An Optimal Labeling Scheme for Workflow Provenance Using Skeleton Labels

Zhuowei Bao, Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy
Department of Computer and Information Science, University of Pennsylvania
Philadelphia, PA 19104, USA
{zhuowei, susan, sanjeev, sudeepa}@cis.upenn.edu

ABSTRACT

We develop a compact and efficient reachability labeling scheme for answering provenance queries on workflow runs that conform to a given specification. Even though a workflow run can be structurally more complex and can be arbitrarily larger than the specification due to fork (parallel) and loop executions, we show that a compact reachability labeling for a run can be efficiently computed using the fact that it originates from a fixed specification. Our labeling scheme is optimal in the sense that it uses labels of logarithmic length, runs in linear time, and answers any reachability query in constant time. Our approach is based on using the reachability labeling for the specification as an effective *skeleton* for designing the reachability labeling for workflow runs. We also demonstrate empirically the effectiveness of our skeleton-based labeling approach.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*scientific databases*

General Terms

Algorithms, Performance

1. INTRODUCTION

Efficiently maintaining the provenance of data produced by scientific workflow systems is of great current interest. Using provenance information, scientists can examine the data, parameter settings, and analysis tools that were used in an "in-silico" experiment to produce a good data result, or determine which downstream data objects were affected by a bad data result. Tracking such provenance information entails answering reachability queries on a directed acyclic graph (DAG) which encodes the dependencies between data and modules in a workflow execution.

As observed in [8, 11], two immediate approaches to answering reachability queries – using graph traversals and pre-computing the transitive closure – are prohibitively expensive for large graphs. A better approach is to use *reachability labels*, *i.e.*, to assign each vertex in the graph a label such that by comparing only the labels of any two vertices, we can decide if one can reach the other. However, the effectiveness of this approach crucially depends on the ability to develop a *compact* and *efficient* labeling scheme, where compactness refers to the space used by the labels, and efficiency refers to the time complexity of creating and comparing the labels.

Motivated by applications in XML database systems, several compact and efficient labeling schemes for trees have been developed; see, for example, [15, 13]. These labeling schemes are generally considered to be *optimal* in the sense that they use labels of logarithmic length, run in linear time, and answer queries in constant time. In contrast, any labeling scheme for general DAGs may require labels of linear length (in the number of vertices), even if arbitrary construction and query time are allowed.

However, workflow executions are not arbitrary DAGs: Each execution (a.k.a. *run*) of a workflow follows the same basic structure as the directed graph representing its specification, but may become much larger due to fork (parallel) and loop executions. Building on this observation, we propose a *skeleton-based* labeling scheme, that uses the reachability labeling for the specification for designing the reachability labeling for workflow runs. Our approach labels a run in two phases: (1) label its specification using any labeling scheme for directed graphs, and (2) extend the reachability labels on the specification (called the *skeleton labels*) with additional run-time information about the forking and looping behavior to label the run.

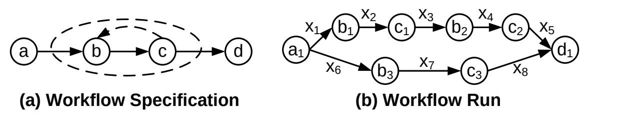

Figure 1: Workflow Example

As an example, consider the workflow specification and one of its runs shown in Figure 1, in which vertices represent modules (a.k.a. tasks) and edges indicate potential data flow between the modules. In the specification, the dotted oval around b and c indicates a *fork* and the dotted backarrow from c to b a *loop*. In the run, the fork is executed twice in parallel; in one fork copy, the loop is executed twice, while in the other fork copy it is executed only once.[1] Edges in the run are also annotated with x_i's, indicating data that was created by the first module and passed to the second.

[1]A realistic run may execute the fork and loop hundreds of times.

To illustrate the approach, consider the following three provenance queries: (1) *Does x_8 (output of c_3) depend on x_1 (input to b_1)?* Note that we need to check if c_3 is reachable from b_1. The answer is 'no', since b_1 and c_3 are in parallel fork copies. This would be encoded by the extended labels, and skeleton labels would not be checked; (2) *Does x_4 (output of b_2) depend on x_2 (input to c_1)?* The answer is 'yes' since, despite the fact that in the specification b is not reachable from c, c_1 and b_2 are in successive loop iterations. This would again be encoded by the extended labels; and (3) *Does x_3 (output of c_1) depend on x_1 (input to b_1)?* Given that b_1 and c_1 are in the same fork and loop copy, c_1 is reachable from b_1 iff c is reachable from b. This query can therefore be answered using the skeleton labels.

At first glance it would appear that extending skeleton labels to workflow runs is straightforward as long as unique process ids are given for different executions of the same module. But the example above shows that the reasoning must take into account the hierarchy of nested forks and loops within which the module execution occurs. Note that it is non-trivial to encode the interaction between nested fork and loop executions using only labels of logarithmic length; even a unique process id takes that much space. Our primary technical contribution is to design logarithmic length labels for the run in linear time, and, using these labels along with the given skeleton labels, answer reachability queries in constant time. Furthermore, we extract the hierarchy of fork and loop executions from the run graph in linear time, so our approach works even if no additional information is available to identify each fork and loop copy in the run.

The effectiveness of our skeleton-based labeling approach follows from three observations: First, the size of the specification is typically much smaller than the size of a run. Therefore, the specification can be efficiently labeled using short labels. Second, once a workflow is created it will be executed repeatedly (using different parameter settings or data inputs). Therefore, the cost (both space and construction time) of labeling a specification can be amortized over a large number of runs. Third, using our algorithm, reachability queries on the run may frequently be answered using only the extended labels without comparing the skeleton labels. Therefore, our approach is effective even when reachability queries on the specification are slow.

Contributions. The contributions of this paper are as follows. First, we propose a two-phase labeling scheme for workflows with well-nested forks and loops, which extends skeleton labels on the specification to labels on a run by capturing fork and loop executions. Second, we show that the scheme is compact and efficient, yielding logarithmic size labels that are created in linear time and which can be used to answer reachability queries in constant time. The scheme is thus *optimal* for labeling runs that conform to a given specification; it in fact matches the bounds achievable for trees even though runs can have an arbitrarily more complex network structure. Third, we extend the reachability labels to data, and show how those labels can be used to answer provenance queries which test dependencies between data or between data and modules. Finally, we perform extensive experimental evaluations on both real and synthetic datasets. The results empirically validate our complexity analysis, and show that our skeleton-based labeling scheme is robust against the approach used for labeling the underlying specification.

Paper Organization. Related work is discussed in Section 2. Section 3 introduces the workflow model and formulates the reachability labeling problem. Section 4 presents and analyzes the skeleton-based labeling scheme. Section 5 describes a linear time algorithm that is used by our labeling scheme to calculate the execution plan and context. Section 6 extends the labeling techniques to answer the data provenance. Issues of labeling the specification are discussed in Section 7, followed by experimental results in Section 8.

2. RELATED WORK

The reachability labeling problem for trees has been extensively studied. The well-known *interval* scheme was first proposed by [15], which uses labels of length at most $2 \log n$ bits, where n is the number of nodes in the tree. A considerable amount of work [1, 16, 3] has been devoted to reduce the label length. [13] also proposed a *prefix* scheme which, though not theoretically best, is likely to be the winner for labeling XML files in practice. Note that an obvious lower bound for maximum label length is $\log n$, however, the tight lower bound for this problem remains open.

The reachability labeling problem for general graphs has also attracted a lot of attention. Existing approaches fall into three categories: Chain Decomposition [10], Tree Cover [2] and 2-Hop [6]. Other variants of Tree Cover include Label+SSPI [5], Dual Labeling [18] and GRIPP [17]. The most recent work hybridizes three main labeling techniques: Path-Tree Cover [12] combines Chain Decomposition with Tree Cover; and 3-Hop [11] combines Chain Decomposition with 2-Hop. A detailed comparison is available in [11]. Thus existing research tries to find a good balance among label length (*i.e.*, index size), construction time and query time. Note that encoding general directed graphs may require labels of length $\Omega(n)$ bits, where n is the number of vertices, even if we allow arbitrary construction and query time.

To provide efficient provenance storage and queries in workflow systems, [4] identifies several factorization and inheritance techniques. [8] proposes a reachability labeling scheme to encode the dependency graph of a workflow execution by transforming a general graph into a tree and then applying the interval scheme [15]. The problem with this approach is that the size of the transformed tree may be exponential in the size of the original graph.

3. MODEL AND PROBLEM STATEMENT

We start with an informal description of our workflow model in Section 3.1 before formalizing in Section 3.2. In Section 3.3 we formulate the reachability labeling problem, and summarize the main result of this paper.

3.1 Workflow Model Description

Our workflow model has two components: A *workflow specification* defines the control and data flow between a set of modules, and serves as a template for executions; an execution of a workflow is called a *workflow run*. Both are represented by directed graphs, in which each vertex denotes a *module*, and each edge denotes a *data channel* associated with a set of *data items*. For the purpose of this paper, we treat the data item as a logical data unit used by the workflow, and the module as a black box that takes a set of data items as input and produces a set of data items as output. A data channel is directed from a module u to another module v if the associated data items are generated by u as output and then used by v as input. This graph captures the dependency between modules and data items.

To present our main labeling techniques, we first study a simplified workflow model which considers only the control flow between modules (*i.e.*, vertices) and ignores the data flow over data channels (*i.e.*, edges). Consequently, only vertices in the graph are labeled with their module names. We will extend the model to include data flow in Section 6.

Workflow Specification. A workflow specification is a directed acyclic graph whose vertices are labeled with *unique* module names. Without any loss of generality, we assume that there exist a single source (a vertex with no incoming edges) and a single sink (a vertex with no outgoing edges). The source denotes a virtual start module that sends out all source data and initializes the execution; similarly, the sink denotes a virtual finish module that collects all final data products and stops the execution. Due to the presence of these two terminals, every module in the workflow lies on some path from the source to the sink. In addition, we are given two sets of subgraphs of the specification, called *forks* and *loops*, which are allowed to be executed repeatedly in a parallel and serial manner respectively. These fork and loop subgraphs must be (1) *self-contained*: each subgraph has a single source and a single sink, and there are no edges coming into or going out from this subgraph through any internal vertices (see Definition 1); and (2) *well-nested*: for any two subgraphs, either one contains another or they are disjoint (see Definition 2).

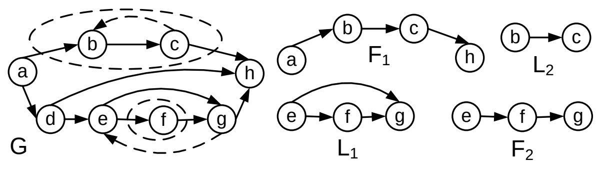

Figure 2: Workflow Specification $(G, \mathcal{F}, \mathcal{L})$

Example 1. A simple workflow specification $(G, \mathcal{F}, \mathcal{L})$ is shown in Figure 2, where G denotes the specification graph, and $\mathcal{F} = \{F_1, F_2\}$ and $\mathcal{L} = \{L_1, L_2\}$ denote two sets of well-nested fork and loop subgraphs respectively. The letter inside each vertex indicates its module name. We also use the name as a unique identifier for each vertex in G. We denote a fork subgraph by a dotted oblong which includes only the internal vertices, and a loop subgraph by a dotted backward edge directed from the sink to the source.

Workflow Run. Each run of a workflow must follow all the paths in the specification, and may repeatedly execute the given fork and loop subgraphs. A *fork* execution replicates one or more copies of a fork subgraph, and combines them in *parallel*; a *loop* execution replicates one or more copies of a loop subgraph, and combines them in *series* (see Definition 4). Two copies of the same fork or loop may differ due to the presence of some inner forks and loops.

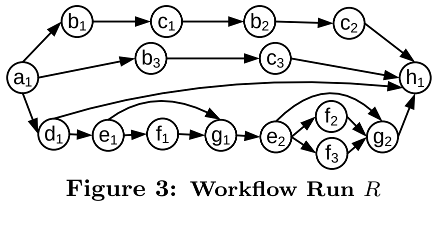

Figure 3: Workflow Run R

Example 2. Figure 3 shows a run R of the specification $(G, \mathcal{F}, \mathcal{L})$ given in Figure 2. The module names in R are not unique due to the fork and loop executions. We thus add a subscript to the module name to obtain a unique identifier for each vertex in R. Comparing Figure 3 with Figure 2, we can see that F_1 is executed twice: in one copy $(a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow b_2 \rightarrow c_2 \rightarrow h_1)$, L_2 is executed twice, while in the other copy $(a_1 \rightarrow b_3 \rightarrow c_3 \rightarrow h_1)$, L_2 is executed only once.

3.2 Workflow Model Formalization

Given a directed graph G, we denote the vertex set of G by $V(G)$, and the edge set of G by $E(G)$. We use a pair $(u, v) \in E(G)$ to denote an edge directed from a vertex $u \in V(G)$ to another vertex $v \in V(G)$. We say a directed acyclic graph G is an *acyclic flow network* if it has a single source $s(G)$ and a single sink $t(G)$. Let $V^*(G)$ be the set of internal vertices of G. That is, $V^*(G) = V(G) \backslash \{s(G), t(G)\}$.

Definition 1. (**Self-Contained Subgraph**) Given an acyclic flow network G, we say a subgraph H of G is *self-contained* if (1) H has a single source $s(H)$ and a single sink $t(H)$ $(s(H) \neq t(H))$; (2) For any $u \in V^*(H)$ and $v \in V(G) \setminus V(H)$, $(u, v) \notin E(G)$ and $(v, u) \notin E(G)$; and (3) For any $u, v \in V(H)$, $(u, v) \in E(G)$ and $(u, v) \notin E(H)$ only if $u = s(H)$ and $v = t(H)$.

A self-contained subgraph H of an acyclic flow network G is said to be (1) *atomic*, if there is no self-contained subgraph H' of G such that $s(H') = s(H)$ and $t(H) = t(H')$ and $E(H') \subset E(H)$; and (2) *complete*, if for any $v \in V(G) \setminus V(H)$, $(s(H), v) \notin E(G)$ and $(v, t(H)) \notin E(G)$.

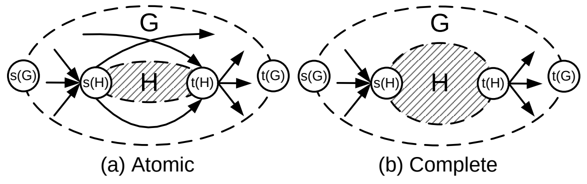

Figure 4: Self-Contained Subgraphs

Figure 4 illustrates the above definitions. Intuitively, a self-contained subgraph connects to other portions of the graph through only its source and sink. It is (1) atomic, if it forms a single branch between its source and sink (*i.e.*, it cannot be split into two parallel self-contained subgraphs) (see Figure 4(a)); and (2) complete, if it contains all branches between its source and sink. In addition, any incoming edge must go through the source and any outgoing edge must go through the sink (see Figure 4(b)).

Definition 2. (**Well-Nested Fork and Loop System**) Given an acyclic flow network G, a set $\mathcal{F}$ of atomic self-contained subgraphs of G, and a set $\mathcal{L}$ of complete self-contained subgraphs of G, we say a pair $(\mathcal{F}, \mathcal{L})$ forms a *well-nested fork and loop system* for G if (i) $\mathcal{F} \cap \mathcal{L} = \emptyset$ and (ii) for any $H_1, H_2 \in \mathcal{F} \cup \mathcal{L}$, exactly one of the following is true:

1. $\texttt{DomSet}(H_1) \subseteq \texttt{DomSet}(H_2)$ and $E(H_1) \subset E(H_2)$; or
2. $\texttt{DomSet}(H_1) \supseteq \texttt{DomSet}(H_2)$ and $E(H_1) \supset E(H_2)$; or
3. $\texttt{DomSet}(H_1) \cap \texttt{DomSet}(H_2) = \emptyset$ and $E(H_1) \cap E(H_2) = \emptyset$.

where $\texttt{DomSet}(H)$ denotes the set of vertices *dominated* by a subgraph $H \in \mathcal{F} \cup \mathcal{L}$, and is defined as

$$\texttt{DomSet}(H) = \begin{cases} V^*(H) & \text{if } H \in \mathcal{F} \\ V(H) & \text{if } H \in \mathcal{L} \end{cases}$$

Note that a fork subgraph dominates only the internal vertices, since its source and sink may be shared by other edge-disjoint fork or loop subgraphs.

Definition 3. (**Workflow Specification**) A *workflow specification* is denoted by a triple $(G, \mathcal{F}, \mathcal{L})$, where G is a uniquely labeled acyclic flow network, and $(\mathcal{F}, \mathcal{L})$ forms a well-nested fork and loop system for G.

Definition 4. (**Parallel and Serial Composition**) Given two acyclic flow networks G_1 and G_2, a *parallel composition* of G_1 and G_2 forms a new acyclic flow network G by identifying $s(G_1)$ with $s(G_2)$ and $t(G_1)$ with $t(G_2)$; and a *serial composition* of G_1 and G_2 forms a new acyclic flow network G by adding a new edge directed from $t(G_1)$ to $s(G_2)$.

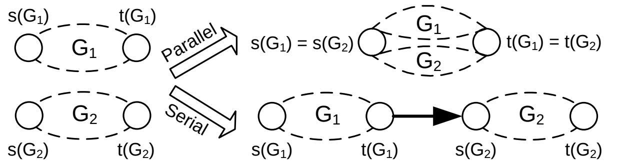

Figure 5: Parallel and Serial Composition

Figure 5 illustrates series and parallel compositions. Note that this definition can be extended to an arbitrary number of acyclic flow networks combined in a single step.

Definition 5. (**Replacement**) Given an acyclic flow network G, *replacing* a self-contained subgraph H_1 of G with another acyclic flow network H_2 will identify $s(H_1)$ with $s(H_2)$ and $t(H_1)$ with $t(H_2)$.

Definition 6. (**Workflow Run**) Given a workflow specification $(G, \mathcal{F}, \mathcal{L})$, a *workflow run* of this specification is denoted by a labeled graph R that can be derived from G by applying a sequence of the following operations:

- **Fork Execution:** Replace a fork subgraph $H \in \mathcal{F}$ with the parallel composition of two copies of H.
- **Loop Execution:** Replace a loop subgraph $H \in \mathcal{L}$ with the serial composition of two copies of H.

LEMMA 3.1. *For any workflow run R, let $\mathcal{F}'$ and $\mathcal{L}'$ be the sets of all fork and loop copies in R respectively. Then $(\mathcal{F}', \mathcal{L}')$ forms a well-nested fork and loop system for R.*

PROOF. (Sketch) This lemma can be proved by an induction on the number of fork and loop executions used to construct R. □

3.3 Problem Statement and Main Result

In this paper, we develop a compact and efficient reachability labeling scheme for workflow runs. More precisely, we will assign each vertex in the run with a *reachability label* such that, using only the labels of any two vertices, we can quickly decide if one can reach the other.

Definition 7. (**Reachability Labeling Scheme**) A *reachability labeling scheme* (in short, labeling scheme) is denoted by a triple $(\mathcal{D}, \phi, \pi)$, where $\mathcal{D}$ is the label domain, $\phi : V(G) \to \mathcal{D}$ is a labeling function (*i.e.*, encoding algorithm) that, given a directed graph G, assigns each vertex $v \in V(G)$ a reachability label $\phi(v) \in \mathcal{D}$, and $\pi : \mathcal{D} \times \mathcal{D} \to \{0, 1\}$ is a binary predicate (*i.e.*, decoding algorithm), such that for any two vertices $v, v' \in V(G)$, $\pi(\phi(v), \phi(v')) = 1$ if and only if there exists a path from v to v' in G.

The quality of a labeling scheme $(\mathcal{D}, \phi, \pi)$ is measured by three criteria: (1) *label length*: the length of reachability labels used from the domain $\mathcal{D}$; (2) *construction time*: the time taken to compute the labeling function ϕ; and (3) *query time*: the time taken to evaluate the binary predicate π.

Problem Statement. This paper addresses the problem that whether there is a labeling scheme for workflow runs that optimizes the above three parameters simultaneously.

Main Result. We develop a *skeleton-based* labeling scheme for workflow runs which uses reachability labels from the underlying specification (called the *skeleton labels*). The following theorem summarizes our main result.

THEOREM 1. **(Main Theorem)** *Given any fixed labeled specification, there exists a reachability labeling scheme for any run conforming to the specification that uses logarithmic length labels, takes linear construction time, and answers queries in constant time.*

Note that just to index n distinct nodes we need $\log n$ bits per vertex, to read the whole graph we need linear time and to answer any query we need at least constant time; in this sense our labeling scheme is *optimal.* However, in general it is impossible to obtain an optimal labeling scheme for acyclic flow networks: We can construct a family of networks such that encoding the reachability for these graphs requires labels of length $\Omega(n)$, even if we allow arbitrary construction and query time. However, our workflow runs originate from a *fixed* specification, and thus have some inherent structure that we can exploit to obtain an optimal labeling scheme.

In the rest of this paper, we will use $(G, \mathcal{F}, \mathcal{L})$ to denote a specification and R to denote a run of this specification. When $\mathcal{F}$ and $\mathcal{L}$ are clear from the context, we may simply refer to the specification as G. To illustrate our labeling techniques, we will use the running examples of specification and run given in Figure 2 and 3 respectively.

4. SKELETON-BASED LABELING SCHEME

We now present a skeleton-based labeling scheme for workflow runs. We assume that the run to be labeled is given along with its underlying specification, that is already labeled by some other labeling scheme. The main idea is described as follows. Recall that each run follows the same basic structure as the specification, but becomes much larger due to the fork and loop behavior. We will design a compact labeling scheme that encodes the basic structure of a run using the given reachability labels from its specification (*i.e.*, skeleton labels), and encode the well-nested fork and loop structure using some extended labels. Combining these two labels, we are able to efficiently answer reachability queries.

The rest of this section is organized as follows. Section 4.1 introduces some preliminary notions, namely origin, execution plan and context, which play a central role in our skeleton-based labeling scheme. Section 4.2 describes how to answer reachability queries using context and skeleton labels. A compact three-dimensional encoding for the context is presented in Section 4.3. We complete the description of our skeleton-based labeling scheme in Section 4.4, and finally present the correctness and quality analysis in Section 4.5.

4.1 Preliminaries

Origin. We start with the notion of *origin*, which establishes the correspondence between the vertices of a specification and a run based on their module names. Let $\textsc{Module}(v)$ denote the module name of a vertex $v \in V(G)$ or $V(R)$.

Definition 8. (**Origin**) Given a specification G and a run R of G, a vertex $u \in V(G)$ is said to be the *origin* of a vertex $v \in V(R)$, denoted by $u = \texttt{Orig}(v)$, if $\textsc{Module}(u) = \textsc{Module}(v)$.

Note that the origin is well-defined for each vertex in R, since the module names in G are unique. The *skeleton* label for a vertex $v \in V(R)$ then refers to the given reachability label on its origin $\texttt{Orig}(v) \in V(G)$.

Fork and Loop Hierarchy. All fork and loop subgraphs given a specification $(G, \mathcal{F}, \mathcal{L})$ are well-nested, and therefore can be captured by an unordered tree T_G, called the *fork and loop hierarchy*, where the root of T_G corresponds to the entire specification graph G, and any other node in T_G corresponds to a unique fork or loop subgraph $H \in \mathcal{F} \cup \mathcal{L}$.

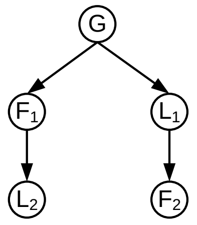

Figure 6: Fork and Loop Hierarchy T_G

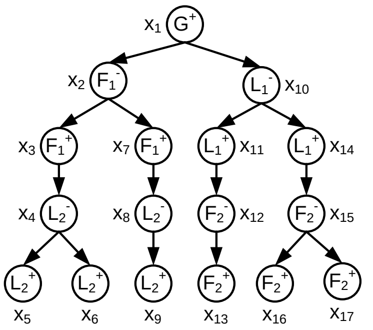

Figure 7: Execution Plan T_R

Example 3. Figure 6 shows the fork and loop hierarchy T_G for the running example, where the label inside each node indicates its corresponding subgraph.

Execution Plan. By Lemma 3.1, all fork and loop copies generated in a run R are also well-nested. We can thus similarly describe them by a tree T_R, called the *execution plan*, where the root of T_R corresponds to the entire run graph R (called a G^+ node), and any other node in T_R either corresponds to a single fork or loop copy (called an $\mathcal{F}^+$ or an $\mathcal{L}^+$ node respectively) or corresponds to all copies of the same fork or loop (called an $\mathcal{F}^-$ or an $\mathcal{L}^-$ node respectively). Note that T_R is a *semi-ordered* tree in the sense that the children of an $\mathcal{L}^-$ node are ordered while the children of any other node are unordered. In the following, we may simply refer to an $\mathcal{F}^+$ node as a $+$ or $\mathcal{F}$ node, and follow a similar convention for other types of nodes.

Example 4. Figure 7 shows the execution plan T_R for the running example, where the label inside each node indicates its type, and the label beside each node indicates a unique identifier. For instance, x_3 is an $\mathcal{F}^+$ node, corresponding to a single fork copy of F_1. Similarly, its sibling x_7 corresponds to a second copy of F_1. Their parent x_2 is then an $\mathcal{F}^-$ node, corresponding to the parallel composition of these two fork copies. Comparing Figure 7 with Figure 6, we can see that an execution plan essentially describes how many times each fork and loop subgraph is executed in the corresponding run.

For any node $x \in V(T_R)$, let $\texttt{Sub}_R(x)$ denote the corresponding fork or loop copy for x (*i.e.*, a subgraph of R), and let $\texttt{Sub}_G(x)$ denote the corresponding fork or loop subgraph for x (*i.e.*, a subgraph of G) from which $\texttt{Sub}_R(x)$ is derived.

Lemma 4.1. *For any node $x \in V(T_R)$, let $y_1, \ldots, y_k$ be the children of x in T_R, then*

- *If x is an $\mathcal{F}^-$ node, then $\texttt{Sub}_R(x)$ is obtained by a parallel composition of all $\texttt{Sub}_R(y_i)$'s.*
- *If x is an $\mathcal{L}^-$ node, then $\texttt{Sub}_R(x)$ is obtained by a serial composition of all $\texttt{Sub}_R(y_i)$'s.*
- *If x is a $+$ node, then $\texttt{Sub}_R(x)$ is obtained from $\texttt{Sub}_G(x)$ by replacing each $\texttt{Sub}_G(y_i)$ with $\texttt{Sub}_R(y_i)$.*

Proof. Let us denote by $k(x)$ the number of children of a node $x \in T_R$. Consider the definition of run (Definition 6). Though in the definition of run there is no fixed order of serial and parallel compositions, and only two copies of fork or loop are made at a time, given the execution plan T_R of a run we can assume wlog. that the subgraph $\texttt{Sub}_G(x)$ for a $-$ node $x \in T_R$ is replicated $k(x)$ times (i.e. formed by a serial or parallel execution of $k(x)$ copies at the same time). We can also note that $\texttt{Sub}_R(x)$ is only formed only when all the serial and parallel executions of all fork and loop subgraphs are complete. Let us consider any post-order listing $x_1, \cdots, x_{|T_R|}$ of the nodes in T_R. Now we will prove the lemma by an induction on this order.

For the base case consider x_1 which must be a leaf node and also a $+$ node. Hence it trivially follows that $\texttt{Sub}_R(x)$ is identical to $\texttt{Sub}_G(x)$. Suppose our claim holds till $j-1$ and now consider x_j. (a) If x_j is an $\mathcal{F}^-$ node, in the run $k(x_j)$ copies of $\texttt{Sub}_G(x_j)$ were combined by a parallel composition. Each of these $k(x_j)$ copies corresponds to the child y_i of x_j in T_R and therefore precedes x_j in the post-order traversal order. By the induction hypothesis, they are replaced by $\texttt{Sub}_R(y_i)$ in the run R. Since all $\texttt{Sub}_R(y_i)$'s are atomic self-contained subgraphs, $\texttt{Sub}_R(x_j)$ can be assumed to be formed by parallel executions of all $\texttt{Sub}_R(y_i)$'s. (b) If x_j is an $\mathcal{L}^-$ node, a similar argument as above holds (here $\texttt{Sub}_R(x_j)$ is formed by serial executions of all $\texttt{Sub}_R(y_i)$'s in order.) (c) If x_j is a $+$-node, i.e. a single copy of a fork or loop subgraph, all its fork and loop subgraphs (corresponding to children of x_j in T_R) $\texttt{Sub}_G(y_i)$'s are replaced by $\texttt{Sub}_R(y_i)$'s in the run R, and since each of them is an atomic or complete self-contained subgraph in the run R, $\texttt{Sub}_R(x_j)$ is obtained from $\texttt{Sub}_G(x_j)$ by replacing each $\texttt{Sub}_G(y_i)$ with $\texttt{Sub}_R(y_i)$. $\square$

It follows from the above lemma that given an execution plan T_R along with the specification $(G, \mathcal{F}, \mathcal{L})$, the corresponding run R can be obtained by computing the function $\texttt{Sub}_R(x)$ for each node x in T_R in a bottom-up fashion, and $R = \texttt{Sub}_R(r)$, where r is the root of T_R. Consequently, the execution plan can be thought of a compact description of a run. The compactness comes from the following lemma showing that the size of the execution plan is bounded by the size of the run within a constant factor. Note that in practice, the execution plan is always much smaller.

Lemma 4.2. $|V(T_R)| \leq 4|E(R)|$.

Proof. First, we delete all $-$ nodes in T_R, and obtain a new tree T'_R. More precisely, deleting a node in a tree means making the children of this node become the children of its parent and then removing this node from the tree. Since each $-$ node has at least one $+$ child and the root is a $+$ node, the number of $-$ nodes in T_R is less than the number of $+$ nodes in T_R. So we have $|V(T_R)| \leq 2|V(T'_R)| - 1$.

Next, we delete all $+$ nodes in T'_R having exactly one child, and obtain another tree T''_R. Let m_1 be the number of $+$ nodes deleted from T'_R, then $|V(T'_R)| = |V(T''_R)| + m_1$.

Now consider the tree T''_R. Let m_2 be the number of leaves in T''_R. Since each non-leaf node in T''_R has at least two children, we have $|V(T''_R)| \leq 2m_2 - 1$.

By Definition 2 and Lemma 3.1, the edge sets of all fork and loop copies in R are well-nested. Therefore, each leaf node in T''_R corresponds to at least one distinct edge in R. On the other hand, since each $+$ node deleted from T'_R has only one child, each of them (except for the root if the root has only one child) also corresponds to at least one distinct edge in R. Hence, $\max\{0, m_1 - 1\} + m_2 \leq |E(R)|$.

Combining all the above results, we obtain

$$\begin{aligned}|V(T_R)| &\leq 2|V(T'_R)| - 1 = 2(|V(T''_R)| + m_1) - 1\\ &\leq 2(2m_2 - 1 + m_1) - 1\\ &< 4(\max\{0, m_1 - 1\} + m_2) \leq 4|E(R)|\end{aligned}$$

$\square$

Context. Intuitively, the context of a vertex in R refers to the smallest fork or loop copy that dominates this vertex. Recall that each $+$ node in T_R corresponds to a single fork or loop copy. We therefore assign each vertex in R a unique $+$ node in T_R as its context. Formally,

Definition 9. (**Context**) Given a run R and its execution plan T_R, a node $x \in V(T_R)$ is said to be the *context* of a vertex $v \in V(R)$, denoted by $x = \mathcal{C}(v)$, if x is the deepest $+$ node in T_R such that $v \in \texttt{DomSet}(x)$, where

$$\texttt{DomSet}(x) = \begin{cases} V^*(\texttt{Sub}_R(x)) & \text{if } x \text{ is an } \mathcal{F} \text{ node} \\ V(\texttt{Sub}_R(x)) & \text{otherwise} \end{cases}$$

In Definition 9, $\texttt{DomSet}(x)$ denotes the set of vertices in R *dominated* by a node x in T_R. This extends the notion of domination[2] introduced in Definition 2. Note that the context is well-defined for each vertex in R, since Lemma 3.1 ensures that the dominating sets are well-nested in T_R.

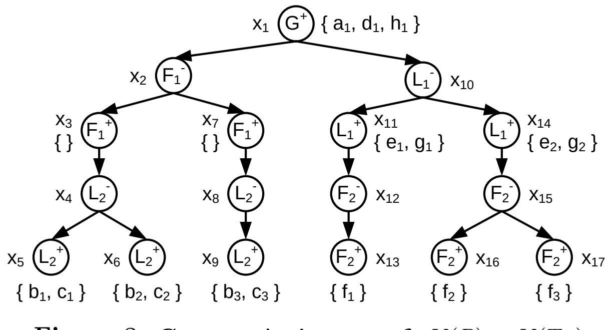

Figure 8: Context Assignment $\mathcal{C} : V(R) \rightarrow V(T_R)$

Example 5. Figure 8 shows the context assignment for the running example. Note that by Definition 9, each fork copy dominates only the internal vertices, since its source and sink may be shared by other edge-disjoint fork or loop copies. For instance, a_1 and h_1 are shared by two fork copies represented by x_3 and x_7 respectively. To ensure that the context for a_1 and h_1 is well-defined, we will assign the least common $+$ ancestor x_1 of x_3 and x_7 as their context.

We say a $+$ node $x \in V(T_R)$ is *empty* if there does not exist any vertex $v \in V(R)$ such that $\mathcal{C}(v) = x$. For example, as shown in Figure 8, x_3 and x_7 are two empty $+$ nodes. Obviously, the context of a vertex $v \in V(R)$ always refers to a nonempty $+$ node $x \in V(T_R)$.

4.2 Answering Reachability Queries

We now show that the reachability between any two vertices in the run can be determined using only their context and given skeleton labels. The main idea is to check the *least common ancestor* of their context in the execution plan. If this ancestor is an $\mathcal{F}^-$ or an $\mathcal{L}^-$ node, then we can immediately determine the reachability by showing that they are dominated by two distinct copies of the same fork (unreachable) or loop (reachable). Otherwise (this ancestor is a $+$ node), we show that the reachability between these two vertices in the run is the same as the reachability between their origins in the specification, and therefore can be determined using the given skeleton labels. Recall that for any vertex $v \in V(R)$, $\mathcal{C}(v)$ denotes the context of v and $\texttt{Orig}(v)$ denotes the origin of v. The lemmas below formalize this idea.

LEMMA 4.3. *For any two vertices $v, v' \in V(R)$, if the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R is*

- *an $\mathcal{F}^-$ node, then there does not exist a path from v to v' or from v' to v in R.*
- *an $\mathcal{L}^-$ node, then there exists a path either from v to v' or from v' to v in R.*

PROOF. Let x be the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R. If x is an $\mathcal{F}^-$ node, since both $\mathcal{C}(v)$ and $\mathcal{C}(v')$ are $+$ nodes, neither of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ is an ancestor of the other (otherwise, x will be a $+$ node). Therefore, there exist two children y and y' of x in T_R such that y is an ancestor of $\mathcal{C}(v)$ and y' is an ancestor of $\mathcal{C}(v')$. Note that both y and y' are $\mathcal{F}^+$ nodes. Therefore, by Definition 9, $v \in \texttt{DomSet}(y) = V^*(\texttt{Sub}_R(y))$ and $v' \in \texttt{DomSet}(y') = V^*(\texttt{Sub}_R(y'))$. Moreover, by Lemma 4.1, $\texttt{Sub}_R(y)$ and $\texttt{Sub}_R(y')$ are combined by a parallel composition. Hence, by Definition 4, there does not exist a path from v to v' or from v' to v in R.

If x is an $\mathcal{L}^-$ node, we can prove the lemma in a similar manner. Since neither of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ is an ancestor of the other, there exist two children y and y' of x in T_R such that y is an ancestor of $\mathcal{C}(v)$ and y' is an ancestor of $\mathcal{C}(v')$. Note that both y and y' are $\mathcal{L}^+$ nodes. Therefore, by Definition 9, $v \in \texttt{DomSet}(y) = V(\texttt{Sub}_R(y))$ and $v' \in \texttt{DomSet}(y') = V(\texttt{Sub}_R(y'))$. Moreover, by Lemma 4.1, $\texttt{Sub}_R(y)$ and $\texttt{Sub}_R(y')$ are concatenated by a serial composition. Hence, by Definition 4, there exists a path either from v to v' or from v' to v in R. Furthermore, we can determine the direction of this path as follows. Note that the children of an $\mathcal{L}^-$ node in T_R are ordered (from left to right). If y is on the left of y', then there exists a path from v to v', otherwise a path from v' to v. $\square$

LEMMA 4.4. *For any two vertices $v, v' \in V(R)$, if the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R is a $+$ node,*

[2]Note that our definition of dominating set differs from the one typically used in graph theory.

then there exists a path from v to v' in R if and only if there exists a path from $\texttt{Orig}(v)$ to $\texttt{Orig}(v')$ in G.

PROOF. For any directed graph G and any two vertices $v, v' \in V(G)$, we use $v \overset{G}{\leadsto} v'$ to denote that there exists a path from v to v' in G. Then this lemma is written as

$$v \overset{R}{\leadsto} v' \iff \texttt{Orig}(v) \overset{G}{\leadsto} \texttt{Orig}(v') \quad (1)$$

Let x be the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R. Clearly, both v and $v' \in V(\texttt{Sub}_R(x))$. Since $\texttt{Sub}_R(x)$ is a self-contained subgraph of R,

$$v \overset{R}{\leadsto} v' \iff v \overset{\texttt{Sub}_R(x)}{\leadsto} v' \quad (2)$$

Similarly,

$$\texttt{Orig}(v) \overset{G}{\leadsto} \texttt{Orig}(v') \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \quad (3)$$

From (1), (2) and (3), it suffices to show that

$$v \overset{\texttt{Sub}_R(x)}{\leadsto} v' \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \quad (4)$$

Let $y_1, \ldots, y_k$ be the children of x in T_R. Note that each y_i is a $-$ node. Moreover, by Lemma 4.1, $\texttt{Sub}_R(x)$ is derived from $\texttt{Sub}_G(x)$ by replacing each $\texttt{Sub}_G(y_i)$ with $\texttt{Sub}_R(y_i)$. Now consider four cases.

(i) If $\mathcal{C}(v) = \mathcal{C}(v')$, then $x = \mathcal{C}(v) = \mathcal{C}(v')$. For any child y_i of x, by Definition 9, both v and $v' \notin \texttt{DomSet}(y_i)$. So both v and $v' \notin V^*(\texttt{Sub}_R(y_i))$. Now consider the transformation from $\texttt{Sub}_G(x)$ to $\texttt{Sub}_R(x)$. Since $\texttt{Sub}_R(y_i)$ and $\texttt{Sub}_G(y_i)$ are self-contained subgraphs, (4) holds for Case (i).

(ii) If $\mathcal{C}(v) \neq \mathcal{C}(v')$ and $\mathcal{C}(v)$ is an ancestor of $\mathcal{C}(v')$, then $x = \mathcal{C}(v)$ and let y_i be the child of x in T_R such that y_i is an ancestor of $\mathcal{C}(v')$. By Definition 9, $v \notin \texttt{DomSet}(y_i)$ and $v' \in \texttt{DomSet}(y_i)$. So $v \notin V^*(\texttt{Sub}_R(y_i))$ and $v' \in V(\texttt{Sub}_R(y_i))$. Let s_i and t_i be the source and sink of $\texttt{Sub}_R(y_i)$ in R respectively. We first show that

$$v \overset{\texttt{Sub}_R(x)}{\leadsto} v' \iff v \overset{\texttt{Sub}_R(x)}{\leadsto} s_i \quad (5)$$

Consider two cases. (a) If y_i is an $\mathcal{F}^-$ node, then $v' \in V^*(\texttt{Sub}_R(y_i))$. Therefore, (5) holds for Case (a); and (b) If y_i is an $\mathcal{L}^-$ node, then $v' \in V(\texttt{Sub}_R(y_i))$, but $\texttt{Sub}_R(y_i)$ is a complete self-contained subgraph. Therefore, (5) also holds for Case (b) even if $v' \in \{s_i, t_i\}$. This completes our proof for (5). Similarly, we can show that

$$\texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(s_i) \quad (6)$$

Now consider the transformation from $\texttt{Sub}_G(x)$ to $\texttt{Sub}_R(x)$ as before. Since for any child y_j of x, both v and $s_i \notin V^*(\texttt{Sub}_R(y_j))$, we again have

$$v \overset{\texttt{Sub}_R(x)}{\leadsto} s_i \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(s_i) \quad (7)$$

Combining (5), (6) and (7), (4) holds for Case (ii).

(iii) If $\mathcal{C}(v) \neq \mathcal{C}(v')$ and $\mathcal{C}(v')$ is an ancestor of $\mathcal{C}(v)$, then $x = \mathcal{C}(v')$ and let y_i be the child of x in T_R such that y_i is an ancestor of $\mathcal{C}(v)$. Using the sink t_i of $\texttt{Sub}_R(y_i)$, in a similar way as done in Case (ii), we can show that

$$v \overset{\texttt{Sub}_R(x)}{\leadsto} v' \iff t_i \overset{\texttt{Sub}_R(x)}{\leadsto} v' \quad (8)$$

$$t_i \overset{\texttt{Sub}_R(x)}{\leadsto} v' \iff \texttt{Orig}(t_i) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \quad (9)$$

$$\texttt{Orig}(t_i) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \quad (10)$$

Combining (8), (9) and (10), (4) holds for Case (iii).

(iv) If $\mathcal{C}(v) \neq \mathcal{C}(v')$ and neither of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ is an ancestor of the other, then there exist two children y_i and y_j of x in T_R such that y_i is an ancestor of $\mathcal{C}(v)$ and y_j is an ancestor of $\mathcal{C}(v')$. Using both the sink t_i of $\texttt{Sub}_R(y_i)$ and the source s_j of $\texttt{Sub}_R(y_j)$, we can show that

$$v \overset{\texttt{Sub}_R(x)}{\leadsto} v' \iff t_i \overset{\texttt{Sub}_R(x)}{\leadsto} s_j \quad (11)$$

$$t_i \overset{\texttt{Sub}_R(x)}{\leadsto} s_j \iff \texttt{Orig}(t_i) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(s_j) \quad (12)$$

$$\texttt{Orig}(t_i) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(s_j) \iff \texttt{Orig}(v) \overset{\texttt{Sub}_G(x)}{\leadsto} \texttt{Orig}(v') \quad (13)$$

Combining (11), (12) and (13), (4) holds for Case (iv). This completes our proof for (4). Hence, the lemma follows. $\square$

Example 6. We can easily verify above three rules using Figure 3. First, consider two vertices b_1 and c_3 in R. We can see from Figure 8 that the least common ancestor of their context (*i.e.*, x_5 and x_9) is an $\mathcal{F}^-$ node (*i.e.*, x_2). Therefore, by Lemma 4.3, there does not exist a path between b_1 and c_3, which is confirmed by Figure 3. Similarly, consider f_1 and e_2. The least common ancestor of their context (*i.e.*, x_{13} and x_{14}) is an $\mathcal{L}^-$ node (*i.e.*, x_{10}), and there exists a path from f_1 to e_2, which again verifies our Lemma 4.3. Finally, consider c_1 and d_1. The least common ancestor of their context (*i.e.*, x_5 and x_1) is a $+$ node (*i.e.*, x_1). Moreover, we can see from Figure 2 that there does not exist a path between their origins (*i.e.*, c and d) in G. Consequently, by Lemma 4.4, there also does not exist a path between c_1 and d_1 in R, which is again confirmed by Figure 3.

4.3 Encoding Context

As seen in Section 4.2, the reachability query against two vertices in the run can be reduced to a related query against their context in the execution plan. Recall that the context always refers to a nonempty $+$ node. The related query is therefore stated as follows. Given two nonempty $+$ nodes in the execution plan, we want to decide if their least common ancestor is an $\mathcal{F}^-$ or an $\mathcal{L}^-$ or a $+$ node. In this section, we will show that we can efficiently answer this related query using a three-dimensional encoding for the context.

Algorithm 1 `GenerateThreeOrders`

```
Input:  $T_R$ : an execution plan
Output: $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ : three total orders
  /* Generate $\mathcal{O}_1$ */
  Do a preorder traversal on $T_R$: for each node $x \in V(T_R)$, visit the children of $x$ from left to right.
  /* Generate $\mathcal{O}_2$ */
  Do a preorder traversal on $T_R$: for each node $x \in V(T_R)$, if $x$ is an $\mathcal{F}^-$ node, visit the children of $x$ from right to left; otherwise, visit the children of $x$ from left to right.
  /* Generate $\mathcal{O}_3$ */
  Do a preorder traversal on $T_R$: for each node $x \in V(T_R)$, if $x$ is an $\mathcal{L}^-$ node, visit the children of $x$ from right to left; otherwise, visit the children of $x$ from left to right.
  return $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$
```

Generate Three Total Orders. We first present a simple algorithm to generate three total orders $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ on all nonempty + nodes in the execution plan T_R. The algorithm performs three *preorder* traversals[3] on T_R, and records the order in which all nonempty + nodes are visited in the ith traversal as the total order $\mathcal{O}_i$ ($i = 1, 2, 3$). Recall that only the children of an $\mathcal{L}^-$ node are ordered in T_R, for any other node, we fix an arbitrary ordering of its children. The details of three preorder traversals are described in Algorithm 1. Note that these traversals differ only in the order in which the children of an $\mathcal{F}^-$ or an $\mathcal{L}^-$ node are visited.

The above three total orders lend themselves to a compact three-dimensional encoding for the context: Each nonempty + node in T_R is simply encoded by three natural numbers indicating its positions in $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ respectively.

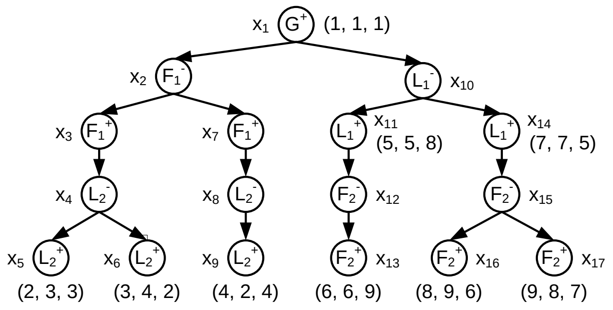

Figure 9: Context Encoding

Example 7. Figure 9 shows the context encoding for the running example. Note that Algorithm 1 will traverse every node in the execution plan, but records only the ordering of nonempty + nodes. In particular, x_3 and x_7 are empty + nodes (as shown in Figure 8) and are therefore skipped.

Compare Three Total Orders. We formally present how to answer the related query using above context encoding and show the correctness by the following lemma. Let $x <_{\mathcal{O}} x'$ denote that x precedes x' in the total order $\mathcal{O}$.

LEMMA 4.5. *For any two nonempty + nodes $x, x' \in V(T_R)$,*

- *If $x <_{\mathcal{O}_1} x'$ and $x' <_{\mathcal{O}_2} x$, then (1a) the least common ancestor of x and x' is an $\mathcal{F}^-$ node and (1b) $x <_{\mathcal{O}_3} x'$.*
- *If $x <_{\mathcal{O}_1} x'$ and $x' <_{\mathcal{O}_3} x$, then (2a) the least common ancestor of x and x' is an $\mathcal{L}^-$ node and (2b) $x <_{\mathcal{O}_2} x'$.*
- *If $x <_{\mathcal{O}_1} x'$, $x <_{\mathcal{O}_2} x'$ and $x <_{\mathcal{O}_3} x'$, then (3) the least common ancestor of x and x' is a + node.*

PROOF. Let y be the least common ancestor of x and x'. Consider three cases. (**Case 1**) If $x <_{\mathcal{O}_1} x'$ and $x' <_{\mathcal{O}_2} x$, we first show that neither of x and x' is an ancestor of the other. Suppose not. First we assume that x is an ancestor of x'. Since $\mathcal{O}_2$ is produced by a preorder traversal on T_R, $x <_{\mathcal{O}_2} x'$, which is a contradiction. Similarly, x' cannot be an ancestor of x. Therefore, there exist two children z and z' of y such that z is an ancestor of x and z' is an ancestor of x'. Next, we show that y is an $\mathcal{F}^-$ node. Suppose not. Then both $\mathcal{O}_1$ and $\mathcal{O}_2$ will visit z and z' in the same order. Since z is an ancestor of x and z' is an ancestor of x', $x <_{\mathcal{O}_1} x'$ if and only if $x <_{\mathcal{O}_2} x'$, which is a contradiction. This proves Part (1a). Since y is an $\mathcal{F}^-$ node (not an $\mathcal{L}^-$ node), both $\mathcal{O}_1$ and $\mathcal{O}_3$ will visit z and z' in the same order. Consequently, it follows from $x <_{\mathcal{O}_1} x'$ that $x <_{\mathcal{O}_3} x'$. This proves Part (1b). (**Case 2**) If $x <_{\mathcal{O}_1} x'$ and $x' <_{\mathcal{O}_3} x$, we can prove Part (2a) and (2b) in a similar manner. (**Case 3**) If $x <_{\mathcal{O}_1} x'$, $x <_{\mathcal{O}_2} x'$ and $x <_{\mathcal{O}_3} x'$, suppose y is not a + node. Then y is either an $\mathcal{F}^-$ node or an $\mathcal{L}^-$ node. Since both x and x' are + nodes, neither of x and x' is an ancestor of the other (otherwise, y will be a + node). Therefore, there exist two children z and z' of y such that z is an ancestor of x and z' is an ancestor of x'. If y is an $\mathcal{F}^-$ node, then $\mathcal{O}_1$ and $\mathcal{O}_2$ will visit z and z' in the reverse order. Therefore, $x <_{\mathcal{O}_1} x'$ if and only if $x' <_{\mathcal{O}_2} x'$, which is a contradiction. Similarly, if y is an $\mathcal{L}^-$ node, then $\mathcal{O}_1$ and $\mathcal{O}_3$ will visit z and z' in the reverse order. Therefore, $x <_{\mathcal{O}_1} x'$ if and only if $x' <_{\mathcal{O}_3} x$, which is again a contradiction. Hence, y must be a + node. Part (3) follows. □

Example 8. We can easily verify the above three rules using Figure 9. For instance, consider two nonempty + nodes x_5 and x_9 in T_R. Observe that $x_5 <_{\mathcal{O}_1} x_9$ and $x_9 <_{\mathcal{O}_2} x_5$. Hence, by Lemma 4.5, the least common ancestor of x_5 and x_9 (*i.e.*, x_2) is an $\mathcal{F}^-$ node and $x_5 <_{\mathcal{O}_3} x_9$, which is confirmed by Figure 9. Using (x_{13}, x_{14}) and (x_5, x_1), we can verify the other two rules in a similar way.

4.4 Labeling Scheme Description

We are now ready to present the skeleton-based labeling scheme. Recall that a labeling scheme is denoted by a triple $(\mathcal{D}, \phi, \pi)$, where $\mathcal{D}$ is the label domain, ϕ is the labeling function (*i.e.*, encoding algorithm) and π is the binary predicate (*i.e.*, decoding algorithm). In the following, let $(\mathcal{D}_g, \phi_g, \pi_g)$ denote the given labeling scheme used to label the specification, and $(\mathcal{D}_r, \phi_r, \pi_r)$ denote our proposed skeleton-based labeling scheme used to label the run.

Label Domain ($\mathcal{D}_r$). Each label assigned by our skeleton-based labeling scheme consists of a three-dimensional encoding for the context (*i.e.*, three natural numbers) and a skeleton label from the specification, *i.e.*, $\mathcal{D}_r = \mathbb{N}^3 \times \mathcal{D}_g$.

Labeling Function (ϕ_r). A high level description of the labeling function ϕ_r is shown in Algorithm 2. Given a specification G labeled by ϕ_g, we label a run R of G in four steps: First, we compute the origin function $\texttt{Orig} : V(R) \to V(G)$ by matching their module names (Line 2). Second, we construct the execution plan T_R for R, and compute the context function $\mathcal{C} : V(R) \to V(T_R)$ (Line 5). Since this part of computation may be of independent interest, we defer the description of the algorithm `ConstructPlan` to Section 5. Third, we build three-dimensional context encoding by generating three total orders $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ using Algorithm 1 (Line 8). Finally, we assign each vertex $v \in V(R)$ a reachability label $\phi_r(v) = (q_1, q_2, q_3, \phi_g(u))$, where (q_1, q_2, q_3) denotes its context encoding (*i.e.*, the positions of $x = \mathcal{C}(v)$ in $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$ respectively) and $\phi_g(u)$ denotes its skeleton label (*i.e.*, the reachability label on $u = \texttt{Orig}(v)$ given by ϕ_g). (Line 12 to Line 15).

Binary Predicate (π_r). Given the above labels, we can answer any reachability query on R by a simple binary predicate π_r described in Algorithm 3. Observe that evaluating the predicate π_r for a pair of labels from $\mathcal{D}_r$ only requires comparing three pairs of natural numbers plus one potential evaluation of the predicate π_g for a pair of labels from $\mathcal{D}_g$.

Example 9. The reachability labels assigned by ϕ_r for the running example are shown in Figure 10, where ϕ_g denotes

[3]A *preorder* traversal on a tree starts at the root of the tree, and recursively visits the root of a subtree before visiting its children.

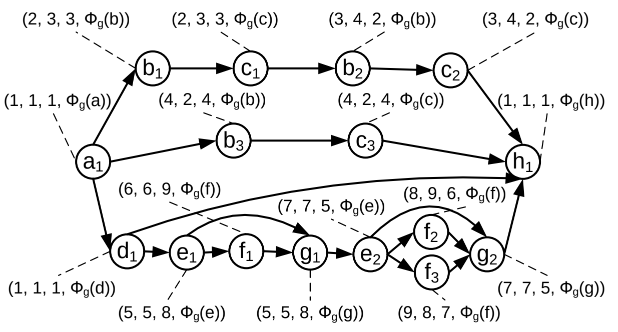

Figure 10: Skeleton-Based Labeling Scheme

the given skeleton labels. To illustrate how to answer reachability queries using these labels, let us consider two vertices c_1 and d_1 in R. We start by comparing their reachability labels $(2, 3, 3, \phi_g(c))$ and $(1, 1, 1, \phi_g(d))$ using the predicate π_r described in Algorithm 3. Since $(3-1) \times (3-1) > 0$, we need to further compare their skeleton labels $\phi_g(c)$ and $\phi_g(d)$ using the given predicate π_g. Observe from Figure 2 that there is no path from c to d in G (*i.e.*, $\pi_g(\phi_g(c), \phi_g(d)) = 0$). Hence, we can conclude that there is also no path from c_1 to d_1 in R, which is confirmed by Figure 10.

Algorithm 2 *Labeling Function* $\phi_r : V(R) \to \mathcal{D}_r$

```
Input:  $G$ : a specification
        $\phi_g : V(G) \to \mathcal{D}_g$ : a labeling function for $G$
        $R$ : a run of $G$
Output: $\phi_r : V(R) \to \mathcal{D}_r$ : a labeling function for $R$
 1: /* Step 1: Compute Origin */
 2: Orig $\leftarrow$ ComputeOrigin($G, R$)
 3:
 4: /* Step 2: Construct Execution Plan and Context */
 5: $< T_R, \mathcal{C} > \leftarrow$ ConstructPlan($G, R$)
 6:
 7: /* Step 3: Build Context Encoding */
 8: $< \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3 > \leftarrow$ GenerateThreeOrders($T_R$)
 9:
10: /* Step 4: Assign Reachability Labels */
11: for each vertex $v \in V(R)$ do
12:     $x \leftarrow \mathcal{C}(v)$ : the context of $v$
13:     $q_i \leftarrow$ the position of $x$ in $\mathcal{O}_i$ $(1 \le i \le 3)$
14:     $u \leftarrow$ Orig($v$) : the origin of $v$
15:     $\phi_r(v) \leftarrow (q_1, q_2, q_3, \phi_g(u))$.
16: end for
17:
18: return $\phi_r$
```

4.5 Correctness and Quality Analysis

The following lemma shows the correctness of our skeleton-based labeling scheme.

Lemma 4.6. *Given a run R labeled by our skeleton-based labeling scheme $(\mathcal{D}_r, \phi_r, \pi_r)$, for any two vertices $v, v' \in V(R)$, $\pi_r(\phi_r(v), \phi_r(v')) = 1$ if and only if there exists a path from v to v' in R.*

Proof. Let $\phi_r(v) = (q_1, q_2, q_3, d_g)$ and $\phi_r(v') = (q'_1, q'_2, q'_3, d'_g)$. Recall that T_R denotes the execution plan for R, and G denotes the specification of R. In addition, for any vertex $v \in V(R)$, $\mathcal{C}(v)$ denotes the context of v, and $\texttt{Orig}(v)$ denotes the origin of v. We will prove the theorem by doing a case analysis according to Algorithm 3.

Algorithm 3 *Binary Predicate* $\pi_r : \mathcal{D}_r \times \mathcal{D}_r \to \{0, 1\}$

```
Input:  $d_r = (q_1, q_2, q_3, d_g) \in \mathcal{D}_r$
        $d'_r = (q'_1, q'_2, q'_3, d'_g) \in \mathcal{D}_r$
        $\pi_g : \mathcal{D}_g \times \mathcal{D}_g \to \{0, 1\}$
Output: $\pi_r(d_r, d'_r)$
 1: if $(q_2 - q'_2) \times (q_3 - q'_3) < 0$ then
 2:     if $q_1 < q'_1$ and $q_3 > q'_3$ then
 3:         $\pi_r(d_r, d'_r) \leftarrow 1$
 4:     else
 5:         $\pi_r(d_r, d'_r) \leftarrow 0$
 6:     end if
 7: else
 8:     $\pi_r(d_r, d'_r) \leftarrow \pi_g(d_g, d'_g)$
 9: end if
10: return $\pi_r(d_r, d'_r)$
```

If $(q_2 - q'_2) \times (q_3 - q'_3) < 0$, then there are four cases: (1) $q_1 < q'_1, q_2 < q'_2, q_3 > q'_3$; (2) $q_1 > q'_1, q_2 < q'_2, q_3 > q'_3$; (3) $q_1 < q'_1, q_2 > q'_2, q_3 < q'_3$; and (4) $q_1 > q'_1, q_2 > q'_2, q_3 < q'_3$. First, consider Case (2) and (3). By Lemma 4.5, the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R is an $\mathcal{F}^-$ node. Therefore, by Lemma 4.3, there does not exist a path from v to v' or from v' to v in R. The theorem holds for Case (2) and (3). Next, consider Case (1) and (4). By Lemma 4.5, the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ in T_R is an $\mathcal{L}^-$ node. Therefore, by Lemma 4.3, there exist a path from v to v' or from v' to v in R. Furthermore, since $q_1 < q'_1$ in Case (1) and $q_1 > q'_1$ in Case (4), from the proof of Lemma 4.3, we can see that there exist a path from v to v' in Case (1) and a path from v' to v in Case (2). Again, the theorem holds for these two cases.

If $(q_2 - q'_2) \times (q_3 - q'_3) \ge 0$, then there are the other five cases: (5) $q_1 = q'_1, q_2 = q'_2, q_3 = q'_3$; (6) $q_1 < q'_1, q_2 < q'_2, q_3 < q'_3$; (7) $q_1 > q'_1, q_2 < q'_2, q_3 < q'_3$; (8) $q_1 < q'_1, q_2 > q'_2, q_3 > q'_3$; and (9) $q_1 > q'_1, q_2 > q'_2, q_3 > q'_3$. Note that by Lemma 4.5, Case (7) and (8) are impossible. Consider the other three cases. By Lemma 4.5, in Case (6) and (9), the least common ancestor of $\mathcal{C}(v)$ and $\mathcal{C}(v')$ is a $+$ node. Note that in Case (5), $\mathcal{C}(v) = \mathcal{C}(v')$. So the above claim is also true. Therefore, by Lemma 4.4, there exists a path from v to v' in R if and only if there exists a path from $\texttt{Orig}(v)$ to $\texttt{Orig}(v')$ in G. Hence, the theorem follows. □

The following lemma analyzes the quality of our skeleton-based labeling scheme in terms of label length, construction time and query time. We do not count the cost of labeling the specification, since it is fixed for all runs.

Lemma 4.7. *Given a fixed labeled specification G, for any run R of G, our skeleton-based labeling scheme for R guarantees (1) the label length $\le 3 \log n_R + \log n_G$, (2) the construction time is $O(m_R + n_R)$, and (3) the query time is $O(1) + t_G$, where $n_R = |V(R)|$, $n_G = |V(G)|$, $m_R = |E(R)|$ and t_G is the query time for G using the given skeleton labels.*

Proof. Let T_R be the execution plan of R, and n^+_T be the number of nonempty $+$ nodes in T_R. Then each label in R consists of three natural numbers between 1 and n^+_T and a skeleton label from G. Note that there are at most n_G distinct skeleton labels in total, so we only need $\log n_G$ bits to encode each skeleton label. Therefore, the total label length $\leq 3 \log n_T^+ + \log n_G$. On the other hand, it is easy to see that $n_T^+ \leq n_R$, because each nonempty $+$ node in T_R serves as the context for at least one distinct vertex in R. Consequently, the label length $\leq 3 \log n_R + \log n_G$.

According to Algorithm 2, labeling a run R involves four steps: (1) computing the origin. Since the module names in G are unique, finding the origin for each vertex in R needs only $O(n_R)$ time; (2) constructing the execution plan and context, for which we will present in Section 5 a linear time algorithm. So by Lemma 5.2, this step takes $O(m_R + n_R)$ time; (3) building context encoding (Algorithm 1), which uses three preorder traversals on T_R. So it takes a total of $O(n_T)$ time, where $n_T = |V(T_R)|$; and (4) assigning the reachability labels, which requires only one scan on $V(R)$ and therefore takes $O(n_R)$ time. By Lemma 4.2, $n_T \leq 4m_R$. So the overall construction time is $O(m_R + n_R)$.

Finally, according to Algorithm 3, evaluating the predicate π_r for a pair of labels in R requires only comparing three pairs of natural numbers plus one potential evaluation of the predicate π_g for a pair of skeleton labels in G. Hence, the query time is $O(1) + t_G$. $\Box$

Note that for a fixed specification G, we can treat both n_G and t_G in Lemma 4.7 as constants. Consequently, our skeleton-based labeling scheme guarantees logarithmic label length, linear construction time, and constant query time (in terms of the size of the run) and therefore is optimal. This proves Theorem 1 claimed in Section 3.3.

5. EXECUTION PLAN AND CONTEXT

In this section we will describe a linear time algorithm to generate the execution plan T_R and the context function $\mathcal{C}$ given a run R, the specification $(G, \mathcal{F}, \mathcal{L})$ along with the fork and loop hierarchy T_G [4]. Our algorithm, namely `ConstructPlan`, will process the run R in a bottom-up fashion in terms of the fork and loop hierarchy T_G to generate T_R. In this section we will first discuss the challenges in implementing the algorithm in linear time and give an overview of the algorithm (Section 5.1), then we will describe the algorithm in detail and discuss how the context function $\mathcal{C}(v)$ for each node $v \in R$ can be efficiently computed as we generate T_R (Section 5.2). Finally we will analyze the running time of the algorithm (Section 5.3).

5.1 Algorithm Overview

Let $T_G(i)$ and $T_R(i)$ denote the i-th level of T_G and T_R respectively. Let d be the depth of T_G. Recall that, for each level $T_G(i)$ except $T_G(1)$, there are two levels $T_R(2i-2)$ and $T_R(2i-1)$. Our algorithm, namely `ConstructPlan`, generates T_R in d iterations in a bottom-up manner; in iteration i ($i = d$ to 1), $T_R(2i-1)$ and $T_R(2i-2)$ are constructed. Note that since there is a one-one correspondence between the nodes of T_G and the subgraphs in $\mathcal{F} \cup \mathcal{L}$, we directly refer to any subgraph $H \in \mathcal{F} \cup \mathcal{L}$ as a node $H \in T_G$.

The main difficulties in constructing T_R in linear time are (1) to identify all copies of each fork and loop subgraph in T_G in the run-graph R efficiently and (2) to avoid traversing the same set of nodes and edges multiple times as we construct T_R in a bottom-up fashion. We take care of these two problems in the following way.

[4] T_G serves as a compact description of $(\mathcal{F}, \mathcal{L})$, and can be efficiently constructed from any other explicit description of $(\mathcal{F}, \mathcal{L})$.

(1) Identifying all copies of fork and loop subgraphs: If $H \in T_G$ is a leaf node, then we can count the number of copies of any edge $e \in E(H)$ in R to count the number of copies of H in R. But if $H \in T_G$ is a non-leaf node, we may not have a unique edge in $E(H)$ which does not belong to any other fork or loop subgraph (for example, consider a line-graph of length 2, $u \rightarrow v \rightarrow w$, and define three forks on $\{u, v\}$, $\{v, w\}$ and $\{u, v, w\}$). For each leaf node $H \in T_G$, we assign any edge $e \in E(H)$ as the *leader* of H and call it `leader`(H). If $H \in T_G$ is a non-leaf node, we assign any arbitrary child H_c of H as a *candidate* for `leader`(H); as the algorithm `ConstructPlan` executes in the bottom-up fashion, it constructs an edge from a copy of H_c during its execution. By traversing R only once we can compute all copies of `leader`(H) for each leaf node $H \in T_G(d)$ and subsequently before the i-th iteration starts ($i = d-1$ to 1), we have all copies of `leader`(H) for each node $H \in T_G(i)$.

(2) Avoiding visiting any edge and node in R more than once: In iteration i, we identify and visit all copies of $H \in T_G(i)$ in R. To avoid visiting the vertices and edges in these copies in future iterations, after $T_R(2i-1)$ is constructed, we replace each *copy* of $H \in T_G(i)$ by a directed edge from the source to the sink of this copy preserving only these two vertices. Similarly, after $T_R(2i-2)$ is constructed, we replace *all copies* of H created due to a fork or loop execution by a directed edge (this preserves the source and sink of a fork execution and the source of the first copy and the sink of the last copy of a loop execution). We refer to the new edges added in this process as *special edges*.

Let H' be a copy of a subgraph $H \in T_G(i)$ in R. In iteration i, we start with an endpoint of a copy of `leader`(H) in H' as a seed and explore the vertices and edges in H' using a modified depth first search. In this search procedure, we use the fact that in the modified run-graph R at each iteration i (after deleting and adding some edges in the previous iterations), (1) the induced subgraph on the vertices in H' is a connected subgraph if $H \in \mathcal{L}$; and (2) the induced subgraph on the vertices in H' excluding $s(H'), t(H')$ is a connected subgraph if $H \in \mathcal{F}$. We can construct the execution plan T_R and compute the context function $\mathcal{C}(v)$ for each vertex $v \in V(R)$ as we execute the above search procedure.

5.2 Detailed Description

Recall that for a node $v \in V(R)$, $\tau(v)$ denotes the origin of the node v in G. We give the pseudocode of `ComputeContext` in Algorithm 4. In iteration i, if H' is a copy of a subgraph $H \in T_G(i)$, `ComputeContext` replaces all edges in $E_i(H')$ by a directed edge from $s(H')$ to $t(H')$. The procedure `SearchNodes` (given in Algorithm 5) identifies the edges in $E_i(H')$. This procedure takes an edge $e \in E_i(H')$ (e can possibly be a special edge added in a previous iteration) and the subgraph H. The edge e is used as a *seed* to identify the edges in $E_i(H')$.

The correctness of the procedure `SearchNodes` depends on the following fact: if H is a loop subgraph then the induced graph on $V_i(H')$ forms a connected component in R_i, whereas if H is a fork subgraph then the induced graph on $V_i(H') - \{s(H'), t(H')\}$ forms a connected component. Since each loop subgraph is a complete self-contained subgraph, it

Algorithm 4 *Algorithm* `ComputeContext`

Input: Run R, Specification G, Fork and loop hierarchy T_G.
Output: Execution Plan T_R

```
 1: /* Compute leader($H$) for each leaf-node $H \in T_G$ */
 2: for each subgraph $H$ that is a leaf in $T_G$ do
 3:     Traverse $R$ to find all copies of leader($H$)
 4:     If $H \in T_G(i)$, include all copies of leader($H$) to
        leaderSet($i$)
 5: end for
 6:
 7: /* Construct $T_R$ from $T_R(2d-1)$ to $T_R(1)$ */
 8: $R_d = R$.
 9: for $i = d$ to 1 do
10:     /* Construct $T_R(2i-1)$, add edges to $T_R(2i)$ */
11:     for each edge $e \in$ leaderSet($i$) do
12:         Suppose $e$ is a copy of leader($H$), $H \in T_G(i)$.
13:         Suppose $e \in E_i(H')$ where $H'$ is a copy of $H$.
14:         Create a node $x$ in $T_R(2i-1)$ for $H'$.
15:         Find $E_i(H'), s(H'), t(H')$ by SearchNodes($H, e$).
16:         Add edges between $T_R(2i-2)$ and $T_R(2i-1)$ when
            special edges are visited by SearchNodes
17:         Replace the edges in $E_i(H')$ by a new special edge
            $(s(H'), t(H'))$.
18:     end for
19:
20:     /* Construct $T_R(2i-2)$, add edges to $T_R(2i-1)$ */
21:     while there exists a node $x \in T_R(2i-1)$ which does
        not have a parent in $T_R(2i-2)$ do
22:         Let $x$ corresponds to a copy $H'$ of $H \in T_G(i)$.
23:         if $H \in \mathcal{F}$ then
24:             Let $s' = s(H'), t' = t(H')$.
25:             If there is a node $y \in T_R(2i-2)$ for $H$ with source
                $s'$ and sink $t'$, add $x$ as a child of $y$, and delete
                a special edge $(s', t')$ from $R_i$. Otherwise, create
                a node $y \in T_R(2i-2)$ for $H$ with source $s'$ and
                sink $t'$.
26:         else {/* $H \in \mathcal{L}$ */}
27:             Find the nodes $x_1, \cdots, x_k \in T_R(2i-1)$ cor-
                responding to all serial executions of $H$, say
                $H'_1, H'_2, \cdots, H'_k$, in order. This can be done by
                searching from $H$ in forward and backward di-
                rection using the edges $(t(H'_{j-1}), s(H'_j))$.
28:             Create a node $y$ in $T_R(2i-2)$ and add $x_1, \cdots, x_k$
                as its children in order.
29:             Delete all the edges and vertices on the path from
                $s(H_1)'$ to $t(H'_k)$ in $R_i$.
30:             Add the special edge $e_1 = (s(H'_1), t(H'_k))$ to $R_i$.
31:         end if
32:         If $H$ is the candidate for the leader of its parent in
            $T_G$, then add the special edge for $y$ to leaderSet($i-$
            1).
33:     end while
34:     Call the modified graph as $R_{i-1}$.
35: end for
36: return $T_R$
```

can be easily seen that the induced graph on $V_i(H')$ remains connected. In the following lemma, we prove that the desired connectedness property holds also for a fork subgraph H. This will complete the correctness argument of the algorithm `ComputeContext`.

Algorithm 5 *Procedure* `SearchNodes`(H, e)

Input: Subgraph $H \in T_G(i)$, an edge $e \in E_i(H')$ where H' is a copy of H in R.
Output: $S = E_i(H'), s(H'), t(H')$

```
 1: Initialize $S = \phi$.
 2: Start DFS from an end point of $e$ ignoring the direction
    of the edges and collect edges visited to $S$.
 3: if $H$ is a fork subgraph then
 4:     /* The induced subgraph on $V_i(H') \setminus \{s(H'), t(H')\}$ is
        a connected subgraph */
 5:     Prune the search at a vertex $v \in V_i(H')$ such that
        $\tau(v) \in \{s(H), t(H)\}$, i.e. do not explore any further
        edges from $s(H'), t(H')$.
 6: else
 7:     /* $H$ is a loop subgraph and the induced subgraph on
        $V_i(H')$ is connected */
 8:     When a vertex $v \in V_i(H')$ is visited such that $\tau(v) \in$
        $\{s(H), t(H)\}$, then explore only those edges $e'$ inci-
        dent on $v$ such that $e'$ is an outgoing edge from $s(H')$
        or $e'$ is an incoming edge to $t(H')$ (in the directed
        sense).
 9: end if
10: return $S$, $s(H')$, $t(H')$
```

Lemma 5.1. *If H is a fork subgraph in $T_G(i)$, and H' is a copy of H in R, then in R_i, the induced subgraph on $V_i(H') - \{s(H'), t(H')\}$ in R_i forms a connected component.*

Proof. We prove by an induction on $i = d$ to 1. Consider the base case $i = d$, hence $R_d = R$. Since H is a fork subgraph, it is an atomic self-contained subgraph. First we claim that $V(H) - \{s(H), t(H)\}$ is connected. Suppose not. Consider any maximal connected subgraph $H^1 \subseteq H$ formed by removing $s(H), t(H)$. Let H^2 denote the induced subgraph on $V(H^1) \cup \{s(H), t(H)\}$ and we set $s(H^2) = s(H)$, $t(H^2) = t(H)$. Since $H^2 \subset H$ is a maximal conneted component formed by removing $s(H), t(H)$. If $(u, v) \in E(G)$ for $u \in V(H^2)$, $v \in V(G) \setminus V(H^2)$, then $u = s(H^2)$ or $t(H^2)$. Hence we get another self-contained subgraph $H^2 \subset H$ with the same source and sink. This contradicts that H is a valid fork subgraph since H is not atomic. Now H' is isomorphic to H by definition of the run graph R and since H does not have any child in T_G. Hence $V(H) - \{s(H), t(H)\} = V_d(H') - \{s(H'), t(H')\}$ must be connected.

Suppose the hypothesis holds for all $j > i$ and consider $H \in T_G(i)$. If H has no child in T_G the same argument as above holds for H'. Otherwise let $H_1, \cdots, H_k$ be the children of H in T_G. In R_i we have replaced all the executions of the subgraphs H_j-s by a single edge. This replacement does not change the connectivity of H' in R_i. Hence the above argument holds once again. □

We can compute the context function $\mathcal{C}(v)$ for a node $v \in V(R)$ as we execute the procedure `SearchNodes`(H, e). Suppose we have created the node $x \in T_R(2i-1)$ for a copy H' of H where $e \in E_i(H')$. When a vertex v is visited by the `SearchNodes` procedure such that $v \notin \{s(H'), t(H')\}$ and v is not an endpoint of a special edge, then we assign $\mathcal{C}(v) = x$. If $v \in \{s(H'), t(H')\}$, we assign $\mathcal{C}(v) = x$ only if H is a loop subgraph. If v is an endpoint of a special edge that corresponds to all parallel executions of a fork subgraph, then also we assign $\mathcal{C}(v) = x$. Since we process R in a bottom-up fashion, it is easy to check that the assignments are consistent with the definition of context function.

5.3 Running Time

Let $m_R = |E(R)|$ and $n_R = |V(R)|$ be the number of edges and nodes in a run R, respectively. Since we do not traverse any edge more than once, the time complexity of the algorithm `ConstructPlan` can be bounded by $O(m_R + n_R + m_{sp})$, where m_{sp} is the number of special edges added in all the iterations. By Lemma 4.2, $m' = O(|V(T_R)|) = O(m_R)$, and therefore the time complexity is $O(m_R + n_R)$.

LEMMA 5.2. *Given a fixed specification* $(G, \mathcal{F}, \mathcal{L})$ *and a run* R*, the algorithm* `ComputeContext` *to compute the execution plan* T_R *and the context function* $\mathcal{C}$ *can be implemented in* $O(m_R + n_R)$ *time where* $m_R = |E(R)|$ *and* $n_R = |V(R)|$.

PROOF. `ComputeContext` first traverses the graph R to collect all the copies of `leader`(H) in R where H is a subgraph in T_G with no child. Whether an edge $(u', v') \in E(R)$ is a copy of `leader`(H) can be checked in $O(1)$ time by checking if $\tau(u') = s(H)$ and $\tau(v') = t(H)$. Hence this step takes $O(m_R + n_R)$ time.

In the i-th iteration, the time complexity of `ComputeContext` is dominated by creating nodes in $T_R(2i-1), T_R(2i-2)$ and all the calls to the `SearchNodes` procedure. The time required to create the nodes in T_R in iteration i is $O(|T_R(2i-1)| + |T_R(2i-2)|)$ hence summing over all i, the total time needed for this purpose can be bounded by $O(|V(T_R)|)$.

When `SearchNodes`(H, e, R_i) procedure is called where H' is a copy of the subgraph $H \in T_G(i)$ and $e \in E_i(H')$, we traverse the edges in $E_i(H')$ exactly once since we prune the search at $s(H'), t(H')$ and thus do not visit any edge $\notin E_i(H')$. Also all the edges in $E_i(H')$ get deleted (which includes both the edges originally in $E(R)$ and the special edges added in the previous iteration) though we may again add some special edges in iteration i. The search steps used in the algorithm can be implemented efficiently using hash functions and therefore the total running time can be bounded by $O(m_R + n_R + m_{sp})$ where m_{sp} denotes the total number of special edges added in all the iterations.

Next we count the number of special edges added. Whenever we add a special edge, we create a node in T_R. Hence the total number of special edges added can be bounded by $|V(T_R)|$. Hence we can bound the total time complexity of `ComputeContext` by $O(m + n + |V(T_R)|)$. But we know from Lemma 4.2 that $|V(T_R)| = O(m_R)$. Hence total time complexity of `ComputeContext` is $O(m_R + n_R)$. $\square$

6. ANSWERING DATA PROVENANCE

To efficiently query dependency relations between data or between data and modules, we extend module labels to the data that flow over the data channels (edges) in a run.

We start by associating with each edge $e = (u, v)$ the set of data items that flow from module u to module v in the run, *i.e.*, that are produced (written) by u and consumed (read) by v. We call this set $\textsc{Data}(e)$. Note that each data item must be created by a unique module, *i.e.*, if $x \in \textsc{Data}(u, v)$ and $x \in \textsc{Data}(u', v')$ then $u = u'$. We say a data item x *depends on* another data item x' if there exists a sequence of modules $v_0, \ldots, v_k$ and a set of data items $x_1, \ldots, x_k$ such that $x_1 = x'$, $x_k = x$, and for all $1 \leq i \leq k$, $x_i \in \textsc{Data}(v_{i-1}, v_i)$.

Given a run whose modules (vertices) are labeled by a labeling scheme $(\mathcal{D}, \phi, \pi)$, we label the data as follows. For any data item x, let $\textsc{Output}(x)$ be the (unique) module that writes x as output and $\textsc{Inputs}(x)$ the set of all modules that read x as input. In other words, if $x \in \textsc{Data}(u, v)$ then $u = \textsc{Output}(x)$ and $v \in \textsc{Inputs}(x)$. We label x by

$$(\phi(\textsc{Output}(x)), \{\phi(v) \mid v \in \textsc{Inputs}(x)\})$$

where $\phi(\textsc{Output}(x))$ denotes the reachability label for the output module of x, and $\{\phi(v) \mid v \in \textsc{Inputs}(x)\}$ denotes the set of reachability labels for all input modules of x.

Using these labels, we can quickly determine the dependency between data as follows. A data item x depends on another data item x' if and only if there exists a module $v \in \textsc{Inputs}(x')$ such that $\pi(\phi(v), \phi(\textsc{Output}(x)) = 1$ (*i.e.*, there is a path from v to $\textsc{Output}(x)$). Similarly, we can determine the dependency between data and modules. For example, to decide if a data item x depends on a module v, we only need to check if $\pi(\phi(v), \phi(\textsc{Output}(x))) = 1$.

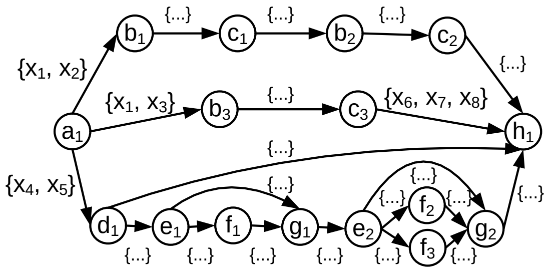

Figure 11: Query Data Dependency

Example 10. Figure 11 shows the running example of R, where each edge is associated with a set of data items. Note that one data item may be read by multiple modules. For instance, x_1 appears on both (a_1, b_1) and (a_1, b_3). Therefore, by the above labeling scheme, x_1 will be labeled by $(\phi(a_1), \{\phi(b_1), \phi(b_3)\})$, where ϕ denotes the given module labels. Similarly, x_6 will be labeled by $(\phi(c_3), \{\phi(h_1)\})$. To tell if x_6 depends on x_1, we need to check if at least one of $\textsc{Inputs}(x_1) = \{b_1, b_3\}$ can reach $\textsc{Output}(x_6) = c_3$. That is, if $\pi(\phi(b_1), \phi(c_3)) = 1$ or $\pi(\phi(b_3), \phi(c_3)) = 1$.

While the construction time for data labels remains linear in the size of the input $(\Sigma_e|\textsc{Data}(e)|)$, the label length will increase by a factor of $k + 1$ and the query time by a factor of k, where k is the maximum number of input modules of a data item (*i.e.*, $k = \max_x(|\textsc{Inputs}(x)|)$. Note that k is bounded by the maximum outdegree of a vertex in the run.

7. LABELING THE SPECIFICATION

In order to apply our skeleton-based approach, one question remains to be answered: How to label the specification? Since any labeling scheme for acyclic flow networks may require labels of linear length, obtaining an optimal labeling scheme for specifications is impossible in general.

Fortunately, within the skeleton-based labeling framework, we do not need to worry about the quality of the scheme used to label the specification, due to the following three observations. *First, the size of the specification is expected to be much smaller than the size of the run.* For instance, the largest workflow specification we have collected has fewer than 150 vertices, while due to the fork and loop behavior, a run of this workflow may have more than $10K$ vertices. *Second, the cost (both storage space and construction time) of labeling a specification can be amortized over a large number of runs.* Since workflows tend to be repeatedly executed, we need to label each specification only once and the resulting skeleton labels can be reused by all runs of this specification. We thus assume in Lemma 4.7 that each skeleton label, when used to label the run, can be encoded by $\log n_G$ bits regardless of its actual size ($\Omega(n_G)$ bits). *Third, reachability queries on the run may be frequently answered using only the context encodings without comparing the skeleton labels.* Therefore, our approach is effective even when reachability queries on the specification are slow. Interestingly, our experimental results show that given a specification, queries on large runs may even be faster than queries on small runs. A more detailed explanation is given in Section 8.2.

Hence any reasonable labeling scheme can be used to label the specification. Here we briefly describe two extreme and well-known approaches, which entail an expensive encoding and decoding step respectively.

TCM. The first approach, called TCM, precomputes the transitive closure matrix M of the specification graph G (where $M[i][j] = 1$ if the i-th vertex can reach the j-th vertex and 0 otherwise) and assigns the i-th row $M[i]$ as the reachability label of the i-th vertex. While this approach achieves a constant query time over the specification (*i.e.*, $t_G = O(1)$), the label length is n_G and the construction time is $O(\min\{m_G \times n_G, n_G^{2.376} \times \log n_G\})$[5].

BFS/DFS. The other approach, called BFS or DFS, answers a reachability query in G by standard graph traversal in a BFS or DFS manner. Note that this approach may or may not be used as a labeling scheme. If it is used as a labeling scheme, then the entire graph G has to be coded in each label. Since no extra index structure is used, we can treat the label length and construction time to be zero, but the query time using BFS or DFS will be linear in terms of the size of the specification (*i.e.*, $t_G = O(m_G + n_G)$).

The experimental results that follow show that even when the above two simple schemes are used, our skeleton-based labeling scheme is scalable to label a run with up to $100K$ vertices. Links to more sophisticated labeling schemes for general graphs are given in Section 2. These schemes generally provide a better tradeoff between the efficiency of encoding and decoding, and therefore, if used to label the specification, can further improve the overall labeling performance.

8. EXPERIMENTAL EVALUATION

All experiments were performed on a local PC with Intel Pentium 2.80GHz CPU and 2GB memory running Windows XP. All labeling schemes are implemented in Java. Our experiments are conducted on both real and synthetic datasets.

Real Dataset. We collected real-life scientific workflows from the myExperiment repository [14], which contains a large public collection of workflows across multiple workflow systems, including Taverna, Kepler and Triana.

Synthetic Dataset. We randomly generated synthetic workflows using the following parameters: (1) n_G: the number of vertices; (2) m_G: the number of edges; (3) $|T_G|$: the size of the fork and loop hierarchy (*i.e.*, the total number of forks and loops plus one); and (4) $[T_G]$: the depth of the fork and loop hierarchy.

To simulate the execution of a workflow, we randomly replicated each fork or loop one or more times. For each specification, we generated a set of runs, varying their size (*i.e.*, the number of vertices) from 0.1K to 102.4K by a factor of 2. Both the specification and runs are stored as XML files. In all experiments, the time to parse the XML file is omitted.

We apply the three labeling schemes (and their different combinations) discussed earlier in the paper: (1) TCM, which precomputes the transitive closure matrix; (2) BFS, which traverses the graph by BFS; and (3) SKL, which denotes our proposed skeleton-based labeling scheme. To evaluate their performance, we measured three parameters: label length, construction time and query time. In all results reported in this section, each point for label length and construction time is an average over 10^3 sample runs, and each point for query time is an average over 10^6 sample queries.

8.1 SKL Performance

In the first set of experiments, we measure the performance of SKL using six selected, real-life scientific workflows. Characteristics of these specifications are listed in Table 1. We first label the specification using TCM, and then label the runs using SKL. Note that the cost of labeling the specification is omitted in this experiment. We only report the results for the QBLAST workflow. Results for other workflows are similar.

Figure 12 reports the maximum and average label length [6] for the QBLAST workflow. As expected, both increase logarithmically with the size of the run, and the average label length is always within a small constant of the maximum. More interestingly, as shown in the proof of Lemma 4.7, the tight upper bound of label length is $3 \log n_T^+ + \log n_G$ (rather than $3 \log n_R + \log n_G$), where n_T^+ is the number of nonempty + nodes in the execution plan and $n_T^+ \leq n_R$. This is confirmed by Figure 12: The actual maximum label length is less than the asymptotic upper bound $3 \log n_R$ (*i.e.*, the dotted line) by a small constant.

Figure 13 reports the construction time for the QBLAST workflow under two experiment settings. In the default setting, the run is given as a graph. In the other setting, we assume that the run is given along with its execution plan and context. Figure 13 shows that the construction time increases proportionally with the size of the run, confirming our linear time complexity result in Lemma 4.7. Comparing the two settings, we also observe that the main time cost of skeleton-based labeling lies in the computation of the execution plan and context. Hence, we can significantly speed up the labeling process if this run-time information is readily available. For example, in Taverna [9], the execution plan and context can be directly extracted from the system log.

Figure 14 reports the query time for the QBLAST workflow. Since the specification is labeled by TCM, it achieves a constant query time over the specification (*i.e.*, $t_G = O(1)$ in Lemma 4.7). Hence, the overall query time over the run is also constant, which is confirmed by Figure 14.

To conclude, this experiment empirically validates that SKL guarantees logarithmic label length, linear construction time, and constant query time.

[5]Using matrix multiplication [7], we can compute the transitive closure matrix in $O(n_G^{2.376} \times \log n_G)$ time.

[6]The average length is measured only for the variable-size labels.

Table 1: Characteristics of Real-life Scientific Workflows

	n_G	m_G	$\|T_G\|$	$[T_G]$
EBI	29	31	4	2
PubMed	35	45	3	3
QBLAST	58	72	6	3
BioAID	71	87	10	4
ProScan	89	119	9	4
ProDisc	111	158	9	3

Table 2: Complexity Comparison (with Amortized Cost)

	Label Length	Construction Time	Query Time
TCM+SKL	$3\log n_R + \log n_G + \frac{n_G^2}{k \times n_R}$	$O(m_R + n_R + \frac{m_G \times n_G}{k})$	$O(1)$
BFS+SKL	$3\log n_R + \log n_G$	$O(m_R + n_R)$	$O(m_G + n_G)$
TCM	n_R	$O(m_R \times n_R)$	$O(1)$
BFS	0	0	$O(m_R + n_R)$

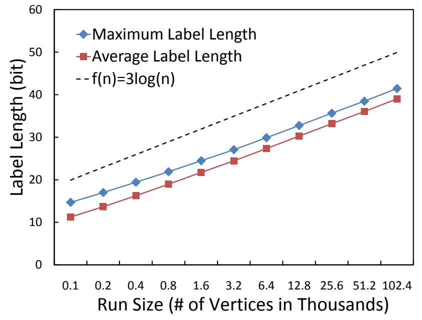

Figure 12: Real Scientific Workflow (Label Length for QBLAST)

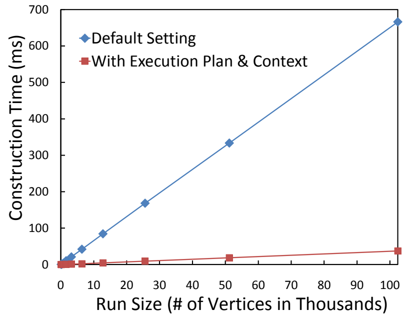

Figure 13: Real Scientific Workflow (Construction Time for QBLAST)

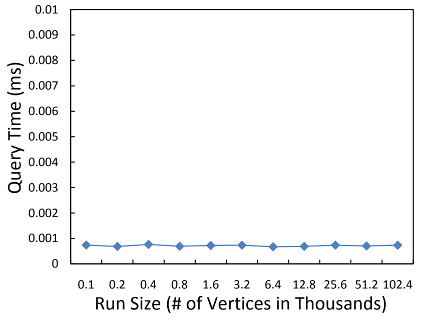

Figure 14: Real Scientific Workflow (Query Time for QBLAST)

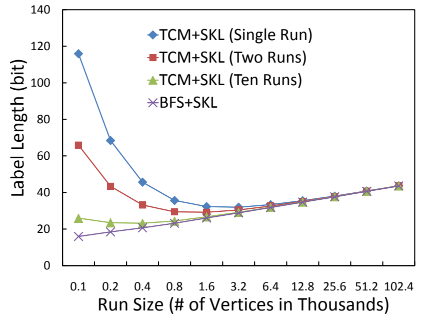

Figure 15: TCM+SKL vs BFS+SKL (Label Length)

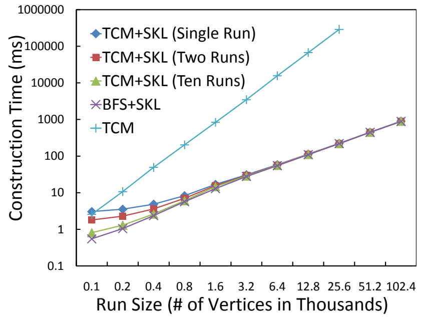

Figure 16: TCM+SKL vs BFS+SKL (Construction Time)

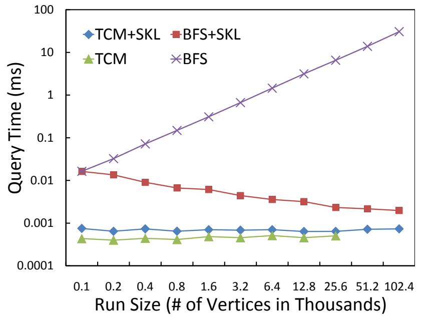

Figure 17: TCM+SKL vs BFS+SKL (Query Time)

8.2 TCM + SKL vs BFS + SKL

In the second set of experiments, we compare different combinations with SKL: (1) TCM+SKL and (2) BFS+SKL, which first label the specification using TCM or BFS and then label the run using SKL. To demonstrate the scalability, we also compare them with two extreme approaches: (3) TCM and (4) BFS, which label the run using TCM or BFS directly. Note that to obtain a fair comparison, we count the initial cost (both storage space and construction time) of labeling a specification, and amortize this cost over all the runs. We analyze the complexity of these four labeling schemes (with amortized cost) in Table 2, where k is the number of runs. This experiment was conducted on a synthetic workflow with $n_G = 100$, $m_G = 200$, $|T_G| = 10$ and $[T_G] = 4$. We measured the amortized cost over a single run, two runs and ten runs respectively.

Figure 15 reports the maximum label length (with amortized cost) for TCM+SKL and BFS+SKL. As expected, BFS+SKL builds much shorter labels than TCM+SKL for small runs, but when the run becomes large, they perform equally well. This is because, as shown in Table 2, the amortized storage cost for TCM+SKL is $n_G^2/(k \times n_R)$. It dominates the total label length for small runs, but as the run becomes large, quickly decreases to zero. Comparing the three curves for TCM+SKL, we also observe that this amortized cost can be significantly reduced, if the skeleton labels are reused by many runs. Note that the label length for BFS+SKL is not affected by the number of runs. We see a simple logarithmic increase for BFS+SKL, confirming our complexity result in Table 2.

Figure 16 reports the construction time (with amortized cost) for TCM+SKL and BFS+SKL, and shows a similar result to Figure 15. Also observe that both TCM+SKL and BFS+SKL label the runs faster than TCM [7] by several orders of magnitude. Note that TCM has a polynomial time complexity (X and Y axises use a logarithmic scale).

Figure 17 reports the query time for all four labeling schemes. Not surprisingly, both TCM+SKL and TCM achieve a truly constant query time. Note that TCM+SKL is slightly slower than TCM, due to a more complex decoding step. In contrast, BFS+SKL is slower than TCM+SKL, especially for small runs; BFS suffers from a linear query time, and is slower than the other three approaches by several orders of magnitude. More interestingly, we can observe a slightly decreasing query time for BFS+SKL. This counter-intuitive behavior

[7]In our experiments, TCM is only scalable to runs with no more than 25.6K vertices, due to memory constraints.

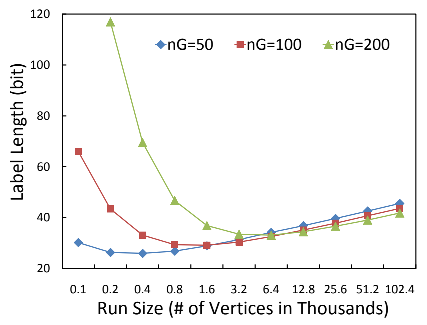

Figure 18: Influence of Specification (Label Length for TCM+SKL)

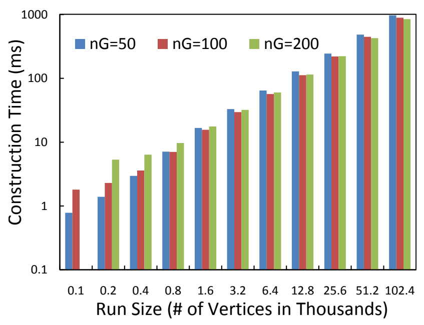

Figure 19: Influence of Specification (Construction Time for TCM+SKL)

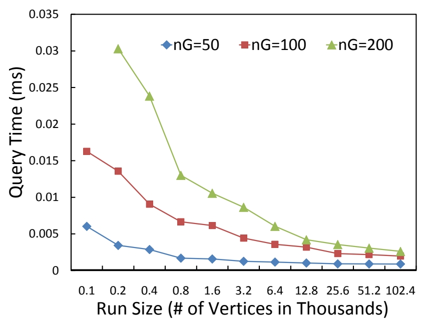

Figure 20: Influence of Specification (Query Time for BFS+SKL)

is explained as follows: As discussed in Section 7, we may avoid the expensive graph search on the specification if the reachability between two vertices can be immediately determined by their context encodings, *i.e.*, if the vertices are dominated by two distinct copies of the same fork or loop. Since large runs may have more fork and loop copies, the chance to avoid the expensive graph search increases with the size of the run. In particular, when $n_R = n_G$, we have to search the specification graph for each query (there is no chance). This results in exactly the same performance as BFS, which is confirmed by Figure 17.

To conclude, SKL (combined with either TCM or BFS) is scalable to large runs: For a run with 102.4K vertices, it builds labels shorter than 50 bits in 1s, and answers a query in 0.01ms [8]. The comparison between TCM+SKL and BFS+SKL shows that there is no clear winner, and only a small tradeoff between the efficiency of encoding and decoding. In particular, when labeling large runs, SKL is insensitive to the quality of the labeling scheme used to label the specification.

8.3 Influence of Specification

In the last set of experiments, we evaluate the influence of specifications on the performance of SKL. Among all parameters of a specification, we focus only on n_G. We generated three synthetic workflows by setting $n_G = 50$, 100 and 200 respectively and fixing $m_G/n_G = 2$, $|T_G| = 10$ and $[T_G] = 4$. Note that we only measured the label length and construction time (both with amortized cost over 2 runs) for TCM+SKL and the query time for BFS+SKL; As shown in Table 2, only those parameters are significantly affected by n_G.

Figure 18 reports the maximum label length (with amortized cost) for TCM+SKL. We can observe that the smaller specification results in much shorter labels for small runs, but slightly longer labels for large runs. This is because for small runs, the amortized storage cost for skeleton labels (*i.e.*, $n_G^2/(k \times n_R)$) dominates the total label length. Clearly, the smaller specification uses shorter skeleton labels. In contrast, for large runs, the context encoding (*i.e.*, $3 \log n_R$) dominates the total label length. Since the smaller specification may contain smaller forks and loops, to obtain a run with the same size, we must replicate more fork and loop copies. Hence, the run of a smaller specification may have a larger execution plan, and therefore is labeled by larger numbers as the context encoding.

[8] In real database systems, the query time may be much longer due to the slow data loading from the database.

Figure 19 reports the construction time (with amortized cost) for TCM+SKL, and shows a similar result to Figure 18.

Figure 20 reports the query time for BFS+SKL. As expected, the query time increases with the size of the specification, since the main query cost lies in the graph search over the specification. Clearly, a smaller specification allows a faster graph search. We can also observe that the query time decreases with increasing size of the run (similar to Figure 17). When the run becomes large, the three specifications achieve very close query times.

To conclude, when the size of the run is close to the size of the specification, SKL works significantly better with smaller specifications; but for large runs, the size of specification has a very weak influence on the overall performance of SKL.

9. CONCLUSIONS

In this paper, we present a skeleton-based labeling scheme for workflows with well-nested forks and loops, which uses a labeled specification to efficiently create compact reachability labels for large runs. For any fixed specification, our scheme for labeling runs is optimal in the sense that it uses logarithmic length labels, takes linear construction time, and answers queries in constant time, even though runs can have arbitrarily complex network structure. The experimental results demonstrate the effectiveness of our approach.

An interesting direction for future work is to design efficient and compact *dynamic* or *online* labeling schemes, so that data can be labeled and stored in a database along with its label as soon as it is generated. Given that scientific workflows can take a long time to execute, this would enable efficient provenance queries on intermediate data results even before the workflow completes.

10. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments. This work was supported in part by US National Science Foundation grants IIS-0803524 and IIS-0629846.

11. REFERENCES

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *SODA*, pages 547–556, 2001.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD Conference*, pages 253–262, 1989.

[3] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *SODA*, pages 947–953, 2002.

[4] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD Conference*, pages 993–1006, 2008.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[7] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, pages 1–6, 1987.

[8] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD Conference*, pages 1007–1018, 2008.

[9] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[10] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[11] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[12] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[13] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *SODA*, pages 954–963, 2002.

[14] D. D. Roure, C. A. Goble, and R. Stevens. The design and realisation of the my$_{\text{experiment}}$ virtual research environment for social sharing of workflows. *Future Generation Comp. Syst.*, 25(5):561–567, 2009.

[15] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *Comput. J.*, 28(1):5–8, 1985.

[16] M. Thorup and U. Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001.

[17] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

[18] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.