Theoretical Framework for Eliminating Redundancy in Workflows

Dhrubajyoti Saha
dhrsaha1@in.ibm.com
IBM India, Bangalore

Abhishek Samanta
absamant@in.ibm.com
IBM India, Bangalore

Smruti R. Sarangi
srsarangi@in.ibm.com
IBM India, Bangalore

Abstract

In this paper we look at combining and compressing a set of workflows, such that computation can be minimized. In this context, we look at two novel theoretical problems with applications in workflow systems and services research, which are duals of each other. The first problem looks at merging the maximum number of vertices in two DAGs (directed acyclic graphs) without creating a cycle. We prove that the dual of this problem is the problem of maximizing the length of the LCS (longest common subsequence) between all pairs of topological orderings of the two DAGs. This formulation generalizes to a new definition of LCS between complex structures like workflows or XML documents, which we call M-LCS. Subsequently, we present a taxonomy of the different kinds of problems in this set, and find the M-LCS solution for a tree and a chain with a dynamic programming algorithm. Along with this theoretical formulation, we implement the algorithms in C++ and run it on representative workflows. We evaluate the performance of the M-LCS algorithm on a set of random workflows and observe that it is substantially better than traditional AI based approaches.

1 Introduction

Workflow systems [9, 14] have been in use in different branches of engineering, science, and services. Recently, they have become very important in grid and cloud computing scenarios. We look at a set of workflows, especially for parameter sweep applications and try to compress them by eliminating redundancy. We look at the problem from a theoretical angle in this paper, and conclude with practical implementations of the theoretical results. We focus on the simpler problem of trying to compress two workflows. We will extend this result in future papers.

To summarize, the aim of the paper is as follows. Take two workflows expressed as DAGs (directed acyclic graphs), and produce another structure, which has the same information as both the original workflows, but has a smaller size. There are two variables here. The first is the topology of the final data structure, and the nature of the information that we are trying to preserve. We observe (described in detail in Section 2.1) that it is better if the final graph is a DAG. The reason for this is that it is much simpler to visualize, prove correctness properties, and work with other standards. For the second problem, we can have a couple of measures. If two nodes take the same input and produce the same output, then they can be merged to one node. If two nodes take different inputs, but do the same computation, then also they can be merged in a different sense. We can logically merge them, which basically means that the scheduler can co-locate them on the same computing node, to maximize temporal locality. We don't put any strict restrictions on this, as our approach is more theoretical. We assume that every vertex has a label. If two vertices have the same label, they can potentially be merged. The problem reduces to that of finding maximal common regions in two DAGs.

This problem of finding matching regions in two acyclic structures has been studied for a long time in different contexts. The LCS (longest common subsequence) problem is a classic problem in computer science [7]. It finds the longest matching subsequence in two sequences and has been solved with a dynamic programming algorithm [7]. For example, if sequence 1 is "ABCBAD", and sequence 2 is "CBDE", the longest common subsequence between the two sequences is "CBD". Please note that the LCS need not be contiguous.

There is a need for generalizing it to higher dimensional structures. Proteins have a secondary and tertiary structure that is modeled by trees [4]. This problem requires the largest subtree between two trees. RNAs have secondary and tertiary structures that require common regions between two directed acyclic graphs [3, 13]. Sadly other than the basic LCS problem and variants of it like TreeLCS [15], other problems have either not deserved enough attention or have been proven to be NP-Complete. For example, a recent paper [1] generalizes LCS to matrices and forests, and proves both the problems to be NP Complete.

There is one similarity in the definition of LCS for higher dimensional structures in all the previous works [1, 15]. Their variant of LCS is the dual of the minimum edit dis-

tance problem, which can be stated as follows. Edit distance is the minimum number of node deletions, modifications, and substitutions to make two structures isomorphic. For example the tree edit distance/LCS problem is defined as:
Definition TreeLCS : The LCS of two rooted and labelled trees is the largest forest that can be obtained by deleting nodes.

However, in this paper we are motivated by a different subset of problems that require a different definition of LCS, which we call *M-LCS*. This problem will find the largest common sub-regions in two DAGs satisfying some further constraints. We show in Section 3 that the problem of M-LCS, and merging two DAGs are the duals of each other. The former is a much more general and powerful framework. It leads to a very simple implementation with very promising results (see Section 6).

2 M-LCS and Related Problems

2.1 The DAG-Merge Problem

This is the main problem regarding compressing workflows that we try to solve in this paper. A distributed job like a workflow is typically represented as a directed acyclic graph(see Figure 1(a)). Each vertex is labeled with a number. If two vertices have the same label, then they run a similar program possibly with different inputs. We can have another job with the structure (see Figure 1(b)).

We wish to merge both the jobs into one DAG (Figure 1(c)). In Figure 1(c), we have merged vertices with the same labels and taken the union of all the edges. If v_1 and v_2 are merged to make vertex v in the final graph, then all incoming edges to v_1 and v_2 are incoming edges to v, and the same holds for outbound edges also. For the vertices with label 3, we have the incoming edges $2 \rightarrow 3$ and $1 \rightarrow 3$ in the merged DAG, and also the outgoing edges $3 \rightarrow 4$ and $3 \rightarrow 5$. When we merge DAGs, we wish to merge only one vertex from one DAG with at most one vertex from the other DAG for the sake of simplicity. Secondly, in Figure 1(c) we don't merge the vertices with label 4 because it would lead to a cycle. If a vertex cannot be merged, then it still needs to appear in the final graph. For example, vertex 4 cannot be merged, hence both of its instances appear. One instance is for the DAG in Figure 1(a) and another instance is for Figure 1(b). We wish to merge the maximum number of vertices without creating a cycle.

This problem has interesting applications. A workflow scheduler can take two DAGs G_1 and G_2 and merge them into one DAG G_f in the way we described. If two nodes v_1 and v_2 are being merged into v_f in G_f, then v_1 and v_2 could be made to run on the same machine. Since they have the same label(run the same program/function), they can take advantage of caching and paging on the machine. We can thus use this merging scheme for speeding up computation by making similar jobs run on the same machine. Similar jobs can constructively interfere and speed each other up. Finally, we wish to ensure that the final graph is a DAG to ensure the simplicity of analyses. If G_f had cycles it would be much more difficult to implement and prove the correctness of distributed algorithms. This problem is applicable to work-sharing techniques in other areas of service science like scheduling, databases, and compilers.

2.2 M-LCS

We define the $M - LCS$ problem as follows. Given a DAG G_1, and a DAG G_2, consider all the topological orderings of G_1, and G_2. A topological ordering is defined as an enumeration of all the vertices in a sequence such that there is no edge from vertex u to v, when u succeeds v in the sequence. For the DAG in Figure 1(a) 1234 is a valid ordering. For the DAG in Figure 1(c), 142345 is a valid ordering. Please note that this ordering is not unique and the number of such enumerations is exponential in the worst case. The M-LCS problem considers all pairs of topological orderings between two DAGs, finds the LCS between each of them, and then reports the LCS that has the maximum length out of these. The $M - LCS$ solution for Figure 1 is as follows. The maximal LCS corresponds to the topological orderings: 1234 and 14235 for the two DAGs. The LCS of these sequences is 123. Please note that Figure 1(c) merges the nodes 1, 2, and 3. We prove in Section 3 that the DAG merge problem is a dual of the $M - LCS$ problem.

2.3 Relevance of M-LCS and Dag-Merge to Other Problems

Both of these problems are different from the traditional higher dimensional LCS problems. In the traditional version, treeLCS always yields a tree or a forest. However, in our case M-LCS yields a DAG. We can thus define our version of generalized LCS as the optimal solution of the M-LCS problem. It has several interesting applications.

1. Solution of the DAG-Merge problem for distributed workflow scheduling and database query optimization.

2. Nowadays, RNA sequences are typically expressed as partially ordered graphs [3, 13]. [3] talks about the problem of finding the longest sequence that satisfies both the orders. This is an instance of the M-LCS problem.

3. Let us consider M-LCS between a DAG and a chain. This problem has extensive use in pattern matching, noise filtering, and XML document retrieval. Let us consider a sequence of symbols. This can be a set of RNA base pairs, signal values, or a binary XML document stream. There can be some noise in this sequence in the form of extra symbols. This often happens in genomics [12] and in XML document transmission [6]. If

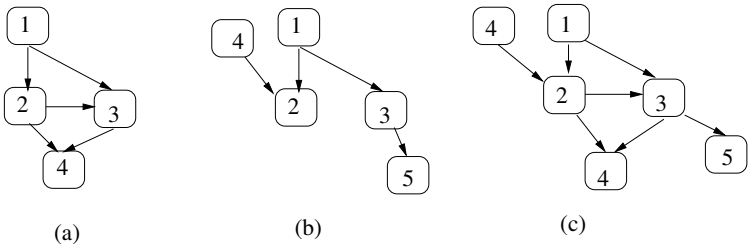

Figure 1. Merging two DAGs

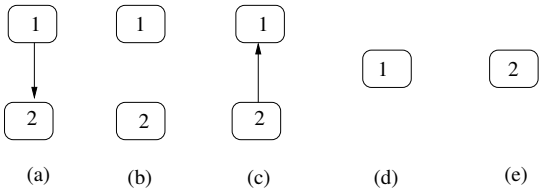

Figure 2. Example DAGs

$Graph_1$	$Graph_2$	Solution
Chain	Chain	Traditional LCS
Tree	Chain	$O(n^2)$ (this paper)
Tree	Tree	???
Dag	Chain	???
Dag	Tree	???
Dag	Dag	???

Table 1. Taxonomy of M-LCS problems

we want to filter the noise out, then we need to find the largest subsequence that obeys a certain partial order. The partial order can be a DAG like that of an XML schema [6]. We thus have an instance of the M-LCS problem.

2.4 Taxonomy of Problems

To the best of our knowledge there are no solutions or NP completeness proofs for the DagMerge and M-LCS problems in prior work. This is the first work to exclusively look at them. Table 1 shows the different problems in this framework in increasing order of perceived difficulty. In this table we assume that the vertices are unique. If there are two chains, then it is the traditional LCS problem. We find an algorithm for the M-LCS solution for a Tree and a Chain in this paper in Section 4. If the vertices are non-unique then we don't have a solution yet. The rest of the problems are as yet unsolved.

Our set of problems are very different from the traditional LCS problems for higher dimensional structures [15, 1]. All of them are duals of the edit distance problem and our problems aren't. Let us merge the graphs in Figure 2(a) and 2(b). The optimal solution according to M-LCS is Figure 2(a). However, the optimal solution in edit-distance based node deletion is either the graph in Figure 2(d) or (e). If we allow node relabeling or edge deletion then the merged Dag of the graphs in (a) and (c), would be either of the two graphs. However, the solution according to M-LCS is just a single node (either (d) or (e)). Hence, we see that these are a totally different class of problems.

3 DAG-Merge is the dual of M-LCS

3.1 Problem Definitions

We start out by defining both the problems formally.

DAG-Merge Problem : Let there be two labeled directed acyclic graphs, G_1 and G_2. We need to produce another DAG $G_f = G_1 \oplus G_2$, where G_f satisfies the following properties:

Properties of G_f

1. Let us define a one to one, onto function $f_{map} : V(G_1) \cup V(G_2) \rightarrow V(G_f)$.
2. Every vertex in G_1 and G_2 is mapped to exactly one vertex in G_f with the same label.
3. Every vertex in G_f is mapped to at most one vertex in G_1, and at most one vertex in G_2, with the same label. No vertex in G_f is unmapped.
4. If there is an edge (u, v) in G_1 or in G_2, then there is an edge between $f_{map}(u)$ and $f_{map}(v)$ in G_f. G_f doesn't contain any other extra edges.

The problem is to minimize $|V(G_f)|$, where G_f is a directed acyclic graph. This basically means that we have to maximize the number of merged vertices.

M-LCS Problem : Given two DAGs G_1, and G_2, find the longest LCS between all pairs of topological orderings.

Definition: Merging two vertices Let there be a vertex $v_1 \in V(G_1)$ and $v_2 \in V(G_2)$. If we map both the vertices to a vertex v in G_f such that $f_{map}(v_1) = v$ and $f_{map}(v_2) = v$, then we *merge* the two vertices v_1 and v_2.

Definition: Merged Dag Let us call G_f, which is obtained by the operation $G_f = G_1 \oplus G_2$, a *Merged Dag*.

Definition: Common Subsequence Graph Let us topologically order the vertices of G_1 and G_2 (see Figure 3). Figure 3(a) and Figure 3(b) show two DAGs with labeled vertices. Figure 3(c) shows the topological order. We note that the topological order shows the labels of the vertices. Let us now find a common subsequence. One such common subsequence is shown in Figure 3(c). Here we merge three pairs of nodes with the same label. The final graph,

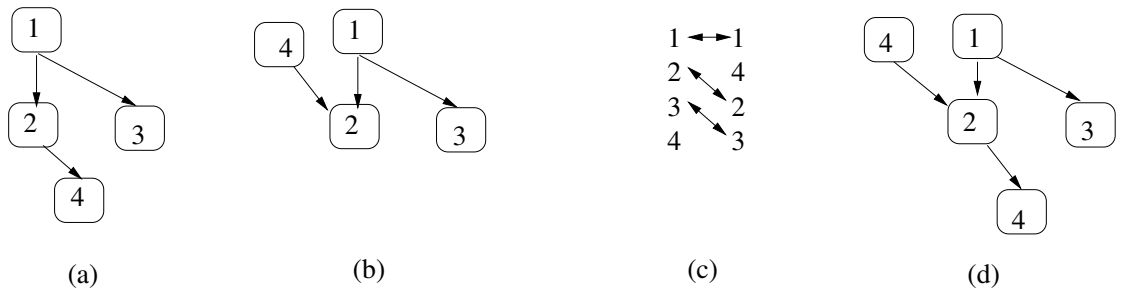

Figure 3. Common Subsequence Graph

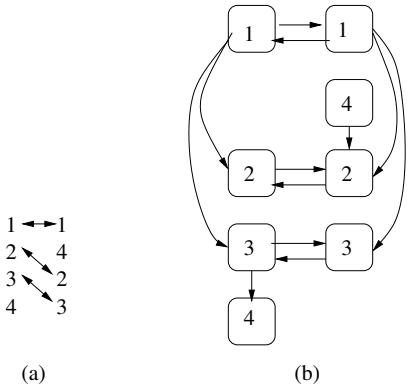

Figure 4. Ladder Graph

G_f, is shown in Figure 3(d). We call such a graph that is created by merging subsequences in the topological order, a *common subsequence graph*.

3.2 Proofs

Theorem 3.1 *Any common subsequence graph is a merged DAG.*

Proof: Let us consider two DAGs G_1, and G_2. Let G_f be any common subsequence graph composed out of G_1, and G_2. It satisfies all the properties of G_f defined in Section 3.1 by construction. We need to prove that it is acyclic.

Let us assume to the contrary that G_f has a cycle. Let it have the nodes $v_1...v_n$. Let us create a ladder shaped graph(Figure 4(b)) as follows. Let us consider Figure 4(a) where both the graphs are arranged in topological order.

Let us draw each of the graphs in a linear fashion vertically arranged in topological order. Further let us add a pair of directed edges between two vertices that are a part of the common subsequence, and will be merged with each other. Let us draw them at the same level. We note that if there is an edge from vertex u to v, and they are part of the same graph, then one needs to be below the other. Secondly, if vertices u, and v are merged with each other, then they need to be at the same level.

Now, let us consider any traversal of vertices in G_f. If we are going from vertex u to v. Then v should be below u in a ladder graph. This is because there must be an edge from u to v in either G_1 or G_2. In either case, we will only descend in the ladder graph. Extending the idea, we observe that any sequence of traversals will only make us descend the ladder graph. Sometimes if a vertex is mapped, we might have to take a horizontal edge and move to the other graph. However, this does not change the level.

Hence, v_n needs to be below v_1 in the ladder graph. Since v_n will have an edge to v_1 if a cycle exists in G_f, v_1 should be below v_n. There is a contradiction. Hence, a cycle cannot exist, and the premise of the theorem is proved.

Lemma 3.1 *Let us consider a subsequence $U = u_1...u_n$, where $u_i \in V(G), 1 \leq i \leq n$, and is topologically ordered in a directed acyclic graph G. There must be a topological ordering of G, which has $u_1...u_n$ as a subsequence.*

Proof : Let $v \in (V(G) - U)$. Let us try to place v in the sequence $u_1...u_n$ such that the final sequence is topologically ordered. We will always find such a slot. If there is no such slot then it means that there is a cycle in the union of U and v. This is not the case.

Let us remove v from $V(G) - U$, and proceed likewise removing a vertex from the set at each step and inserting it into the topological sequence consisting of $u_1...u_n$ and all previously inserted vertices. The same argument holds, and we will always find a slot in this sequence to insert the new vertex. Proceeding in this manner we will have a topologically ordered sequence at the end consisting of all the vertices in $V(G)$. We note that $u_1...u_n$ is a subsequence of this sequence.

Theorem 3.2 *Any merged DAG is a common subsequence graph.*

Proof : Let us consider a merged DAG, $G_f = G_1 \oplus G_2$. We need to prove that a common subsequence exists between

any two topological sorts of G_1, and G_2, such that G_f is a common subsequence DAG.

Let us topologically sort G_f. Let us name all the merged vertices as $v_1...v_n$ in topological order. We note that $\forall i, j, j > i$ there is no path from v_j to v_i. Now, each vertex in this set corresponds to a vertex in G_1 and a vertex in G_2. These vertex sets need to be topologically ordered. Let us assume that this is not the case. Without any loss of generality let us assume that G_1 breaks this order. Let $v_1...v_n$ correspond to $u_1...u_n$ in G_1. There has to be a path from u_j to u_i where $j > i$ to break the topological ordering. In G_f, there has to be a path from v_j to v_i also since we don't delete any edges in G_f. Hence, we prove by contradiction, that the corresponding vertex sets $u_1...u_n$ need to be topologically ordered.

Now, by Lemma 3.1 there exists a topological ordering of G_1, which has $u_1...u_n$ as a subsequence. The same holds for G_2. Hence, we have topological orderings for G_1 and G_2, which have $v_1...v_n$ as a common subsequence. Thus G_f is a common subsequence graph.

Theorem 3.3 *DAG-Merge is the dual of M-LCS*

Proof : Let us consider DAG $G_{fmin} = G_1 \oplus G_2$, which minimizes $|V(G_f)|$. According to Theorem 3.2, G_{fmin} is a common subsequence graph. If it does not correspond to the longest common subsequence(LCS), let G'_{fmin} be the common subsequence graph corresponding to the LCS. By Theorem 3.1, G'_{fmin} is a merged DAG, which will have strictly lesser merged vertices than G_{fmin}. This is a contradiction. Hence, G_{fmin} must be created out of the LCS.

Let us consider a common subsequence graph created out of the LCS of all combinations of topological orderings. Let it be G_{LCS}. By Theorem 3.1, it is a merged DAG. We can prove that $|V(G_{LCS})| = |V(G_{fmin})|$ by contradiction using a logic similar to the former case.

4 M-LCS Between a Tree and a Chain

In this section, we outline an algorithm to find the M-LCS solution between a Tree and a Chain (sequence of vertices) with unique vertices. Let us consider a tree T and a chain C, with m and n vertices respectively. Let the vertices in the Tree be arranged in topological order starting with the root. Equation 1 shows the dynamic programming formulation.

Let LCS(T(i), C(j)) be the LCS between the subtree rooted at $T[i]$ and $C[j...n]$. Let $left(T(i))$ be the left child of $T(i)$, and $right(T(i))$ be the right subchild. We have:

Since the vertices are unique, the subsequences in each subchild of any node in the tree are disjoint. Hence, the LCS is the sum of the LCS for each subchild. The rest of the proof is on the same lines as that of the classic LCS algorithm [7]. This algorithm has $O(mn)$ time complexity.

5 Evaluation Setup

Given a set of workflows, we wish to evaluate the degree of compression that can be achieved by combining workflows. Secondly, we wish to find the quality of the output and time taken. We thus require a random workflow generator, and an algorithm to combine workflows based on the results presented in Section 3.

5.1 Random Workflow Generator

We studied scientific workflows from several sources (major corporation and academic papers). We focussed mainly on parameter sweep applications, which were very common applications. We observed that most workflows have a similar structure. Most of the nodes have a degree less than 10. Secondly vertices are connected to other vertices, which are not very far away. Based on this, we constructed a random workflow generator. This generator creates workflows with nodes that have a bounded degree. Given this constraint, the number of outbound edges follows a normal distribution. Secondly, vertices are connected to other vertices within a certain window. This basically means that we avoid vertices that are connected to vertices that are far away in a planar embedding of the workflow. Based on observations, we set the size of the window to three times the degree. We compared the statistical properties of the DAGs generated to that of a representative set of workflow DAGs. They matched within a 10% interval.

5.2 Implementation of Algorithms

We look at two kinds of algorithms. We first try to solve the DAG merge algorithm using an AI based Simulated Annealing approach. Secondly, we try to solve it using our M-LCS formulation. We coded both the algorithms in C++. They were compiled with -O3 optimization and run on a dual core Intel system running at 2.2 GHz. For every run, we ran it 10 times, removed the outliers, and took an average of the rest of the numbers, for our runtime measurements.

6 Evaluation

6.1 Overview

We evaluate both of our algorithms. We first evaluate our DAG-Merge algorithm. This takes two random workflows, and tries to combine them. We use a simulated annealing based approach to avoid local maxima. The algorithm is as follows. We randomly merge vertices, till there are no free vertices left. This represents a local maxima. After that we unmerge a random number of vertices. We continue the same process again. However, we found that this

$$LCS(T(i), C(j)) = \begin{cases} T(i) = C(j) & LCS(left(T(i)), C(j+1)) + LCS(right(T(i)), C(j+1)) + 1 \\ T(i) \neq C(j) & max(LCS(T(i), C(j+1)), (LCS(left(T(i)), C(j)) + LCS(right(T(i)), C(j)))) \end{cases} \quad (1)$$

approach was also not sufficient. At some point, we unmerge all the vertices and start from scratch. We continue this process for several hundred iterations, and report the best solution. The asymptotic time complexity per iteration is roughly $O(h(V+E))$. Here h is the number of matched vertices. For matching each vertex, we need to check if there is a cycle by doing depth first search. Hence, we need to multiply h by the asymptotic time complexity of doing a depth first search, which is $\theta(V+E)$. There are also other operations like finding the next free pair of vertices to merge. However, we assume that these have $O(1)$ complexity by implementing proper hash structures.

We secondly, evaluate the M-LCS algorithm. This algorithm is much simpler to evaluate with much less code. We construct a random topological ordering of DAG_1 and DAG_2. We subsequently, find the longest common subsequence between these two orderings. Assuming that both the DAGs have the same number of nodes, the complexity of LCS is $O(V^2)$.

Based on the asymptotic complexity, it is not very clear which algorithm is faster. We leave that to our Monte Carlo simulations.

6.2 Results

6.2.1 Vertices Merged vs Number of Iterations

Our first experiment is to find the number of iterations it takes for each algorithm to converge. We consider two random DAGs with 40 vertices. The nodes are randomly labelled following the constraints described in Section 5, with each vertex having a unique label in a DAG. We limit the number of outbound edges to 2 in Figure 5, 5 in Figure 6, and 10 in Figure 7.

We observe two interesting features. The solution converges by roughly 20 iterations. We were not able to significantly improve the quality of the solution after 20 iterations. The second thing is that for smaller degrees, DAG-Merge is much better than M-LCS. After the maximum degree exceeds 3 or 4, M-LCS is a much better solution. We observe that for larger degrees, there are more edges, and the graph is more connected. Probably, DAG-Merge gets stuck in local minima. M-LCS does not suffer from this problem. Every iteration takes it to different corners of the parameter space, thus yielding a better solution.

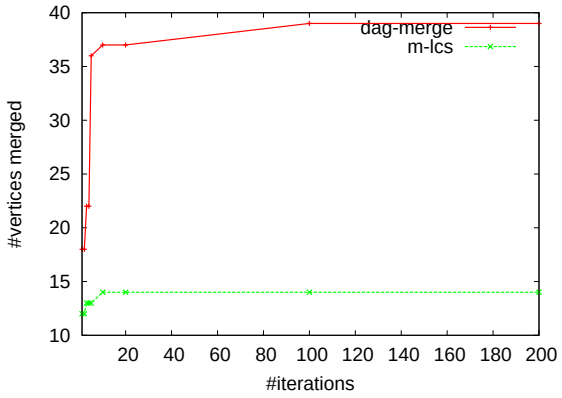

Figure 5. matched vertices vs iterations for maximum degree 2

6.2.2 Vertices Merged vs Maximum Degree

Since we know that about 20 iterations are sufficient, we now look at varying the degree. Figure 8 shows the results. We see that only when the degree is 2, DAG-Merge is better than M-LCS. After that, M-LCS is consistently and significantly better than DAG-Merge. We purposely don't collect data for the case when the degree is 1, because such kind of workflows are not found in practice.

6.2.3 Vertices Merged vs Number of Labels

Now, keeping the degree fixed, and the number of iterations fixed, let us try to modify the number of labels. We observed in our empirical analyses that typically 2,3, and sometimes more nodes, can have the same label. For this experiment, we keep the degree fixed at 5, and change the number of labels that can be assigned. We observe in Figure 9 that if the number of labels is less, then more vertices can be matched. As we increase the number of labels, the number of matched vertices tends to decrease. In Figure 9 we show the results for different values of maximum degrees. The results are inconclusive.

6.2.4 Percentage of Nodes Merged vs Number of Nodes

In this experiment, we keep the maximum degree fixed at 5, and change the number of nodes. The number of labels is equal to the number of nodes divided by 3. Figure 10 shows the results. It is easy to conclude that DAG-Merge does not perform very well. However, we observe that uptil 100

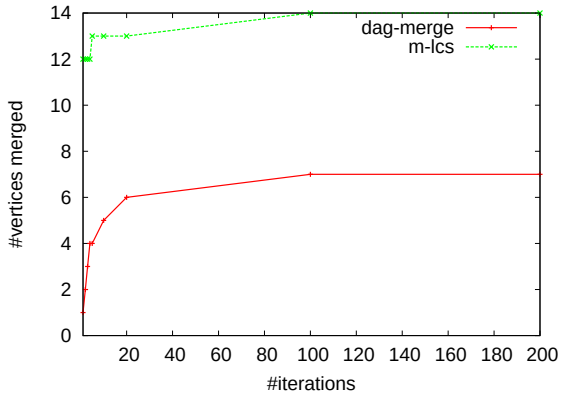

Figure 6. matched vertices vs iterations for maximum degree 5

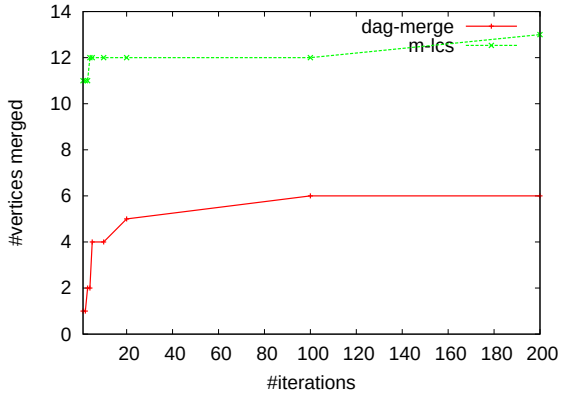

Figure 7. matched vertices vs iterations for maximum degree 10

nodes, we are able to merge around 30% of the nodes. This is a sizeable compression. Less than 50 nodes, we are able to compress about 50% of the nodes, which is even better. Between 100 to 500, the percentage decreases from 30 to about 11.

6.2.5 Time Taken

Lastly, we evaluate a very important criterion namely the wall clock time for the M-LCS algorithm. Figure 11 plots the time taken in milliseconds vs the number of nodes. Undoubtedly, M-LCS takes more time than DAG-Merge. It also scales super linearly. However, the overhead is still around half a second for 500 nodes, which is a very big number for a practical workflow.

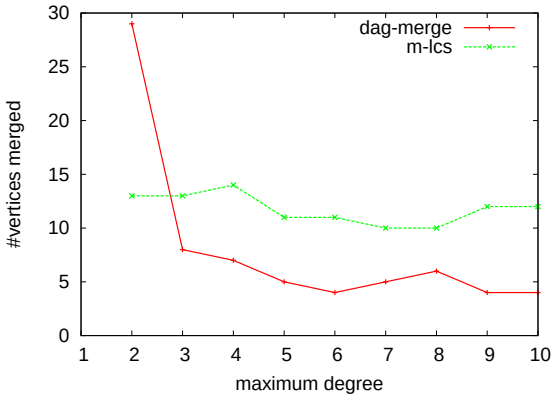

Figure 8. Varying the maximum degree of a vertex

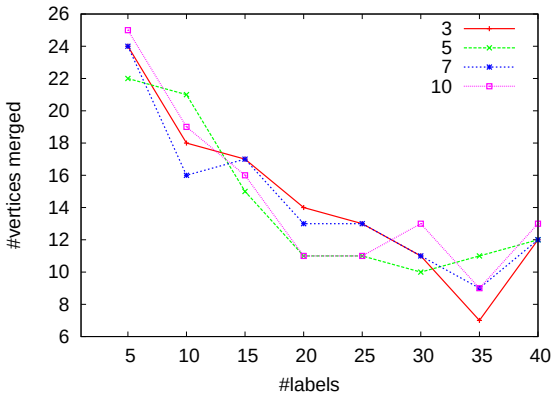

Figure 9. Varying the number of labels

7 Related Work

To the best of our knowledge this is the first work that explicitly looks at the problem of combining two workflows. The M-LCS and Dag-Merge problems are also novel.

[5] talks about grid scheduling algorithms using peer to peer systems as the framework. They define a task group as a set of tasks that use the same files. They are scheduled on the same set of machines to maximize temporal locality. The SMARTS system [8] looks at a single workflow, and tries to find nodes that do similar computations. They are scheduled on the same resource to increase temporal locality.

On the theoretical side, LCS(Longest Common Subsequence) is a classic problem described in various textbooks [7]. Different variants of LCS have been studied in [2]. It proposes different algorithms, which are more efficient than the classic $O(mn)$ time dynamic programming algorithm in terms of time and space under various con-

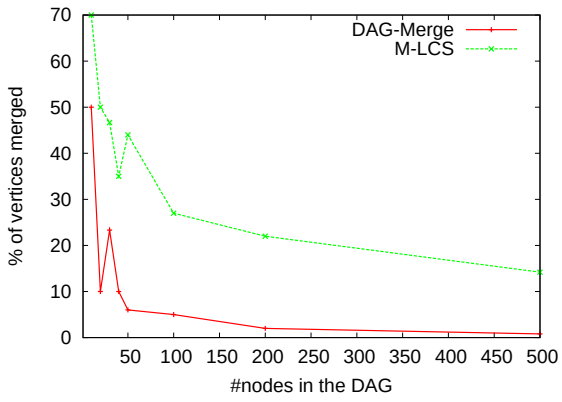

Figure 10. Varying the number of nodes

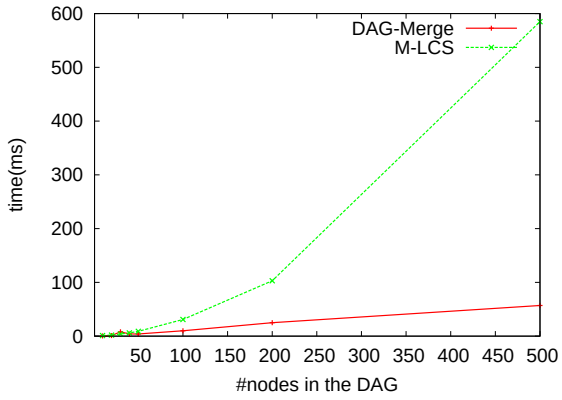

Figure 11. Time taken (ms)

straints. [10, 11] extend the problem to the LCS of two strings where each literal can have a set of values. This is solved by a dynamic programming algorithm.

[15] tries to generalize the problem to find the LCS of two trees. Here the dual of the LCS is a node deletion based edit distance metric. A later approach [1] extends the idea to an LCS of two matrices and an LCS of two forests. It proves the problem to be NP-Complete in both the cases.

8 Conclusion

In this paper we looked at combining two workflows, and compressing them by eliminating redundancy. After a preliminary theoretical investigation, we arrive at two problems namely DAG-Merge, and M-LCS, which were proved to be duals of each other. The M-LCS problem turned out to very important in other fields like XML document transmission and genomics. We subsequently, evaluated practical implementations of both the algorithms. We observed that DAG-Merge was suitable for only a very limited set of workflows. However, M-LCS proved to be a very versatile algorithm. It was able to compress around 40-50% of the vertices for random workflows, which is a very significant reduction in the size of the workflows.

References

[1] A. Amir, T. Hartman, O. Kapah, B. R. Shalom, and D. Tsur. Generalized LCS. *Theor. Comput. Sci.*, 409(3):438–449, 2008.

[2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[3] E. Bloomberg. Partial-order alignment of RNA structures. Master's thesis, Brown University, 2005.

[4] C. Branden and J. Tooze. *Introduction to Protein Structure*. Garland Publishing, New York, 1999.

[5] C. Briquet, X. Dalem, S. Jodogne, and P.-A. de Marneffe. Scheduling data-intensive bags of tasks in p2p grids with bittorrent-enabled data distribution. In *UPGRADE '07: Proceedings of the second workshop on Use of P2P, GRID and agents for the development of content networks*, pages 39–48, 2007.

[6] H. Chen and P. Mohapatra. A context-aware html/xml document transmission process for mobile wireless clients. *World Wide Web*, 8(4):439–461, 2005.

[7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 2003.

[8] S. V. et. al. Smarts: Exploiting temporal locality and parallelism through vertical execution. In *ICS*.

[9] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3(2):119–153, 1995.

[10] D. S. Hirschberg and L. L. Larmore. The set LCS problem. *Algorithmica*, 2(1-4):91–95, Nov 1987.

[11] D. S. Hirschberg and L. L. Larmore. The set-set LCS problem. *Algorithmica*, 4(1-4):91–95, June 1989.

[12] http://en.wikipedia.org/wiki/Junk_DNA. Junk dna. Technical report, wikipedia, 2008.

[13] C. Lee. Generating consensus sequences from partial order multiple sequence alignment graphs. *Bioinformatics*, 19(8):999–1008, May 2003.

[14] N. Mandal, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Integrating existing scientific workflow systems: the kepler/pegasus example. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 21–28, 2007.

[15] S. Mozes, D. Tsur, O. Weimann, and M. Ziv-Ukelson. Fast algorithms for computing tree LCS. In *Proceedings of the 19th annual symposium on Combinatorial Pattern Matching*, pages 230–243, 2008.