Differencing Provenance in Scientific Workflows

Zhuowei Bao  
*University of Pennsylvania*, zhuowei@cis.upenn.edu

Sarah Cohen-Boulakia  
*Université Paris-Sud*

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

Anat Eyal  
*University of Pennsylvania*

Sanjeev Khanna  
*University of Pennsylvania*, sanjeev@cis.upenn.edu

Follow this and additional works at: [http://repository.upenn.edu/cis_reports](http://repository.upenn.edu/cis_reports)

Recommended Citation


This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_reports/961](http://repository.upenn.edu/cis_reports/961)

For more information, please contact repository@pobox.upenn.edu.
Differencing Provenance in Scientific Workflows

Abstract
Scientific workflow management systems are increasingly providing the ability to manage and query the provenance of data products. However, the problem of differencing the provenance of two data products produced by executions of the same specification has not been adequately addressed. Although this problem is NP-hard for general workflow specifications, an analysis of real scientific (and business) workflows shows that their specifications can be captured as series-parallel graphs overlaid with well-nested forking and looping. For this natural restriction, we present efficient, polynomial-time algorithms for differencing executions of the same specification and thereby understanding the difference in the provenance of their data products. We then describe a prototype called PDiff View built around our differencing algorithm. Experimental results demonstrate the scalability of our approach using collected, real workflows and increasingly complex runs.

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/961
Differencing Provenance in Scientific Workflows

Zhuowei Bao #1, Sarah Cohen-Boulakia *2, Susan B. Davidson #3, Anat Eyal#4, Sanjeev Khanna #5

*University of Pennsylvania, USA
1zhuowei, 3susan, 4anate, 5sanjeev@seas.upenn.edu

1Université Paris-Sud, France
2cohen@lri.fr

Abstract—Scientific workflow management systems are increasingly providing the ability to manage and query the provenance of data products. However, the problem of differencing the provenance of two data products produced by executions of the same specification has not been adequately addressed. Although this problem is NP-hard for general workflow specifications, an analysis of real scientific (and business) workflows shows that their specifications can be captured as series-parallel graphs overlaid with well-nested forking and looping. For this natural restriction, we present efficient, polynomial-time algorithms for differencing executions of the same specification and thereby understanding the difference in the provenance of their data products. We then describe a prototype called PDiffView built around our differencing algorithm. Experimental results demonstrate the scalability of our approach using collected, real workflows and increasingly complex runs.

I. INTRODUCTION

Answering scientific questions – e.g. Is a given gene involved in a syndrome? What are the evolutionary relationships between a set of species? What is the impact of a set of chemical compounds on a cell? – involves conducting a complex set of analyses or “in-silico” experiments. Such experiments are typically defined as workflows and executed repeatedly. Each execution may vary the parameters and data inputs to the tools used as modules in the workflow; furthermore, alternative paths of the workflow may be followed. In this process, the scientist’s goal is to identify parameter settings and approaches which lead to “good” biological results. Comparing workflow runs and understanding the difference between two executions of an experiment is thus of paramount importance to scientists.

To manage these complex experiments as well as the large number of intermediate and final data products they produce, a number of workflow systems have been developed for scientific applications which provide support to track provenance of derived data products. To understand the similarities and differences of these systems with respect to provenance, a Provenance Challenge Workshop was held [1]. One of the challenge queries was the differencing problem for a dataflow, the execution model commonly supported in scientific workflow systems. While most of the participating systems gave reasonable answers for this simple model, the techniques used do not extend to more complex execution models, i.e., those that support forked executions over an unknown number of elements of an input set (implicit iteration), looping until some condition is met (explicit iteration), and parallel executions.

As an example of a complex workflow, consider a classical scientific analysis involving protein annotation (Fig. 1(a)). The aim of this analysis is to infer the biological function of a new sequence from other sequences. The underlying biological assumption is that a protein’s biological function is a composition of the biological function of its domains (consecutive parts of its sequence). The main steps are: The user provides the protein sequence (1), which is converted into Fasta format (2) and compared using the BLAST similarity tool against protein sequences in the major proteomic resources: SwissProt (3), TrEMBL (4), and PIR (5). The most similar protein sequence (Top-1) found is selected (6) and BLASTed against SwissProt/PIR/TrEMBL sequences (“reciprocal best hits”). The result is compared with previous ones until a set of very close proteins is found. The following steps are done for each sequence output of step (6), independently from each other. If proteomic domains are unknown, they are searched for in major domain resources such as ProDom (8) or PFAM (9); otherwise these steps are skipped. Domain sequences are then extracted (10) and each domain is used as input to be annotated by ontologies (Gene Ontology (11) then FunCat (12)) or using enzymatic terms (Brenda (13) then Enzyme (14)). Annotations obtained are eventually gathered (15).

The graph in Fig. 1(a) uses a dataflow notation annotated with control flow information for forks and loops. A loop is indicated by a dotted back arrow, e.g., from module 6 (collect-Top1&Compare) to module 2 (FastaFormat), and forking is indicated by a dotted oblong, e.g., the oblong around module 3 (BlastSwP) indicates that similar proteins can be searched for simultaneously. Note that this workflow could also be expressed using BPEL [2], a standard which is becoming increasingly popular within bioinformatics. However, to simplify the presentation we will use a simpler notation that is also closer to what is used in most scientific workflow systems.

In a run, loops are unrolled and the number of fork executions is given explicitly. For example, two runs of the protein annotation workflow specification are shown in Fig. 1(b) and (c). Observe that Run (b) has two fork executions between modules 6 and 15, while Run (c) has two executions of the loop from module 6 to module 2.

In a dataflow execution, module names do not repeat and there is an immediate pairing between nodes in the two executions. Therefore, the naive approach of taking the difference of the nodes and edges in the two runs to calculate their difference works well. However, for the runs in Fig. 1 this approach does not work since node names repeat and hence there are many possible ways of pairing nodes. To determine
the best pairing of nodes, a global computation must be performed to match copies that are most similar overall in terms of the control structure and dataflow.

The difference or edit distance between a pair of valid runs of the same specification is defined as a minimum cost sequence of edit operations that transforms one run to the other. While many edit operations could be considered (e.g., insert or delete a node, and insert or delete an edge), it is important that they transform a valid run to another valid run, are atomic, and are complete. While inserting or deleting a node or an edge are atomic operations that can be used to transform between any two valid runs, they do not guarantee the validity of intermediate results. We use therefore as edit operation the insertion or deletion of elementary paths.

This notion of edit distance has a simple appealing interpretation: It is the shortest path connecting the given pair of runs in the space of all valid runs, where two valid runs are adjacent if they differ by a single elementary path.

While differencing network flows is NP-hard for general graphs [3], the structure of most workflows can be captured as a series-parallel graph (SP-graph) overlaid with well-nested forks and loops. Such graphs capture the structure of most scientific workflows we have encountered in practice as well as well-structured business process and other workflows [4]. For this natural restriction, we present efficient, polynomial-time algorithms for differencing workflow runs of the same specification. The algorithms are based on a well-known tree representation of SP-graphs in which internal nodes are annotated with series (in which the children are ordered) or parallel (in which case the child nodes are unordered). We then add annotations to represent loop (ordered) and fork (unordered) executions (annotated SP-trees).

In addition to capturing well-structured workflows, SP-graphs are in some sense the most complex graphs that allow efficient differencing algorithms: The simplest graph that is not an SP-graph has four nodes [5], and the differing problem already becomes NP-hard on this graph [3].

An equally important difference in the provenance of two data products are parameter settings and input data sets. Two executions could have exactly the same control flow but produce very different results due to the data used. Data affects the differing problem in two ways: It is a factor in the matching between nodes in the executions; and once the matching is done the data differences can be highlighted as annotations on nodes (for parameter settings) and edges (for data flowing between modules). For simplicity of presentation, however, we will focus solely on control flow in this paper.

A. Contributions and Overview

Our contributions are four-fold: First, we present a model of workflows that is sufficiently general to capture workflows that we have encountered in practice and collected from articles and sample workflows on the web (Section III). Second, for this model of workflows we present efficient, polynomial-time algorithms for differencing workflow executions, first considering forks (Sections IV and V), and then extending the techniques for loops (Section VI). Our algorithms work under fairly general cost models, allowing us to capture a variety of application-specific notions of distance. Third, we describe a prototype called Provenance Difference Viewer (PDiffView) built around our differencing algorithm (Section VII). Fourth, we provide experimental results showing the scalability of our approach and the effect of the cost model (Section VIII).

II. RELATED WORK

Related work includes differencing strings [6], [7], [8], [9], trees, and programs. Work on recording the edit history between workflow versions has also been studied [10] and extended to runs of different specifications [11]. Computing deltas of RDF graphs was studied in [12]. In this paper we build on tree differencing techniques, which can handle hierarchically structured data [13].

Ordered tree differencing has been studied extensively (e.g. [14], [15], [16], [17]). [14] first formulated the edit problem between two ordered labeled trees as a generalization of the string edit problem; a constrained edit model that preserves the validity of the data was shown in [18]. Various restricted cost models and edit operations have been proposed, including [19] where insertions and deletions are restricted to leaves, and [20] where deleting (inserting) a node deletes (inserts) the subtree rooted at this node. Our work differs in the model considered (graph rather than tree) and edit operations considered.

The differing problem for unordered trees is known to be NP-hard for the general case [21]. By constraining the possible mappings between the two trees, so that disjoint subtrees are mapped to disjoint subtrees, [22] proposed a quadratic time algorithm. [23] introduced a less-constrained edit distance model which requires that matches are between nodes at the same level; this does not make sense for workflow runs. The constrained model we consider, series-parallel graphs, is similar to [22], however our cost model and edit operations are again different and therefore the differencing algorithm is different.

The problem of program differencing is also related. Most of the work focuses on a static comparison of two program versions (e.g. [24], [25], [26], [27]). Other algorithms that
compare different program executions (e.g. [28], [29], [30]) work at the binary level and cannot be used for workflow differencing.

Although many different notions of process equivalence have been proposed (e.g., trace equivalence, bisimulation, branching bisimulation, etc.), most give a true/false result. More related to our work is the process mining work of [31], [32], which develops a notion of quantified process equivalence. Process mining compares different models (specifications), whereas we compare different executions of the same model.

III. MODEL AND PROBLEM STATEMENTS

In this section, we introduce the general workflow model, and formulate the workflow difference problem. We then develop in some detail a natural restriction of the general problem, called the SP-workflow difference problem, that is studied in detail in the remainder of this paper.

A. Definitions and Notation

Given a node-labeled directed graph $G$, we let $V(G)$ denote the set of all nodes in $G$ and $E(G)$ denote the set of all edges in $G$. For any node $v$ in $V(G)$, let $\text{Label}(v)$ denote the label on $v$. In addition, let $s(G)$ and $t(G)$ denote the unique source node and unique sink node in $G$.

Definition 3.1: A flow network is a directed graph $G$ in which there exist a single source node $s(G) \in V(G)$ and a single sink source node $t(G) \in V(G)$, and every node $v \in V(G)$ lies on some path from $s(G)$ to $t(G)$.

A sub-class of flow networks that naturally arises when modeling program control and dataflow are series-parallel graphs.

Definition 3.2: A series-parallel graph (also called SP-graph) is a directed multigraph $G$ with a single source $s$ and a single sink $t$ (two terminals) that can be produced by a sequence of the following operations:

- **Basic SP-graph**: Create a new graph consisting of a single edge directed from node $s$ to node $t$.
- **Series Composition**: Given two SP-graphs $G_1$ and $G_2$ with sources $s_1$, $s_2$ and sinks $t_1$, $t_2$ respectively, form a new graph $G = S(G_1, G_2)$ by identifying $s = s_1$, $t_1 = s_2$ and $t = t_2$.
- **Parallel Composition**: Given two SP-graphs $G_1$ and $G_2$ with sources $s_1$, $s_2$ and sinks $t_1$, $t_2$ respectively, form a new graph $G = P(G_1, G_2)$ by identifying $s = s_1 = s_2$ and $t = t_1 = t_2$.

In this definition, $S$ and $P$ are two functions that take a pair of SP-graphs as input and produce their series or parallel composition as output. A straightforward induction on the number of operations used to produce the SP-graph shows that every SP-graph is an acyclic flow network.

The inductive definition of SP-graphs given above naturally lends itself to two special classes of subgraphs of SP-graphs.

Definition 3.3: Given an SP-graph $G$, a subgraph $H$ of $G$ is said to be a **series (parallel) subgraph** if $H$ is the result of a series (parallel) composition step above. In addition, we say any single edge (basic SP-graph) of $G$ is a series subgraph.

Example 3.1: We will use the simplified SP-workflow example shown in Fig. 2 in the remainder of this paper. All four graphs shown are SP-graphs, ignoring the dotted line and oblongs in (a). The number inside the circle indicates the label on the node. We use a superscript on labels to obtain a unique identifier for each node in a run.

B. General Workflow Model

A workflow model has two components: a specification that serves as a template for executions, and the set of valid runs for the given specification. Informally, a workflow specification consists of a set of different modules and defines the order in which they can be executed. A workflow run is a partial order of steps where each step is an instance of a module defined in the underlying specification, and the partial order conforms to the ordering constraints in the given specification.

Formally, a workflow specification is given by a flow network $G$ with unique labels on the nodes. Given a workflow specification $G$, a flow network $R$ with labels on the nodes (not necessarily unique) is said to be a **valid run** with respect to $G$ if $R$ is acyclic, and there exists a homomorphism $h : V(R) \rightarrow V(G)$ such that 1) $\forall v \in V(R), \text{Label}(v) = \text{Label}(h(v))$; 2) $h(s(R)) = s(G), h(t(R)) = t(G)$; and 3) $\forall (u, v) \in E(R), (h(u), h(v)) \in E(G)$.

Notice that even if the specification $G$ has cycles, a valid run $R$ is always acyclic, since we unfold the cycles in the specification to capture the sequential order of all iterations in a workflow run. Consequently, the node labels in a run $R$ are not necessarily unique.

C. The Workflow Difference Problem

The goal of the workflow difference problem is to find the edit distance and a path script between two valid runs of the same specification. We begin by defining two edit operations, and then propose a cost model for them. Our notion of edit distance has a simple appealing interpretation: It is the shortest path connecting the given pair of valid runs in the space of all valid runs, where two valid runs are adjacent iff one can be transformed into another by a single edit operation and the length of each edge is given by the cost model.

![Fig. 2. SP-workflow specification and runs](image-url)
1) Edit Operations and Edit Script: In the following, we assume that \( R_1 \) and \( R_2 \) are valid runs with respect to a specification \( G \), and use the notion of an elementary path:

**Definition 3.4**: Given a valid run \( R \) with respect to a specification \( G \), a path \( p \) is said to be an elementary path in \( R \) iff 1) each internal node on \( p \) has exactly one incoming edge and one outgoing edge; and 2) \( s(p) \) has at least two outgoing edges and \( t(p) \) has at least two incoming edges.

An edit operation \( \omega \) applied to a valid run \( R_1 \) to produce another valid run \( R_2 \) with respect to a specification \( G \) is written as \( R_1 \xrightarrow{\omega} G R_2 \). We consider the following two path edit operations:

- **Path Insertion**: A path insertion operation creates a new (elementary) path \( p \) between two existing nodes and is denoted by \( \Lambda \rightarrow p \). The restriction we impose on \( p \) is that it is an elementary path in \( R_2 \).
- **Path Deletion**: This operation is the inverse of the path insertion operation. A path deletion operation is denoted by \( p \rightarrow \Lambda \), where \( p \) is an elementary path to be deleted from a given run.

We define an edit script to be a sequence of zero or more edit operations. Formally, a sequence of path edit operations \( \mathcal{E} = \omega_1, \omega_2, \ldots, \omega_k \) is said to be an edit script from \( R_1 \) to \( R_2 \), written as \( R_1 \xrightarrow{\mathcal{E}} G R_2 \), if there exists a sequence of valid runs with respect to \( G \), say \( S_0, S_1, \ldots, S_k \), such that \( S_0 = R_1, S_k = R_2 \) and \( S_{i-1} \xrightarrow{\omega_i} G S_i \) for \( 1 \leq i \leq k \).

There are several principles that motivate our choice of edit operations. Firstly, they preserve the validity of the run. Other path edit operations, such as inserting a node, deleting a node, and relabeling a node, may violate the validity of the run, and hence make the notion of distance meaningless with respect to the underlying specification. Secondly, they are atomic. More complex operations can be decomposed to a sequence of elementary path edit operations. For example, one could define a path replacement operation that replaces one path by another or a subgraph insertion operation that creates an SP-graph between two nodes in one step. Such operations may be detected by post-processing the output of our algorithm. Finally, they are complete. One can easily show that every pair of valid runs can be transformed from one to another by this set of operations. These two elementary path edit operations are therefore a natural choice.

2) Cost Model and Edit Distance: Given two valid runs, there may be many edit scripts that transform one to another. Among them, we are interested in finding one with the minimum cost. To this end, we introduce a cost model for edit operations and edit scripts.

There is a tradeoff between the generality of the cost model and the difficulty in computing a minimum-cost edit script. For example, a simple unit cost model would assign each edit operation a cost of one, and the cost of an edit script would be the number of its operations. On the other hand, a very general cost model would have a user-defined function to determine the cost of each edit operation, based on the type of the edit operation, as well as the particular path on which it operates. However, since the number of paths in an SP-graph can be exponentially large, we need a cost function with a compact representation that is still general.

The model we will therefore use is that the cost of each edit operation is given by a function \( \gamma \) that is determined by both the length of the elementary path to be edited, and the labels on its two terminals. That is, for all elementary paths \( p \),

\[
\gamma(p \rightarrow \Lambda) = \gamma(|p|, \text{Label}(s(p)), \text{Label}(t(p)))
\]

In addition, we constrain \( \gamma \) to be a distance metric with respect to elementary path insertions and deletions, which satisfies the following conditions:

1) **non-negativity**: \( \gamma(p \rightarrow \Lambda) \geq 0 \);
2) **identity**: \( \gamma(p \rightarrow \Lambda) = 0 \) iff \( |p| = 0 \) and \( s(p) = t(p) \);
3) **symmetry**: \( \gamma(p \rightarrow \Lambda) = \gamma(p \rightarrow \Lambda) \); and
4) **quadrangle inequality**: for all elementary paths \( p_1, p_2, p_3 \) such that \( p_1 \circ p_2 \circ p_3 \) and \( p_1 \circ p_2 \circ p_3 \) are well-defined, \( \gamma(p_1 \rightarrow p_2 \circ p_3) \leq \gamma(p_1 \rightarrow p_2) + \gamma(p_2 \rightarrow p_3) \).

In terms of our cost function, we can rephrase the quadrangle inequality as follows: For all labels \( A, B, C, D \) and \( l_1, l_2, l_3 \) such that there exist in the specification elementary paths \( p_1 \) from \( A \) to \( B \) of length \( l_1 \), \( p_2 \) from \( B \) to \( C \) of length \( l_2 \), \( p_3 \) from \( B \) to \( C \) of length \( l_3 \), as shown in Figure 4, we have

\[
\gamma(l_1 + l_2 + l_3, A, D) \\
\leq \gamma(l_1 + l_2 + l_3, A, D) + \gamma(B, C, l_2) + \gamma(B, C, l_2)
\]

![Fig. 3. A path edit script from \( R_1 \) to \( R_2 \)](image)

![Fig. 4. Quadrangle inequality](image)
The quadrangle inequality essentially says that the cost of inserting an elementary path \( p \) directly is never more than the cost of inserting another elementary path \( p' \), and then replacing a part of \( p' \) to make it identical to \( p \).

Our cost model is general enough to capture a wide spectrum of cost functions. For example, any sublinear function \( \gamma(l, A, B) = l^d \) where \( \epsilon \leq 1 \) is eligible. When \( \epsilon = 0 \), this is exactly the unit cost function mentioned above.

Finally, the cost of an edit script is the sum of the costs of its individual operations. To express this, we extend the cost function \( \gamma \) to an edit script \( \mathcal{E} \) by letting \( \gamma(\mathcal{E}) = \sum_{\omega \in \mathcal{E}} \gamma(\omega) \).

**Definition 3.5:** Given a cost function \( \gamma \), the edit distance between \( R_1 \) and \( R_2 \), denoted by \( \delta(R_1, R_2) \), is defined as the minimum cost of an edit script from \( R_1 \) to \( R_2 \). Formally, \( \delta(R_1, R_2) = \min\{\gamma(\mathcal{E}) \mid R_1 \xrightarrow{\mathcal{E}} R_2\} \).

**Problem Statement:** Given two valid runs \( R_1 \) and \( R_2 \) with respect to a specification graph \( G \), and a cost function \( \gamma \), we want to compute the edit distance \( \delta(R_1, R_2) \) as well as the corresponding minimum-cost edit script from \( R_1 \) to \( R_2 \).

**D. The SP-Workflow Difference Problem**

The problem of computing the workflow difference under a general workflow model is at least as hard as subgraph isomorphism, a well known NP-hard problem. Indeed, the workflow difference problem becomes NP-hard when the specification graph is not an SP-graph. In fact, we will show that the NP-hardness holds on a specification graph with four nodes, with a very simple cost function, namely, one that assigns each elementary path a cost equal to its length. Interestingly, the specification graph underlying our reduction is precisely the forbidden minor for a directed acyclic SP graph [5].

**Theorem 1:** The Workflow Difference Problem is NP-hard for general flow graphs.

**Proof:** Consider the specification graph \( G_s = (V_s, E_s) \) where \( V_s = \{s, v_1, v_2, t\} \) and \( E_s = \{(s, v_1), (s, v_2), (v_1, v_2), (v_1, t), (v_2, t)\} \). We will do a reduction from the bipartite clique problem where we are given an undirected bipartite graph \( H = (X \cup Y, E) \) with \( |X| = |Y| = n \), and an integer \( \ell \). The goal is to decide if there is an \( \ell \times \ell \) bipartite clique in \( H \). We construct the two runs \( R_1, R_2 \) to encode this as a workflow difference problem. For run \( R_1(V_1, E_1) \), the vertex set \( V_1 \) is \( X \cup Y \cup \{s_1, t_1\} \) and there is a directed edge \((x, y) \in E_1 \) if \((x, y) \in E \). Moreover, we add a directed edge from \( s \) to each node in \( X \cup Y \) and from each node in \( X \cup Y \) to \( t \). Each node in \( X \) is given the same label as \( v_1 \), and each node in \( Y \) is given the same label as \( v_2 \). The nodes \( s_1, t_1 \) are given the same label as the node \( s, t \) in \( G_s \). For run \( R_2(V_2, E_2) \), the vertex set \( V_2 \) is \( X' \cup Y' \cup \{s_2, t_2\} \) where \( |X'| = |Y'| = \ell \). There is a directed edge \((x', y') \in E_2 \) for each \((x', y') \in X' \times Y' \). Moreover, we add a directed edge from \( s_2 \) to each node in \( X' \cup Y' \) and from each node in \( X' \cup Y' \) to \( t_2 \). Each node in \( X' \) is given the same label as \( v_1 \), and each node in \( Y' \) is given the same label as \( v_2 \). The nodes \( s_2, t_2 \) are given the same label as the node \( s, t \) in \( G_s \).

The cost function \( \gamma(p) \) is defined to be simply the number of edges on the path \( p \). Let \( m \) denote the number of edges in the graph \( H \). Then the cost of any edit script to convert \( R_1 \) into \( R_2 \) must be at least \( \Gamma = |E_1 \setminus E_2| = (m - \ell^2) + 4(n - \ell) \). We claim that there is an edit script of cost at most \( \Gamma \) iff \( H \) has an \( \ell \times \ell \) bipartite clique.

First consider the case when \( H \) has an \( \ell \times \ell \) bipartite clique. Let \( X_1 \subseteq X \) and \( Y_1 \subseteq Y \) denote the vertices of the bipartite clique in \( H \). Consider the following edit script for converting \( R_1 \) to \( R_2 \). We first delete all edges in \( E_1 \) which have one end-point in \( X \setminus X_1 \) and another in \( Y \) or one end-point in \( X \) and another in \( Y \setminus Y_1 \). Clearly, each such edge is an elementary path, and the total cost of these operations is \((m - \ell^2) \). Next we delete all elementary paths of length \( 2 \) that are of the form \((s_1 \rightarrow x \rightarrow t_1) \), where \( x \in X \setminus X_1 \). The total cost of these deletions is \(2(n - \ell) \). Similarly, we delete all elementary paths of length \( 2 \) that are of the form \((s_1 \rightarrow y \rightarrow t_1) \), where \( y \in Y \setminus Y_1 \). The total cost of these deletions is \(2(n - \ell) \). Thus the overall cost of this edit script is precisely \( \Gamma = (m - \ell^2) + 4(n - \ell) \). It is easy to see that at the end of these edit operations, the run \( R_1 \) is transformed into run \( R_2 \).

Now suppose that \( H \) does not have an \( \ell \times \ell \) bipartite clique. We will show that any edit script \( \mathcal{E} \) must have cost at least \( \Gamma + 2 \). Note that any edit script \( \mathcal{E} \) must delete at least \((n - \ell) \) edges of the form \((s_1, x) \) where \( x \in X \), and at least \((n - \ell) \) edges of the form \((y, t_1) \) where \( y \in Y \).

We consider two cases. Suppose the elementary paths in \( \mathcal{E} \) contain exactly \((n - \ell) \) edges of the form \((s_1, x) \) where \( x \in X \), and exactly \((n - \ell) \) edges of the form \((y, t_1) \) where \( y \in Y \). Then there is a subset \( X_1 \subseteq X \) of size \( \ell + 1 \) and a subset \( Y_1 \subseteq Y \) of size \( \ell \) such that the vertices in \( X_1 \) and \( Y_1 \) are never deleted by \( \mathcal{E} \). But since there is no \( \ell \times \ell \) bipartite clique in \( H \), the total number of edges in the set \( \{(x, y) \mid x \in X_1, y \in Y_1\} \) is at most \(\ell^2 - 1 \). Thus \( \mathcal{E} \) must insert at least one edge of the form \((x, y) \) with \( x \in X_1 \) and \( y \in Y_1 \). The total number of edges deleted by \( \mathcal{E} \) is at least \((m - (\ell^2 - 1)) + 4(n - \ell) \). Thus the total cost of \( \mathcal{E} \) is at least \(\Gamma + 2 \). On the other hand, suppose \( \mathcal{E} \) deletes at least \((n - \ell + 1) \) edges of the form \((s_1, x) \) where \( x \in X \). Then it must also insert an edge of the form \((s_1, x) \) for some \( x \in X \) since in \( R_2 \) there are \( \ell \) such edges. In addition, \( \mathcal{E} \) must delete at least \((m - \ell^2) + 3(n - \ell) \). Thus \( \mathcal{E} \) deletes at least \((n - \ell + 1) \) edges of \( \mathcal{E} \) to obtain \( R_2 \). Thus the cost of \( \mathcal{E} \) is at least \(\Gamma + 2 \). The case when \( \mathcal{E} \) deletes at least \((n - \ell + 1) \) edges of the form \((y, t_1) \) where \( y \in Y \) is analogous, and it can be similarly shown that the cost of \( \mathcal{E} \) is at least \(\Gamma + 2 \).

Hence the Workflow Difference problem is NP-hard.

**Definition 3.6:** Let \( F \) be a collection of subsets over a
ground set \( U \). Then \( \mathcal{F} \) is a **laminar family** if for any pair of sets \( H_1, H_2 \) in \( \mathcal{F} \), one of the following is true: (i) \( H_1 \subset H_2 \); or (ii) \( H_2 \subset H_1 \); or (iii) \( H_1 \cap H_2 = \emptyset \).

In the basic model, an **SP-workflow specification** is then given by a pair \((G, \mathcal{F})\), where \( G \) is an SP-specification graph with unique labels on the nodes, and \( \mathcal{F} \) is a laminar family of series subgraphs of \( G \) describing the well-nested set of allowed fork executions. Furthermore, we consider three kinds of executions for an SP-workflow specification \((G, \mathcal{F})\):

- **Series Execution**: For any series subgraph \( H \) of \( G \), a series execution of \( H \) executes its two sequential components in series. In the case where \( H \) is a basic SP-graph, it returns \( H \) itself as a valid run.

- **Parallel Execution**: For any parallel subgraph \( H \) of \( G \), a parallel execution of \( H \) executes either one of or both of its two branches in parallel.

- **Fork Execution**: For any series subgraph \( H \) of \( G \) belonging to \( \mathcal{F} \), a fork execution of \( H \) replicates one or more copies of \( H \) and executes them in parallel: They are split at the forking point (source) \( s(H) \) and then joined together at the synchronization point (sink) \( t(H) \), generating the parallel composition of one or more valid runs with respect to \( H \). Note that these runs (graphs) may differ from each other as long as they are all valid with respect to the same part of the specification. The fork execution is defined over series subgraphs of \( G \), since a forking over a parallel subgraph is equivalent to forking over each of its series component subgraphs.

We may abstract the above three executions by a **nondeterministic recursive** function, called an **execution function** (see Fig. 5), from SP-graphs to SP-graphs:

\[
\begin{align*}
f(H) &= \begin{cases} 
H & \text{if } H = (s(H), t(H)) \\
S(f(H_1), f(H_2)) & \text{if } H = S(H_1, H_2) \\
f(H_1) & \text{if } H = P(H_1, H_2) \\
\text{or } f(H_1) \text{ or } f(H_2) & \text{if } H = P(H_1, H_2) \\
P(f(H), f(H)) & \text{if } H \in \mathcal{F}
\end{cases}
\]

![Fig. 5. Execution function f](image)

A valid workflow run is now naturally defined as a graph that can be produced by applying a sequence of series, parallel, and fork executions recursively on the given SP-specification. Formally, given an SP-specification \((G, \mathcal{F})\), a node-labeled directed acyclic graph \( R \) is said to be a **valid workflow run** with respect to \((G, \mathcal{F})\) if \( R = f(G) \) where \( f \) is the execution function for \((G, \mathcal{F})\).

**Example 3.3**: Fig. 2 shows a pair of valid runs \( R_1 \) and \( R_2 \) that are both produced from the SP-specification \((G, \mathcal{F})\) by applying a sequence of series, parallel and fork executions. Note that in Fig. 2(a) the fork executions are defined over the series subgraphs \((2, 3, 6), (2, 4, 6), (2, 5, 6)\) and the entire graph \( G \). We defer the discussion of the loop implied by the dotted line to Section VI.

One can show by induction that any graph \( f(G) \) generated above is an SP-graph and admits a graph homomorphism to the specification graph \( G \). Thus, this new definition of the validity is consistent with our original definition for the general model. However, it further restricts the class of valid runs.

**IV. AN EQUIVALENT PROBLEM**

We now describe a well-known tree representation of SP-graphs [33]. By using SP-trees for both specifications and valid runs, we convert the SP-workflow difference problem into an equivalent edit distance problem on SP-trees.

**A. SP-trees**

The SP-tree representation \( T \) (a.k.a. tree decomposition) of an SP-graph \( G \) [33] captures the sequence of operations used to construct \( G \) as follows:

- If \( G \) is a basic SP-graph, then \( T \) is a single node \( v \) with Type\((v) = Q \).
- If \( G \) is the series or parallel composition of \( G_1 \) and \( G_2 \), then \( T \) has a root \( v \) with Type\((v) = S \) or \( P \), and its two children are the SP-trees for \( G_1 \) and \( G_2 \). The children of an \( S \) node are ordered while the children of a \( P \) node are unordered.

A linear time algorithm for the tree decomposition problem has been given by [33]. We abstract the decomposition as a recursive function \( h \) from SP-graphs to SP-trees:

\[
h(G) = \begin{cases} 
Q() & \text{if } G = (s(G), t(G)) \\
S(h(G_1), h(G_2)) & \text{if } G = S(G_1, G_2) \\
P(h(G_1), h(G_2)) & \text{if } G = P(G_1, G_2)
\end{cases}
\]

In this definition, the \( Q \), \( S \) and \( P \) functions applied to SP-trees create a new node with the corresponding type as the root, and make all input SP-trees the children of this root. Note that we use the same name as the \( S \) and \( P \) functions (applied to SP-graphs) defined in Definition 3.2, because they essentially perform the same compositions but on different domains.

A key observation is that the SP-tree representation of SP-graphs is not unique. We therefore compress a binary SP-tree into a **canonical SP-tree** by repeatedly merging two adjacent nodes with the same type. The canonical SP-tree representation of SP-graphs is unique [33] up to reordering of the children of a \( P \) node.

**Example 4.1**: The canonical SP-tree \( T \) for the SP-specification graph \( G \) (see Fig. 2(a)) is shown in Fig. 6(a). In this figure, we use a pair of node identifiers to denote the edge represented by each \( Q \) node (leaf).

In the remainder of this paper, given an SP-tree \( T \), we let \( T[v] \) denote the subtree rooted at a node \( v \) in \( T \) and let
\( p(v) \) denote the parent of \( v \). Also, we will use \( \text{Graph}(T) \) to represent the graph from which \( T \) is constructed, and use \( \text{Leaf}(T) \) to denote the leaves (\( Q \) nodes) of \( T \).

**B. Annotated SP-trees for Specifications**

An SP-workflow specification is given by an SP-graph, together with a laminar family over its series subgraphs to represent allowed fork executions. The canonical SP-tree for the SP-graph captures the series and parallel executions implied by this specification. To capture allowed forking, we annotate it using the given laminar family. In doing so, we use the following properties of series subgraphs.

**Lemma 4.1:** Let \( T \) be the canonical SP-tree for an SP-graph \( G \). Each series subgraph of \( G \) is represented either by a consecutive subsequence of two or more children of an \( S \) node in \( T \), or by a \( Q \) node (leaf) in \( T \).

**Proof:** (Sketch) A \( Q \) node trivially represents a series subgraph. For \( S \) nodes, note that a series composition is an associative operation e.g. \( S(S(G_1,G_2),G_3) \) is the same graph as \( S(G_1,S(G_2,G_3)) \). Using this rewriting, any consecutive subsequence of two or more children of an \( S \) node in \( T \) can be shown to represent a series subgraph of \( G \).

Given an SP-specification \((G,F)\), the **annotated SP-tree for \((G,F)\)** is obtained as follows: We first construct the canonical SP-tree for \( G \), and then, for each series subgraph in \( F \), insert an \( F \) node as a parent of the root of the subtree which represents this series subgraph. The detailed construction is described in Algorithm 1.

**Example 4.2:** The annotated SP-tree \( T_G \) for the SP-specification \((G,F)\) (see Fig. 2(a)) is shown in Fig. 6(b).

We can show by Lemma 4.1 that Algorithm 1 covers all scenarios when we insert an \( F \) node for an individual series subgraph. Furthermore, the property of laminar family, given in Definition 3.6, prevents potential conflicting annotations of two series subgraphs on the same tree.

**Lemma 4.2:** The annotated SP-tree for a specification has the following properties:

1) each internal node is an \( S \) or a \( P \) or an \( F \) node;
2) each leaf is a \( Q \) node;
3) each node has a different type from its parent;
4) each \( S \) or \( P \) node has at least two children; and
5) each \( F \) node has only one child of type either \( S \) or \( Q \).

**Proof:** The first two properties are obvious. Now consider Property 5. In Algorithm 1, we observe that any \( F \) node inserted in either case 1 or case 2 must have only one child. Moreover, by Lemma 4.1, we know that for the first case, the child of the \( F \) node inserted must be an \( S \) or a \( Q \) node, and for the second case, it must be an \( S \) node. Therefore, Property 5 holds. Properties 3 and 4 hold for all \( S \) and \( P \) nodes by the construction of a canonical SP-tree, and are not isolated by the insertion of \( F \) nodes. Property 5 shows that Property 3 also holds for any \( F \) node, and Property 3 holds trivially for \( Q \) nodes.

The annotated SP-tree is a semi-ordered tree: For an \( S \) node the left-to-right order among its children is significant, but for a \( P \) or an \( F \) node it is irrelevant.\(^1\) We thus say two annotated SP-trees \( T \) and \( T' \) are *equivalent*, \( T \equiv T' \), if they differ only in the order of children of \( P \) or \( F \) nodes.

**Lemma 4.3:** The annotated SP-tree representation of an SP-specification is unique. That is, if two annotated SP-trees \( T_G \) and \( T_G' \) for the same SP-specification \((G,F)\) are produced, then \( T_G \equiv T_G' \).

\(^1\)In the specification, \( F \) nodes can only have one child, however in a run they can have multiple children.
Proof: We prove this lemma by induction on $|\mathcal{F}|$. The base case where $|\mathcal{F}| = 0$ holds because of the uniqueness of the canonical SP-tree representation of SP-graphs. For the inductive step, let $H$ be a minimal subgraph in $\mathcal{F}$, $T_H$ be the canonical SP-tree of $H$, and $v, v'$ be the unique children of $F$ nodes representing the fork operation on subgraph $H$ in $T_G$, $T'_G$ respectively (i.e. $\text{Leaf}(T_G[v]) = \text{Leaf}(T'_G[v']) = \text{Leaf}(T_H$). Type$(v)$ = Type$(v') = F$. Type$(v)$ = Type$(v') = S$).

Since $H$ is minimal, $T_G[v]$ and $T'_G[v']$ do not contain any $F$ nodes and are canonical SP-trees of $H$, i.e. $T_G[v] = T'_G[v']$. Now remove from $T_G, T'_G$ the nodes $p(v), p(v')$ respectively, making $v, v'$ the child of their respective parents; if the new parents of $v, v'$ are $S$ nodes, then repeat this step. The resulting trees, $T''_G$ and $T''_G$, are annotated SP-trees of $(G, \mathcal{F} - \{H\})$.

By the inductive hypothesis, $T''_G \equiv T''_G$.

C. Annotated SP-Trees for Valid Runs

We now define a tree execution function $f'$ that takes the annotated SP-tree for a specification as input and produces as output the annotated SP-tree for a valid run. Formally, $f'$ is a nondeterministic recursive function from annotated SP-trees to annotated SP-trees:

$$f'(T) = \begin{cases} T & \text{if } T = Q() \\ S(f'(T_1), \ldots, f'(T_k)) & \text{if } T = S(T_1, \ldots, T_k) \\ P(f'(T_i), \ldots, f'(T_j)) & \text{if } T = P(T_1, \ldots, T_k) \\ F(f'(T_1), \ldots, f'(T_j)) & \text{if } T = F(T_1) \end{cases}$$

where $\{i_1, \ldots, i_j\}$ is a nonempty subset of $\{1, \ldots, k\}$ and $F$ takes one or more copies of $f'(T_1)$ as input.

Given a valid run $R$ with respect to an SP-specification $(G, \mathcal{F})$, the annotated SP-tree for $R$ is obtained as follows: we start by constructing the annotated SP-tree $T_G$ for $(G, \mathcal{F})$ and the canonical SP-tree $T_R$ for $R$, and then generate the annotated SP-tree $T_R$ for $R$ by a deterministic variant of the tree execution function $f''$ such that $f''(T_G, T_R) = T_R$.

Intuitively, $f''$ simulates the original nondeterministic tree execution function $f'$ and leads the tree derivation to the corresponding annotated SP-tree in terms of the given valid run. In each step of the tree derivation described in $f'$, we make the decision (e.g. which subset of children is chosen for a $P$ node, or how many copies are replicated for an $F$ node) by doing a case analysis on the current $T_G$ and $T_R$, matching zero or more subtrees in $T_R$ with each subtree in $T_G$ based on the leaves contained in each subtree. Note that even with a series composition, there may be multiple matches in the subtrees of $T_R$ due to a fork execution. The full definition of $f''$ is given in Algorithm 2.

Example 4.3: The annotated SP-trees $T_1$ and $T_2$ for the runs $R_1$ and $R_2$ (see Fig. 2(b) and (c)) are shown in Fig. 6(c) and (d) respectively.

Lemma 4.4: The annotated SP-tree for a valid run differs from the annotated SP-tree for an SP-specification in the following aspects:

1) A $P$ node is allowed to have only one child; and
2) An $F$ node is allowed to have more than one child but each child has the same type.

Algorithm 2 Annotated-SP-Tree-for-Valid-Run (a.k.a. $f''$)

**Input:** annotated SP-tree $T_G$

**Output:** annotated SP-tree $T_R$ of valid run $R$

1: $T_G = Q()$ then
2: $T_R = T_G$
3: end if
4: if $T_G = S(T_1, T_2, \ldots, T_k)$ then
5: $T_R = S(T_1', T_2', \ldots, T_k')$
6: for $i = 1$ to $k$ do
7: $X \leftarrow \{T_j' \mid \text{Leaf}(T_j') \cap \text{Leaf}(T_i) \neq \phi\}$
8: if $|X| > 1$ then
9: $T_i' \leftarrow S(X)$
10: else
11: $T_i' \leftarrow \text{element}(X)$
12: end if
13: end for
14: $T_R \leftarrow P\{f''(T_1', T_i') \mid 1 \leq i \leq k\}$
15: end if
16: if $T_G = P(T_1, T_2, \ldots, T_k)$ then
17: $T_R = P(T_1', T_2', \ldots, T_k')$
18: for $i = 1$ to $k$ do
19: $X \leftarrow \{T_j' \mid \text{Leaf}(T_j') \cap \text{Leaf}(T_i) \neq \phi\}$
20: if $|X| > 1$ then
21: $T_i' \leftarrow P(X)$
22: else
23: $T_i' \leftarrow \text{element}(X)$
24: end if
25: end for
26: $T_R \leftarrow P\{f''(T_1', T_i') \mid 1 \leq i \leq k, \text{Leaf}(T_R) \neq \phi\}$
27: end if
28: if $T_R = S(T_1', T_2', \ldots, T_l')$ or $Q()$ then
29: choose $i$ s.t. $\text{Leaf}(T_i') \cap \text{Leaf}(T_R) \neq \phi$
30: return $P\{f''(T_1', T_i') \mid 1 \leq i \leq k, \text{Leaf}(T_R) \neq \phi\}$
31: end if
32: end if
33: if $T_G = F(T_1)$ then
34: if $T_R = P(T_1', T_2', \ldots, T_l')$ then
35: return $F\{f''(T_1', T_j') \mid 1 \leq j \leq l\}$
36: end if
37: if $T_R = S(T_1', T_2', \ldots, T_l')$ or $Q()$ then
38: return $F\{f''(T_1', T_R')\}$
39: end if
40: end if
Proof: This is a direct consequence of Lemma 4.2 and the definition of the tree execution function \( f' \).

**Lemma 4.5:** The annotated SP-tree representation of a valid run is unique. That is, if two annotated SP-trees \( T_R \) and \( T'_R \) for the same valid run \( R \) with respect to an SP-specification \( (G,F) \) are produced, then \( T_R \equiv T'_R \).

Proof: Let \( T_R \) and \( T'_R \) be two annotated SP-tree representations of \( R \) constructed by Algorithm 2. Since the algorithm is deterministic, then equivalent but different input trees for the annotated SP-tree specification of \( (G,F) \) or the canonical SP-tree of \( R \) (or both) must have been given as input.

**Case 1:** Different annotated SP-trees for the specification are provided. Since the annotated SP-tree for \( (G,F) \) constructed by Algorithm 1 is unique (Lemma 4.3), the two input trees can differ only in the order of children of \( P \) nodes. The order of \( P \) nodes which is used only in Line 16, and the output trees \( T_R \) and \( T'_R \) will differ only in the order of children of \( P \) nodes.

**Case 2:** Different canonical SP-trees for \( R \) are provided. Since the canonical SP-tree for \( R \) is unique, these trees differ only in the order of children of \( P \) nodes, which is used in Lines 17 and 34. Line 17 affects the order of children of \( P \) nodes while Line 34 affects the order of children of \( F \) nodes in the output tree.

**Case 3:** Both the annotated SP-trees for the specification and the canonical SP-trees for \( R \) are different. The argument follows from Cases 1 and 2 above.

Thus, \( T_R \) and \( T'_R \) differ only in the order of children of \( P \) or \( F \) nodes, and hence \( T_R \equiv T'_R \).

**D. Edit Distance on Annotated SP-trees**

Based on the tree representation of SP-workflows, we now propose an edit distance problem on annotated SP-trees that is equivalent to our SP-workflow difference problem.

Recall that given an SP-tree \( T \), we let \( T[v] \) denote the subtree rooted at a node \( v \) in \( T \) and let \( p(v) \) denote the parent of \( v \). Also, we will use \( \text{Graph}(T) \) to represent the graph from which \( T \) is constructed, and use \( \text{Leaf}(T) \) to denote the leaves \( (Q \text{ nodes}) \) of \( T \). In addition, let \( s(v) \) and \( t(v) \) be the labels on two terminals of \( \text{Graph}(T[v]) \). In the following, we say a node is true if it has more than one child, otherwise call it a pseudo node. We first give the notion of a branch-free subtree and an elementary subtree:

**Definition 4.1:** Given an annotated SP-tree \( T \), \( T[v] \) is said to be a branch-free subtree in \( T \) iff \( T[v] \) does not contain any true \( P \) or true \( F \) node. Furthermore, we say \( T[v] \) is an elementary subtree in \( T \) iff (1) \( T[v] \) is a branch-free subtree in \( T \); and (2) \( p(v) \) is a true \( P \) or a true \( F \) node.

We consider two subtree edit operations over the annotated SP-trees: **Subtree Insertion** and **Subtree Deletion**. Following the notation for path edit operations, we denote a subtree insertion by \( \Lambda \rightarrow T[v] \) and a subtree deletion by \( T[v] \rightarrow \Lambda \), where \( T[v] \) is an elementary subtree to be edited. The following lemma shows the correspondence between an elementary subtree and an elementary path.

**Lemma 4.6:** Given the annotated SP-tree \( T \) for a valid run \( R \), if \( T[v] \) is an elementary subtree in \( T \), then \( p = \text{Graph}(T[v]) \) is an elementary path in \( R \). Conversely, if \( p \) is an elementary path in \( R \), then there exists an elementary subtree \( T[v] \) in \( T \) such that \( p = \text{Graph}(T[v]) \).

Proof: Given that \( T[v] \) is an elementary subtree, by the first condition in Definition 4.1, we know that \( \text{Graph}(T[v]) \) is a simple path such that each internal node has exactly one incoming edge and one outgoing edge. By the second condition, we know that there exists another SP-graph between \( s(\text{Graph}(T[v])) \) and \( t(\text{Graph}(T[v])) \). Hence, \( s(\text{Graph}(T[v])) \) has at least two outgoing edges and \( t(\text{Graph}(T[v])) \) has at least two incoming edges. These are exactly the two conditions required for an elementary path defined in Definition 3.4. Thus \( \text{Graph}(T[v]) \) is an elementary path in \( R \).

Conversely, let \( p \) be an elementary path in \( R \). Let \( T[v] \) be a subtree with the minimum number of leaves such that the leaves of \( T[v] \) contain all edges in \( p \). Among all such subtrees, we choose the one that maximizes the number of internal nodes (i.e. grab as many pseudo-nodes as possible in root).

We start by showing that \( T[v] \) is a branch-free subtree. First, \( v \) cannot be a true \( P \) or a true \( F \) node. Otherwise, there exists a child \( c(v) \) of \( v \) such that the leaves of \( T[c(v)] \) contain all edges in \( p \) but it has fewer leaves than \( T[v] \), a contradiction. Second, no descendant \( u \) of \( v \) can be a true \( P \) or a true \( F \) node. Otherwise, either \( s(\text{Graph}(T[u])) \) or \( t(\text{Graph}(T[u])) \) is an internal node on \( p \) and has at least two incoming or outgoing edges, contradicting Definition 3.4. It follows that \( \text{Graph}(T[v]) \) is a simple path such that each internal node has exactly one incoming edge and one outgoing edge. Since \( p \) is a subgraph of \( \text{Graph}(T[v]) \) and \( p \) is an elementary path, we have \( \text{Graph}(T[v]) = p \).

We now show that \( p(v) \) is a true \( P \) or \( F \) node. Otherwise, if \( p(v) \) is a pseudo \( P \) or \( F \) node then \( T[p(v)] \) has the same set of leaves as \( T[v] \) but more internal nodes, a contradiction. If \( p(v) \) is an \( S \) node then \( v \) must be a \( Q \) node or a pseudo \( P \) or \( F \) node. If this case, then either \( s(p) = s(\text{Graph}(T[v])) \) has only one outgoing edge or \( t(p) = t(\text{Graph}(T[v])) \) has only one incoming edge, contradicting Definition 3.4.

**Example 4.4:** Fig. 7 shows a subtree edit script from \( T_1 \) to \( T_2 \) (see Fig. 6) which corresponds to the path edit script between the underlying runs \( R_1 \) and \( R_2 \) (see Fig. 3).

Given a cost function \( \gamma \) over path edit operations, we extend \( \gamma \) to subtree edit operations by letting

\[
\gamma(\Lambda \rightarrow T[v]) = \gamma(\Lambda \rightarrow \text{Graph}(T[v]))
\]

By Lemma 4.6 and Eq. 1 in Section III-C.2, we have the following appealing interpretation of \( \gamma \) on trees: For any elementary subtree \( T[v] \), we have

\[
\gamma(\Lambda \rightarrow T[v]) = \gamma([\text{Leaf}(T[v])], s(v), t(v))
\]

where \( [\text{Leaf}(T[v])] \) is the number of leaves of \( T[v] \) and \( s(v), t(v) \) are the labels of two terminals of \( \text{Graph}(T[v]) \). Note that \( s(v) \) and \( t(v) \) are two invariants associated with each node \( v \) and will not be changed by any subtree edit operation.
Definition 4.2: Given a cost function $\gamma$, the edit distance between $T_1$ and $T_2$, denoted by $\delta(T_1, T_2)$, is defined as the minimum cost of a subtree edit script from $T_1$ to $T_2$. Formally, $\delta(T_1, T_2) = \min\{\gamma(\mathcal{E}) \mid T_1 \xrightarrow{\mathcal{E}} T_2\}$.

The following theorem shows that the two edit distance problems are equivalent.

Lemma 4.7: Let $R_1$ and $R_2$ be a pair of valid runs and let $T_1$ and $T_2$ be their annotated SP-trees respectively. Given a path edit script $\mathcal{E}$ that transforms $R_1$ to $R_2$, there exists a subtree edit script $\mathcal{E}'$ that transforms $T_1$ to $T_2$ such that $\gamma(\mathcal{E}') = \gamma(\mathcal{E})$. Conversely, for any subtree edit script $\mathcal{E}'$, there exists a path edit script $\mathcal{E}$ such that $\gamma(\mathcal{E}) = \gamma(\mathcal{E}')$.

Proof: We prove the first part by induction on $|\mathcal{E}|$. In the base case, $\mathcal{E}$ is a single path edit operation that transforms $R_1$ to $R_2$. Without loss of generality, let $\mathcal{E} = \Lambda \rightarrow p$ where $p$ is an elementary path in $R_2$. By Lemma 4.6, there exists an elementary subtree $T_2[v]$ in $T_2$ such that $p = \operatorname{Graph}(T_2[v])$. Thus, $\mathcal{E}' = \Lambda \rightarrow T_2[v]$ is a subtree edit operation (script) that transforms $T_1$ to $T_2$ and $\gamma(\mathcal{E}') = \gamma(\mathcal{E})$. We now consider the inductive step. Let $\omega$ be the last edit operation of $\mathcal{E}$ and $\mathcal{E} = \mathcal{E}_1 \cdot \omega$. Let $R_1'$ be the run that results after $\mathcal{E}_1$ is applied to $T_1$ and let $R_2'$ be the run represented by $T_1'$. By inductive hypothesis, there exists a path edit script $\mathcal{E}_1$ that transforms $R_1$ to $R_1'$ and $\gamma(\mathcal{E}_1) = \gamma(\mathcal{E}')$. On the other hand, as constructed in the base case, let $\omega$ be the corresponding path edit operation for $\omega'$. Since $\omega'$ transforms $T_1'$ to $T_2$, it follows from the base case that $\gamma(\omega) = \gamma(\omega')$. Thus, $\mathcal{E} = \mathcal{E}_1 \cdot \omega$ is a path edit script that transforms $R_1$ to $R_2$ and $\gamma(\mathcal{E}) = \gamma(\mathcal{E}')$. We now consider the inductive step. Let $\omega'$ be the last edit operation of $\mathcal{E}'$ and $\mathcal{E}' = \mathcal{E}_1' \cdot \omega'$. Let $T_1'$ be the tree that results after $\mathcal{E}_1'$ is applied to $T_1$ and let $R_1'$ be the run represented by $T_1'$. By inductive hypothesis, there exists a path edit script $\mathcal{E}_1'$ that transforms $R_1$ to $R_1'$ and $\gamma(\mathcal{E}_1') = \gamma(\mathcal{E}')$.

Theorem 2: Let $R_1$ and $R_2$ be a pair of valid runs and let $T_1$ and $T_2$ be their annotated SP-trees respectively. Then $\delta(R_1, R_2) = \delta(T_1, T_2)$.

Proof: This is a direct consequence of Lemma 4.7.

V. Algorithm

The algorithm to compute the edit distance as well as the corresponding minimum-cost edit script between two valid runs $R_1$ and $R_2$ with respect to an underlying SP-specification $(G, \mathcal{F})$ has two steps:

1) Generating the annotated SP-trees $T_G$, $T_1$ and $T_2$ for the specification $(G, \mathcal{F})$ and the pair of valid runs $R_1$ and $R_2$ respectively.

2) Computing the edit distance $\delta(T_1, T_2)$ as well as the corresponding minimum-cost edit script from $T_1$ to $T_2$.

The first subproblem was solved in the previous section. We now study the second subproblem. For clarity of exposition, we focus on computing the edit distance, since the corresponding minimum-cost edit script can be easily produced by bookkeeping. To compute the edit distance between $T_1$ and $T_2$, our algorithm needs to find a minimum-cost well-formed mapping from $T_1$ to $T_2$, which is first defined in Section V-A. We then describe a preprocessing step in Section V-B, before presenting the full algorithm in Section V-C. Finally, Section V-D analyzes the time complexity of our algorithm.

A. Well-Formed Mapping

We now formalize the notion of mapping implied by an edit script, and show the correspondence between them. Intuitively, an edit script transforming a tree $T_1$ to another tree $T_2$ keeps some of the nodes in $T_1$ unchanged and inserts and deletes other nodes to create a tree $T_2'$ that is isomorphic to $T_2$. The bijection between $T_2'$ and $T_2$ gives rise to a partial one-to-one mapping between the nodes of $T_1$ and $T_2$.

Example 5.1: The dashed lines between $T_1$ and $T_2$ in Fig. 6 show a mapping that corresponds to the subtree edit script depicted in Fig. 7.

For any node $v$ in $T_i (i = 1, 2)$, let $h(v)$ be the node in $T_G$ such that $T_i[v]$ is derived from $T_G[h(v)]$. Formally, $T_i[v] = f'(T_G[h(v)])$ where $f'$ is the tree execution function defined in Section IV-C. In addition, for any pair of nodes $(v_1, v_2)$ in $T_1$ and $T_2$, we say $v_1$ and $v_2$ are homologous if $h(v_1) = h(v_2)$. That is, $T_1[v_1]$ and $T_2[v_2]$ represent two valid runs with respect to the same part of a specification.

Definition 5.1: A set $M$ of pairs of nodes is said to be a well-formed mapping from $T_1$ to $T_2$ if
1) **one-to-one:** $M$ is a one-to-one mapping from $T_1$ to $T_2$. Formally, for any pair of $(v_1, v_2) \in M$ and $(v'_1, v'_2) \in M$, $v_1 = v'_1$ iff $v_2 = v'_2$.

2) **root mapped:** The roots of $T_1$ and $T_2$ are mapped by $M$. Formally, $(r_1, r_2) \in M$, where $r_1$ and $r_2$ are the roots of $T_1$ and $T_2$ respectively.

3) **specification preserved:** If a pair of nodes is mapped by $M$, then they are homologous. Formally, for any $(v_1, v_2) \in M$, $h(v_1) = h(v_2)$.

4) **parent preserved:** If a pair of nodes is mapped by $M$, then their parents are also mapped. Formally, for any $(v_1, v_2) \in M$, $(p(v_1), p(v_2)) \in M$; (recall that $p(v)$ denotes the parent of a node $v$).

5) **children of an S node preserved:** If a pair of $S$ nodes is mapped by $M$, then each pair of their children is also mapped. Formally, for any $(v_1, v_2) \in M$ such that Type$(v_1) = Type(v_2) = S$, $(c_i(v_1), c_i(v_2)) \in M$ for all $i$, where $c_i(v)$ denotes the $i$th child of a node $v$.

**Definition 5.2:** Let $M$ be a well-formed mapping from $T_1$ to $T_2$. A pair of nodes $(v_1, v_2)$ mapped by $M$ is said to be **unstably matched** iff (1) $(v_1, v_2)$ is a pair of $P$ nodes; (2) both $v_1$ and $v_2$ have only one child; and (3) their children are homologous and not mapped by $M$. A pair in $M$ that is not unstably matched is called **stably matched**.

Given a well-formed mapping $M$ from $T_1$ to $T_2$, let $I_1$ and $I_2$ be the sets of nodes mapped by $M$ in $T_1$ and $T_2$ respectively. We define the cost of a pair of nodes $(v_1, v_2) \in M$ as follows.

**Case 1** If $(v_1, v_2)$ is stably matched then we sum up the minimal cost of deleting or inserting all unmapped children of $v_1$ and $v_2$. That is,

$$\gamma(v_1, v_2) = \sum_{p(c_1) = v_1, c_1 \notin I_1} X_{T_1}(c_1) + \sum_{p(c_2) = v_2, c_2 \notin I_2} X_{T_2}(c_2)$$

where $X_T(c)$ is the minimum cost of deleting the subtree $T[c]$.

**Case 2** If $(v_1, v_2)$ is unstably matched then by Definition 5.2 both of them must have only one child and they are homologous, say $c_1$ and $c_2$. Thus,

$$\gamma(v_1, v_2) = X_{T_1}(c_1) + X_{T_2}(c_2) + 2 \cdot W_{T_G}(h(v_1), h(c_1))$$

where $W_{T_G}(h(v_1), h(c_1))$ is the minimum cost of inserting or deleting an elementary subtree rooted at a child of $h(v_1)$ that is distinct from the subtree rooted at $h(c_1)$ in $T_G$. We then define the cost of a mapping $M$ to be the sum of the cost of all pairs of nodes mapped by $M$. That is,

$$\gamma(M) = \sum_{(v_1, v_2) \in M} \gamma(v_1, v_2)$$

A formal treatment is given as follows. Let $M$ be a well-formed mapping from $T_1$ to $T_2$ and $E$ be an edit script from $T_1$ to $T_2$, we say that $E$ conforms to $M$ if for any node $v_1$ in $T_1$, the node $v_1$ is mapped by $M$ iff it remains in the tree during the transformation by $E$.

**Lemma 5.1:** $\gamma(M) = \min \{\gamma(E) \mid E$ is an edit script from $T_1$ to $T_2$ conforming to $M\}$

**Proof:** We first show that the cost of any edit script from $T_1$ to $T_2$ conforming to $M$ is no less than $\gamma(M)$. Let $E$ be an edit script from $T_1$ to $T_2$ conforming to $M$. Consider a pair of nodes $(v_1, v_2)$ mapped by $M$. Since $E$ conforms to $M$, by the definition we know that for any child $c_1$ of $v_1$ that is not mapped by $M$, $T_1[c_1]$ must be deleted during the transformation by $E$, which gives rise to a cost of at least $X_{T_1}(c_1)$. Similarly, for any child $c_2$ of $v_2$ that is not mapped by $M$, it also gives rise to a cost of at least $X_{T_2}(c_2)$. Moreover, for any unstably matched pair of nodes $(v_1, v_2)$, we have to insert some redundant subtree as the child of $v_1$ before we can delete the only child $c_1$ of $v_1$, and finally remove this redundant subtree. By the definition of $W_{T_G}(h(v_1), h(c_1))$, we know that such operations give rise to an extra cost of at least $2 \cdot W_{T_G}(h(v_1), h(c_1))$. By the definition of $\gamma(M)$, we have that $\gamma(E) \geq \gamma(M)$.

On the other hand, we will show that there exists an edit script $E$ from $T_1$ to $T_2$ conforming to $M$ and $\gamma(E) = \gamma(M)$. We prove this claim by constructing a desired edit script. Consider a pair of nodes $(v_1, v_2)$ mapped by $M$ such that there exists at least one child of $v_1$ or $v_2$ that is not mapped by $M$. Since by Definition 5.1 all children of an $S$ node must be preserved, $(v_1, v_2)$ must be a pair of $P$ or $F$ nodes.

If both $v_1$ and $v_2$ are $F$ nodes, then $(v_1, v_2)$ must be stably matched by $M$. We construct an edit script $E(v_1, v_2)$ by first inserting all children of $v_2$ that are not mapped by $M$ and then deleting all children of $v_1$ that are not mapped. Note that during this transformation $v_1$ always has at least two children and is a true $F$ node. Therefore, by Lemma 5.6, any unmapped children of $v_1$ and $v_2$ can be deleted or inserted by
a sequence of elementary subtree edit operations. Furthermore, we always insert or delete it by a minimum cost sequence of edit operations, and thus \( \gamma(E(v_1, v_2)) = \gamma(v_1, v_2) \).

If both \( v_1 \) and \( v_2 \) are \( P \) nodes, we consider the following three subcases:

1) If there exists a pair of children of \( v_1 \) and \( v_2 \) that are mapped by \( M \), then \((v_1, v_2)\) must be stably matched by \( M \). We construct an edit script \( E(v_1, v_2) \) by first deleting all children of \( v_1 \) that are not mapped by \( M \) and then inserting all children of \( v_2 \) that are not mapped. Note that \( v_1 \) is always a true \( P \) node due to the presence of a mapped child. Moreover, \( v_1 \) does not have two homologous children at any intermediate state, and therefore, each intermediate tree is valid. Again, \( \gamma(E(v_1, v_2)) = \gamma(v_1, v_2) \).

2) If no pair of children \( v_1 \) and \( v_2 \) is mapped by \( M \), and \((v_1, v_2)\) is stably matched by \( M \). By Definition 5.2, we claim that there exists a pair of non-homologous children \((c_1, c_2)\) of \( v_1 \) and \( v_2 \). Then we construct an edit script \( E(v_1, v_2) \) as follows: delete the child of \( v_1 \) which is homologous with \( c_2 \), if it exists; insert \( c_2 \); delete all remaining children of \( v_1 \); and insert all other children of \( v_2 \). The fact that \( v_1 \) is always a true \( P \) node follows from the presence of \( c_1 \) in the first two steps and follows from the presence of \( c_2 \) in the remaining steps. We can also check that \( v_1 \) does not have two homologous children at any intermediate step. Moreover, \( \gamma(E(v_1, v_2)) = \gamma(v_1, v_2) \).

3) If no pair of children \( v_1 \) and \( v_2 \) is mapped by \( M \), and \((v_1, v_2)\) is unstably matched by \( M \). Let \( c_1 \) and \( c_2 \) be the only children of \( v_1 \) and \( v_2 \) such that \( c_1 \) and \( c_2 \) are homologous and are not mapped by \( M \). Then we construct an edit script \( \gamma(E(v_1, v_2)) \) as follows: insert a minimum cost elementary subtree \( T' \) rooted at a child \( c'_1 \) of \( v_1 \) such that \( c'_1 \) is not homologous with \( c_1 \); delete \( T_1[c_1] \); insert \( T_2[c_2] \); and delete the elementary subtree \( T' \) inserted in the first step. We can check that \( v_1 \) is always a true \( P \) node during the transformation and all intermediate trees are valid. Moreover, by the definition of \( \gamma(v_1, v_2) \) for an unstably matched pair of nodes and that of \( W_{T'}(h(v_1), h(c_1)) \), we have \( \gamma(E(v_1, v_2)) = \gamma(v_1, v_2) \).

Note that the unmapped children of all pairs of nodes \((v_1, v_2)\) considered above are disjoint. On the other hand, they cover all the nodes that are not mapped by \( M \) but their parent is mapped in both trees. Finally, we construct an edit script by letting \( E = \{E(v_1, v_2)\} \cup \{v_1, v_2\} \in M \}. \) By the construction shown above, \( E \) is an edit script from \( T_1 \) to \( T_2 \) conforming to \( M \) and \( \gamma(E) = \gamma(M) \), as desired.

Hence, \( \gamma(M) = \min \{ \gamma(E) \mid E \) is an edit script from \( T_1 \) to \( T_2 \) conforming to \( M \} \).

**Lemma 5.2**: Given a well-formed mapping \( M \) from \( T_1 \) to \( T_2 \), there exists an edit script \( E \) from \( T_1 \) to \( T_2 \) such that \( \gamma(E) \leq \gamma(M) \).

**Proof**: This is a direct consequence of Lemma 5.1.

We next show that well-formed mappings can be composed in a natural manner. If \( M_1 \) is a well-formed mapping from \( T_1 \) to \( T_2 \), and \( M_2 \) is a well-formed mapping from \( T_2 \) to \( T_3 \) where \( T_1, T_2 \) and \( T_3 \) are valid runs for the same specification tree \( T_G \), we define \( M_1 \circ M_2 = \{(i, j) \mid \exists k \text{ s.t. } (i, k) \in M_1 \text{ and } (k, j) \in M_2 \} \).

**Lemma 5.3**: If \( M_1 \) is a well-formed mapping from \( T_1 \) to \( T_2 \), and \( M_2 \) is a well-formed mapping from \( T_2 \) to \( T_3 \), then

1) \( M_1 \circ M_2 \) is a well-formed mapping from \( T_1 \) to \( T_3 \).
2) \( \gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2) \).

**Proof**: The first property can be verified by Definition 5.1. We now prove the second property. Let \( E_1 \) be the minimum cost edit script from \( T_1 \) to \( T_2 \) conforming to \( M_1 \) and \( E_2 \) be the minimum cost edit script from \( T_2 \) to \( T_3 \) conforming to \( M_2 \). By Lemma 5.1, we have \( \gamma(E_1) = \gamma(M_1) \) and \( \gamma(E_2) = \gamma(M_2) \). Now consider the edit script \( E = E_1 \cup E_2 \). By the definition of \( M_1 \circ M_2 \), we know that \( E \) is an edit script from \( T_1 \) to \( T_3 \) conforming to \( M_1 \circ M_2 \). By Lemma 5.1, we have \( \gamma(M_1 \circ M_2) \leq \gamma(E) = \gamma(E_1) + \gamma(E_2) = \gamma(M_1) + \gamma(M_2) \).

The lemma follows.

**Lemma 5.4**: Given an edit script \( E \) that transforms \( T_1 \) to \( T_2 \), there exists a well-formed mapping \( M \) from \( T_1 \) to \( T_2 \) such that \( \gamma(M) \leq \gamma(E) \).

**Proof**: This lemma can be proved by induction on \( |E| \).

In the base case, \( E \) is a single edit operation. We consider the case when \( E = T_1[v_1] \rightarrow \Lambda \) is an elementary deletion, an elementary insertion can be handled similarly. Since \( E \) transforms \( T_1 \) to \( T_2 \), it gives rise to a partial one-to-one mapping \( M \) between their nodes. In this mapping, only nodes in \( T_1[v_1] \) are not mapped. Since \( T_1[v_1] \) is an elementary subtree in \( T_1 \), by Definition 4.1, we know that \( p(v_1) \) is a true \( P \) or a true \( F \) node. Thus we can verify that all five conditions in Definition 5.1 are satisfied by this mapping \( M \). Moreover, \( \gamma(M) = X_{T_1}(v_1) = \gamma(E) \). We now consider the inductive case. Let \( \omega \) be the last edit operation of \( E \) and \( E = E' \cup \omega \). Suppose \( E' \) is an edit script from \( T_1 \) to \( T'_1 \), then \( \omega \) is a single edit operation from \( T'_1 \) to \( T_2 \). By the inductive hypothesis, there exists a mapping \( M_1 \) from \( T_1 \) to \( T'_1 \) such that \( \gamma(M_1) \leq \gamma(E') \). Let \( M_2 \) be the mapping for \( \omega \) as constructed in the base case. Then \( \gamma(M_2) = \gamma(\omega) \). By Lemma 5.3, we have that \( M_1 \circ M_2 \) is a mapping from \( T_1 \) to \( T_2 \) such that \( \gamma(M_1 \circ M_2) \leq \gamma(M_1) + \gamma(M_2) \leq \gamma(E') + \gamma(\omega) = \gamma(E) \).

**Theorem 3**: Let \( T_1 \) and \( T_2 \) be the annotated SP-trees for a pair of valid runs. Then \( \delta(T_1, T_2) = \min \{ \gamma(M) \mid M \) is a well-formed mapping from \( T_1 \) to \( T_2 \} \).

**Proof**: This is a direct consequence of Lemma 5.2 and Lemma 5.4.

**B. Subtree Deletion**

As a preprocessing step, we present an algorithm for computing the minimum cost of deleting a subtree. In the following, let \( T \) be an annotated SP-tree for a run.

Firstly, note that the subtree to be deleted here is not necessarily elementary, and therefore such a deletion may involve a sequence of elementary subtree edit operations.

**Lemma 5.5**: For any node \( v \) in \( T \), \( T[v] \) can be reduced to a branch-free subtree by a sequence of elementary subtree deletions.
Proof: Let $k$ be the number of true $P$ and true $F$ nodes in $T[v]$. We prove this lemma by induction on $k$. In the base case where $k = 0$, $T[v]$ is already a branch-free subtree. We now consider the inductive case. Let $u$ be the deepest true $P$ or true $F$ node in $T[v]$. Thus for no child $c$ of $u$ does $T[c]$ contain a true $P$ or true $F$ node, and $p(c) = u$ is a true $P$ or true $F$ node. By Definition 4.1, $T[c]$ is an elementary subtree. Hence, we can repeatedly delete a subtree $T[c]$ until only one child of $u$ is left. At that time $u$ becomes a pseudo $P$ or a pseudo $F$ node. By the inductive hypothesis, the lemma follows.

Lemma 5.6: For any node $v$ in $T$, $T[v]$ can be deleted by a sequence of elementary subtree deletions iff $p(v)$ is a true $P$ or an true $F$ node.

Proof: By Lemma 5.5, there exists a sequence of elementary subtree deletions that reduces $T[v]$ to a branch-free subtree, say $T'[v]$. By Definition 4.1, $T'[v]$ is an elementary subtree iff $p(v)$ is a true $P$ or true $F$ node.

Furthermore, the quadrangle inequality of our cost model guarantees that the minimum-cost subtree deletion is always achieved by a sequence of elementary subtree deletions.

Lemma 5.7: For any node $v$ in $T$ such that $p(v)$ is a true $P$ or true $F$ node, there exists a sequence of elementary subtree deletions $E$ such that $T[v]$ is deleted by $E$ and for any edit script $E'$ that deletes $T[v]$, we have $\gamma(E) \leq \gamma(E')$.

Proof: We prove this lemma by contradiction. Assume that any minimum cost edit script that deletes $T[v]$ contains at least one elementary subtree insertion. Let $E_m = \omega_1, \omega_2, \ldots, \omega_k$ be a minimum cost edit script with the minimum number of insertions. Suppose $\omega_i$ is the last insertion in $E_m$, and it inserts an elementary subtree $T[v_1]$ as the child of a $P$ or $F$ node $v_2$. After $\omega_i$ is applied, $v_2$ must be a true $P$ or true $F$ node. To delete the whole subtree $T[v]$, we must convert $v_3$ into a pseudo node by the remaining sequence of deletions.

We first claim that when $v_2$ is converted into a pseudo node, its only child must be $v_1$. Otherwise, there exists an elementary subtree deletion $\omega_j$ that deletes $T[v_1]$, which is exactly the complement of $\omega_i$. Let $E'_m = E_m - \{\omega_i, \omega_j\}$. Thus $E'_m$ is also a minimum cost edit script that deletes $T[v]$ and has less insertions than $E_m$ has, contradicting our choice of $E_m$. Let $\omega_j$ be the last elementary subtree deletion that deletes a child $T[v'_1]$ of $v_2$, such that after $\omega_j$ is applied, $v_2$ becomes a pseudo node with only child $v_1$. Finally, let $\omega_i = T[v_3] \rightarrow \Lambda$ be the elementary subtree deletion that removes $T[v_1]$ along with its ancestors, where $v_3$ is an ancestor of $v_1$. Then we construct an edit script $E_m$ by removing $\omega_i$ and $\omega_j$ from $E_m$ and replacing $w_i$ by an elementary subtree deletion $w'_i$ that deletes a subtree rooted at the same node $v_3$. Observe that all the edit operations in $E_m$ are the same as those in $E'_m$, except that $w_i$ in $E_m$ deletes an elementary subtree $T[v'_1]$ which contains $T[v_1]$ as a subtree, but $w'_i$ in $E'_m$ deletes the subtree $T[v'_3]$ which contains $T[v'_1]$ as a subtree.

We now consider the cost of $E'_m$. Let $T[v_3]$ represent an elementary path $p_2$ from $B$ to $C$ with length $l_2$. Since $T[v'_1]$ and $T[v_1]$ are both children of $v_2$ which is either a $P$ or an $F$ node, $T[v'_1]$ has the same terminals as $T[v_1]$. Thus let $T[v'_1]$ represent an elementary path $p_2$ from $B$ to $C$ with length $l_2$. In $E_m$, $T[v_3]$ is an elementary subtree that contains $T[v_1]$. Therefore $T[v_3]$ represents an elementary subtree $p_2$. Suppose $p$ has two terminals $A$ and $D$ and is concatenated by a path from $A$ to $B$ with length of $l_1$, a path from $B$ to $C$ with length of $l_2$ and a path from $C$ to $D$ with length of $l_3$. On the other hand, by a similar analysis, we know that $T[v_3]$ in $E_m$ represents an elementary path $p'$ that is concatenated by a path from $A$ to $B$ with length of $l_1$, a path from $B$ to $C$ with length $l_2'$ and a path from $C$ to $D$ with length $l_3$. Thus, by the definition of $\gamma$,

$$\gamma(E'_m) - \gamma(E_m) = \gamma(\omega_i') - \gamma(\omega_i) - \gamma(\omega_j)$$

$$= \gamma(l_1 + l_2 + l_3, A, D) - \gamma(l_2, B, C) - \gamma(l_2', B, C) - \gamma(l_1 + l_2 + l_3, A, D)$$

It follows from the quadrangle inequality on $\gamma$ that $\gamma(E'_m) \leq \gamma(E_m)$. Hence, $E'_m$ is also a minimum cost edit script that deletes $T[v]$. However, $E'_m$ has less insertions than $E_m$ has. Contradiction. The lemma follows.

We now describe the algorithm to compute the minimum cost of deleting a subtree. Given an annotated SP-tree $T$, let $X_T(v)$ denote the minimum cost of deleting the subtree $T[v]$. We now do a bottom-up computation on $T$ and compute $X_T(v)$ for each node $v$ in $T$ as follows: (1) If $v$ is a $Q$ node, then $X_T(v)$ is given by the cost model; (2) If $v$ is a $P$ or an $F$ node, then $X_T(v)$ is equal to the sum of $X_T(c)$'s for all children $c$ of $v$; and (3) If $v$ is an $S$ node, then $X_T(v)$ is derived by a dynamic programming which computes the minimum cost of reducing each of the first $i$ children of $v$ to a branch-free subtree with a total of $l$ leaves.

Let $Y_T(v)[l]$ be the minimum cost of a sequence of elementary subtree deletions that reduces $T[v]$ to a branch-free subtree with exactly $l$ leaves. For each $S$ node $v$, let $Z_T(v)[i,l]$ be the minimum cost of a sequence of elementary subtree deletions that reduces each of the first $i$ children of $v$ to a branch-free subtree with a total of $l$ leaves. Finally, let $l(v)$ be the maximum number of leaves in a branch-free subtree that can be derived from $T[v]$ using a sequence of elementary subtree deletions. A pseudo code presentation of this algorithm is given in Algorithm 3, and its correctness is proved by Theorem 4.

Theorem 4: Algorithm 3 is correct.

Proof: We will show that $Y_T(v)[l]$ computed by Algorithm 3 is the minimum cost of a sequence of elementary subtree deletions that reduces $T[v]$ into a branch-free subtree with $l$ leaves, and $X_T(v)$ is the minimum cost of a sequence of elementary subtree deletions that deletes $T[v]$. This is proved by induction on the depth of $v$. In the base case where $v$ is a $Q$ node (leaf), $T[v]$ is a branch-free subtree with only one leaf. Thus $Y_T(v)[1] = 0$, and $X_T(v)$ is the cost of deleting $T[v]$ given by the cost model. Now consider the inductive case. Let $E$ be the minimum cost sequence of elementary subtree deletions that reduces $T[v]$ to a branch-free subtree with $l$ leaves. We consider the following two cases:

1) If $v$ is a $P$ or an $F$ node, then $E$ must convert $v$ into a pseudo $P$ or a pseudo $F$ node by keeping only one child of
Algorithm 3 Subtree-Deletion

Input: $X_T(c_i(v)), Y_T(c_i(v))[l], l(c_i(v))$
Output: $X_T(v), Y_T(v)[l], l(v)$

1: if Type($v$) = $Q$ then
2: \hspace{1em} $l(v) \leftarrow 1$
3: \hspace{1em} $Y_T(v)[l] \leftarrow 0$
4: \hspace{1em} $X_T(v) \leftarrow \gamma(1, s(v), t(v))$
5: end if
6: if Type($v$) = $P/F$ then
7: \hspace{1em} $l(v) \leftarrow \max_{i=1}^{d(v)} \{l(c_i(v))\}$
8: \hspace{1em} for $l = 1$ to $l(v)$ do
9: \hspace{2em} $Y_T(v)[l] \leftarrow \min_{i=1}^{d(v)} \{Y_T(c_i(v))[l] + \sum_{j \neq i} X_T(c_j(v))\}$
10: \hspace{1em} end for
11: \hspace{1em} $X_T(v) \leftarrow \min_{i=1}^{d(v)} \{Y_T(v)[l] + \gamma(l, s(v), t(v))\}$
12: end if
13: if Type($v$) = $S$ then
14: \hspace{1em} $l(v) \leftarrow \sum_{i=1}^{d(v)} l(c_i(v))$
15: \hspace{1em} $Z_T(v)[0, 0] \leftarrow 0$; $Z_T(v)[0, l] \leftarrow +\infty$ for all $l > 0$
16: \hspace{1em} for $i = 1$ to $d(v)$ do
17: \hspace{2em} for $l = 1$ to $\sum_{j=1}^{d(v)} l(c_j(v))$ do
18: \hspace{3em} $Z_T(v)[i, l] \leftarrow \min_{i=1}^{\sum_{j=1}^{d(v)} l(c_j(v))} \{Z_T(v)[i-1, k] + Y_T(c_i(v))[l-k]\}$
19: \hspace{2em} end for
20: \hspace{1em} end for
21: \hspace{1em} $Y_T(v)[l] \leftarrow Z_T(v)[d(v), l]$
22: end for
23: \hspace{1em} $X_T(v) \leftarrow \min_{i=1}^{d(v)} \{Y_T(v)[l] + \gamma(l, s(v), t(v))\}$
24: end if
25: end if
26: return $X_T(v), Y_T(v)[l], l(v)$

$v$ (say $c_k(v)$) and deleting all other children, and then reduce this child to a branch-free subtree with $l$ leaves. Thus, by the induction hypothesis,

$$\gamma(\mathcal{E}) \geq Y_T(c_k(v))[l] + \sum_{j \neq k} X_T(c_j(v)) \geq Y_T(v)[l]$$

where $Y_T(v)[l]$ is computed by Algorithm 3 for a $P$ or $F$ node. On the other hand, for any child $c_i(v)$ of $v$, $Y_T(c_i(v))[l] + \sum_{j \neq i} X_T(c_j(v))$ is equal to the cost of a sequence of elementary subtree deletions that reduces the $i$th child of $v$ into a branch-free subtree with $l$ leaves and deletes all other children. That is, it reduces $T[v]$ into a branch-free subtree with $l$ leaves. Thus, by the minimality of $\mathcal{E}$, we know that for all $1 \leq i \leq d(v),$

$$\gamma(\mathcal{E}) \leq Y_T(c_i(v))[l] + \sum_{j \neq i} X_T(c_j(v))$$

It follows that $\gamma(\mathcal{E}) \leq Y_T(v)[l]$. Hence, $Y_T(v)[l] = \gamma(\mathcal{E})$ is the minimum cost.

2) If $v$ is an $S$ node, then by Lemma 5.6 none of the children of $v$ can be deleted. Therefore, for any sequence of elementary subtree deletions that reduces $T[v]$ to a branch-free subtree with $l$ leaves, it must reduce each child of $v$ to a branch-free subtree, and all these subtrees collectively have $l$ leaves. We solve this problem by a dynamic programming as shown in Algorithm 3. We now prove that $Z_T(v)[i, l]$ computed by this dynamic programming is the minimum cost of a sequence of elementary subtree deletions that reduces each of the first $i$ children of $v$ to a branch-free subtree with a total of $l$ leaves. It is proved by an induction on $i$. In the base case where $i = 0$, $Z_T[0, 0] = 0$ and $Z_T[0, l] = +\infty$ for all $l > 0$. Now suppose $Z_T(v)[i-1, l]$ is the optimal solution for all $l$. Consider the optimal solution $\mathcal{E}'$ for the first $i$ children with a total of $l$ leaves. Observe that $\mathcal{E}'$ must reduce each of the first $i-1$ children into a branch-free subtree with a total of $x$ leaves for some $x \in [0, \ldots, l-1]$ (if $i = 1$ then $0$ is possible), and then reduce the $i$th child to a branch-free tree with exactly $l-x$ leaves. Thus, by the induction hypothesis,

$$\gamma(\mathcal{E}') \geq Z_T(v)[i-1, x] + Y_T(c_i(v))[l-x] \geq \min_{k=0}^{l-1} \{Z_T(v)[i-1, k] + Y_T(c_i(v))[l-k]\}$$

On the other hand, for any $0 \leq k \leq l-1$, $Z_T(v)[i-1, k] + Y_T(c_i(v))[l-k]$ is the cost of a solution that reduces each of the first $i$ children of $v$ to a branch-free subtree with a total of $l$ leaves. Thus, by the optimality of $\mathcal{E}'$, $\gamma(\mathcal{E}') \leq \min_{k=0}^{l-1} \{Z_T(v)[i-1, k] + Y_T(c_i(v))[l-k]\}$

Hence, $Z_T(v)[i, l] = \gamma(\mathcal{E}')$ is the optimal solution for first $i$ children with a total of $l$ leaves. By the definition of $Z_T(v)[i, l]$, we know that $Y_T(v)[l] = Z_T(v)[d(v), l] = \gamma(\mathcal{E})$ is the minimum cost.

Finally, for any sequence of elementary subtree deletions that deletes $T[v]$, it must first reduce $T[v]$ to an elementary subtree with $l$ leaves, and then delete this elementary subtree. Furthermore, the cost of deleting an elementary subtree only depends on the labels of two terminals (which are two invariants associated with each node) and the number of leaves, as defined in Section III-C.2. Hence, $X_T(v) = \min_{i=1}^{l(v)} \{Y_T(v)[l] + \gamma(l, s(v), t(v))\}$ is the minimum cost of a sequence of elementary subtree deletions that deletes $T[v]$. By Lemma 5.7, $X_T(v)$ is also the minimum cost of any edit script that deletes $T[v]$.

C. Edit Distance on SP-Trees

We are now ready to give the algorithm which computes the edit distance $\delta(T_1, T_2)$. By Theorem 3, we only need to compute the minimum-cost well-formed mapping from $T_1$ to $T_2$. Let $M(v_1, v_2)$ be the minimum-cost well-formed mapping from $T_1[v_1]$ to $T_2[v_2]$ such that $v_1$ and $v_2$ are homologous (i.e., $h(v_1) = h(v_2)$ where $h(v)$ denotes the node in $T_G$ from which a node $v$ in $T_R$ is derived). Also, let $d(v)$ be the degree of a node $v$ (i.e., the number of children of $v$) and $c_i(v)$ be the $i$th child of $v$.

The complete algorithm is described as below. We do a bottom-up computation on $T_1$ and $T_2$ and compute $M(v_1, v_2)$ for each pair of homologous nodes $(v_1, v_2)$. Note that by
Definition 5.1 (root is mapped) \( v_1 \) and \( v_2 \) are always mapped by \( M(v_1, v_2) \). Based on their types, we then have four cases:

Case 1 If \((v_1, v_2)\) is a pair of \( Q \) nodes, then \( M(v_1, v_2) \) consists only of a pair of nodes \((v_1, v_2)\) (Lines 1–3).

Case 2 If \((v_1, v_2)\) is a pair of \( S \) nodes, then they agree in the number of children. Moreover, each pair of their children are homologous. By Definition 5.1 (children of an \( S \) node are preserved) they must be mapped by \( M(v_1, v_2) \) (Lines 4–6).

Case 3 If \((v_1, v_2)\) is a pair of \( P \) nodes, we consider the following two subcases: Case 3a If both \( v_1 \) and \( v_2 \) have only one child and they are homologous, say \( c_1 \) and \( c_2 \), then the pair \((c_1, c_2)\) will be included in \( M(v_1, v_2) \) only if the minimum cost of a mapping between them is no greater than the minimum cost of deleting and inserting the corresponding subtrees plus the cost of inserting and deleting a minimum-cost elementary subtree rooted at a child of \( v_1 \) which is not homologous with \( c_1 \), denoted by \( W_{T_1}(h(v_1), h(c_1)) \) (Lines 8–13). Case 3b Otherwise, note that for each child of \( v_1 \), there exists at most one child of \( v_2 \) that is homologous with it, and vice versa. A pair of homologous nodes will be included in \( M(v_1, v_2) \) only if the minimum cost of the mapping between them is no greater than the minimum cost of deleting and inserting the corresponding subtrees (Lines 14–16).

Case 4 If \((v_1, v_2)\) is a pair of \( F \) nodes, then all children of \( v_1 \) and \( v_2 \) are homologous. We need to find the minimum-cost matching between them by setting up the following bipartite graph between the children of \( v_1 \) and \( v_2 \). Each pair of children of \( v_1 \) and \( v_2 \) are connected by an edge associated with the minimum cost of a mapping between them; each child of \( v_1 \) has an edge to a special node “−” associated with the minimum cost of deleting the corresponding subtree; and each child of \( v_2 \) has an edge to a special node “+” associated with the minimum cost of inserting the corresponding subtree. Now let \( M^* \) be the minimum-cost bipartite matching in this graph. Then a pair of children of \( v_1 \) and \( v_2 \) will be included in \( M(v_1, v_2) \) only if they are mapped by \( M^* \) (Lines 18–21).

A pseudo code presentation of this algorithm is given in Algorithm 4, and its correctness is proved by Theorem 5. Note that \( \delta(T_1, T_2) = \gamma(M(v_1, v_2)) \), where \( v_1 \) and \( v_2 \) are the roots of \( T_1 \) and \( T_2 \) respectively, and the corresponding minimum-cost edit script can be easily derived from \( M(v_1, v_2) \).

Theorem 5: Algorithm 4 is correct.

Proof: We will show that \( M(v_1, v_2) \) computed by Algorithm 4 is the minimum cost mapping from \( T_1[v_1] \) to \( T_2[v_2] \). It can be proved by induction on the depth of \( v_1 \) and \( v_2 \). The base case where both \( v_1 \) and \( v_2 \) are \( Q \) nodes (leaves) is straightforward. Let us consider the inductive case. Let \( M \) be any mapping from \( T_1[v_1] \) to \( T_2[v_2] \). By Definition 5.1 (root mapped), \( v_1 \) and \( v_2 \) must be mapped by \( M \). Moreover, by Definition 5.1 (parent preserved), we claim that any child \( c_1 \) of \( v_1 \) is either mapped to some child (say \( c_2 \)) of \( v_2 \) or not mapped by \( M \). Further, in the former case, any descendant of \( c_1 \) can only be mapped to some descendant of \( c_2 \). In the latter case, any descendant of \( c_1 \) is not mapped by \( M \). Therefore, \( M \) consists of the pair \((v_1, v_2)\) and the mappings between all pairs of children of \( v_1 \) and \( v_2 \) that are mapped by \( M \).

The inductive hypothesis and the construction of \( M(v_1, v_2) \) in Algorithm 4, we have that \( \gamma(M(v_1, v_2)) \leq \gamma(M) \).

Algorithm 4 Edit-Distance-on-Trees

Input: \( T_1[v_1], T_2[v_2] \)

Output: \( M(v_1, v_2) \)

1: if Type\( (v_1) = \text{Type}(v_2) = Q \) then
2: \( M(v_1, v_2) \leftarrow \{(v_1, v_2)\} \)
3: end if
4: if Type\( (v_1) = \text{Type}(v_2) = S \) then
5: \( M(v_1, v_2) \leftarrow \cup_{\text{Type}(c_1) = 1, \text{Type}(c_2) = 2 \cup \{(v_1, v_2)\} \}
6: end if
7: if Type\( (v_1) = \text{Type}(v_2) = P \) then
8: \( d(v_1) = d(v_2) + 1 \) and \( h(c_1) = h(c_2) \) then
9: \( \gamma(M(c_1, c_2)) \leq X_{T_1}(c_1) + X_{T_2}(c_2) + 2 \times W_{T_1}(h(v_1), h(c_1)) \) then
10: \( M(v_1, v_2) \leftarrow M(c_1, c_2) \cup \{(v_1, v_2)\} \)
11: else
12: \( M(v_1, v_2) \leftarrow \{(v_1, v_2)\} \)
13: end if
14: else
15: \( M(v_1, v_2) \leftarrow \cup\{M(c_1(v_1), c_j(v_2)) \mid h(c_1(v_1)) = h(c_j(v_2)) \} \) \( \gamma(M(c_1(v_1), c_j(v_2))) \) \( \leq X_{T_1}(c_1(v_1)) + X_{T_2}(c_j(v_2)) \} \cup \{(v_1, v_2)\} \)
16: end if
17: end if
18: if Type\( (v_1) = \text{Type}(v_2) = F \) then
19: \( M^* \leftarrow \text{minimum-cost bipartite matching} \)
20: \( M(v_1, v_2) \leftarrow \cup\{M(c_1(v_1), c_j(v_2)) \mid (c_1(v_1), c_j(v_2)) \in M^* \} \cup \{(v_1, v_2)\} \)
21: end if
22: return \( M(v_1, v_2) \)

Fig. 9. Bipartite matching for \( F \) nodes

D. Algorithm Complexity

Let \(|E|\) be the total number of edges in both \( R_1 \) and \( R_2 \), and let \(|V|\) be the total number of their vertices. Since both
$R_1$ and $R_2$ are SP-graphs, we have $|V| \leq 2|E|$ and therefore $|V| = O(|E|)$. Moreover, let $T_1$ and $T_2$ be their annotated SP-trees respectively. Let $N$ be the total number of nodes in both $T_1$ and $T_2$. By the derivation of annotated SP-trees, we have $N \leq 2|E| + 2|F| - 1$, where $|F|$ is the number of allowed forks in the specification $(G, F)$. Since we compute the edit distance for a fixed specification, we conclude that $N = O(|E|)$.

We now analyze the time complexity of our algorithm in terms of the dominant parameter $|E|$. First of all, a linear algorithm for tree decomposition problem of series-parallel graphs has been given by [33]. Generating the annotated SP-trees takes $O(|E|^2)$ time. In addition, the main overhead of computing the minimum-cost subtree deletion (Algorithm 3) lies in the dynamic programming used for $S$ nodes, which runs in $O(|E|^3)$ time. Finally, the main overhead of computing the minimum-cost well-formed mapping (Algorithm 4) is solving the weighted bipartite matching problem (a.k.a. the assignment problem) for each pair of $F$ nodes. This step can be done in $O(|E|^3)$ time, using the Hungarian algorithm [34]. Hence, the overall time complexity of our algorithm is $O(|E|^3)$.

VI. EXTENDED SP-WORKFLOW MODEL

We now present an extended SP-workflow model capable of expressing both fork and loop executions. In the extended model, an SP-workflow specification is given by a triple $(G, F, \mathcal{L})$ where $F$ and $\mathcal{L}$ represent well-nested forking and looping imposed over an SP-specification $G$ as a set of subgraphs, such that (1) $F \cap \mathcal{L} = \emptyset$; and (2) the edge sets of $F \cup \mathcal{L}$ form a laminar family. As before, elements of $F$ are series subgraphs. Elements of $\mathcal{L}$ are complete subgraphs. A complete subgraph of $G$ is either a series subgraph or a parallel subgraph of $G$ and contains all the paths from its source to its sink. In the canonical SP-tree for $G$, it corresponds to a nonempty proper subset of consecutive children of an $S$ node.

Intuitively, a loop execution of $H \in \mathcal{L}$ replicates one or more copies of $H$ and executes them in series; They are concatenated by an implicit edge from the sink of one copy to the source of the next copy, generating the series composition of one or more valid runs with respect to $H$ along with all implicit edges $(t(H), s(H))$ between them. As before, we may abstract the loop execution by the execution function $f$ below:

$$f(H) = S(f(H), (t(H), s(H)), f(H)) \quad \text{if } H \in \mathcal{L}$$

Example 6.1: Fig. 2(d) shows a run $R_3$ in which the loop has been executed twice; note the implicit edge $(6^a, 2^b)$.

The annotated SP-trees for the specification $(G, F, \mathcal{L})$ and the valid run $R$ are then constructed as before, adding $L$ nodes to represent allowed loop executions in $\mathcal{L}$. In particular, when generating the annotated SP-tree for $R$, we may capture the implicit edges from the sink of one iteration to the source of the next iteration by the order of children of an $L$ node.

While subtree insertion and deletion over annotated SP-trees remain the same, the corresponding operations over SP-graphs are more complicated since iterations are connected in series by implicit edges. However, to preserve the atomicity of our edit operations, at most one iteration of a loop should be inserted or deleted by a single edit operation. We therefore introduce two more path edit operations:

- **Path Expansion:** A path expansion operation creates a new iteration of a loop by inserting an elementary path between two existing consecutive iterations. Note that this operation also involves a set of necessary insertion and deletion of implicit edges.
- **Path Contraction:** This operation is the inverse of the path expansion operation. Intuitively, a path contraction operation removes an iteration of a loop by contracting the last elementary path.

Example 6.2: To delete the second iteration of the loop in $R_3$ (Fig. 2(d)), we first delete the path $(2^b, 5^a, 6^b)$, then contract the path $(2^b, 4^c, 6^b)$ by replacing the path $(6^a, 2^b, 4^c, 6^b, 7^a)$ with the edge $(6^a, 7^a)$.

To extend our differencing algorithm to handle loops, we need to consider one more case analysis for loops in Algorithms 1, 2, 3 and 4. Note that in Algorithms 1 and 3, loops are handled in exactly the same way as forks. We therefore discuss only the changes to Algorithms 2 and 4.

In Algorithm 2, we need to consider one more case where $T_G$ is rooted at an $L$ node. By counting the number of edges from the sink of one iteration to the source of the next iteration (i.e., $(t(T_G), s(T_G)))$, we can decide how many iterations of this loop are executed in $T_R$. A pseudo code presentation of our extended algorithm is given in Algorithm 5.

### Algorithm 5 Annotated-SP-Tree-for-Valid-Run (Extended for loops)

**Input:** annotated SP-tree $T_G$, canonical SP-tree $T'_R$

**Output:** annotated SP-tree $T_R = f''(T_G, T'_R)$

1: if $T_G = L(T_1)$ then
2:  if $T'_R = S(T'_1, T'_2, \ldots, T'_l)$ then
3:      $Y \leftarrow \{j_1 < j_2 < \ldots < j_{|Y|} \mid T'_j = (t(T_G), s(T_G))\}$
4:      $j_0 \leftarrow 0, j_{|Y|} + 1 \leftarrow j_0 + 1$
5:  for $i \in \{1, \ldots, |Y|\}$ do
6:      $X \leftarrow \{T'_j \mid j_{i-1} < j < j_i\}$
7:     if $|X| > 1$ then
8:        $T_{i}^R \leftarrow S(X)$
9:     else
10:        $T_{i}^R \leftarrow \text{element}(X)$
11:  end if
12: end for
13: return $L(\{f''(T_i, T_{i}^R) \mid 1 \leq i \leq |Y| + 1\})$
14: end if
15: if $T'_R = P(T'_1, T'_2, \ldots, T'_l)$ or $Q()$ then
16:   return $L(f''(T_1, T_{R}^R))$
17: end if
18: end if

In Algorithm 4, we also do one more case analysis: If $(v_1, v_2)$ is a pair of $L$ nodes, we set up the same bipartite graph between children of $v_1$ and $v_2$ as described in $F$ case. However, instead of finding a minimum-cost bipartite matching for $F$ nodes, we only need to compute a minimum-cost non-crossing bipartite matching for $L$ nodes, since the
children of an $L$ node are ordered. This problem can be efficiently solved by dynamic programming in $O(|E|^2)$ time. Note that the computation for $F$ nodes dominates the cost. So the overall time complexity is still $O(|E|^3)$. A pseudo code presentation of our extended algorithm is given in Algorithm 6.

**Algorithm 6** Edit-Distance-on-Trees (Extended for loops)

**Input:** $T_1[v_1], T_2[v_2]$  
**Output:** $M(v_1, v_2)$

1. if $\text{Type}(v_1) = \text{Type}(v_2) = L$ then
2. $M^* \leftarrow $ minimum-cost noncrossing bipartite matching
3. $M(v_1, v_2) \leftarrow \cup \{M(c_i(v_1), c_j(v_2)) \mid (c_i(v_1), c_j(v_2)) \in M^* \} \cup \{(v_1, v_2)\}$
4. end if
5. return $M(v_1, v_2)$

With all the above changes to our algorithms, we can obtain an efficient differencing algorithm for workflows characterized as SP-graphs with well-nested forking and looping.

**VII. Prototype-PDiffView**

We have developed a prototype system called Provenance Difference Viewer (PDiffView)$^2$ which allows users to view, store, generate and import/export SP-specifications and their associated runs. The user may then see the difference between two runs of the same specification by stepping through the set of edit operations in the minimum-cost edit script, or by seeing an overview. Since the graphs can be large, users may successively cluster modules in the specification to form a hierarchy of composite modules. The difference between two runs of that specification can then be viewed at any level in the defined hierarchy, giving the user the ability to zoom in on composite modules that indicate a large amount of change and ignore others that indicate no change.

**Example 7.1:** A snapshot of our prototype system is shown in Fig. 10. The big pane on the left-hand side shows the source run, with green edges indicating inserted paths and red edges indicating deleted paths in the edit script. The target run is shown in the big pane on the right-hand side. The small pane on top shows the specification, and the small pane on the top right gives the context for the edit operation being applied. The small panes on the bottom right and left corners display miniatures of the respective runs, and brief summaries of their statistics are listed above.

**VIII. Evaluation**

We empirically evaluate our differencing algorithm on both real and synthetic datasets. All experiments were performed on a local Pentium IV 2.8GHZ PC with 2GB memory running Fedora Core 6 with kernel version 2.6.20. The algorithm is implemented in Java 6, and specifications and runs are stored as XML files. In all experiments, the time to parse the XML file is omitted.

**A. Real Scientific Workflows**

In the first set of experiments, we evaluate the performance of our differencing algorithm over six collected, real scientific workflows$^3$. Characteristics of these specifications are listed in Table I. $|F|$ and $|L|$ are the number of forks and loops annotated in the specification, respectively, and $||F||$ and $||L||$ are the total number of edges in the forks and loops. For each specification, we randomly generate a pair of valid runs, varying their total number of edges from 200 to 2000, and then measure the execution time of computing the minimum-cost edit script under the unit cost model. Each point is an average over 100 sample pairs.

**TABLE I**  
Characteristics of Real Workflow Specifications

| WORKFLOW | $|V|$ | $|E|$ | $|F|$ | $||F||$ | $|L|$ | $||L||$ | $|F|+|L|$ |
|-----------|-----|-----|-----|-------|-----|-------|---------|
| PA        | 11  | 13  | 3   | 6     | 1   | 6     | 10      |
| EMBROSS   | 17  | 22  | 4   | 10    | 2   | 10    | 12      |
| SAXPF     | 27  | 36  | 7   | 18    | 1   | 7     | 19      |
| MB        | 17  | 19  | 2   | 6     | 1   | 6     | 4       |
| PGAQ      | 37  | 41  | 4   | 22    | 2   | 26    | 24      |
| BAIDD     | 29  | 36  | 8   | 17    | 2   | 12    | 20      |

Fig. 11 shows that our differencing algorithm performs well even on large runs. In the worst case, we can compute the edit distance between a pair of PGAQ workflow runs with a total of 2000 edges in less than one minute. In practice, most workflow runs have fewer than 200 edges, which can be done in less than one second.

Fig. 11 also shows that the execution time varies between specifications. However, it is hard to understand this variation using only the statistics listed in Table I. In the remainder of this section, we will show the effect of factors such as the proportion of series to parallel compositions, the number of nested forks and loops, and so on.

$^2$Available at http://www.seas.upenn.edu/~zhuowei/diff/index.html.

$^3$Real workflows can be found at http://www.myexperiment.org/.
we randomly generate a pair of valid runs with 
probabilities that each fork and loop copy is taken by the run, respectively. For instance, when \( r = +\infty \) the (series) specification becomes a single path, and when \( r = 0 \) the (parallel) specification consists only of two vertices and a set of multi-edges between them. First of all, we randomly generate a synthetic workflow specification with no forks and loops, varying the number of edges from 100 to 1000 and setting \( r \) to be 3, 1 and \( \frac{1}{3} \) respectively. For each specification, we randomly generate a pair of valid runs with \( \text{prob}_F = 95\% \), \( \text{max}_F \) and \( \text{max}_L \), and randomly generate a run with many loops (and no forks). Finally, we generate a run with many loops (and no forks). We then measure the execution time and edit distance under the unit cost model. Each point is an average over 200 sample specifications.

Fig. 12 shows that computing the edit distance between a pair of runs of a series specification is expensive: Since there are no forks and loops in the specification, finding the minimum-cost non-crossing bipartite matching \((O(|E|^3))\). In contrast, the minimum cost of deleting a subtree rooted at a \( P \) node can be easily computed in linear time \( O(|E|) \).

Fig. 12 also shows that the execution time increases with the size of specification, confirming our polynomial time complexity result. More interestingly, comparing Fig. 12 with Fig. 11, we observe that forks and loops bring a significant complexity to the differencing problem. Note that each run is randomly generated by taking on average 95\% of branches in the specification, thus making Fig. 12 comparable to Fig. 11.

Fig. 13 shows that a pair of runs of a series specification have a smaller edit distance than a pair of runs of a parallel specification. There are two reasons for this: 1) deleting a long path only incurs a cost of one under the unit cost model; and 2) fewer parallel branches in the series specification means that the runs generated will be more similar. Comparing Figs. 13 and 12, we conclude that there is little correlation between the running time of the algorithm and the edit distance.

### B. Series vs Parallel

In the second set of experiments, we compare series specifications with parallel specifications. Let \( r \) be the ratio of series compositions to parallel compositions used to construct the specification. For instance, when \( r = +\infty \) the (series) specification becomes a single path, and when \( r = 0 \) the (parallel) specification consists only of two vertices and a set of multi-edges between them. First of all, we randomly generate a synthetic workflow specification with no forks and loops, varying the number of edges from 100 to 1000 and setting \( r \) to be 3, 1 and \( \frac{1}{3} \) respectively. For each specification, we randomly generate a pair of valid runs with \( \text{prob}_F = 95\% \), where \( \text{prob}_F \) is the probability that each parallel branch in the specification is taken by the run. We then measure the execution time and edit distance under the unit cost model. Each point is an average over 200 sample specifications.

Fig. 12 shows that computing the edit distance between a pair of runs of a series specification is expensive: Since there are no forks and loops in the specification, finding the minimum-cost non-crossing bipartite matching \((O(|E|^3))\). In contrast, the minimum cost of deleting a subtree rooted at a \( P \) node can be easily computed in linear time \( O(|E|) \).

### C. Fork vs Loop

In the third set of experiments, we compare forks with loops. First, we randomly generate a synthetic workflow specification with 100 edges and a series/parallel ratio of 0.5, annotating it with 5 forks and 5 loops. To generate a random valid run, we use the following parameters: 1) \( \text{max}_F \) and \( \text{max}_L \) are the maximum number of copies replicated by each fork and loop executions respectively; and 2) \( \text{prob}_F \) and \( \text{prob}_L \) are the probabilities that each fork and loop copy is taken by the run, respectively. For instance, the product of \( \text{max}_F \) and \( \text{prob}_F \) is the average number of copies in a fork execution. We now fix \( \text{prob}_F = 1 \) and \( \text{max}_F = \text{max}_L = 20 \), and randomly generate a run with many forks (and no loops) by varying \( \text{prob}_F \) from 0 to 1 and setting \( \text{prob}_L \) to be 0. Similarly, we generate a run with many loops (and no forks). Finally, we measure the execution time and edit distance between different combinations of runs under the unit cost model. Each point is an average over 200 sample specifications.

Fig. 14 shows that computing the edit distance between a pair of runs with many forks is extremely expensive when \( \text{prob}_F \) is high, and that computing the edit distance between one run with many forks and one run with many loops is cheapest. This is because we do a minimum-cost bipartite matching to pair fork copies across the runs \((O(|E|^3 \log_2 |E|))\), whereas to pair loop copies we calculate a minimum-cost non-crossing bipartite matching \((O(|E|^2))\).
Furthermore, when we pair a run with many forks and a run with many loops, the bipartite matching instances are small because forked copies are never matched with loop copies.

Fig. 15 shows that the edit distance between a pair of runs with many forks (loops) will eventually drop to 0 when the fork (loop) probability approaches 1: Each fork copy will be replicated exactly \( max_F \) times, and the runs generated will have the same shape. In contrast, the edit distance between one run with many forks and one run with many loops monotonically increases, since a higher fork and loop probability results in a larger difference between the two runs. Comparing Figs. 15 and 14 again confirms that there is little correlation between running time and edit distance.

D. Influence of Cost Model on Edit Scripts

In the last set of experiments, we evaluate the influence of varying cost models on the minimum-cost edit script produced. Recall that any sublinear function \( \gamma(l) = l^\epsilon \), where \( \epsilon \leq 1 \) and \( l \) is the length of path to be edited, can be used.

![Image](a)

(a) (b)

Fig. 17. Influence of Cost Model on Edit Scripts

Consider the example in Fig. 17(a). Two edit scripts that transform \( R_1 \) to \( R_2 \) are: \( \mathcal{E}_1 = \{(1, 2, 3, 5) \rightarrow \Lambda, (1, 4, 5, 6) \rightarrow \Lambda\} \) and \( \mathcal{E}_2 = \{(1, 4, 5) \rightarrow \Lambda, (1, 2, 3, 5, 6) \rightarrow \Lambda\} \). Which script is better depends on \( \epsilon \): 1) when \( \epsilon = 0 \) (unit cost) or 1 (cost equal to length), \( \gamma(\mathcal{E}_1) = \gamma(\mathcal{E}_2) \); 2) when \( 0 < \epsilon < 1 \), \( \gamma(\mathcal{E}_1) > \gamma(\mathcal{E}_2) \); and 3) when \( \epsilon < 0 \), \( \gamma(\mathcal{E}_1) < \gamma(\mathcal{E}_2) \). Thus different cost models may lead to different minimum-cost edit scripts between the same pair of valid runs.

To empirically evaluate the effect of different cost models, we use the synthetic workflow specification in Fig. 17(b). The specification \( G \) contains a fork subgraph connecting a pair of nodes \( u \) and \( v \) by 10 parallel paths. The length of the \( i \)th path is \( i^2 \). We now randomly generate a pair of valid runs by setting \( max_F = 5 \), \( prob_F = 1 \) and \( prob_P = 0.5 \). Each random run then contains exactly 5 fork copies, and each copy includes a random subset of roughly 5 parallel paths. We then compute the minimum-cost edit scripts between a pair of runs under different cost models, by varying \( \epsilon \) from 0 to 1. Finally, we measure the percent error between the edit distance (i.e., the minimum cost) and the cost of these edit scripts under the unit (\( \epsilon = 0 \)) and length (\( \epsilon = 1 \)) cost models. We test for 100 pairs of sample runs and evaluate both average error and worst-case error.

Fig. 16 shows that the minimum-cost edit script produced by one cost model may be suboptimal for another, and that the corresponding cost may be far away from the edit distance (i.e., the minimum cost). As shown in Fig. 16, the average error under the unit cost model monotonically increases, while the average error under the length cost model monotonically decreases. The minimum-cost edit script produced by the length cost model has an average error of 14% and worst case of 50% under the unit cost model; the minimum-cost edit script produced by unit cost model has an average percent error of 16% and worst case of 64% under the length cost model. Not surprisingly, the minimum-cost edit scripts produced by other cost models show a tradeoff between the errors with respect to the unit and length cost models. This is due to the way in which fork copies of \( H \) are matched: In the unit cost model, copies which agree on the largest number of paths are matched, ignoring the lengths of unmatched paths. In the length cost model, matched copies may differ in many paths but agree on some of the longer paths.

IX. Conclusions

We show that the problem of differencing workflow runs of the same specification, described by series-parallel graphs overlaid with well-nested forks and loops, can be efficiently solved in \( O(|E|^3) \) time, where \( |E| \) is the number of edges in both graphs. The edit distance between a pair of valid runs is naturally defined as a minimum-cost set of elementary path insertions and deletions that transform the first into the second run, and preserves the validity of each intermediate run. The cost function used for each edit operation is compact yet general, allowing us to capture a variety of application-specific notions of distance, and depends on the start and end nodes as well as the length of the path. Experimental results show the scalability of our approach.

Using the prototype, we are currently working with scientists to help them 1) visualize the provenance of data results; 2) see the execution difference in the provenance of data results; and 3) see the parameter and input data difference of data results. In particular, we are developing techniques to help them see where the most salient differences lie when the provenance is large, and experimenting with the cost functions to see which works best in practice and under what conditions.

X. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by US National Science Foundation grants IIS 0513778, SEII 0612177 and IIS 0803524.

References


