International Journal of Mathematical Archive-7(10), 2016, 128-133
Available online through www.ijma.info ISSN 2229 – 5046

TOPOLOGICAL SORT ALGORITHM: A BFS APPROACH

ISHWAR BAIDARI[1*], AJITH HANAGWADIMATH[2]

[1,2]Department of computer science, Karnatak University, Dharwad, India.

(Received On: 12-09-16; Revised & Accepted On: 30-09-16)

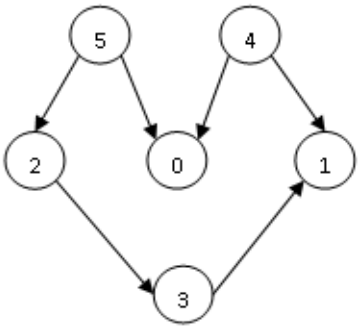
ABSTRACT

Topological sorting for directed acyclic graph (DAG) is linear ordering of vertices such that for directed edge uv, vertex u comes before v in the ordering. Topological sorting can be implemented using DFS. In this paper topological sorting is implemented using modified BFS algorithm.

***Keywords*:** *Topological sorting, DAG, DFS.*

1. INTRODUCTION

Let us consider the following graph



A topological sorting of the above graph is 5, 4, 2, 3, 1, 0. These can be more than one topological sorting for a graph. For example another topological sorting of the following graph is “4, 5, 2, 3, 1, 0”. The first vertex in topological sorting is always a vertex with in-degree 0.

A. Topological sorting Vs Depth First Traversal (DFS)

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS the vertex ‘4’ should also be printed before vertex ‘0’. So topological sorting is different from DFS. For example a DFS of the above graph is “523104”, but it is not a topological sorting.

We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. For example, given a digraph G =(V,E), DFS traverses all vertices of G and

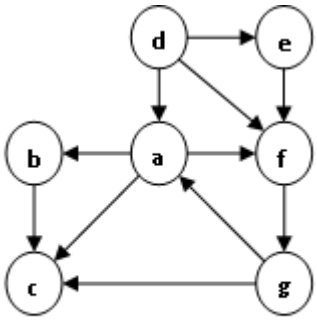
1) Constructs a forest together with a set of source vertices: and
2) Output two time unit arrays, d[v]/f[v]

B. DFS Forest

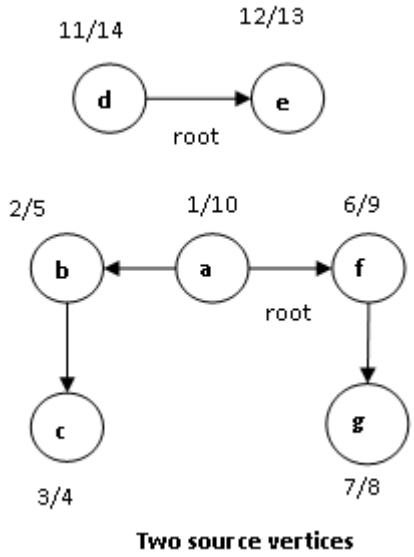
DFS constructs a forest F= (V, E_f), a collection of trees, where E_f = {(pred[v], v)| where DFS calls are made}

Corresponding Author: Ishwar Baidari[1]*
[1,2]Department of computer science, Karnatak University, Dharwad, India.

Example:



Original Graph



Two source vertices

C. Classification of the edges:

Tree edges: Those on the DFS forest

The remaining edges fall in three categories

Forward edges: (u, v) where v is a proper descendent of u in the tree [$u \neq v$]

Back edges: (u, v) where vertex v is an ancestor of u in the tree [u = v is allowed]

Cross edges: (u, v) where u and v are not ancestors or decedents of one another (in fact; the edges may go between different trees).

I. TOPOLOGICAL SORT

A Topological sort of a DAG G={V,E} is a linear ordering of all its vertices such that if G contains an edge (u,v), then 'u' appears before 'v' in the ordering. This can be accomplished using DFS algorithm.

But in this algorithm we are Topologically sorting a DAG using BFS algorithm. And this algorithm works for almost all DAG's and in few DAG's it satisfies the condition of a Topological sort leaving some (very few) edges. There is also a mechanism in this algorithm to distinguish such "Unsafe Edges"(unsafe for Topological sort) and them separately. [i.e., G.E – {unsafe edges} gives a perfect Topological sort].

A. *The assumptions made in the algorithm are as follows:*

- The algorithm assumes that the input graph G = (V,E) is represented using adjacency lists representation.
- It attaches several additional attributes to each vertex in the graph. We use three different colours to distinguish between vertices. White colour specifies that the vertex is not net discovered, Grey colour specifies that the vertex is discovered for the first time, and Black colour specifies that the vertex is fully discovered.

- We store colour of each vertex u∈V in the attribute u.colour, and the predecessor of 'u' in the attribute u.π. If 'u' has no predecessor then u.π =NIL. The attribute u.d holds the distance from the 'root' to vertex 'u' computed by the algorithm.
- The algorithm attaches the colour attribute to every edge in the graph. We use two different colours Green and Red to distinguish between edges. Green colour specifies that the edge is safe for Topological sort and Red specifies that the edge is unsafe for Topological sort.

B. The Algorithm

```
1)  for each vertex u ∈ G.V
2)  {
3)  u.colour = WHITE
4)  u.d = ∞
5)  u.π = NIL
6)  }
7)  for each edge (u, v) ∈ G.E
8)  {
9)    (u, v).colour = GREEN
10) }
11) Unsafe_Edge_Set = 0
12) for each vertex u ∈ G.V
13) {
14)   if(u.colour == white)
15)   {
16)             u.colour = GRAY
17)             u.d = 0
18)             u.π = NIL
19)             Q = 0
20)             ENQUEUE(Q, u)
21)             While (Q ≠ 0 )
22)             {
23)                     u = DEQUEUE(Q)
24)                     for each v ∈ G.Adj[u]
25)                     {
26)                             if(v.colour == WHITE)
27)                             {
28)                               v.colour = GRAY
29)                               v.d = u.d+1
30)                               v.π = u
31)                               ENQUEUE(Q, v)
32)                             }
33)                             else
34)                             {
35)                                     if( v.d>u.d)
36)                                     {
37)                                       (u, v).colour = Red
38)                     Add edge (u, v) to Unsafe_Edge_Set
39)                                       }
40)                             }
41)                     }End of for
42)                     u.colour = BLACK
43)                     List (u, u.d)
44)             }End of while
45)   }End of if
46) }End of for
```

II. THE ALGORITHM USES THE FOLLOWING DATA STRUCTURES

- A FIFO queue 'Q' to manage the set of Gray vertices.
- A set to store all the Unsafe edges of the given graph.
- A Linked-list to store all the vertices of the given graph in a linear order. All vertices at distance 0 will be at the beginning of the list, followed y the vertices at distance 1 which will be followed by the vertices at distance 2 and so on...
- The for loop in lines

A. One of the way in which we can implement the List (u, u.d) procedure is shown belowList (u, u. d)

```
{
    Switch (u.d)
    {
        Case"0": Attach 'u' to the rear of "List_0"
        Case"1": Attach 'u' to the rear of "List_1"
        Case"2": Attach 'u' to the rear of "List_2"
        .
        .
        .
        Case"n": Attach 'u' to the rear of "List_n"
    }
}
```

[Finally join all 'n' lists one after the other according to their numbering.]

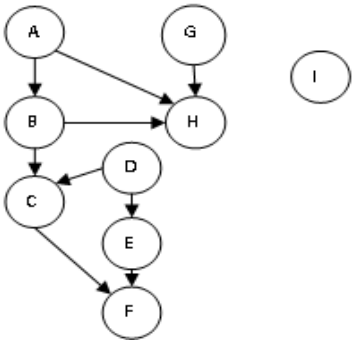
III. WORKING OF THE ALGORITHM IS AS EXPLAINED BELOW

- The for loop in line 1-6 initializes all the vertices in the graph. It colours every vertex white, set u.d to be infinity for each vertex u, and set the predecessor of every vertex to be NIL.
- The for loop in lines 7-10 initializes all the edges in the graph. It colours every edge to (u, v) to green.
- In line 11 a Set by name 'Unsafe_Edge_Set' is declared and initializes to 'null'.
- In lines 8-9 the procedure BFS (G, u) is called for the first time with specified 'root' vertex.
- The for loop in lines 12-46 check every vertex to find any undiscovered vertex in the graph (i.e. A vertex with a colour white). If such an undiscovered vertex is found then, the colour of that vertex is changed to gray in the line 16. Line 17 initializes u.d to 0 & line 18 sets the predecessor of that vertex to be NIL. Line 19 initializes a queue with null & line 20 enqueues the discovered vertex into it.

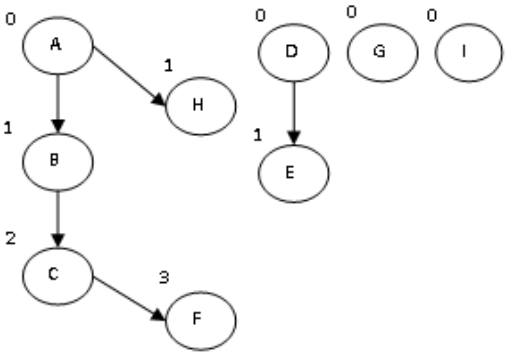
  The while loop of lines 21-44 iterates as long as the queue is not empty. The line 23 determines the gray vertex 'u' at the head of the 'Q' and removes it from 'Q'. The for loop of lines 24-41 considers each vertex 'v' in the adjacency list of 'u'.

  If 'v' is the white then in lines27-30 the procedure paints vertex 'v' gray, sets its distance v.d to u.d+1, records 'u' as its predecessor in v.π, and places it at the end of the queue 'Q'. If 'v' is not white then the procedure checks whether the distance of 'v' (v.d) is less than the distance of 'u' (u.d), if so then the colour of edge (u,v) is set to 'Red' and that edge is added to the set 'Unsafe_Edge_Set'.
- Once the procedure has examined all the vertices in u's adjacency list, it blackens 'u' in line 42.
- And in line 43, the procedure places the latest finished vertex 'u' into a linked-list data structure according to its distance i.e., u.d value.

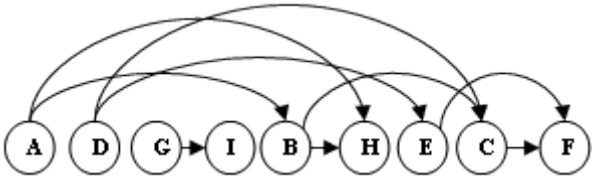
Example 1: consider the Directed acyclic graph (Dag) given below



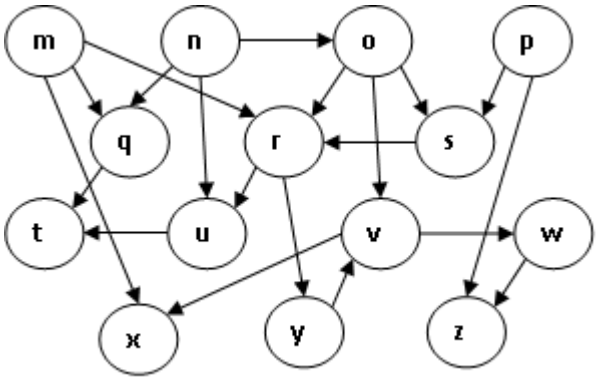
Output 1: The output forest formed by the above algorithm is as follows



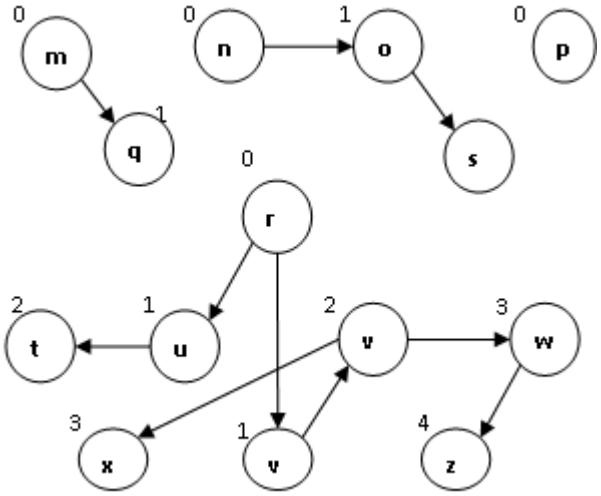
Output 2: The topological sort formed by the above algorithm is follows:



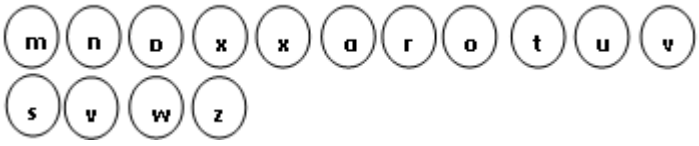
Example 2: Consider the Directed acyclic graph (Dag) shown below:



Output 1: The output forest formed by the above algorithm is as follows:



Output 2: The topological sort formed by the above algorithm is as follows:



In this case it satisfies the condition of the Topological sort leaving some (very few) edges.
[i.e., G.E – {Unsafe edges} gives a perfect Topological sort]

➔ In this case the Unsafe_Edge_Set = {(o-r), (s-r)}

So in this case leaving those 2 Unsafe edges (unsafe or Topological sort), the linear ordering of the vertices has a topologically sorted all the edges in the given graph.

III. TIME COMPLEXITY

- The for loop in lines 1-6 initializes all the vertices in the graph and hence it takes O(V) time.
- The for loop in lines 7-10 initializes all the edges in the graph and hence it takes O (E) time.
- The for loop in lines 12-46 iterates once for each vertex in the graph and the while loop in the lines 21-44 scans the adjacency list of the vertices only when the colour of the vertex is WHITE.
  Hence the total lengths of the edges scanned is given as follows:

$$\sum_{v \in V} |Adj[v]| = \theta(E)$$

  Hence the total time spent in scanning adjacency lists is O (E).
- And the total time taken by the 'for and while' loop is:
  O (V + E) time.
- Therefore, the total time complexity of the algorithm is:
  O (V) + O (E) + O (V + E) = O (V + E)

REFERENCES

[1] AIELLO. W. Chung. F., AND LU, L, 2000. A random graph model for power low graphs. In Proceedings of the ACM Symposium on the theory of computing (STOC). 172-180.
[2] AJWANI. D., Friedrich T. And Meyer. U. 2006. Ah O(n2.75) algorithm for online topological ordering. In proceedings of the Seandinavian workshop on algorithm theory (SWAT). Lecture Notes in Computer Science Vol. 4059. Springer-verlag, New York. 53-74.
[3] S. R Arikati and C.P Rangan, Linear algorithm for optimal path cover problem on interval graphs, Inform Process. Ltt.35(1190 PT. 149-153)
[4] A Brandstadt, V.B.LE and J.P.Spinrad, Graph classes: A survey, SIAM, Philadelphia, 1999.
[5] George B. Mertzios and Derek G. Corneil, A Simple Polynomial Algorithm for the Longest Path Problem on cocomparability Graphs, SIAMJ, Discrete Math. Vol. 26, No.3 PP. 940-963.
[6] The Longest Path Problem on Permutation Graphs Ruei-Yuam Chang, Cheng-Hsien Hsu, and Shen-Lung, The 29th workshop on Combinatorial Mathematics and Computation Theory.
[7] A Simple Polynomial Algorithm for the Longest Path Problem on Cocomparability Graphs, George B Mertzios, Derek G Corneil, arxiv:1004.456001[CS.DM] 26 Apr 2010.

Source of support: Nil, Conflict of interest: None Declared