A Low Complexity Topological Sorting Algorithm for Directed Acyclic Graph

Renkun Liu

***Abstract*—In a Directed Acyclic Graph (DAG), vertex A and vertex B are connected by a directed edge AB which shows that A comes before B in the ordering. In this way, we can find a sorting algorithm totally different from Kahn or DFS algorithms, the directed edge already tells us the order of the nodes, so that it can simply be sorted by re-ordering the nodes according to the edges from a Direction Matrix. No vertex is specifically chosen, which makes the complexity to be O*E. Then we can get an algorithm that has much lower complexity than Kahn and DFS. At last, the implement of the algorithm by matlab script will be shown in the appendix part.**

***Index Terms*—DAG, algorithm, complexity, matlab.**

I. Introduction

In computer science, a topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time.

Kahn Algorithm works by choosing vertices in the same order as the eventual topological sort [1]. First, find a list of "start nodes" which have no incoming edges and insert them into a set S; at least one such node must exist in an acyclic graph. Then:

```
L ← Empty list that will contain the sorted
elements
S ← Set of all nodes with no incoming edges
while S is non-empty
   do remove a node n from S
      insert n into L
   for each node m with an edge e from n to
m
     do remove edge e from the graph
        if m has no other incoming edges
       then insert m into S
       if graph has edges
        then return error (graph has at
least one cycle)
        else return L (a topologically
sorted order)
```

Because we need to check every vertex and every edge for the "start nodes", then sorting will check everything over again, so the complexity is O(E+V).

An alternative algorithm for topological sorting is based on Depth-First Search (DFS) [2]. For this algorithm, edges point in the opposite direction as the previous algorithm. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort:

```
L ← Empty list that will contain the sorted
nodes

while there are unmarked nodes

    do select an unmarked node n
    visit(n)
    function visit(node n)
    if n has a temporary mark
    then stop (not a DAG)
    if n is not marked (i.e. has not been
visited yet)
    then mark n temporarily
    for each node m with an edge from n to
m
    do visit(m)
    mark n permanently
    add n to head of L
```

In theoretical computer science, DFS is typical used to traverse an entire graph, and takes time O(|E|), linear in the size of the graph. In these applications it also uses space O(|V|) in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus the complexity of DFS is also O(E+V) [3]-[5].

II. How the New Algorithm Works

In the new algorithm, we don't have to list any vertex, or try to go through the loop. We just need to list all the edges and choose which meet the condition.

First we place the vertexes in order from A, B, C, D … to XX. We assume it is in this order.

Then we make a matrix like (Fig. 1):

We can say A results to B.

This matrix describes a directed edge B to A with array: (2, 1). (We put the column first, row second and assume '1'

Manuscript received October 30, 2013; revised December 27, 2013.

Renkun Liu is with Beijing R&D Center, Beijing, China (e-mail: liurenkun1985@163.com).

stands for 'A' and '2' stands for 'B'.)

So if we firstly place vertexes A, B, C, D (1, 2, 3, 4 in the matrix) in order, and I want the final topology is in the order of 'right' results from 'left'. I will read the position of the array, 'left'=2, 'right'=1, if vertex (find(left)) > vertex (find(right)) ('>' means on the right, since we place the elements in the order of 1, 2, 3, 4) then we change the position of the vertexes, so that we can make sure in the new order, the left always results to the right. In this example, 'A, B, C, D' is changed to 'B, A, C, D'.

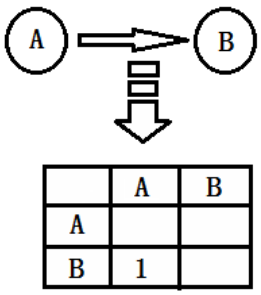

Fig. 1. Matrix example.

If vertex (position on the left) < vertex (position on the right), then leave it and jump to the next array.

Then orderly do the same thing with other edges in the matrix, we ensure the new order of the sorted vertexes match the 'left to right' matrix.

So for a topology that has E edges, what we do is just re-arranging the orders based on the matrix, the calculation times we do for sorting is less than E times. And the complexity for this algorithm is O1*E which is far less than O2*(E+V).

We can simply describe this algorithm as:

```
M ← Matrix that contains all the directed edges' information
E ← directed arrays of the element '1's in the Matrix
k=1; %loop continue indicator
while k==1
    k=0;
for every directed edge
    x=E(i,1);
    y=E(i,2);
    if x's positon is on the right of y's position, then exchange the nodes on these two positions.
    k is set to 1 again, because sort is not done, reset the indicator for one more loop
```

III. The Topology Sorting Result of the 3 Algorithms

There is an example (Fig. 2):

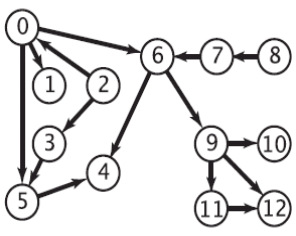

Fig. 2. A topology example.

This topology result is 2->8->0->3->7->1->5->6->9->4->11->10->12 by Kahn algorithm, and 8->7->2->3->0->6->9->10->11->12->1->5->4 by DFS.

In the new algorithm, we first change the graph into matrix (Fig. 3):

	A	B	C	D	E	F	G	H	I	J	K	L	M
A			1										
B	1												
C													
D			1										
E						1	1						
F	1			1									
G	1							1					
H									1				
I													
J							1						
K										1			
L										1			
M										1		1	

Fig. 3. Matrix that describes the topology praph.

Then we know the edge arrays that listed in the Matrix (Fig. 4):

E=

3	1
1	2
3	4
6	5
7	5
1	6
4	6
1	7
8	7
9	8
7	10
10	11
10	12
10	13
12	13

Fig. 4. Directed edges described by arrays.

This new matrix tells us: for each raw, the left results to the right.

The starting sequence is 1,2,3,4,5,6,7,8,9,10,11,12,13

With the first array ('3','1'), we find '3' is on position (3) and '1' is on position (1), and (3) is on the right of (1), so we change the position of '1' and '3', then we get a new sequence of 3,2,1,4,5,6,7,8,9,10,11,12,13.

With the second array ('1','2'), we find '1' is on position(3), and '2' is on position (2), but (3) is on the right of (2), so we change the position of '1' and '2', then we get a new sequence of 3,1,2,4,5,6,7,8,9,10,11,12,13.

With the third array ('3','4'), we find '3' is on position(1), , and '4' is on position (4), we can see (1) is on the left of (4), so the sequence will not be changed, and it's still 3,2,1,4,5,6,7,8,9,10,11,12,13.

We keep doing the same thing. When we have searched from the first raw of the arrays until the last raw, we still need to repeat from the beginning of the array and check again, in order to make sure that if the last exchange has impacted the first exchange, this situation will be fixed.

At last, according to the sorting with the arrays in the new matrix, starting with the order: A, B, C, D, E ….M (stands for 0, 1, 2, 3, 4…12), in the new order of 'left' results to 'right'. We get the final sequence of 3, 1, 2, 4, 6, 9, 8, 7, 5, 10, 11, 12, 13.

The sequence stands for the topology in the graph: 2->0->1->3->5->8->7->6 ->4 ->9->10->11->12 (for each sequence element, minus 1, to align with the graph) which stands for 3->1->2->4->6->9->8->7->5->10->11->12->13 of the graph.

The Most important is that in this example, the position only changes **9** times before we get the result (Fig. 5). It is even less than the edge number of 15. It is far less complex than Kahn and DFS algorithms.

But we need to notice that it takes 3 loops to finish the sequencing. Because the earlier position changes may be overlapped by the later changes, and makes the sequence out of the designed order, we need to check the loop again if changes were detected in the previous loop.

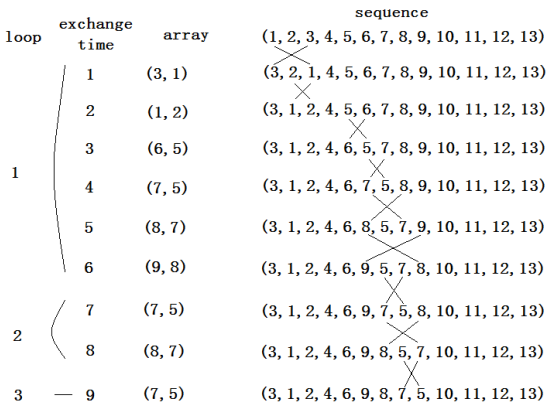

Fig. 5. The sorting process of the example.

IV. Conclusion

For a large system which has many vertexes, complexity is very important. This new low complexity algorithm can greatly decrease the computing time and increase the efficiency.

It needs to traverse the entire graph, and takes time O(|E|), linear with the size of the graph. However, there is no search path to store, what exists is a sequence with a stable size. It makes the new algorithm more efficient and use less storage than DFS in DAG sorting.

But there is a defect in the new algorithm that may increase the complexity. If the system is large, and complex, where are many triangles like the 5, 7, 8 structure in the previous graph, even triangles in the triangle, how many computing times is for the new algorithm and for DFS? It is sure the new algorithm must have less complexity than DFS in large systems, but the advantages would be smaller perhaps.

There also could be a tricky process to detect and simplify the triangle system, modify the triangle in a proper way before the formal sequencing, which would optimize the algorithm and make the complexity much lower.

The computing times and efficiency comparison in large systems, as well as the tricky process, should be studied in the next steps.

Appendix

```
% =THE IMPLEMENT OF THE ALGORITHM BY MATLAB SCRIPT=
%==================START==================

% load the matrix.
[num,txt,raw] = xlsread('file path');

% calculate the sequence length:
p=size(raw(1,:));

sequence_length=p(2)-1;

% create the initial sequence: 1,2,3,4……
node=(1:1: sequence_length);

% calculate the matrix(a) of arrays from the loaded matrix.
% record the number of the raws in 'a' with index 't'.
t=1;

% search the raw.
for j=1:1:size(num,1)

    % search the column.
    for i=1:1:size(num,2)

        % find '1' in the original matrix, and record its
        position in 'a'.
        if num(j,i)==1

            a(t,:)=[node(i),node(j)];

            t=t+1;

        end
end
end

% start sorting
% set the indicator to 1
k=1;

% if the indicator is 1, go on sorting
while k==1

    % first set the indicator to 0
    k=0;

% sort by each raw of matrix a
for i=1:1:size(a,1)

    % x is the first element of the raw
    x=a(i,1);

    % y is the second element of the raw
    y=a(i,2);

    % if x's position is on the right of y's position, then
exchange the two nodes in the sequence
    if find(node==x)>find(node==y)

        m=find(node==x);

        n=find(node==y);

        node(m)=y;

        node(n)=x;
```

```
            % if sort is done, reset the indicator to 1, need to start the 'a' loop and check again
                k=1;

        end

end
end
```

%' node' is the final modified sequence, corresponding to the DAG by the algorithm

%==========THE END==========

REFERENCES

[1] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962, doi: 10.1145/368996.369025.

[2] C. P. Trénaux, "École Polytechnique of Paris (X:1876)," in *Proc. French Engineer of the Telegraph in Public Conference*, December 2, 2010, Annals academic , pp. 1859–1882.

[3] S. A. Cook, "A Taxonomy of Problems with Fast Parallel Algorithms," *Information and Control*, vol. 64, no. 1–3, pp. 2–22, 1985, doi: 10.1016/S0019-9958(85)80041-3.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Section 22.4: Topological sort," *Introduction to Algorithms*, 2nd ed., MIT Press and McGraw-Hill, pp. 549–552, ISBN 0-262-03293-7, 2001.

[5] R. E. Tarjan, "Edge-disjoint spanning trees and depth-first search," *Acta Informatica*, vol. 6, no. 2, pp. 171–185, 1976, doi: 10.1007/BF00268499.

Renkun Liu was born in Dalian, May 12, 1985. He obtained his BS in communication engineering, Beijing Jiaotong University, Beijing, China, in 2008, and MS in optical communication, politecnico di Torino, Turin, Italy, in 2010. Renkun works for Fairchild Semiconductor since 2010, he is working in Beijing R&D as a Characterization and Device Modeling Engineer. He is currently majored in the field of semiconductor area.