Detecting and Resolving Unsound Workflow Views for Correct Provenance Analysis

Peng Sun[1], Ziyang Liu[1], Susan Davidson[2], Yi Chen[1]
Arizona State University[1], University of Pennsylvania[2]
{peng.sun,ziyang.liu,yi}@asu.edu[1]
susan@cis.upenn.edu[2]

ABSTRACT

Workflow views abstract groups of tasks in a workflow into high level composite tasks, in order to reuse sub-workflows and facilitate provenance analysis. However, unless a view is carefully designed, it may not preserve the dataflow between tasks in the workflow, i.e., it may not be *sound*. Unsound views can be misleading and cause incorrect provenance analysis.

This paper studies the problem of efficiently identifying and correcting unsound workflow views with minimal changes. In particular, given a workflow view, we wish to split each unsound composite task into the minimal number of tasks, such that the resulting view is sound. We prove that this problem is NP-hard by reduction from independent set. We then propose two local optimality conditions (weak and strong), and design polynomial time algorithms for correcting unsound views to meet these conditions. Experiments show that our proposed algorithms are effective and efficient, and that the strong local optimality algorithm produces better solutions than the weak local optimality algorithm with little processing overhead.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; H.2.8 [**Database Applications**]: Scientific databases

General Terms

Algorithms, Performance, Theory

Keywords

Workflow, View, Soundness, Provenance, Network, Connectivity

1. INTRODUCTION

Technological advances have enabled the capture of massive amount of data in many different domains, taking us a step closer to solving complex problems such as global climate change and uncovering the secrets hidden in genes. Workflow management systems [4, 21, 12, 5, 6, 27, 26, 28] are therefore increasingly used for managing and analyzing these data, allowing users to specify complex, multi-step, "in-silico" experiments or analyses. To ensure reproducibility and verifiability of results, many workflow systems are now providing support for provenance [14, 13, 22, 31, 7].

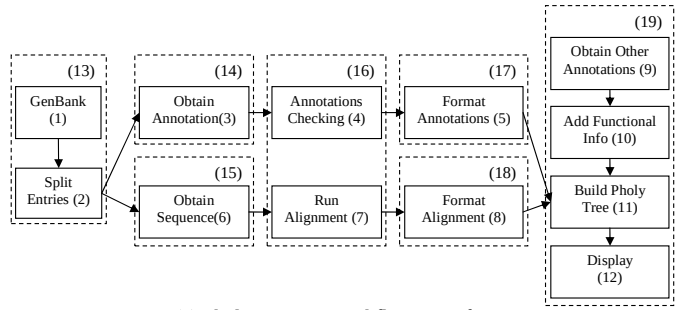
The *provenance* of a data item is the sequence of steps used to produce the data, together with the intermediate data and parameters used as input to those steps. In general, it can be thought of as a graph which captures the causal dependencies between entities such as data and processes (a *provenance graph* [2]), and queries of provenance as calculating transitive closures of dependencies [16]. As workflows become large and complex, the sizes of provenance graphs as well as the cost of answering transitive closure queries become problematic, and a number of techniques have recently been proposed for reducing the size of provenance graphs and the complexity of calculating provenance information [16, 8, 5, 6].

In this paper, we explore the use of *views* for efficient provenance analysis. By abstracting groups of tasks in a workflow into high level composite tasks, a view can hide irrelevant details and be much smaller than the original workflow. Thus analyzing provenance queries that involve transitive closures at the view level can be more efficient than that at the workflow level.

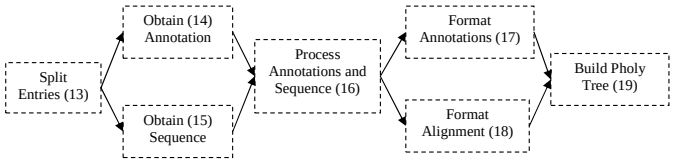
As an example, consider the workflow in Figure 1(a) which describes a common analysis in molecular biology: *Phylogenomic inference of protein biological function*. Tasks are modeled as nodes in a directed graph, where edges represent data dependencies between the tasks. Note that the graph itself is the provenance graph for the final output – a Phylogenomic tree – and that the data items flowing between tasks have been omitted for simplicity. In this workflow, users first select a set of entries in the GenBank database (1), and split the entries (2) to extract a set of sequences (6), and a set of annotations (3). The retrieved annotations are then curated (4) and formatted (5) to be served as input to building a Phylogenomic tree (11). For the extracted sequences, an alignment is performed (7) and then formatted (8). Other annotations for the sequences (9) may also be considered and processed (10) to serve as input to (11). A Phylogenomic tree will then be built and displayed (12).

A view is constructed by abstracting the task(s) in each dotted box into a composite task and preserving all the inter-composite task edges. For instance, Figure 1(b) is workflow

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.



(a) Phylogenomic Workflow Specification



(b) A Workflow View of (a)

Figure 1: Sample Workflow Specification and View of Building Phylogenomic Tree

view of (a). As we can see, the composite task Build Phylo Tree (19) consists of four atomic tasks, and simplifies the provenance graph for users who are not interested in the details of checking additional annotations. Such a view preserves the dataflow between adjacent tasks, however, it does not guarantee that the transitive closure of the edges, the paths, can also be preserved.

In fact, unless a view is carefully designed, it may not preserve the dataflow between two composite tasks, and thus can be misleading and cause incorrect provenance analysis. For example, suppose the user found that the formatted sequence output by task (18) in a workflow execution is not good, and would like to check its provenance with respect to the view in Figure 1(b). Based on the view, all the outputs of tasks (13), (14), (15) and (16) will be considered as the provenance of the output of task (18), as there are paths from each of them to task (18). This is, nevertheless, incorrect. There is no path between node (3) (which is the same as (14)) and (8) (which is the same as (18)) in the workflow in Figure 1(a), i.e., the output of task (14) is not part of the provenance of the output of task (18) according to the original workflow. Provenance analysis based on this view would therefore be incorrect.

Ideally, a view should *preserve all the data dependencies* between two composite tasks, without adding or removing paths. We call such a view *sound* with respect to provenance. Although it would seem natural to design views which are sound, our survey of workflow designs in a well-curated workflow repository [4] revealed unsound views. Furthermore, existing tools for constructing views (e.g., [5, 6]) do not guarantee this soundness property.

Our focus in this paper is therefore on *diagnosing* and *correcting* unsound views. First, we formally define what it means for a view to be sound. However, detecting an unsound view according to its definition is intractable. We thus define an equivalent problem of identifying unsound composite tasks in a view, which can be efficiently achieved.

Then we design algorithms to correct unsound composite tasks. Two alternatives can be pursued for correcting an unsound task: splitting it into multiple smaller tasks, or merging it with other tasks. Note that splitting composite tasks refines the initial view and provides more provenance information. In contrast, merging tasks loses information, as tasks that are important to the user may be invisible after the merger. Therefore, in this paper we focus on techniques that resolve an unsound view by splitting unsound composite tasks rather than merging them.

Our goal is to correct an unsound view by splitting its unsound composite tasks to a minimal number of tasks, each of which is sound. We show that this problem is NP-hard by reduction from the independent set problem. To efficiently tackle this hard problem, we propose two optimality criteria: weak local optimality and strong local optimality. A weakly local optimal solution is one in which no two tasks split from the same composite task can be merged into a sound task, and a strongly local optimal solution is one in which no any number of tasks split from the same composite task can be merged. We show that weak local optimality can be easily achieved with an $O(n^2)$ algorithm. However, achieving strong local optimality is much more challenging, as the straightforward approach takes exponential time. We then present a cleverer algorithm which achieves strong local optimality in polynomial time $O(n^3)$. The proposed algorithms are much more efficient than the algorithm which produces an optimal solution. The strongly local optimal algorithm often has comparable performance to the weakly local optimal algorithm, and produces solutions that are comparable to the optimal one.

Soundness diagnosis and correction can be done either by making suggestions while users are constructing a view, or by correcting unsound views after the view is created. At construction time, once a composite task is specified, we can check and correct it immediately; then the user can proceed to specify other composite tasks. Our proposed approach is applicable to all workflow views, including user-specified views in existing workflow repositories (e.g., Kepler, MyExperiment) and those built by users [4, 21, 15, 25, 11] or view-construction tools [12, 5, 6, 23]. In addition, while we motivate the sound view problem in the application of workflow provenance, this problem is general for diverse types of network data, such as computer networks, electronic circuits, protein-protein interactions; and our proposed techniques can generate network views that correctly preserve node connectivity.

The contributions of our work include:

- This work initiates a novel research problem: how to generate views that can correctly preserve the dataflows of a network, and proposes the notation of "sound views". We motivate this problem in a real application: provenance analysis in workflows.
- We design algorithms to efficiently validate workflow views and identify unsound composite tasks.
- We prove that the problem of correcting an unsound task by splitting it into a minimum number of sound tasks is NP-hard.
- We design efficient algorithms for correcting unsound tasks which produce solutions of different forms of local optimality in polynomial time.
- A system for resolving unsound views was developed and verified for its efficiency and effectiveness through

experimental studies. This provides an effective tool to guide the design of workflow views.

The rest of paper is organized as follows: Section 2 introduces the general workflow model and the definition of view. Section 3 defines sound views, formulates the View Soundness Problem (VSP) and its equivalent problem: Task Soundness Problem (TSP). Section 4 introduces two optimality criteria, weak local optimality and strong local optimality, and presents algorithms which achieve each criterion for unsound view correction. Experimental results are presented in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

2. PRELIMINARIES

In this section, we introduce the background on a general workflow model and views.

Definition 2.1: A *workflow specification* W is a directed graph where each node corresponds to a task and each edge indicates the dataflow between two tasks. $N(W)$ denotes the node set of W, and $E(W)$ the edge set. ∎

A *view* is a directed graph that is an abstraction of a workflow specification by grouping some tasks along with their edges together into single tasks, as defined in [5, 6, 4, 20, 23]. In other words, each task in a view corresponds to a group of tasks in the workflow specification.

For example, consider the workflow specification in Figure 2(a). A sample view is shown in Figure 2(b), where all the nodes in (a) are mapped to a single task M. (c) and (d) are another two views of (a), where each dotted rectangle denotes a composite task. The formal definition of view is given below.

Definition 2.2: A *view* V of workflow specification W is a directed graph induced by a partition of the nodes $N(W)$: $\{P_1, P_2, ..., P_n\}$ such that $\emptyset \neq P_i \subseteq N(W)$, $P_i \cap P_j = \emptyset$ for $i \neq j$ $(1 \leq i, j \leq n)$, and $P_1 \cup P_2 \cup ... \cup P_n = N(W)$.

- The node set of V, $N(V)$, is defined by a bijection from the partition to $N(V)$, $\Phi : 2^{N(W)} \longrightarrow N(V)$. $\forall P_i$, $\exists T_i = \Phi(P_i) \in N(V)$. Conversely, $\forall T_i \in N(V)$, $\exists P_i$, such that $\Phi^{-1}(T_i) = P_i$.
- The edge set of V, $E(V)$, is defined by a surjection from $E(W)$ to $E(V)$, $\Psi : E(W) \longrightarrow E(V)$. $\forall l \in E(W)$ from t_m to t_n, $t_m \in P_i$, $t_n \in P_j$. If $i \neq j$, $\Psi(l) = L \in E(V)$, where L is an edge from $\Phi(P_i)$ to $\Phi(P_j)$; conversely, $l = \Psi^{-1}(L)$. Otherwise if $i = j$, $\Psi(l) = \epsilon$.
- Each $T \in N(V)$ is called a *composite task*, and each $t \in N(W)$ an *atomic task*. An edge $l \in E(W)$ is called an *intra-group edge* with respect to V if $\Psi(l) = \epsilon$, and an *inter-group edge* otherwise.

∎

Intuitively, Φ maps each group in a partition in the workflow specification to a composite task in the view. An inter-group edge l is mapped by Ψ to an edge which connects two composite tasks in the view, while an intra-group edge is mapped to ϵ.

3. VIEW AND TASK SOUNDNESS PROBLEM

In this section we define sound views, and formulate the view soundness problem and task soundness problem. We then prove that both problems are NP-hard.

3.1 Sound Views

According to Definition 2.2, a view *preserves the edges* among composite tasks. Specifically, there is an edge from composite task T_i to T_j in the view, if and only if there exists $t_i \in P_i = \Phi^{-1}(T_i)$ and $t_j \in P_j = \Phi^{-1}(T_j)$ in the corresponding workflow specification, such that there is an edge from t_i to t_j. This is a desirable property of a view, as such a view preserves the dataflow between adjacent tasks.

Intuitively, we may think that the transitive closure of an edge – a path – in the workflow is also preserved in the view, and thus both immediate (i.e., shallow) provenance and transitive (i.e., deep) provenance are correctly captured in the view. Unfortunately, this is not the case. As discussed in Section 1, a view, which preserves the edges by definition, may not preserve the paths in the workflow specification. In particular, a view may have a path between two tasks which actually does not exist in the workflow specification. Consider the view in Figure 2(b), since M is viewed as a single task, there is a path consisting of edges L_1 and L_5. However, in the workflow specification, there is no path that includes both $l_1 = \Psi^{-1}(L_1)$ and $l_5 = \Psi^{-1}(L_5)$. This view, therefore, does not preserve the paths in the workflow specification. Such a view conveys incorrect dataflow information and thus is misleading and may lead to incorrect provenance analysis.

We call a view that preserves the paths in the corresponding workflow specification as a *sound view*, and *unsound view* otherwise, formally defined in the following definition.

Definition 3.1: A view V of a workflow specification W is *sound* if the following conditions hold:

1. $\forall T_i, T_j \in N(V)$, for any path from T_i to T_j consisting of edges $L_1.L_2.\cdots.L_p$, there exists a path in W consisting of edges $\Psi^{-1}(L_1).C_1.\Psi^{-1}(L_2).C_2.\cdots.C_{p-1}.\Psi^{-1}(L_p)$, where $C_m (1 \leq m \leq p-1)$ is a path consisting of intra-group edges only.

2. $\forall t_i, t_j \in N(W)$, for any path from t_i to t_j consisting of edges $l_1.l_2.\cdots.l_p$, there is a path in V consisting of edges $\Psi(l_1).\Psi(l_2).\cdots.\Psi(l_p)$.

∎

As we can see, a view is sound if and only if it preserves the paths, that is, for any path p in the view, there is a corresponding path p' in the workflow specification, and vice versa. A path p in the view corresponds to a path p' in the workflow specification if p' consists of the mapping Ψ of edges in p, with possibe intra-group edges in between. For example, Figure 2(c) is a sound view: for any path in Figure 2(c), there is a corresponding path in (a). For instance, the corresponding path in the workflow specification of path $L_1.L_2.L_4$ is $l_1.l_2.l_3.l_4$, where l_1, l_2 and l_4 map to L_1, L_2 and L_4, and l_3 is an intra-group edge. Besides, for any path in (a), there is also a corresponding path in (c).

Our focus in this paper is to determine unsound views and refine an unsound view into a sound one, e.g., Figure 2(b) is determined to be an unsound view, and (c), (d) are two possible refinements of (b). Next we discuss how to determine unsound views, then define a *view soundness problem*, whose goal is to refine an unsound view with minimal cost. We then transform it to an equivalent problem called *task soundness problem*, and prove it to be NP-hard.

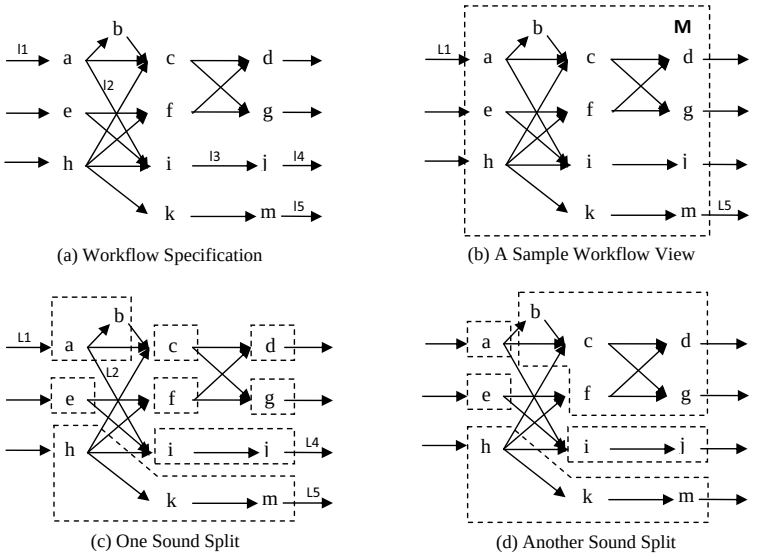


Figure 2: Sound versus Unsound Views

3.2 Determining Unsound Views

We first need to check whether a view is sound. However, it is very expensive to do so by directly applying Definition 3.1, as we will need to check if every path in the view satisfies condition 1, and every path in the data satisfies condition 2 in Definition 3.1. Even if we do not count cycles, the number of paths in a workflow specification is exponential to the number of tasks. Thus checking view soundness by checking paths is intractable.

Interestingly, we discover that, to check view soundness, it is sufficient to check the paths between the input and output of each composite tasks, which can be done in polynomial time. We define the soundness of a task in a view, and show that a view is sound if and only if all its tasks are sound. This property scales the problem down to task-level, such that we can check the soundness of each task to determine the soundness of a view, and later on correct each unsound task individually to resolve an unsound view.

Definition 3.2: Given a composite task T, $T.in$ denotes the atomic tasks in T that receive input from some atomic task $t \notin T$, and $T.out$ denotes the atomic tasks in T that send output to some atomic task $t \notin T$. ∎

Definition 3.3: A composite task T in a workflow view is *sound* if and only if $\forall t_i \in T.in$ and $\forall t_o \in T.out$, there is a directed path from t_i to t_o which is composed of nodes in $\Phi^{-1}(T)$ and intra-group edges. ∎

As an example, the composite task M in Figure 2(b) is unsound because there is no path from $a, e \in M.in$ to $m \in M.out$. All composite tasks in Figure 2 (c) and (d) are sound.

Proposition 3.1: A view V of a workflow specification W is sound if and only if all composite tasks in V are sound.

PROOF. $\Rightarrow$: Suppose V is a sound view, but it contains an unsound task T. Since T is unsound, $\exists t_i \in T.in$ and $t_o \in T.out$, such that there is no directed path from t_i to t_o which is composed of intra-group edges in T. Assume that an incoming edge of t_i is l_i, and an outgoing edge of t_o is l_j. Let $L_i = \Phi(l_i)$ and $L_j = \Phi(l_j)$.

In V, there is a path $L_i.L_j$. By Definition 3.1, there should be a path in the workflow specification, such that it consists of l_i, l_j and intra-group edges in T. However, there is no such path, which is a contradiction to the assumption that V is a sound view. Thus, we can conclude that if a view V of a workflow specification is sound, then all the composite tasks in V are sound.

$\Leftarrow$: Suppose all the composite tasks in V are sound. Take any path p in V consisting of edges $L_1.L_2.\cdots.L_p$. Let $l_i = \Psi^{-1}(L_i)(1 \leq i \leq p)$, and let the endpoints of l_i be n_{i1} and n_{i2}. Consider a path consisting of nodes $n_{11}, n_{12}, n_{21}, n_{22}, \cdots, n_{p1}, n_{p2}$ in the workflow specification. Each n_{i1} and n_{i2} are connected by edge l_i, and each n_{i2} and $n_{i+1,1}$ must be the same task, or belong to the same composite task. Since each composite task in V is sound, there is a path from n_{i2} to $n_{i+1,1}$ for all i consisting of intra-group edges. Therefore, the entire path consisting of edges l_i and intra-group edges, which means V satisfies condition 1 of Definition 3.1.

By Definition 2.2, it is easy to prove that V satisfies condition 2 as well. Therefore, V is a sound view. ∎

According to Proposition 3.1, we can check whether a view is sound by checking whether each composite task in the view is sound. To check the soundness of a composite task, we simply need to check whether there is a directed path composed of intra-group edges from every input to every output of the task, which can be done efficiently. Therefore we have the following proposition.

Proposition 3.2: Checking whether a view is sound can be done in polynomial time with respect to the number of tasks in the workflow specification.

PROOF. A polynomial time algorithm that checks soundness of a view is given in Section 4.1 (Algorithm 1). Thus the proof is omitted. ∎

3.3 Refining Unsound Views

To make a view sound, according to Proposition 3.1, we can make each composite task in the view sound. As dis-

cussed in Section 1, we only correct unsound composite tasks by splitting them for finer granularity. Therefore, we define the notion of view refinement as follows.

Definition 3.4: A view V_1 of workflow W is a *refinement* of another view V_0 of W if and only if $\forall T_1 \in N(V_1)$, $\exists T_0 \in N(V_0)$, such that $\Phi_1^{-1}(T_1) \subseteq \Phi_0^{-1}(T_0)$. The set of tasks $\{T_1 | \Phi_1^{-1}(T_1) \subseteq \Phi_0^{-1}(T_0)\}$, which can be viewed as a split of T_0, is called a *group* and denoted as $S(T_0)$. A refinement/split is *sound* if the resulting view is sound. ∎

According to Definition 3.4, a refinement of an unsound view V_0 is to split some composite tasks into smaller tasks to create a finer grained view V_1. For example, the unsound view in Figure 2(b) is refined into (c) by splitting task M in (b) into eight smaller tasks shown in dotted rectangles in (c), each of which is sound.

Definition 3.5: Given a view V_0, the *cost of splitting* a task T_0 in V_0 into a set of tasks $S(T_0)$ in V_1 is defined as $cost(T_0, S(T_0)) = |S(T_0)| - 1$. The *cost of refining* the view V_0 to another view V_1 is given as:
$cost(V_0, V_1) = \sum_{T_0 \in N(V_0)} cost(T_0, S(T_0))$. ∎

Considering the split of one task into two tasks as an atomic edit operation, our cost function is equivalent as the edit distance between the original view and the refined view. In Figure 2, the cost of refining (b) into (c) is 7, while the cost of refining (b) into (d) is only 4. In fact, the view in (d) is the refinement of (b) with minimum cost.

Given an unsound view, we would like to refine it with minimum cost, thus we define the following view soundness problem.

Definition 3.6: *View Soundness Problem (VSP):* Given an unsound view V_0, find a sound refinement V_1 such that $cost(V_0, V_1)$ is minimized over all sound refinements of V_0. ∎

According to Proposition 3.1, VSP can be converted to a task soundness problem (TSP, Definition 3.7). Also, according to the cost of refinement in Definition 3.5, it is easy to see that to refine a view with minimum cost, we should split each unsound task in the view into the smallest number of sound tasks. In the rest of this paper, we focus on the discussion of TSP.

Definition 3.7: *Task Soundness Problem (TSP):* Given an unsound task T_0, find a sound split $S(T_0)$ such that $cost(T_0, S(T_0))$ is minimized among all the sound splits of T_0. ∎

3.4 NP-hardness of Task Soundness Problem

The decision problem of TSP is defined as: Given a directed graph, can we divide it into at most S disjoint subgraphs, such that for each subgraph, each of its inputs can reach each of its outputs?

We prove the NP-completeness of this problem by reduction from the independent set problem. We first introduce two definitions and a proposition, and then prove the NP-completeness.

Definition 3.8: A complete bipartite task of size p, K_p, is defined as follows:

1. There are two sets of nodes, $a_1, \cdots, a_p$ and $b_1, \cdots, b_p$.
2. $a_i \in K_p.in (1 \leq i \leq p)$, $b_i \in K_p.out (1 \leq i \leq p)$.
3. There is an edge from each $a_i (1 \leqslant i \leqslant p)$ to each $b_j (1 \leqslant j \leqslant p)$.

∎

For example, the composite task shown in Figure 3(a) is K_3.

Definition 3.9: The *join* of two complete bipartite tasks of size p, K_p and K'_p, is defined as follows:

Let the nodes of K_p be $a_1, \cdots, a_p$, $b_1, \cdots, b_p$, and the nodes of K'_p be $a'_1, \cdots, a'_p$, $b'_1, \cdots, b'_p$. Before the join, all nodes in K_p and K'_p are called *open nodes*.

1. Randomly select an open node n_1 in $b_1, \cdots, b_p$ (or $b'_1, \cdots, b'_p$), and another open node n_2 in $a'_1, \cdots, a'_p$ (or $a_1, \cdots, a_p$).
2. Remove the output of n_1 and the input of n_2, and merge the nodes.

After the join, the merged node (n_1, n_2) is called a *closed node*. ∎

For example, consider K_3 and K'_3 in Figure 3(a). If the open nodes b_1 and a'_3 are selected, then the joined task is shown in Figure 3(b), which is an unsound task.

Similarly, we can join multiple complete bipartite tasks by joining every pair of them, and it is easy to see that the resulting task must be unsound.

Proposition 3.3: Suppose there is an unsound task T, which is the result of joining some pairs of M complete bipartite tasks of size p, $(K_p)_1, \cdots, (K_p)_M$. Then, in any sound refinement of this task, the following statements hold:

1. Each resulting task must be either an entire $(K_p)_i$, or a single node, and
2. Suppose $(K_p)_i$ is a resulting task, then for each $(K_p)_j$ which is joined with $(K_p)_i$ (i.e., $(K_p)_j$ has a common node with $(K_p)_i$), each task in $(K_p)_j$ must be in a separate resulting task.

∎

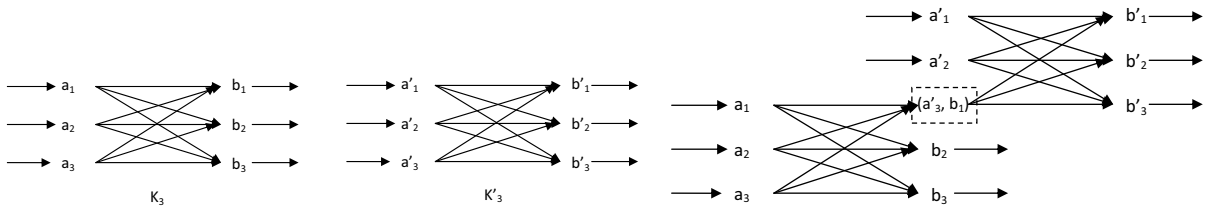
The formal proof is omitted for the space limitation. The idea of proposition 3.3 is that, for a complete bipartite task, either all atomic tasks of it are in a group, or each of its atomic tasks is in a separate group. No other subset of a complete bipartite task can form a sound group, either by itself or with nodes from other complete bipartite tasks. In Figure 3(b), for example, nodes $\{a_1, a_2, a_3, b_1, b_2, b_3\}$ either are in one group, or in six separate groups.

From Proposition 3.3, a minimum cost refinement of joined bipartite tasks can be obtained. The smallest split of Figure 3(b) is 6: we can either put $\{a_1, a_2, a_3, (a'_3, b_1), b_2, b_3\}$ into one group and the other five nodes into five groups, or put $\{a'_1, a'_2, (a'_3, b_1), b'_1, b'_2, b'_3\}$ into one group and the other five nodes into five groups. Based on this proposition, we have the following proof of NP-completeness of the decision version of TSP.

Theorem 3.4: The decision version of TSP is NP-complete.

PROOF. The decision version of TSP is in NP, as we can verify whether a given split of a composite task is sound in polynomial time.

To prove the NP-completeness, consider an arbitrary instance of the independent set problem. Let $G(N, E)$ be an



(a) Selecting b_1 and a'_3 (b) Merging b_1 and a'_3

Figure 3: Joining K_3 and K'_3

undirected graph. Let $m = |N|$. The independent set problem is: Can we find a subset N' of N, such that $|N'| \geq c$ and there is no edge between any two nodes in N'?

Now we construct an instance of TSP. For each node $n \in N$, we prepare a complete bipartite task K_m. For every pair of nodes n_i, $n_j \in N$ that have an edge, we join the corresponding bipartite tasks, $(K_m)_i$ and $(K_m)_j$. Now, we get a big composite task T, which is unsound. Suppose T has M nodes. The following question is asked: Can we split T into at most $(c + M - 2cm)$ groups, such that every group is sound? It is easy to see that this transformation takes polynomial time.

Given a solution N' of the independent set problem, suppose N' has c' nodes ($c' \geq c$), $n_1 ... n_{c'}$. Then in the corresponding TSP, we let each of $(K_m)_1$, $(K_m)_2 ... (K_m)_{c'}$ form a group. Since no two of them are joined, this is legal. Each other node has to be in one group alone. The total number of groups is thus $c' + M - 2c'm \leq c + M - 2cm$.

Now the other direction. Suppose there is a solution for TSP. Note that according to Proposition 3.3, each group either contains a K_m or contains a single node. Therefore, if the number of groups is no more than $c+M-2cm$, it is easy to see that there must be at least c such K_m, each forming a group. This means no two of such K_m are joined. Then, we can get a solution N' to the independent set problem, which consists of the corresponding nodes in N. ∎

4. ALGORITHMS

In this section, we first discuss how to discover unsound tasks in a workflow view efficiently, then propose algorithms for splitting an unsound composite task into a set of groups, each of which is a sound task.

We proved in Section 3 that the task soundness problem is NP-hard. Therefore, instead of aiming for the optimal solution, we propose two other criteria which, as will be shown later, can be achieved in polynomial time: *weak local optimality* and *strong local optimality*. An algorithm that satisfies weak or strong local optimality does not necessarily produce the optimal split for an unsound task, but one that is "good" in a local sense. We show that weak local optimality is easy to achieve (Section 4.2). However, achieving strong local optimality is much more challenging. We present in Section 4.3 an algorithm which is strongly local optimal, as proved in Section 4.4.

4.1 Detecting Unsound Tasks in a Workflow View

We begin with discussing how to determine whether a composite task is sound. First, we introduce the notion of *input set*.

Algorithm 1 Unsound Task Detection Algorithm

DetectAndCorrect (V)
1: **for** each $T \in N(V)$ **do**
2: CalcinSet(T)
3: **if** !IsSound(T) **then**
4: Split(T) {See Algorithm 2 and 3}
5: **end if**
6: **end for**

CalcinSet (T)
1: **for** each $t \in T.out$ **do**
2: go backwards along dataflow from t to traverse each task t'
3: **if** $t' \in T.in$ **then**
4: $t.inSet(T) = t.inSet(T) \cup \{t'\}$
5: **end if**
6: **end for**

IsSound (T)
1: **for** each $t \in T.out$ **do**
2: **if** $t.inSet(T) \neq T.in$ **then**
3: return $false$
4: **end if**
5: **end for**
6: return $true$

Definition 4.1: In a composite task T, the *input set* of a task $t \in N(T)$ with respect to T, denoted as $t.inSet(T)$, is a set of nodes $N \subseteq T.in$, such that $\forall n \in N$, n can reach t through directed edges. ∎

For example, for the composite task T in Figure 4(a), the $inSet(T)$ of each atomic task is annotated next to the atomic task in Figure 4(b).

According to Definitions 3.3 and 4.1, a task T is sound if $\forall t \in T.out$, $t.inSet(T) = T.in$. For the composite task in Figure 4(a), we determine that it is unsound because $m.inSet(T) = \{h\}$ while $T.in = \{a, e, h\}$.

The algorithm for detecting unsound tasks is presented in Algorithm 1. For each composite task T in a view V, procedure DetectAndCorrect calls CalcinSet(V) to compute $inSet(T)$ for each task in $T.out$, which can be obtained by traversing T. Then it calls IsSound(T) to check the soundness of T, and if T is unsound, it calls Split(T) to split it (for which it can use Algorithm 2 for weak local optimality or Algorithm 3 for strong local optimality).

For each task $t \in T.out$, Algorithm 1 finds $t.inSet(T)$ by traversing the task starting from t, which takes $O(n^2)$ time, where n is the number of atomic tasks in T. Therefore, if T has m output tasks, then the complexity of Algorithm 1 is $O(mn^2)$.

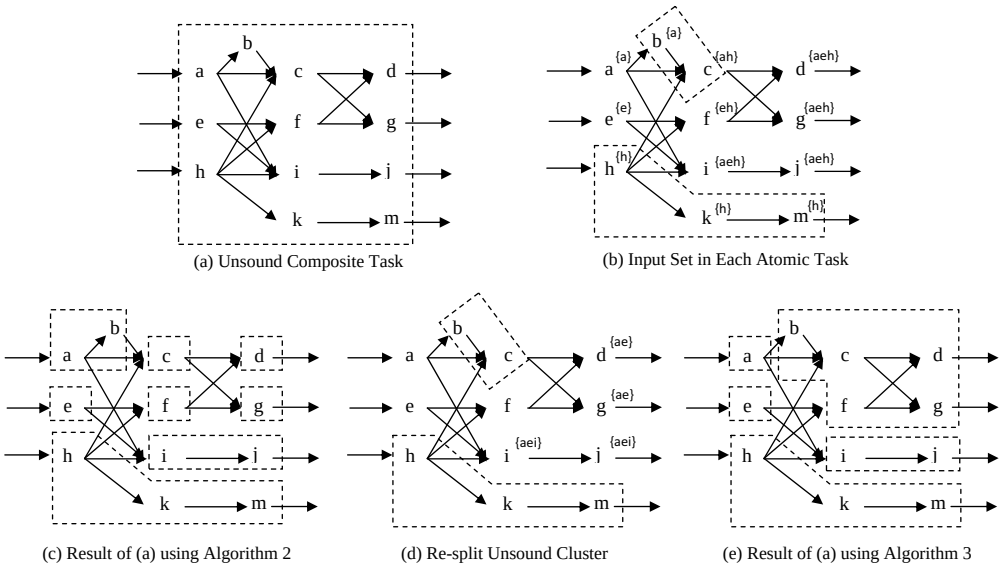


Figure 4: Example about View Correction

4.2 Weakly Local Optimal Algorithm

In this subsection we introduce the *weak local optimality* criterion for judging the quality of correcting an unsound task. Then a polynomial time algorithm satisfying weak local optimality is presented.

Definition 4.2: If two tasks T_1 and T_2 can be merged so that the resulting composite task is sound, then T_1 and T_2 are *combinable*, denoted as $T_1 \asymp T_2$. If a set of tasks $\mathcal{T}$ can be merged so that the resulting composite task is sound, then we say $\asymp(\mathcal{T})$. ∎

Definition 4.3: A split $\mathcal{S} = S_1, S_2, ..., S_n$ of an unsound task T is *weakly local optimal* if and only if there does not exist $S_i, S_j \in \mathcal{S}$, $S_i \asymp S_j$ $(1 \leq i, j \leq n)$. An algorithm for the task soundness problem is called a *weakly local optimal algorithm* if for any unsound task T, it guarantees to produce a split which is weakly local optimal. ∎

In short, weak local optimality indicates that any two groups in the split are not combinable. Weak local optimality can be achieved by an algorithm which works as follows:

1. Split the given unsound task, such that each group contains one atomic task.
2. Test whether any two groups are combinable. If so, merge them.
3. Repeat step 2 until no two groups are combinable.

The pseudo code of this algorithm is shown in Algorithm 2. Procedure Split recursively tries to merge two groups a and b by calling CanMerge(a, b), which checks whether two groups a and b are combinable. If there is no edge from a node in b to a node in a (i.e., $b2a$ is false), then there is no edge connecting b and a, which means that any node in b cannot reach any node in a through the nodes in a and b. Therefore, if b has an input i that does not come from a (b_in = true) and a has an output o that does not go to b (a_out = true), then i cannot reach o through the nodes in a and b, which means the combination of a and b is not sound (lines 20-22). It is similar with lines 23-25.

In our running example, Figure 4(a), during the first iteration we find that $a \asymp b$, $i \asymp j$ and $h \asymp k$, and we merge these three pairs. During the next iteration, we find that the group containing h and k can be merged with m, i.e., $(h, k) \asymp m$, and put h, k, m in one group. During the third iteration, no two groups are found to be combinable, and therefore we stop and output the current split. The result of splitting the task in Figure 4(a) using Algorithm 2 is shown in Figure 4(c).

Algorithm 2 obviously gives a split of an unsound task which is guaranteed to be weakly local optimal. The formal proof is omitted.

Now we analyze the time complexity of Algorithm 2. Let n denote the number of atomic tasks in T, thus we initially have n groups. Each time we execute line 8 to merge two groups, the total number of groups decreases by 1, thus line 8 is executed at most n times. During each "while" loop, suppose there are currently p groups $(p \leq n)$, and line 8 is executed q times to merge $2q$ groups into q groups $(2q \leq p)$. Then in the next iteration of the "while" loop, we have $p-2q$ old groups and q new groups, thus the "for" loop will be executed $q^2 + q(p-2q) < pq \leq nq$ times. This means that, each time line 8 is executed in the current "while" loop, it causes procedure CanMerge (line 7) to execute at most n times in the next iteration of the "while" loop. In addition, CanMerge is executed n^2 times in the first iteration of the "while" loop. Therefore, CanMerge is executed $O(n^2)$ times altogether.

The cost of CanMerge can be reduced to O(1) using a hash table. For each group, the hash table records (1) whether it has an outgoing or incoming edge to/from each other group, and (2) the number of its incoming and outgoing edges (multiple outgoing/incoming edges to/from the same group are counted as 1). Then, each "if" clause in CanMerge takes O(1) time to compute. Since CanMerge is executed $O(n^2)$ times in the whole process, the total cost it poses is $O(n^2)$. Besides, the hash table needs to be maintained every time two groups a and b are merged (line 8 in Split). For a and b's entries, we need to merge them in this

Algorithm 2 Weakly Local Optimal Algorithm

SPLIT (T)

```
1: break T into pieces, i.e., each single task forms a group
2: changed = true
3: while changed = true do
4:    changed = false
5:    for every pair of new groups (a, b) and every pair of one
      old group and one new group (a, b) do
6:       {Initially, all groups are new groups. From the 2nd it-
         eration, a group is a new group if it is merged from two
         groups in the last iteration.}
7:       if CanMerge(a, b) then
8:          merge a and b
9:          changed = true
10:      end if
11:   end for
12: end while
```

CANMERGE (a, b)

```
1: a2b = b2a = a_in = a_out = b_in = b_out = false
2: if a has an outgoing edge to b then
3:    a2b = true
4: end if
5: if a has an incoming edge from b then
6:    b2a = true
7: end if
8: if a has more outgoing edges other than the one to b then
9:    a_out = true
10: end if
11: if a has more incoming edges other than the one from b then
12:    a_in = true
13: end if
14: if b has more outgoing edges other than the one to a then
15:    b_out = true
16: end if
17: if b has more incoming edges other than the one from a then
18:    b_in = true
19: end if
20: if b2a=false AND a_out=true AND b_in=true then
21:    return false
22: end if
23: if a2b=false AND b_out=true AND a_in=true then
24:    return false
25: end if
26: return true
```

way: for each other group c, (a, b) has an outgoing/incoming edge to/from c if and only if either a or b does. The number of outgoing/incomging edges of (a, b) is the summation of that of a and b, subtracted by the number of groups that has edges to both a and b plus 1 (because there must be edge(s) between a and b). This part takes $O(n)$ time. For each other group c's entry, we need to update its content. c has an outgoing/incoming edge to/from (a, b), if and only if c has an outgoing/incoming edge to/from either a or b. If c has outgoing/incoming edges to both a and b, then the number of c's incoming/outgoing edge is decreased by 1. This part also takes $O(n)$. Therefore, maintaining the tables upon a merger takes $O(n)$. Since a merger (line 8 of SPLIT) can happen at most n times, the total cost is $O(n^2)$ in the whole process.

Therefore, the total time complexity of Algorithm 2 is $O(n^2)$.

4.3 Strongly Local Optimal Algorithm

Running Algorithm 2, which is weakly local optimal, on the task in Figure 4(a) produces the split shown in Figure 4(c). As we can see, tasks c, d, f and g in Figure 4(c) are combinable. However, since no two of them are combinable, Algorithm 2 fails to put them in a single group. In this subsection, we propose and discuss a stronger optimization goal, namely *strong local optimality*, for achieving better splits of unsound tasks.

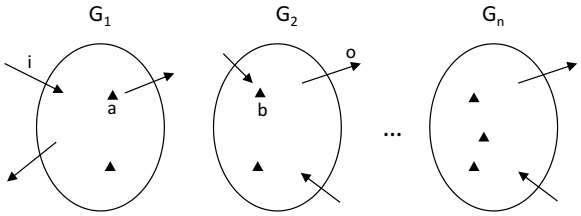


Figure 5: Strongly Connected Component

Definition 4.4: A split $\mathcal{S} = S_1, S_2, ..., S_n$ of an unsound task T is *strongly local optimal* if and only if there does not exist $\mathcal{S}' \subset \mathcal{S}$, $\asymp (\mathcal{S}')$. An algorithm for the task soundness problem is called a *strongly local optimal algorithm* if for any unsound task T, it guarantees to produce a split which is strongly local optimal. ∎

Achieving strong local optimality is much more challenging than achieving weak local optimality. A straightforward way of achieving strong local optimality is to check whether any subset of groups are combinable. Since the number of subsets is exponential with respect to the number of atomic tasks, this takes exponential time.

Before presenting a clever algorithm that achieves strong local optimality in polynomial time, we introduce a lemma and a definition as necessary background.

Lemma 4.1: Given an unsound task T, if there is a strongly connected component $S \subseteq T$, then in any strongly local optimal split of T (hence any optimal split of T as well), all tasks in S must belong to the same group.

PROOF. In Figure 5, an unsound task is split into a number of groups. All the triangle nodes comprise a strongly connected component, which is not placed in one group, but instead separated in n groups $G_1, G_2, ..., G_n$. Assume that the lemma is not true, and this split is strongly local optimal.

Since the triangle nodes are placed in multiple groups, in any such group, e.g., G_1, at least one triangle node, e.g., a, belongs to $G_1.out$, as it needs to connect to triangle nodes in another group. For the same reason, in any other such group, e.g., G_2, at least one triangle node b belongs to $G_2.in$. This indicates that $\asymp (G_1, G_2, ..., G_n)$. The reason is that for any input i of G_1 and any output o of another group G_2, i can reach a through edges in G_1 (because G_1 is sound), a can reach b through edges among the triangle nodes, and b can reach o through edges in G_2. This means that each input of $\bigcup(G_1, G_2, ..., G_n)$ can reach each of its output through edges in it, and thus $\asymp \{G_1, G_2, ..., G_n\}$, which contradicts with the assumption that the split is strongly local optimal. ∎

Lemma 4.1 indicates that if an unsound task has a strongly connected component within it, the nodes in the strongly connected component can be directly merged into one group. In the rest of this section, we focus on discussion on unsound tasks which do not have strongly connected components.

Definition 4.5: In a composite task T, the *complete pre-*

decessor set of a set of nodes $U \subseteq T$, denoted as $CPS(U)$, is a set of nodes $P \subseteq T \backslash U$, such that for every task $p \in P$, each of p's output edges points to a task in U. The *complete predecessor closure* of U, denoted as $CPC(U)$, is equal to $U \cup CPS(U) \cup CPS(CPS(U)) \cup CPS(CPS(CPS(U))) \cup ....$ ∎

For example, in Figure 4(a), let $U = \{d, g, i, j\}$. Then $CPS(U) = \{c, f\}$. $h \notin CPS(U)$ because h has an output edge $(h \rightarrow k)$ that does not point to any task in U. $CPC(U) = \{a, b, c, d, e, f, g, i, j\}$.

The concept of CPC is crucial for achieving strong local optimality in polynomial time. As to be proved in Section 4.4, for any set of nodes S in a composite task T, if there is a sound group S' such that $S = S'.out$, then $CPC(S)$ is sound. Therefore, we can first split the nodes in T into separate groups similar as Algorithm 2, then find a set of groups that can be merged, if there is one, by testing whether $CPC(S)$ is sound for every possible S.

However, the number of all possible S is exponential. Fortunately, we do not need to test all of them. We will prove in Section 4.4 that for any S, if there is a sound group S' such that $S = S'.out$, then every node in S have the same $inSet(T)$. Therefore, we can cluster the nodes in T according to $inSet(T)$, and iteratively re-cluster until an S is found, such that $CPC(S)$ is sound.

Now we present an algorithm which guarantees to produce a strongly local optimal split for any unsound task in polynomial time. The pseudo code is shown in Algorithm 3.

Similar as Algorithm 2, Algorithm 3 first separates each atomic task in a unique group (lines 3-6 of procedure Split). In each "while" loop, it computes $t.inSet(T)$ for each node $t \in T$ (line 9), then clusters the groups in T into $\mathcal{C} = \{C_1, C_2, ..., C_c\}$ (line 10). Groups with the same $inSet$ are placed in the same cluster.

In Figure 4(a), the $inSet(T)$ of each node is annotated near the node in Figure 4(b). There are six clusters: $C_1 = \{a, b\}$, $C_2 = \{e\}$, $C_3 = \{h, k, m\}$, $C_4 = \{c\}$, $C_5 = \{f\}$, $C_6 = \{d, g, i, j\}$.

Then, Algorithm 3 finds the complete predecessor closure (CPC) of each $C_i \in \mathcal{C}$. In our example, $CPC(C_4) = \{b, c\}$, and $CPC(C_6) = \{a, b, c, d, e, f, g, i, j\}$. The CPC of each other cluster is itself. Each CPC is a potential sound task, and there cannot be a sound task that contains nodes in two or more CPCs. Then the algorithm checks each $C_i \in \mathcal{C}$ to see whether $CPC(C_i)$ is sound and has at least two nodes (lines 1-5 of procedure FindSoundCluster). If so, we merge the groups in $CPC(C_i)$, and then repeat the "while" loop in Split.

In the running example, there are three sound CPCs with at least two nodes: $CPC(C_1) = \{a, b\}$, $CPC(C_3) = \{h, k, m\}$, $CPC(C_4) = \{b, c\}$. We merge one of them, e.g., merging b and c. In the next loop, there is only one such CPC: $CPC(\{h, k, m\}) = \{h, k, m\}$, so we merge h, k and m into one group. Now we have the set of tasks shown in Figure 4(b).

In the third iteration, there is no sound CPC with at least two nodes. In this case, we further cluster the nodes in each C_i with two or more groups based on the $inSet$ with respect to $CPC(C_i)$ (lines 6-7 of procedure FindSoundCluster), and recursively do so until the CPC of a cluster is sound, or no sound CPC containing at least two nodes exists.

Continuing the example, in the third iteration the nodes in T form five clusters: $C_1 = \{a\}$, $C_2 = \{(b, c)\}$, $C_3 = \{e\}$, $C_4 = \{(h, k, m)\}$, $C_5 = \{d, g, i, j\}$. There is only one CPC with at least two nodes, which is $CPC(C_5) = \{a, (b, c), d, e, f, g, i, j\}$ and is unsound. Therefore, we compute the $inSet$ of each node in C_5 with respect to $CPC(C_5)$, which is shown next to each node in Figure 4(d). Then we cluster them accordingly into two clusters: $\{d, g\}$ and $\{i, j\}$. Now we find that $CPC(\{d, g\}) = \{(b, c), d, f, g\}$, which is sound. Therefore, we merge (b, c), d, f and g into one group. The procedure is continued until the result is produced as shown in Figure 4(e).

Now we analyze the time complexity of Algorithm 3. Each "while" loop in procedure Split merges at least two groups into one, therefore the "while" loop is executed at most n times, where n is the number of nodes initially in T. During each "while" loop, to compute $inSet(T)$ of the nodes $t \in T$ (line 9), we use a pre-calculated reachability matrix of the graph (which can be built in $O(n^2)$ time). Therefore lines 9-10 take $O(n)$ time. The "for" loop in lines 11-20 recursively processes each cluster $C_j \in \mathcal{C}$. If $CPC(C_j)$ is sound and contains more than one group, then we merge them and enter the next "while" loop. The soundness of $CPC(C_j)$ can be checked in $O(|C_j|)$ time, where $|C_j|$ is the

Algorithm 3 Strongly Local Optimal Algorithm

```
Split (T)
 1: {Initially, each atomic task in T is represented by a node.
    Each atomic task is in a separate group}
 2: changed = true
 3: group = ∅
 4: for each atomic task t ∈ T do
 5:    group = group ∪ {t}
 6: end for
 7: while changed = true do
 8:    changed = false
 9:    Compute t.inSet(T) for each node t ∈ T
10:    Cluster the groups in T according to their inSet into C =
       {C_1, C_2, ..., C_c}. Nodes with the same inSet are placed in
       the same cluster.
11:    for each C_j ∈ C do
12:       if (soundCluster = FindSoundCluster(C_j)) ≠ NULL
          then
13:          changed = true
14:          for each group g ∈ soundCluster do
15:             group = group − {g}
16:          end for
17:          group = group ∪ soundCluster
18:          break
19:       end if
20:    end for
21: end while
FindSoundCluster (cluster)
 1: if CPC(cluster) has only 1 node then
 2:    return NULL
 3: else if IsSound(CPC(cluster)) then
 4:    return CPC(cluster)
 5: end if
 6: Compute t.inSet(CPC(cluster)) for each node t ∈ cluster
 7: Cluster the nodes in cluster according to their inSet into
    C = {C_1, C_2, ..., C_c}. Nodes with the same inSet are placed
    in the same cluster.
 8: for each C_j ∈ C do
 9:    if (soundCluster = FindSoundCluster(C_j)) ≠ NULL
       then
10:       return soundCluster
11:    end if
12: end for
13: return NULL
```

number of groups in C_j. Since $\sum_{C_j \in \mathcal{C}} |C_j| \leq n$, checking the soundness of all such $CPC(C_j)$ takes $O(n)$ time. Those C_js whose CPC is not sound are then recursively separated into several clusters (lines 6-7 of FindSoundCluster); checking the soundness of all their CPCs takes $O(n)$ time for the same reason. Since the recursive procedure FindSoundCluster can run at most n rounds, the "for" loop in Split takes $O(n^2)$ time. Therefore, the total complexity of Algorithm 3 is $O(n^3)$.

4.4 Proof of Strong Local Optimality

To prove that Algorithm 3 is strongly local optimal, we start with some lemmas.

Lemma 4.2: In an unsound composite task T, for any set of tasks $S \subseteq T$ such that S is sound, $\forall t_1, t_2 \in S.out$, $t_1.inSet(T) = t_2.inSet(T)$.

Proof. Since S is sound, $\forall t \in S.out$, $t.inSet(T) = \bigcup_{t' \in S.in} t'.inSet(T)$. Therefore, any two nodes in $S.out$ has the same $inSet$ with respect to T. ∎

Lemma 4.2 indicates that, after we cluster tasks in T into $\mathcal{C} = C_1, C_2, ..., C_c$ in line 10 of procedure $SPLIT$, for any sound composite task $S \subseteq T$, $S.out$ must be a subset of a cluster C_i $(1 \leq i \leq c)$.

Lemma 4.3: In an unsound composite task T, for any set of tasks $S \subseteq T$, if there exists $S' \subseteq T$ such that S' is sound and $S'.out = S$, then $S' \subseteq CPC(S)$.

Proof. Suppose there is an $S' \subseteq T$, S' is sound, $S'.out = S$ but there is a node $s \in S'$ such that $s \notin CPC(S)$. Then according to the definition of CPC, s must have an output edge that goes to somewhere else other than the nodes in S. Since every task has at least one input and one output edge, this output of s goes out of S' via a set of output edges of S', which means there must be at least one output edge of S' that does not belong to S. This contradicts with the assumption. ∎

Lemma 4.4: In an unsound composite task T, $\forall S \subseteq T$, if there exists $S' \subseteq T$, such that S' is sound and $S'.out = S$, then $CPC(S)$ is sound.

Proof. Suppose there exists an S' such that S' is sound, $S'.out = S$ and $S' \neq CPC(S)$. According to Lemma 4.3, $S' \subset CPC(S)$. Then for any node $s \in CPS(S')$, $\{s\} \asymp S'$. The reason is that since $s \in CPS(S')$, each of s's output edges serves as an input edge of S, as illustrated in Figure 6. Transitively, all other nodes in $CPC(S')$ can be merged with S', i.e., $CPC(S')$ is sound. Since $S = S'.out$, it is easy to see that $CPC(S) = CPC(S')$. Thus, $CPC(S)$ is sound. ∎

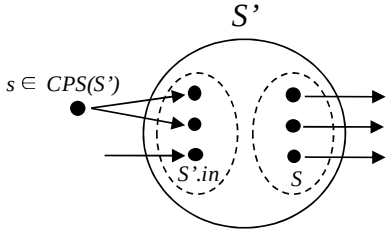


Figure 6: Lemma 4.4

Lemma 4.4 indicates that, given a set of nodes S, if we want to find out whether there exists a sound task S' whose output node set is exactly S (i.e., $S'.out = S$), we can simply see if the complete predecessor closure of S is sound. Besides, according to Lemma 4.3, if $CPC(S)$ is sound, then it is the largest sound composite task whose output node set is exactly S.

Now we present the proof of strong local optimality of Algorithm 3.

Theorem 4.5: Algorithm 3 is strongly local optimal.

Proof. To show that the algorithm is strongly local optimal, we only need to prove that, during any "while" loop between lines 7 and 21 of Split, if the current split is not strongly local optimal, then the algorithm guarantees to find a set of groups that can be merged. If this is true, then it means that the number of groups decreases at each iteration, and the iteration stops when the split reaches strong local optimality.

Suppose that in a "while" loop, the current composite task is T. Each group is represented by a single node in T. Suppose there is a set of groups, S, that can be merged. Then our algorithm will find S in the following way:

1. Line 10 of Split splits T into $\mathcal{C} = C_1, C_2, ..., C_c$. According to Lemma 4.2, $\exists C_j \in \mathcal{C}$, $S.out \subseteq C_j$.
2. If $S.out = C_j$, then according to Lemma 4.4, $CPC(C_j)$ is sound. Therefore, when we visit $CPC(C_j)$ during the "for" loop between lines 11 and 20, a sound group is found.
3. Otherwise, since $S.out \subseteq C_j$, $CPC(S.out) \subseteq CPC(C_j)$. Line 7 of procedure FindSoundCluster recursively clusters tasks in C_j into $\mathcal{C}' = C'_1, C'_2, ..., C'_{c'}$ based on the $inSet$ of each node in C_j with respect to $CPC(C_j)$. According to Lemma 4.2, $\exists C'_{j'} \in \mathcal{C}'$, $S.out \subseteq C_{j'}$.
4. If $S.out = C_{j'}$, then according to Lemma 4.4, $CPC(C_{j'})$ is sound. Therefore, a sound group is found. Otherwise, procedure FindSoundCluster recursively cluster tasks in $C_{j'}$ into a set of groups. The procedure continues in such a way. We can observe that:
   - At each clustering (line 10 in Split and line 7 in FindSoundCluster), since the set of tasks being clustered is not sound, we at least produce two clusters. This means that the algorithm will terminate, as the clusters become smaller and smaller.
   - At each clustering, according to Lemma 4.3, we guarantee to produce a cluster whose CPC is a superset of S, and this cluster becomes smaller each time (as the nodes in it are further clustered into at least two clusters each time). So we guarantee to find S eventually.

Therefore, this algorithm will always terminate, and if in the current "while" loop, we find such an S, we merge the nodes in it, treat it as a single task and then enter the next "while" loop. Otherwise, the current split is strongly local optimal and we output it. ∎

5. EVALUATION

In our experimental evaluation, we begin with surveying workflow views appearing in the real world (Section 5.1). To

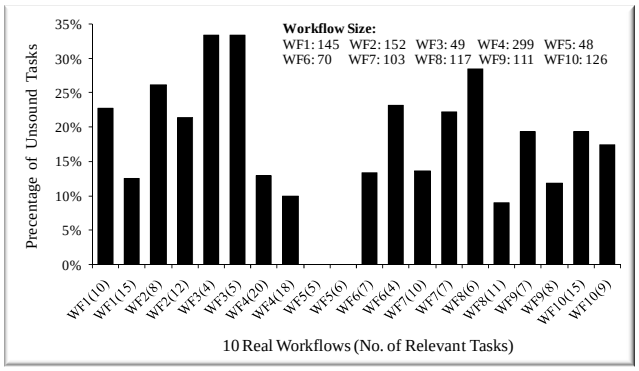


Figure 7: The Percentage of Unsound Views Constructed by Zoom

evaluate our approach, we test them on synthetic workflows from two perspectives: the number of tasks produced when correcting an unsound view (Section 5.2), and the time they take to correct an unsound view (Section 5.3).

The experiments were performed on a laptop with Intel Core(TM) 2 CPU 1.66GHZ, 2GB memory, running Windows XP.

5.1 Applicability of Our Approach

We have checked workflows in the Kepler repository [4], which contains the well-curated workflows with composite tasks specified by domain experts. Most views in Kepler are sound, which indicates that sound views are generally desirable in practice, but unsound views exist even in a well-curated workflow repository.

Manual view construction is very time consuming and can be error prone since a workflow can have hundreds of nodes. Workflow view construction tools, such as Zoom [5, 6], are designed to help users construct views of workflows. Zoom takes as input a set of user specified relevant tasks, and generates a view that is driven by the user's interest. Here, we choose ten workflows from Kepler repository; for each workflow, we randomly specify 2 sets of relevant tasks, whose sizes are no more than 10% of the workflow size. We then use Zoom to construct a view for it, and count the number of unsound tasks among all the composite tasks created by Zoom.

As shown in Figure 7, only 2 out of 20 of views generated by Zoom are sound. In addition, the percentage of unsound tasks within an unsound view varies from 9% to 33.3%, which is a measurement of the view quality. We also observe that there is no obvious relationship between the view quality and the size of the workflow.

This set of experiments suggests that although sound views are generally desirable, unsound views are often created.

5.2 Quality of Unsound View Correction

The effectiveness of our algorithms is tested on the synthetic data sets in Table 1, which is measured as the size of the refined view.

To generate synthetic workflows, we take commonly used workflow patterns[1] (e.g., parallel, sequential, etc.) and randomly combine them, varying the number of atomic tasks in the workflow. To combine two workflow patterns, we randomly connect the inputs of one pattern to the outputs of the other. We repeatedly combine workflow patterns until the desired workflow size is reached. Seven sets of workflows are generated based on their sizes, as shown in Table 1. Each set has 50 workflows; in each synthetic workflow, the number of inputs/outputs is rougly 25% of the workflow size.

[1]http://www.workflowpatterns.com/

Table 1: Synthesized Workflows with Different Sizes

Set #	Input Size	Output Size	Workflow Size	Task Size
Set 1	3±1	3±1	10±1	8±1
Set 2	20±2	20±2	100±10	90±10
Set 3	50±2	50±2	200±10	190±10
Set 4	80±5	80±5	300±10	290±10
Set 5	110±5	110±5	400±10	390±10
Set 6	140±10	140±10	500±10	490±10
Set 7	170±10	170±10	600±10	590±10

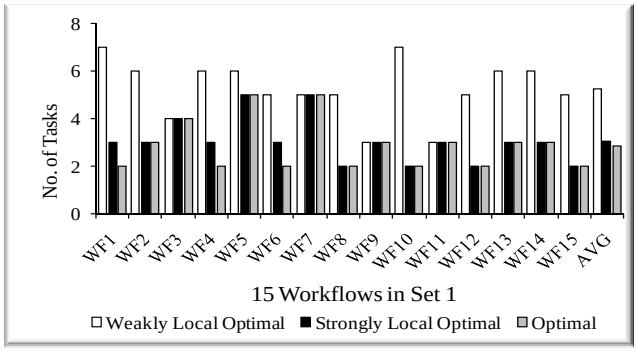


Figure 8: Quality of Weakly, Strongly Local Optimal and Optimal Algorithms

Since the processing time of the optimal algorithm grows exponentially with the size of the composite task, and becomes unacceptable (several hours) when the workflow size is only 12, we only test the optimal algorithm on workflows of Set 1 in Table 1.

Figure 8 shows the quality of the three algorithms on views of the 15 randomly selected workflows of Set 1 in Table 1. As we can see, the number of tasks in the view refined by the strongly local optimal algorithm is the same as that of the optimal algorithm for many views, and is similar to the optimal algorithm for the others. On the other hand, the weakly local optimal algorithm may have many more tasks in the refined view. The last column shows the average quality of the three algorithms on all the workflows of Set 1. It indicates that the strongly local optimal algorithm has an almost perfection quality.

Figure 9 shows the average number of tasks in the refined views using the weakly and strongly local optimal algorithms on workflows of Sets 1-7 in Table 1. As we can see, the numbers of tasks in the refined views using both the weakly and strongly local optimal algorithms increase with the workflow

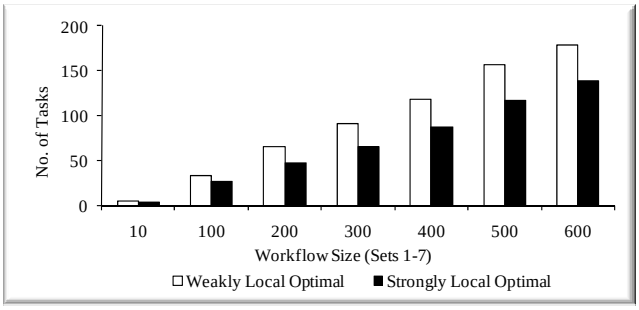


Figure 9: Quality of Weakly, Strongly Local Optimal Algorithms w.r.t Workflow Size

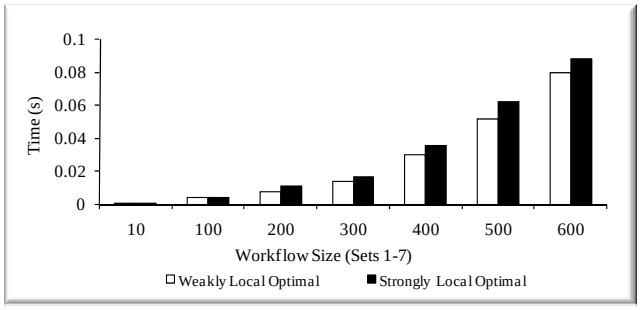


Figure 10: Efficiency of Weakly, Strongly Local Optimal Algorithms w.r.t Workflow Size

size. Moreover, the strongly local optimal algorithm consistently produces fewer tasks than the weakly local optimal one.

From the quality tests, it is shown that the strongly local optimal algorithm produces better quality refinements (roughly 15% better) than the weakly local optimal one. In addition, the quality of the strongly local optimal algorithm is quite close to the quality of the optimal algorithm.

5.3 Processing Time

The average processing time on the views of Set 1 in Table 1 for the weakly and strongly local optimal algorithms and for the optimal algorithm are 104 milliseconds, 41.7 milliseconds and 118×10^6 milliseconds, respectively. Clearly, the processing time of both the weakly and strongly local optimal algorithms are significantly better than that of the optimal algorithm even for small workflows. Note that the strongly local optimal algorithm is faster than the weakly local optimal one for this set of workflows. The reason is that when a workflow is small, it is likely that each complete predecessor closure (CPC) is already sound, and does not need to be recursively split by procedure *FindSoundCluster* in Algorithm 3. However, the weakly local optimal algorithm still needs to compare every pair of tasks.

Figure 10 shows the processing time of workflows of Sets 1-7 in Table 1. As we can see, when the size of workflow is less than 600, the average processing time for both algorithms is less than 0.1 second. The processing time of both the weakly and strongly local optimal algorithms increases with the size of the workflow. Although the strongly local optimal algorithm generally takes more time than the weakly local algorithm in each class, when the size of the workflow is less than 600 the extra processing time is no more than 0.01 second. This indicates that the strongly local optimal algorithm provides a more practical solution with good quality improvements and little processing overhead compared with the weakly local optimal one.

To summarize, the strongly local optimal algorithm is the best overall choice regardless of workflow size. It produces views with similar quality as the ones generated by the optimal algorithm with much better efficiency, which is comparable with the efficiency of the weakly local optimal algorithm.

6. RELATED WORK

Provenance on workflows has been much studied in recent works [14, 13, 22, 31, 7]. As workflows become large and complex, the cost of answering transitive closure queries for provenance information becomes unacceptable. Therefore, a number of techniques [16, 8, 5, 6] have been proposed to reduce the complexity of provenance graph and improve provenance calculation efficiency. One way to simplify the provenance graph of a workflow is to create views in workflow management system, and the provenance information can then be shown to users w.r.t the specified views.

Many workflow systems allow a user to manually create composite tasks which constitute views. For instance, Kepler [4], Taverna/myGrid [21] and MyExperiment [1] allow users to specify a composite task/view through a graphical interface or using files. [15, 25, 11] allow users to specify views using their proposed view definition languages. [29] enables users to retrieve views from a repository of predefined views using simple keywords by dynamically constructing relevant views that contain both query keywords and the dataflow among them.

There are also systems [20, 23, 5, 12, 6, 19] that construct views automatically based on user requirements. The *Zoom* system [12, 5, 6] constructs views for focusing user attention on user-specified "relevant tasks" on a workflow, such that (1) a composite task either contains one relevant composite task or none, (2) there is a path between relevant composite tasks in the view if and only if there is path between their contained relevant tasks in the workflow, (3) the number of non-relevant composite tasks are minimized. [20] constructs "order-preserving" views for easing execution order analysis on workflows. Specifically, if there is an *edge* between two composite tasks T_i and T_j in a view, then for *any* atomic task t_m in T_i and t_n in T_j, there is a path from t_m to t_n. In contrast, in this paper we propose the definition of *soundness* of a view with respect to provenance analysis. None of existing work can guarantee that the generated views are sound.

The term "workflow soundness" has been used in existing works to evaluate the execution of workflow: a workflow execution is sound if and only if every process will be terminated and there are no dangling references or dead tasks [17, 32, 18, 30, 33, 24]. In this paper, we present the notion of "view soundness", which refers to correct provenance analysis at the view level.

Another related field is *Program Slicing* [34], which is a program analysis technique for debugging and understanding programs. It attempts to identify a set of program points whose execution contributes to the value of a variable. Both program slicing and data provenance analyze data dependency [3, 10, 9]. Recent papers [10, 9] identifies the problem

of "incorrect dependency provenance" on programs, which shares the same spirit of the "unsound view" problem on workflows. No existing works, however, provide a solution to resolve incorrect dependency provenance in programs.

7. CONCLUSION AND FUTURE WORK

Unsound views, which do not correctly preserve path information in the workflow specification, can be misleading and lead to incorrect provenance analysis. In this paper, we formalize and study the problem of identifying and correcting unsound views. A view is sound if and only if it preserves the data dependencies between any two composite tasks. After we identify an unsound view, we generate a sound refinement of the view by splitting each unsound composite task into a minimal set of sound tasks. Consequently, we define the view soundness problem and the task soundness problem, and prove that they are NP-hard. In order to provide practical solutions, we introduce two criteria: weak local optimality and strong local optimality. Then we design polynomial time algorithms for correcting unsound views to satisfy these criteria. Empirical evaluations show that they efficiently generate sound view refinements with good quality. Furthermore, the strongly local optimal algorithm produces better solutions with little processing overhead compared with the weakly local optimal algorithm.

In the future, we will investigate approximation or randomized algorithms for this problem, and study how to correct unsound views with minimum cost if tasks are allowed to be either split or merged.

8. ACKNOWLEDGEMENT

This material is based on work partially supported by NSF CAREER award IIS-0845647, NSF IIS-0740129, IIS-0803524, IIS-0629846, IIS-0612177, IIS-0415810 and IIS-0513778.

9. REFERENCES

[1] myExperiment. http://www.myexperiment.org/workflows.

[2] Open provenance model, 2008.

[3] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. In *POPL*, 1999.

[4] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *SSDBM*, 2004.

[5] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and Managing Provenance through User Views in Scientific Workflows. In *ICDE*, 2008.

[6] O. Biton, S. B. Davidson, S. Khanna, and S. Roy. Optimizing user views for workflows. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 310–323, 2009.

[7] S. Bowers, T. M. McPhillips, and B. Ludäscher. Provenance in Collection-Oriented Scientific Workflows. *Concurr. Comput.*, 20(5):519–529, 2008.

[8] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient Provenance Storage. In *SIGMOD*, 2008.

[9] J. Cheney. Program Slicing and Data Provenance. In *IEEE Data Eng. Bull.*, volume 30(4), pages 22–28, 2007.

[10] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as Dependency Analysis. In *Database Programming Languages*, pages 138–152, 2007.

[11] D. K. W. Chiu, S. C. Cheung, S. Till, K. Karlapalem, Q. Li, and E. Kafeza. Workflow View Driven Cross-Organizational Interoperability in a Web Service Environment. *Inf. Technol. and Management*, 5(3-4):221–250, 2004.

[12] S. Cohen, S. C. Boulakia, and S. B. Davidson. Towards a Model of Provenance and User Views in Scientific Workflows. In *DILS*, 2006.

[13] S. M. S. da Cruz, P. M. Barros, P. M. Bisch, M. L. M. Campos, and M. Mattoso. Provenance Services for Distributed Workflows. In *CCGRID*, 2008.

[14] S. B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *SIGMOD*, 2008.

[15] R. Eshuis and P. Grefen. Constructing Customized Process Views. *Data Knowl. Eng.*, 64(2):419–438, 2008.

[16] T. Heinis and G. Alonso. Efficient Lineage Tracking for Scientific Workflows. In *SIGMOD*, 2008.

[17] R. B. A. Kamel Barkaoui and Z. Sbai. Workflow Soundness Verification based on Structure Theory of Petri Nets. In *International Journal of Computing and Information Sciences*, 2007.

[18] E. Kindler, A. Martens, and W. Reisig. Inter-operability of Workflow Applications: Local Criteria for Global Soundness. In *Business Process Management*, pages 235–253, 2000.

[19] D.-R. Liu and M. Shen. Modeling Workflows with a Process-View Approach. In *DASFAA*, 2001.

[20] D.-R. Liu and M. Shen. Workflow Modeling for Virtual Processes: an Order-Preserving Process-View Approach. *Inf. Syst.*, 28(6):505–532, 2003.

[21] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows. In *Bioinformatics*, 2003.

[22] S. Rajbhandari, O. F. Rana, and I. Wootten. A Fuzzy Model for Calculating Workflow Trust Using Provenance Data. In *ACM Mardi Gras*, 2008.

[23] M. R. Ralph Bobrik and T. Bauer. View-Based Process Visualization. In *Business Process Management*, pages 88–95, 2007.

[24] W. Sadiq and M. E. Orlowska. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. In *CAiSE*, 1999.

[25] Z. Shan, D. K. W. Chiu, and Q. Li. Systematic Interaction Management in a Workflow View Based Business-to-Business Process Engine. In *HICSS*, 2005.

[26] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis. EasyTicket: a Ticket Routing Recommendation Engine for Enterprise Problem Resolution. *Proc. VLDB Endow.*, 1(2), 2008.

[27] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis. Efficient Ticket Routing by Resolution Sequence Mining. In *SIGKDD*, 2008.

[28] Q. Shao, P. Sun, and Y. Chen. Efficiently Discovering Critical Workflows in Scientific Explorations. *Future Gener. Comput. Syst.*, 25(5):577–585, 2009.

[29] Q. Shao, P. Sun, and Y. Chen. WISE: a Workflow Information Search Engine. In *Proceedings of ICDE*, 2009.

[30] J. Siegeris and A. Zimmermann. Workflow Model Compositions Preserving Relaxed Soundness. In *Business Process Management*, pages 177–192, 2006.

[31] Y. L. Simmhan, B. Plale, and D. Gannon. A Framework for Collecting Provenance in Data-Centric Scientific Workflows. In *ICWS*, 2006.

[32] W. M. P. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In *Business Process Management*, pages 161–183, 2000.

[33] J. Vanhatalo, H. Völzer, and F. Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *ICSOC*, 2007.

[34] M. Weiser. Program Slicing. In *ICSE*, 1981.