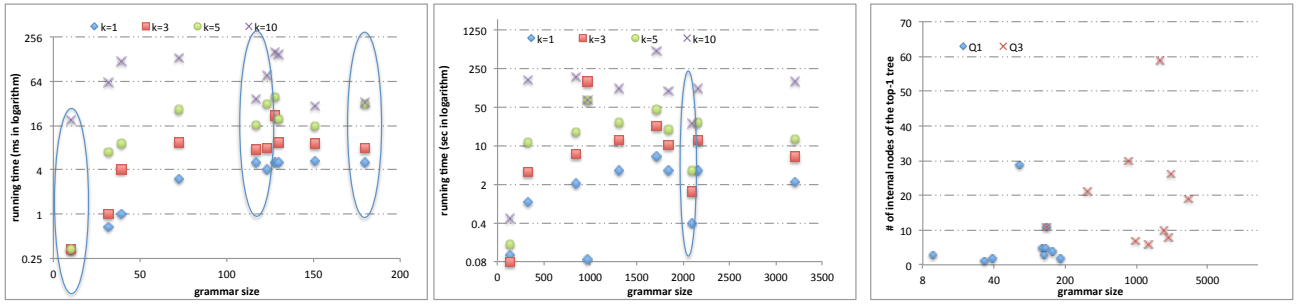
We now demonstrate that the techniques of Section 4 can be implemented efficiently. We first show that the running time of $Score$ (Algorithm 3) is polynomial in grammar size, and that $OptScore$ outperforms $Score$ in most cases. We then show that top-k workflows can be identified efficiently when TA is used to find the promising grammars.

We observed similar trends for average and longest running time of $Score$($optScore$) to that of $Match$($optMatch$). Figure 9a reports the longest running times of $Score$ and $optScore$ for grammars in our repository and demonstrates that the running time of this algorithm is reasonable, and is below 40ms for all grammars. Comparing Figure 9a with Figure 8b, we note that $Score$ is almost three times slower than $Match$. We also observe that our $Score$ algorithm outperforms *Baseline* [12] (which is used for matching). Since a scoring algorithm is necessarily slower than a matching algorithm, we conclude that a scoring algorithm that uses



(a) Longest running time of *Score* and *optScore* for Q_2 (b) Running time of TA (ms). (c) Running time of TA (RA).

Figure 9: Performance of the ranking solution: *Score*, *OptScore* and threshold algorithm (TA).



(a) Q_1 (b) Q_3 (c) Size of the top-1 tree

Figure 10: Performance of finding top-k *rpTrees* for the top-10 grammars (ellipses indicate non-recursive grammars).

a similar framework as *Baseline* will be less efficient than *Score*, and do not run a direct experimental comparison.

We also measured the improvement of *optScore* over *Score* and got a trend very similar to those observed for *OptMatch* (Figure 8c). *OptScore* results in an improvement for the vast majority of grammars, for queries of varying lengths.

Figure 9b reports the running time of Threshold Algorithm (TA), followed by an execution of *OptScore* for the promising grammars, for queries Q_1, Q_2, Q_3. We can see from Figure 9b that it takes under $100ms$ to find the top-5 grammars for Q_1, and around $400ms$ for Q_2 and Q_3. These queries all match between 150 and 180 grammars in our repository, and, as is usually the case for TA, the difference in performance is due to the distribution of scores. Figure 9c gives the running time of TA in terms of the number of random accesses (RA), demonstrating that the stopping condition for TA is reached after only a fraction of all matching grammars have been considered.

In summary, *Score* and *OptScore* are efficient algorithms, and *OptScore* outperforms *Score* for most grammars. Using TA to identify promising grammars, and then invoking *OptScore* for these grammars, allows us to achieve interactive response times when retrieving the top-k grammars from the repository.

6.4 Result presentation

Finally, we evaluate the running time and quality of *topKRep* (Algorithm 4), and show that it can be used to find the highest-scoring representative parse trees in interactive time.

Recall from Section 5 that *topKRep* is invoked on a particular grammar, typically one that is among the highest-scoring grammars for a given query, and computes a fixed number of *rpTrees* for that grammar that match the query.

Figure 10 reports the total running time of *topKRep* over top-10 grammars for queries Q_1 and Q_3, as a function of grammar size. The number of *rpTrees*, denoted k, varies from 1 to 10. Observe from Figure 10a that the top-10 grammars for Q_1 are small (< 200), and that the total running time of *topKRep* is reasonable, under $161.33ms$ for $k = 10$.

We use ellipses to indicate running times for non-recursive grammars. We noted in Section 5 that, because all parse trees of non-recursive grammars are representative, we can design an efficient version of *topKRep* for this case. The difference in running time is not significant for Q_1 (Figure 10a), but becomes more pronounced for Q_3 (Figure 10b).

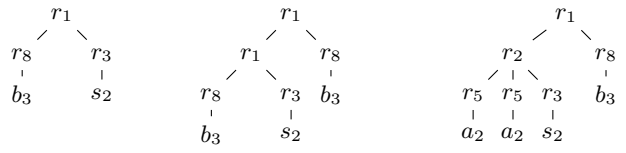
Figure 10b shows that *topKRep* is significantly slower for Q_3 than for Q_1, for three reasons. First, the top-10 grammars for Q_3 are much larger, and have larger parse trees. Figure 10c shows that sizes of top-1 trees for large grammars are usually larger than those for small grammars. Second, there are more parse trees to be constructed for large grammars. Third, it is more common for large grammars to require a larger buffer size when computing the top-k *rpTrees*. Recall that buffer size is an argument in Algorithm 4, and that the algorithm is re-executed with a larger buffer if the original setting does not yield enough *rpTrees*. For the 4^{th} grammar in Figure 10b, the required buffer size for $k = 3$ was 48 (meaning that *topKRep* was executed 5 times). This was higher than for $k = 5$ and $k = 10$, where buffer size of 40 (for 4 and 3 executions of *topKRep*, respectively) was sufficient.

We now present an example that illustrates the effectiveness of *rpTrees*. For query Q_1, the top-1 matching grammar in our repository is one where the start module is annotated with all the keywords in Q_1. Thus all parse trees of this grammar match Q_1. To simplify presentation, we eliminate the keywords of modules and show only the grammar below:

$r_1 : S \Rightarrow \{S, B\}$ $(\frac{1}{3})$ $r_2 : S \Rightarrow \{A, A, S, s_1\}$ $(\frac{1}{3})$
$r_3 : S \Rightarrow \{s_2\}$ $(\frac{1}{3})$ $r_4 : A \Rightarrow \{B, a_1\}$ $(\frac{1}{2})$
$r_5 : A \Rightarrow \{a_2\}$ $(\frac{1}{2})$ $r_6 : B \Rightarrow \{S, b_1\}$ $(\frac{1}{3})$
$r_7 : B \Rightarrow \{S, b_2\}$ $(\frac{1}{3})$ $r_8 : B \Rightarrow \{b_3\}$ $(\frac{1}{3})$

Note that the top-6 trees of this grammar are all *rpTrees*.

The 7^{th} most probable tree T_7 (with a probability 0.004) is shown in Figure 11b. Observe that T_7 is subsumed by the top 2^{nd} tree T_2 (Figure 11a), and so is not an *rpTree*. On the other hand, T_8 (with probability 0.003) is an *rpTree*, and is more interesting to show to the user than T_7, although its probability score is lower.



T_2 (0.037) T_7 (0.004) T_8 (0.003)

(a) top-2 tree (b) top-7 tree (c) top-8 tree

Figure 11: Trees for the top-1 grammar that matches Q_1

In summary, *topKRep* can be used to efficiently compute the highest-scoring representative parse trees for many grammars. For certain large grammars, *topKRep* does not terminate in interactive time, due to parse tree size, and to conservative buffer size requirements. Performance can be improved using alternative strategies for setting buffer size.

7. CONCLUSIONS

In this paper we addressed the problem of searching a repository of workflow specifications in which modules, both atomic and composite, are annotated with keywords. Since search does not interact with the graph structure of workflows, we reduced the problem to one of searching a repository of *bag grammars*. We gave an efficient polynomial-time matching algorithm with respect to *data* complexity, and extended this to search over a repository of bag grammars. We developed efficient algorithms for calculating the relevance score of a grammar to a given query, and for finding the top-k grammars for a given query. Finally, we proposed a novel result presentation method.

This work introduces a novel use of bag grammars, and shows the importance of probabilistic bag grammars. Our approach has been based on efficiency considerations; in the future we would like to gain a deeper understanding of how to use probabilistic bag grammars and continue to explore ways of presenting concise search results. Moving beyond keyword search, we would like to add structural features into queries. We also plan on testing the usability of these ideas on real datasets.

8. ACKNOWLEDGEMENTS

We thank Tova Milo for extensive discussions and feedback on this draft.

9. REFERENCES

[1] Z. Bao et al. Labeling recursive workflow executions on-the-fly. In *SIGMOD*, 2011.
[2] Y. Bar-Hillel et al. On formal properties of simple phrase structure grammars. *Language and Information: Selected Essays on Their Theory and Application*, 1964.
[3] C. Beeri et al. Querying business processes. In *VLDB*, 2006.
[4] S. P. Callahan et al. Managing the evolution of dataflows with VisTrails. In *SciFlow*, 2006.
[5] Y. Chen et al. Keyword search on structured and semi-structured data. In *SIGMOD*, 2009.
[6] Z. Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25, 1999.
[7] D. Chiu et al. Keyword search support for automating scientific workflow composition. In *SSDBM*, 2011.
[8] S. Crespi-Reghizzi and D. Mandrioli. Petri nets and commutative grammars. 74–5, Mar, 1974.
[9] B. B. Dalvi et al. Keyword search on external memory data graphs. *PVLDB*, 2008.
[10] S. B. Davidson et al. Keyword search in workflow repositories with access control. In *AMW*, 2011.
[11] D. Deutch et al. Optimal top-k query evaluation for weighted business processes. *PVLDB*, 3(1), 2010.
[12] D. Deutch and T. Milo. A structural/temporal query language for business processes. *J. Comput. Syst. Sci.*, 78(2), 2012.
[13] J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes, 1997.
[14] J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey, 1994.
[15] K. Etessami and M. Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56, 2009.
[16] R. Fagin et al. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
[17] A. Gándara et al. Knowledge annotations in scientific workflows: An implementation in kepler. In *SSDBM*, 2011.
[18] H. He et al. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.
[19] J. E. Hopcroft et al. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2003.
[20] V. Kacholia et al. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
[21] R. M. Karp and R. E. Miller. Parallel program schemata. *J. of CSS*, 3(2), 1969.
[22] J. Kleinberg and E. Tardos. *Algorithm Design.* 2005.
[23] J. Li et al. Top-k keyword search over probabilistic xml data. In *ICDE*, 2011.
[24] Z. Liu and Y. Chen. Processing keyword search on xml: a survey. *WWW*, 14(5-6), 2011.
[25] Z. Liu et al. Searching workflows with hierarchical views. *PVLDB*, 3(1-2), 2010.
[26] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing.* MIT Press, 2001.
[27] A. Nierman and H. V. Jagadish. Protdb: Probabilistic data in xml. In *VLDB*, 2002.
[28] T. Oinn et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(1), 2003.
[29] D. D. Roure et al. The design and realisation of the myExperiment virtual research environment for social sharing of workflows. *FGCS*, 25(5), 2009.
[30] Q. Shao et al. Wise: A workflow information search engine. In *ICDE*, 2009.
[31] J. Starlinger et al. (re)use in public scientific workflow repositories. In *SSDBM*, 2012.
[32] J. Stoyanovich et al. Exploring repositories of scientific workflows. In *WANDS*, 2010.
[33] J. Stoyanovich and I. Pe'er. Mutagenesys: estimating individual disease susceptibility based on genome-wide snp array data. *Bioinformatics*, 24(3):440–442, 2008.
[34] T. Tran et al. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, 2009.
[35] J. X. Yu et al. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1), 2010.