LDAG: A New Model for Grid Workflow Applications

GUIPING WANG[1] AND YAN WANG[2]
[1]School of Information
Zhejiang University of Finance & Economics
18 Xueyuan Street, Hangzhou, Zhejiang
CHINA 310018
w_guiping@163.com
[2]School of Information
Zhejiang University of Finance & Economics
18 Xueyuan Street, Hangzhou, Zhejiang
CHINA 310018
wangyan@zufe.edu.cn

Abstract: - Grid workflow and its application are one of main focuses of Grid Computing. Due to data or control dependencies between tasks and the requirement of no directed circuit, Directed Acyclic Graph (DAG) is a natural model for Grid workflow, and has been extensively used in Grid workflow modeling. For some workflow applications, there may exist another requirement that each task should be accomplished at an expected stage, that is, at a given level. In this paper, we discuss such workflow applications in depth, and propose a new DAG model, which we called LDAG. In LDAG, each node possesses a level. Several cases of the level of nodes are discussed in detail. For a reasonable one of these cases, we propose the topological sorting algorithm. The algorithm consists of two phases, namely *Level Adjusting* and *Topological Sorting*. We discuss some relevant problems, such as choice of stack or queue, the determination of directed circuit, complexity of the algorithm, etc. The experiment and analysis of LDAG and topological sorting algorithm show its correctness and efficiency in modeling grid workflow.

Key-Words: - Directed Acyclic Graph (DAG), LDAG; Grid workflow; Level; Topologic sorting; Directed circuit

1 Introduction

Grid computing [1] is considered as a cornerstone of next generation distributed computing that coordinates large-scale resource sharing and problem solving in dynamic, multi-institutional virtual organizations [2]. Based on Grid, a sophisticated job can be decomposed into a great amount of atomic tasks and accomplished with distributed computing resources, and thus achieve minimum job accomplishing time and high resource utilization efficiency.

Grid workflow is defined as the orchestration of a set of atomic tasks processed at distributed resources in a well-defined order to accomplish a large and sophisticated goal [2]. Currently, Directed Acyclic Graph (DAG), and other models such as UML [3, 4] and Petri Net [5, 6], have been extensively used in scientific computational workflow modeling, especially in large-scale data-, computing - or instrumentation-intensive Grid applications, such as high-energy physics, geophysics, astronomy, medical image processing, and bio-informatics.

For some workflow applications, besides data or control dependencies between atomic tasks, there may exist another requirement, expecting each atomic task be accomplished at an expected stage. For example, a large and sophisticated job needs to be partitioned into several stages due to computing resources insufficiency or inherently containing several stages for the job. Therefore each atomic task corresponds to a given stage, which we called Level.

In this paper, we discuss such workflow applications in depth and propose a new DAG model, which we called LDAG. In LDAG, each task possesses a level. The level of a task may be inherent stage of the whole job or expected stage to be executed. Firstly we analyze several cases of the level of tasks. And then based on original topological sorting algorithm [7], we propose the topological sorting algorithm for a reasonable one of

these cases. The algorithm consists of two phases, namely *Level Adjusting* and *Topological Sorting*.

The remainder of this paper is organized as follows. Section 2 gives an overview of related works. Section 3 presents the formal definitions and preliminaries of DAG workflow. Section 4 presents LDAG, a new DAG model with all vertices possessing a level, and discusses several cases of the level of vertices. In section 5, we propose the topological sorting algorithm for one reasonable case of LDAG workflow, and discuss some relevant problems. Experiment and analysis of LDAG model and the algorithm is given in section 6. Finally, we conclude this paper and give some future works in section 7.

2 Related Work

The applications of DAG in Grid workflow mainly converge in two aspects, that is, workflow modeling and workflow scheduling (optimizing inclusively).

Workflow modeling is the first step before further workflow processing, such as scheduling and optimizing. Due to inter-task data or control dependencies between atomic tasks, and the requirement of no directed circuit, DAG is a natural model for Grid workflow, in which nodes represent atomic computing tasks and directed arcs or edges represent inter-task data or control dependencies between tasks. For example, in [8] the nodes of DAG represent the computing tasks, while the dependences between tasks are materialized by file: a task produces a file that is necessary for the processing of some other task. An e-Protein project of the London e-Science Center in [9] presents a complex Grid infrastructure of distributed computational resources with different databases and specialized analysis software that may not be deployed on every resource. A protein annotation workflow is given, as shown in Fig. 1. In this DAG, the nodes represent components of the project, while arcs describe the direction of data flow between components, that is, data dependencies between tasks.

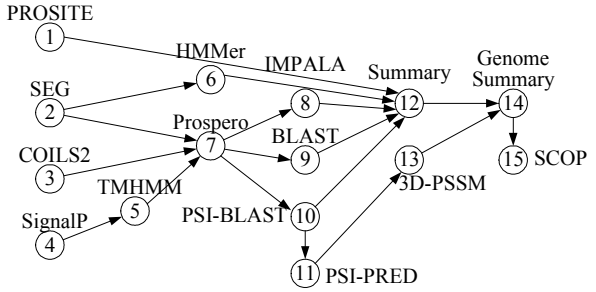

Fig. 1 A protein annotation workflow [9]

Based on DAG, some well-known Grid workflow management systems such as DAGMap [2], GridAnt [10], DAGMan [11], GridLab [12] and PROGRESS [12] have been implemented and applied in Globus and Condor projects.

For DAG model optimizing, the study in [13] shows that a set of workflows modeled by DAG can be combined and compressed in order to eliminate redundancy in workflow and minimize computation.

Based on DAG, some more sophisticated models are studied in order to meet special system requirements. For example, Yuan [14] studies workflow applications described by DAG with deadline constraints, and proposes an effective and efficient heuristic called DET. Tian [15] studies such workflow applications that each processing of a workflow needs to be finished within its deadline, and proposes a critical-region based scheduling algorithm.

Except for DAG, UML [3, 4], Petri nets [5, 6], and even their combination can also be adopted in workflow modeling for special applications. DAG only contains sequence and parallel workflow patterns. However choice and iteration workflow patterns are also required in different fields of science such as bio-informatics, meteorology, etc. Therefore, a workflow specification model, which uses directed graphs and Petri nets, is designed in [16], where directed graphs are used at user-level and Petri nets are used as underlying workflow model. In [12], DAG and Petri nets based approaches are compared in GridLab and PROGRESS projects.

DAG workflow job scheduling in a Grid environment determines how to map all atomic tasks to a bounded number of distributed computing resources [17]. Various workflow scheduling algorithms are discussed from different points of view, such as static vs. dynamic policies, objective functions, applications models, quality of service (QoS) constraints, strategies dealing with dynamic behavior of resources, and so on [15]. The studies in [18] give a state-of-the-art taxonomy of grid scheduling algorithms. Besides basic scheduling algorithms, such as list and group scheduling algorithms, some more effective and efficient heuristic have been studied for some special workflow applications, such as DAGMap [2], DET (Deadline Early Tree)[14], etc.

Some more sophisticated scheduling heuristics that adopt more intelligent strategies are studied in literatures. In [19], a decentralized scheduling algorithm based on genetic algorithms for the problem of DAG scheduling is proposed.

Almost all the literatures about DAG and its applications assume grid workflow being a DAG, and discuss such issues as workflow scheduling and optimizing based on this assumption. Few literatures work on how to model the workflow and determine whether the model is a DAG.

3 Model and Preliminaries

To illustrate workflow job modeling problem clearly, we present the formal definitions for DAG workflow job and topological sorting in this section.

3.1 DAG Grid Workflow

Definition 1(DAG Grid Workflow)[2]: A DAG Grid workflow job can be represented by a directed acyclic graph G(T, E), where T = { t_1, t_2, …, t_N } is the collection of atomic tasks (N is the total number of tasks), and E is the collection of arcs or edges indicating the dependency and precedence constraint between tasks.

For example, in the DAG shown in Fig. 2(a), T = {A, B, C, D, E}, E = {<A, B>, <A, C>, <A, D>, <B, C>, <B, E>, <C, E>, <D, E>}.

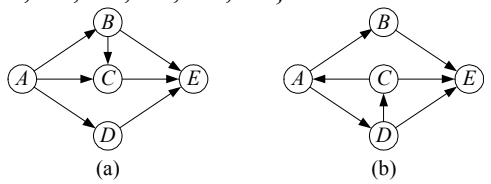

Fig. 2 Two workflow models

Definition 2(job, workflow, vertex, node, task, atomic task, activity): In this paper, job and workflow mean the whole workflow job. The latter 5 terms are synonyms, which mean atomic task in workflow job.

Based on grid, a workflow job can be decomposed into a great amount of atomic tasks and accomplished with distributed computing resources.

The workflow job shown in Fig. 2(a) is decomposed into 4 atomic tasks, that is, A, B, C, D and E.

Definition 3(edge, arc): The directed edge<u, v> (from u to v)∈ E in a DAG may represent data or control dependencies between task u and v.

For example, the execution of task v requires data or file produced by that of task u. It can also represent precedence constraint between task u and v, that is, task u should be executed before task v.

Definition 4(start vertex, end vertex): In a DAG, if <u, v>∈ E, then we call vertex u is the start vertex of edge <u, v>, while vertex v is the end vertex of edge <u, v>.

Definition 5(predecessor and successor): In a DAG, if <u, v>∈ E, then the task u is the direct predecessor of task v, and task v is the direct successor of task u. If there exists a path from task u to task v, that is, <u, u1, u2, …, un, v>, then the task u is the predecessor of task v, and task v is the successor of task u.

For example, in a DAG shown in Fig. 2(a), E has 3 direct predecessors, that is, B, C, and D. A has no direct predecessor. A is a predecessor of E. B has 2 direct successor, that is, C and E, while E has no direct successor, etc.

Notice that, a task can start to be executed only that all of its predecessor tasks have been accomplished. Similarly, only after the execution of a task, all of its successor tasks can start to be executed.

Definition 6(entry task and exit task): In a given DAG graph, a task without any predecessors is called an entry task, and a task without any successors is an exit task.

For example, in a DAG shown in Fig. 2(a), A is an entry task, and E is an exit task.

Entry tasks are entries of the whole workflow job, and exit tasks are exits of the whole workflow job.

Definition 7(dummy node): If there is more than one entry/exit task, a dummy entry/exit task can be added. At the same time, the dummy entry task should be connected to each original entry task with edges. Similarly, each original exit task should be connected to the dummy exit task with edges. This can ensure that there are only one single-entry task (denoted as tentry) and one single-exit task (denoted as texit) in a DAG workflow job.

For example, in Fig. 1, there are 4 entry tasks and 1 exit task, so we add a dummy entry task (numbered as 1), as shown in Fig. 3. The added edges are denoted as dashed lines. Attention: in Fig. 3, except for the dummy entry task, the i-th task corresponds to the (i-1)th task in Fig. 1.

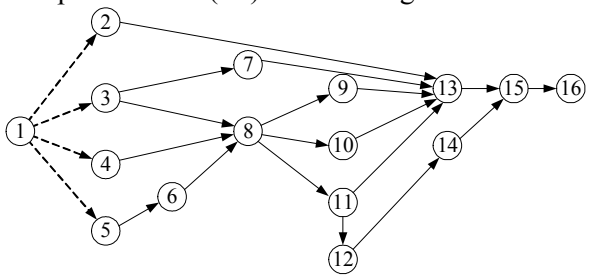

Fig. 3 DAG model of the workflow application sample in Fig 1.

3.2 Topological sorting

After modeling a workflow into a DAG, the foremost job is to judge whether there exists a

directed circuit, that is, whether the DAG is reasonable.

Definition 8(directed circuit): In a given DAG graph, if a directed path, <u, u1, u2, …, un, v>, starts from and ends in a same vertex, that is, u = v, then we call this path a directed circuit.

A DAG is an unreasonable one if there exists a directed circuit, which means one task is a predecessor of itself. If there exists a circuit in a DAG workflow, then it cannot be scheduled and executed in Grid environment.

For example, in Fig. 2(b), A is the predecessor of D, D is the predecessor of C, C is the predecessor of A, and then we get a conclusion that A is the predecessor of A, which is not reasonable.

The method of determining directed circuit in a DAG is to construct its topological sorting sequence (topological sequence for short).

Definition 9(topological sequence): A topological sequence of a DAG is a linear ordered sequence of all vertices that satisfy all the predecessor and successor relations in the DAG.

Definition 10(topological sorting): The topological sorting operation of a DAG is to arrange all vertices into a linear ordered sequence and satisfy all the precedence relations.

For example, one of the topological sorting sequences of Fig. 2(a) is ADBCE, which satisfies the condition that an arc always leads form an anterior node to a posterior node. Another topological sequence is ABCDE, while ACBDE is not a topological sequence because the precedence relation of <B, C> is not satisfied in this sequence.

The original topological sorting algorithm described in section 5.1 can solve topological sorting problem for a DAG.

4 LDAG: each vertex possessing a level

4.1 Level

For some real workflow applications, besides data or control dependencies between tasks, there may exist another requirement, expecting each task be accomplished at an expected stage. For example, due to computing resources insufficiency, a large job and sophisticated has to be executed in several stages. Or the job inherently contains several stages. In such cases, the workflow has to be partitioned into several stages, and each task corresponds to a given stage, that is, Level.

Definition 11(Level): In some workflows, each vertex possesses a weight, which we called Level. Level can be the expected stage for each task.

Definition 12(LDAG): In a given DAG, if each vertex possesses a Level, the model is called LDAG in this paper.

For example, A LDAG is shown in Fig. 4, the number beside each vertex represents its Level, that is, the expected stage for each task.

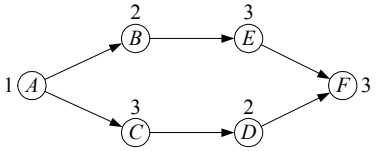

Fig. 4 A LDAG model.

LDAG can be widely used in grid workflow modeling. Consequently, it is important to study this DAG model and relevant problems, such as topological sorting, scheduling and optimizing. In this paper, we focus on the first problem, that is, topological sorting of a LADG.

4.2 Several cases of the level of nodes

There are 3 cases for Level of nodes in LDAG, which listed as follows.

1) The level of each vertex is fixed, that is, the stage of each task cannot be advanced and postponed.

2) Due to computing resources and other reasons, the stages of all vertices are usually expected to be postponed as much as possible under permitting conditions; that is, each task has an expected stage, and the stage can be delayed, but cannot be advanced.

3) Due to the delivery of each task and the time minimization of the whole job, advance of stage is allowed, but postponement is prohibited. That is, each task has an expected stage, and the stage can be advanced but cannot be postponed. At the same time, due to computing resources and other reasons, it is expected that the stage of each task will not be advanced under unnecessary situation.

In addition, it is necessary to explain that, in the same stage, several tasks can be arranged in despite of their precedence constraint. These tasks will be arranged according to their precedence constraint relations.

For the first case, even if there does not exist a directed circuit in a given DAG, it may be impossible to schedule the workflow. For example, in Fig. 4, task C must be executed in stage 3, while task D must be executed in stage 2. It is obviously impossible because that task C is a predecessor of task D, and should be executed before the accomplishment of task D.

For the second case, since it is always possible to postpone all tasks to the last stage, it is meaningless to study this case. For example, in Fig. 4, if the stage of a task can be postpone arbitrarily, all the tasks can be postponed to stage 3 (all the tasks will be executed in turn according to the topological sorting order).

For the third case, it is required that all tasks be executed before the given stage of a task (including the given stage). For example, in Fig. 4, the level of task D is 2, which means task D must be executed before stage 2 inclusively. At the same time, due to computing resources and other reasons, it is expected that all tasks will not be advanced as much as possible. The problem now is how to arrange execution order of all tasks. The order should satisfy not only precedence relations between tasks, but also level constraints, that is, all tasks should be executed before the given stage (inclusively).

Because of the precedence constraints between tasks, not all the tasks can be executed in its given stages. For example, in Fig. 4, task D can be executed in stage 2, but task C cannot be executed in stage 3. The reason is that task C is the predecessor of task D and precedent to D. Thus task C has to be advanced to stage 2, that is, its level has to be advanced to level 2. Note that, task C and D can be arranged in the same stage, and the execution of task C is precedent to that of task D.

For the last case of LDAG, we propose a topological sorting algorithm, which can be used to judge which tasks will be executed in each stage and the execution order of tasks in each stage.

5 Topological Sorting Algorithm for LDAG

5.1 Original Topological Sorting Algorithm for DAG

For the convenience of discussion, we need to introduce another several concepts.

Definition 13(incoming arc, outgoing arc, indegree, outdegree): Incoming arc of vertex u is an arc from another vertex to u. Outgoing arc of vertex u is an arc from u to another vertex. The indegree of vertex u is the number of incoming arcs of u. The outdegree of vertex u is the number of outgoing arcs of u.

Definition 14(zero-indegree vertex): If the indegree of vertex u is zero, then we call u a zero-indegree vertex. A zero-indegree vertex is the one that has no predecessor.

For example, the only one zero-indegree vertex in Fig. 5(a) is A initially. In the process of topological sorting, along with the operation of deleting outgoing edges of a zero-indegree vetex, more zero-indegree vertices will come into being.

When topologically sorting a DAG, it is necessary to establish a stack S to store zero-indegree vertices. The stack can be replaced by a queue, the detail discussion can be found in section 5.3.

The original topological sorting algorithm can be found in [7]. The algorithm consists of three parts: initialization (lines 1-2), topological sorting process (lines 3-12), directed circuit determination (lines 13-14).

The initialization process includes establishing a stack (line 1) and pushing each zero-indegree vertex into S (line 2).

The topological sorting process is a loop procedure. The loop condition is that S is not empty (line 3). At each round, pop top vertex (denoted by u) out of S (line 4) and output it (line 5) firstly. Then check and delete each outgoing edge of u (lines 6-11). When checking an outing edge, the indegree of the end vertex (denoted by v) minus 1 (line 7). If the indegree of v drops to zero (line 8), push v into S (line 9). After checking, delete the outing edge (line 10).

In the last process, if the number of vertices having been output is less than n, then a circuit is determined. Otherwise, the DAG is reasonable.

Algorithm 1(original topological sorting algorithm):

```
1.  establish stack S for zero-indegree vertices;
2.  push each zero-indegree vertex into S;
3.  while( S is not empty )
4.    pop u(top vertex) out of S;
5.    ouput u;
6.    for( each outgoing edge of u )
7.      the indegree of v (end vertex) minus 1;
8.      if( the indegree of v equals to zero)
9.        push v into S;
10.     delete the outgoing edge;
11.   end for
12. end while
13. if( the number of vertices having been output<n)
14.   report: there exist circuit(s) in DAG
```

For example, the topological sorting process of the DAG in Fig. 5(a) is listed as follows. 1) push A into S; 2) pop A out of S, and delete its outgoing edges, push B and D into S in turn; 3) pop D out of S, and delete its outgoing edge; 4) pop B out of S, and delete its outgoing edge, push C into S; 5) pop

C out of S, and delete its outgoing edge, push E into S; 6) pop E out of S.

After topological sorting, we get a vertex sequence, A→ D→ B→ C→ E, as shown in Fig. 5(b).

In Fig. 5(b), all the predecessor/successor relations (denoted by real line) have been reserved. In addition, from vertex D to vertex B, a predecessor/successor relation is added artificially, as shown a dashed line in Fig. 5(b).

Undoubtedly, there may exist more than one topological sorting sequence in a DAG. For example, another two topological sorting sequences of the DAG in Fig. 5(a) are A→ B→ C→ D→ E and A→ B → D→ C→ E.

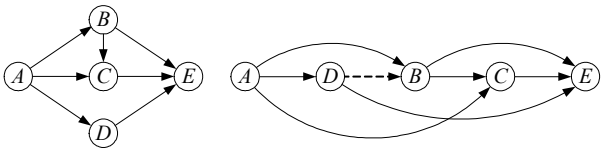

Fig. 5 Topological sorting example

In algorithm 1, if there are some vertices having not been output after topological sorting process (the S is empty, that is, there is no zero-in-degree vertex), then we can report there exist circuit(s) in the DAG.

5.2 Topological Sorting Algorithm for LDAG

For the topological sorting process of the third case of LDAG, the critical step is that, when checking the incoming edges (can be denoted by <u, v>) of each vertex v, if the level of start vertex (denoted by u) is higher than that of end vertex (that is, vertex v), then it is necessary to advance the level of u to L (L is the level of vertex v). It means that the task corresponding to vertex u has to be advanced to stage L. The reason is that, u is a predecessor of v, and u should be executed before v.

When checking the vertices of level L, a queue Q is used to store these vertices. (The queue Q can be replaced by a stack. Its detail discussion can be found in section 5.3.) When topologically sorting the vertices of level L, a stack S is used to store the zero-indegree vertices. (The stack S can also be replaced by a queue, its detail discussion can also be found in section 5.3.)

The main thread of topological sorting algorithm for a LDAG is to check each level from lower to higher. The checking process for level L consists of two phases: *Level Adjusting* and *Topological Sorting*.

The concrete checking process for level L is listed below.

(1) *Level Adjusting* (lines 2-17): Find out all the vertices should be executed in level L, including the vertices whose initial level is L, and the vertices whose level are advanced to L. The method is listed as follows.

a) For each vertex (denoted by u) whose initial level is L, push it into Q (line 4). If the indegree of u is zero (line 5), push u into S at the same time (line 6).

b) Check each vertex (denoted by v) in the queue Q (lines 8-17). It is a while-loop procedure. In each round, pop top vertex (that is, v) out of Q (line 9), and check each incoming arc of v, if the level of start vertex (denoted by w) is higher than L, then advance the level of w to L (line 12), and push w into Q (line 13). If the indegree of w is zero (line 14), push w into S at the same time (line 15).

Note that, if the level of w is lower than L, it means that w has been processed in a lower level and does not need to be considered in level L.

The reason of pushing w into Q is that it is necessary to check each predecessor (denoted by x) of w. And if the level of x is higher than L, it also needs to be advanced to L.

(2) *Topological Sorting* (lines 18-19): Topologically sort all the vertices of level L by virtue of stack S using algorithm 1 (section 5.1).

For all the levels in ascending order, invoke the checking process (*Level Adjusting* and *Topological Sorting*), until all vertices have been output, or a circuit is determined (its detail discussion can be found in section 5.3).

The pseudocode of topological sorting algorithm for LDAG is listed below.

Algorithm 2(topological sorting algorithm: LDAG):

```
1.  for( each level in ascending order )
2.    //Level Adjusting
3.    for( vertex u which initial level is L )        //a)
4.      push u into Q;
5.      if( indegree of u is zero )
6.        push u into S;
7.    end for
8.    while( Q is not empty)                          //b)
9.      pop v (top vertex) out of Q;
10.     for( each incoming arc of v )
11.       if( the level of w (start vertex) > L )
12.         advance level of w to L;
13.         push w into Q;
14.         if( indegree of w is zero )
15.           push w into S;
16.   end for
17.   end while
18.   //Topological Sorting
19.   Topological sort all vertices of level L;
```

20. end for

In addition, if the levels of vertices do not start from 1, or the levels of vertices are not continuous, an array is needed to store all the levels in ascending order. The total topological sorting process will be executed according to the levels stored in the array.

For example, a LDAG is shown in Fig. 6. It consists of 8 atomic tasks and 12 pairs of predecessor/successor relations. The topological sorting process is listed as follows.

(1) Firstly, check the vertices of level 1.

1) Push vertex A and G into queue Q. At the same time, push A into stack S.

2) Pop A out of queue Q. Because A has no incoming arc, there is no further process for A. Pop G out of queue Q. When checking incoming arcs of G, the levels of vertex D and F are advanced to 1, and push into queue Q in turn.

3) Pop D out of queue Q. Pop F out of queue Q, advance the level of vertex C to 1, push C into queue Q.

4) Pop C out of queue Q.

5) Now there is only one vertex, that is, A, in stack S. But by virtue of stack S, C, D, F and G will be pushed into and popped out of stack in turn.

Thus the resulting topological sequence of level 1 is A→ C→ D→ F→ G.

(2) When checking the vertices of level 2, vertex E is push into queue Q. Pop E out of queue Q, and push B into queue Q. After topological sorting, we get a sequence of level 2, that is, B→ E.

(3) Check the vertices of level 3, and get the resulting topological sequence, that is, H.

Therefore, after topological sorting the LDAG in Fig. 6, the tasks that need to be executed in each level are listed below.

Level 1: A→ C→ D→ F→ G;

Level 2: B→ E;

Level 3: H.

At each level, all tasks will be executed according to their topological sorting orders listed above.

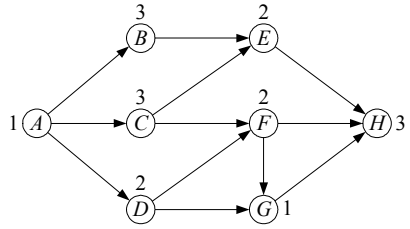

Fig. 6 A topological sorting sample of LDAG.

5.3 Further discussion of Algorithm 2 and complexity analysis

A. Choice of stack or queue

The difference between queue and stack lies in the pop sequence of vertices in queue (or stack). Queue is FIFO (First In and First Out), and stack is LIFO (Last In First Out). Therefore, the adoption of a queue or a stack during the implementation process of an algorithm is determined by whether this pop sequence will affect the correctness of the algorithm. The discussion of choice of queue or stack in algorithm 1 and 2 is listed below.

Algorithm 1: During the implementation process of the algorithm, if there exist more than one zero-indegree vertex, which vertex should be deleted first does not affect the correctness of the algorithm. Therefore, in algorithm 1, the stack S can be replaced by a queue Q to store zero-indegree vertices.

Algorithm 2: When finding out all the vertices of level L, which vertex (and its all incoming arcs) should be checked first does not affect the correctness of the algorithm. Therefore, the queue Q can be replaced by a stack S to store vertices of level L. In addition, the stack S in algorithm 2 can also be replaced by a queue Q, the reason is same to that of Algorithm 1.

B. Determination of directed circuit

In topological sorting algorithm for a LDAG, the determination of directed circuit(s) can be implemented by the method in algorithm 1. The method can be concreted as: in the step (2), if the number of vertices having been output is less than the number of vertices of level L, then the directed circuit(s) is determined. The algorithm also does not need to perform any longer.

For example, there is a directed circuit in a LDAG shown in Fig. 7. The topological sorting process of is listed below.

(1) Firstly, output the only one vertex of level 1, that is, vertex A.

(2) Then find out all the vertices of level 2, that is, B, D, C and E (note: the levels of latter two vertices are advanced to 2). However, after outputting vertex E, there is no zero-indegree vertex in S. Therefore, a directed circuit can be determined. In this example, the directed circuit is: B→ C→ D→ B. Thus, the algorithm's execution can be terminated.

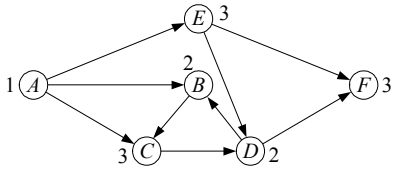

Fig. 7 A LDAG consists of a directed circuit.

C. Complexity analysis of the algorithm 2

Before analysis, we make an assumption that there are n vertices, m directed edges, and L levels initially in a LDAG.

Time complexity: During the topological sorting process in algorithm 2, each vertex is pushed into queue Q one time, and popped out of Q one time. Similarly, each vertex is pushed into and popped out of stack S only one time. Therefore, the processing time required for queue Q and stack S is O(2n). When checking the incoming arcs of each vertex, each incoming arc is scanned one time. When outputting a vertex, each outgoing arc will be deleted. Therefore, the processing time required for processing the incoming and outgoing arcs is O(2m). Consequently, the total time complexity of algorithm 2 is O(2n +2 m).

Space complexity: When implementing algorithm 2, adjacency list is used to store a LDAG. Moreover, incoming arc list and outgoing arc list should be comprised at the same time. The required storage space is O(2n +2m). In addition; the indegree of each vertex should be stored, the required space is O(n). When implementing topological sort algorithm 2, in the worst case, the levels of all vertices are same, the required storage for queue Q to store all the vertices is O(n). And the storage for stack S is not more than O(n). In addition, the storage for L levels is O(L). Consequently, the total space complexity of algorithm 2 is O(5n +2m + L).

6 experiment and analysis

In this section, we will analyze algorithm 2 from two aspects: verifying correctness of the algorithm, and analyzing advance ratio of initial level of nodes during execution of the algorithm.

6.1 verifying correctness of the algorithm

The verification work can be concreted as: inputting a LDAG, and outputting the vertices of each level ultimately. When the vertices are output, they should be listed according to their topological order. If there exists a circuit, then output "directed circuits exist".

A test data set containing 10,000 random test data are generated in the verifying work. The format of each test data consists of three parts. The first part is three integers in one line, N, L and M. $5 \le N \le 26$, $1 \le L \le 10$, $1 \le M \le N*(N-1)/2$, M ensures the ground graph of the LDAG is connected. These three integers mean the number of vertices, the number of levels (level is 1~L), the number of predecessor/successor constraints respectively. The N vertices are represented with anterior N uppercase letters in alphabet. Their sequence number are 1~N. The second part is N integers in one line. The i-th integers Li represents the initial level of the i-th vertex, $1 \le Li \le L$. The third part contains M lines, which describe M pairs of predecessor/successor constraints. Each line contains two uppercase letters (The range of each uppercase letter is within anterior N uppercase letters in alphabet. They are denoted by A and B here), which means vertex A is predecessor of vertex B.

For example, the LDAG shown in Fig. 6 can be formatted as: 8 3 12; 1 3 3 2 2 2 1 3; AB; AC; AD; BE; CE; CF; DF; DG; EH; FG; FH; GH. Each component before a semicolon corresponds a line.

The specification of output content is listed as follows. For each test data, output L lines, with the i-th line representing vertices of i-th level. In the i-th line, firstly output the number of i, indicating the i-th level. A space, a colon and a space are followed. Then if there is no vertex in the i-th level, output 0, otherwise output topological order of vertices in the i-th level.

For the test data shown in Fig. 6, one correct output should be:

1: ACDFG
2: BE
3: H

Notice that, the solution of each test data may be not unique. Therefore a special judge program is written to verify the correctness of the output. We use two methods when judging whether the vertices sequence of each level is in a topological order. We can conclude the output is right only on the condition that both methods judging the vertices sequence are in a topological order.

These 2 methods are listed as follows.

(1) Note the location of each letter in the vertices sequence of each level, and store the locations in a POS array (POS[1] represents the location of letter A, POS[1] represents that of letter B, …). Then judge each direct arc in LDAG whether the location of start vertex is before that of end vertex. As long as there exists one directed arc not satisfying the condition, we can conclude the vertices sequence is not in a topological order.

(2) A double loop is used to determine whether a wrong predecessor/successor constraint can be found in the adjacency matrix of a LDAG. The wrong constraint can be represented with <j, i>, where i represents the i-th letter in the vertices

sequence of each level, and j represents each latter letter of i. If there exists one wrong constraint, we can conclude the vertices sequence is not in a topological order similarly.

When generating a test data set, we can choose whether there exists directed circuit in a LDAG, and get two test data sets. Each data set contains 10,000 random test data. The test result of each data set is listed in Tab. 1, which shows the correctness and high efficiency of algorithm 2. The configuration of the test machine is: Intel Core2 Duo CPU, 2.26GHz; 3G memory.

Tab. 1 Test result of algorithm 2.

	test data set description	run time	result
test data set 1	no directed circuit	563 ms	output is correct
test data set 2	directed circuit is permitted	867 ms	output is correct

6.2 advance ratio of initial level of nodes

In algorithm 2, when checking each incoming edge of vertex v, if the level of start vertex (denoted by u) is higher than that of v, then the level of u will be advanced to L (L is the level of vertex v). After execution of algorithm 2, we can calculate advance ratio: the number of vertices which level are advanced / total number of vertices. Based on the aforementioned two test data sets, the ratios are calculated and listed in Tab. 2.

Tab. 2 change ratio of two test data set.

	test data set description	ratio
test data set 1	no directed circuit	19.25%
test data set 2	directed circuit is permitted	23.67%

Usually the ratio indicates the relation between initial level of vertices and predecessor/successor constraint density between vertices. If there exist two many constraints, the ratio will possibly be high, which means many initial levels of vertices will have to be advanced.

7 Conclusions and future works

In this paper, we introduce a new model, LDAG, for special workflow applications. The three cases of level in LDAG are discussed in detail. For a reasonable one of these cases, we propose the topological sorting algorithm. The algorithm consists of two phases, namely *Level Adjusting* and *Topological Sorting*. We discuss some relevant problems, such as choice of stack or queue, the determination of directed circuit, complexity of the algorithm, etc. The experiment and analysis of the algorithm show its efficiency in grid workflow modeling.

In the future works, we will explore the scheduling heuristics based on LDAG for workflow applications, such as workflow with timing constraints. In addition, optimizing heuristics based on LDAG, such as tradeoff between time and cost, is also in our future research plan.

Acknowledgement

The work has been supported by China National Natural Science Foundation (No. 50975250), Zhejiang Natural Science Foundation of China (No. Y1100177) and Zhejiang Science & Technology Plan of China (No. 2010C31092).

References:

[1] Foster, I. and Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure (Second Edition)*. Elsevier Inc. 2004.

[2] Cao H. J., Jin H., Wu X. X., Wu S., Shi X. H., DAGMap: efficient and dependable scheduling of DAG workflow job in Grid. *The Journal of Supercomputing*. 2010(51): pp. 201-223.

[3] Yousra B. H. and Leila J. B:, Extended UML activity diagram for composing Grid services workflows. *Third International Conference on Risks and Security of Internet and System(CRiSIS '2008)*. pp. 207-212.

[4] Rafe V., Rahmani A. T., Formal Analysis of Workflows Using UML 2.0 Activities and Graph Transformation Systems, *In: Proceeding of 5th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2008, pp. 305-318.

[5] Cao H. J., Jin H., Wu S., and Tao Y. C., PGWFT: A Petri Net Based Grid Workflow Verification and Optimization Toolkit, *In: Proceeding of the 3rd International Conference on Grid and Pervasive Computing*(GPC), 2008, pp. 48-58.

[6] Alt M., Hoheisel A., Pohl H. W., Gorlatch S., A Grid Workflow Language Using High-Level Petri Nets. *In: Proceedings of PPAM*, 2005, pp. 715-722.

[7] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., Introduction to Algorithms (Second Edition). *The MIT Press*. 2001.

[8] Gallet M., Marchal L., Vivien F., Allocating Series of Workflows on Computing Grids, In: Proceedings of the 14th International Conference on Parallel and Distributed Systems, 2008, pp. 48-55.

[9] O'Brien A., Newhouse S., Darlington J., Mapping of scientific workflow within the e-Protein project to distributed resources, *In:*

Proceedings of UK e-Science All Hands Meeting, 2004, pp. 404-409.

[10] Laszweski G. V., Amin K, Hategan M., Zaluzec N. J., Hampto S., Rossi A., GridAnt: a client-controllable grid workflow system, *In: Proceddings of 37th Hawaii International Conference on System Science*, 2004.

[11] Malewicz G., Foster I., Rosenberg A. L., Wilde M. A tool for prioritizing DAGMan jobs and its evaluation, *The Journal of Grid Computing*, 5(2), pp. 197-212.

[12] Kosiedowski M., Kurowski K., Mazurek C., Nabrzyski J., and Pukaski J., Workflow applications in GridLab and PROGRESS projects. *The Journal of Concurrency Computation Practice and Experience*, 18(10), p1141-1154.

[13] Saha D., Samanta A., Sarangi S. R., Theoretical Framework for Eliminating Redundancy in Workflows. *In: IEEE International Conference on Services Computing*, 2009, pp. 41-48.

[14] Yuan Y. C., Li X. P., Wang Q., Zhu X., Deadline division-based heuristic for cost optimization in workflow scheduling. *The Journal of Information Science*, 2009(179), pp. 2562-2575.

[15] Tian G. Z., Yu J., He J. S., Towards critical region reliability support for grid workflows. The Journal of Parallel and Distributed Computing, 2009(69), pp. 989-995.

[16] Sumit W. S., Sanjeev K. A., Song J., Melvin K., Simon S., Modeling and Verifying Non-DAG Workflow for Computational Grids, *IEEE Congress on Services*. 2007.

[17] You S. Y., Kim H. Y., Hwang D. H., Kim S. C., Task scheduling algorithm in GRID considering heterogeneous environment. *In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2004, pp 240-245.

[18] Dong F., Akl S. G., Scheduling algorithms for grid computing: state of the art and open problems, *Technical Report*, No. 2006-504, School of Computing, Queens University Kingston, Ontario.

[19] Pop F., Dobre C., Cristea V., Genetic algorithm for DAG scheduling in Grid environments. *In: Proceedings of IEEE 5th International Conference on Intelligent Computer Communication and Processing*, 2009, pp. 299-305.