International Journal of Mathematical Archive-7(10), 2016, 123-127
Available online through www.ijma.info ISSN 2229 – 5046

ALGORITHM TO FIND TOTAL NUMBER OF PATHS IN DIRECTED ACYCLIC GRAPH

ISHWAR BAIDARI[1*], S. P. SAJJAN[2]

[1,2]Department of Computer Science, Karnatak University, Dharwad, India.

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

ABSTRACT

A directed acyclic graph (DAG) is a graph with directed edges in which there are no cycles. Formally, a directed graph is a pair (N,R⊆N×N) consisting of a set of Nodes N and a binary relation R on it that specifies a directed edge from a node n to another one m whenever (n, m). IN this paper we presented on polynomial time algorithm to find total number of path in DAG.

Keywords*: DAG, Path, Polynomial, Directed graph.*

I. INTRODUCTION

A. Shortest Path

The basic problem: Find the "best" way of getting from s to t where s and t are vertices in a graph. We measure "best" simply as the sum of edges lengths of a path. For instance the graph could be a map representing intersections as vertices and segments as edges you want to find either the shortest or fastest route.

Suppose we already know the distances d(s, r) from s to every other vertices. This isn't a solution to the shortest path because we want to know actual paths having those distances. There are two kinds of shortest paths those formed by a single edge (s,t) and those in which the path from s to to goes through some other vertices: let's say x is the best vertex the path goes through before t. Then in the second case the overall path must itself be a shortest path. A final observation is that d(s,x) must be less than d(s, t) since d(s, x)= d(s, t)+length(x, t) assuming all edges have the length.

B. Dijkstra's Algorithm

The only remaining use of d(s,y) in this algorithm is to determine what order to process the vertices in Dijkstra's algorithm for shortest paths.

C. Topological ordering and shortest path

There is an important class of graphs in which shortest paths can be computed more quickly in linear time. The idea is to go back to algorithms 1&2 which required you to visit the vertices in some order. In those algorithms we defined the order to be sorted by distance from S, which as we have seen works for positive weight edges but not if there are negative weights. Here another order always works define a topological ordering of a directed graph to be one which whenever we have an edge from x to y, the ordering visits x before y. If we define such an ordering be sure that the predecessor of a vertex X is always processed before we process X itself.

D. Shortest path in directed Acyclic Graph

Given a weighted directed Acyclic graph and source vertex in the graph find the shortest path from given source to all other vertices.

For a general weighted graph we can calculate single source shortest distance in O(VE) time using Bellman Ford Algorithm. For a graph with no negative weights we can do better and calculate single source shortest distance in O(E+VlogV) time using Dijkstra's algorithm. We can calculate single source shortest distance in O(V+E) time for DAGs. The idea is to use Topological Sorting.

E. Topological ordering and acyclic graphs

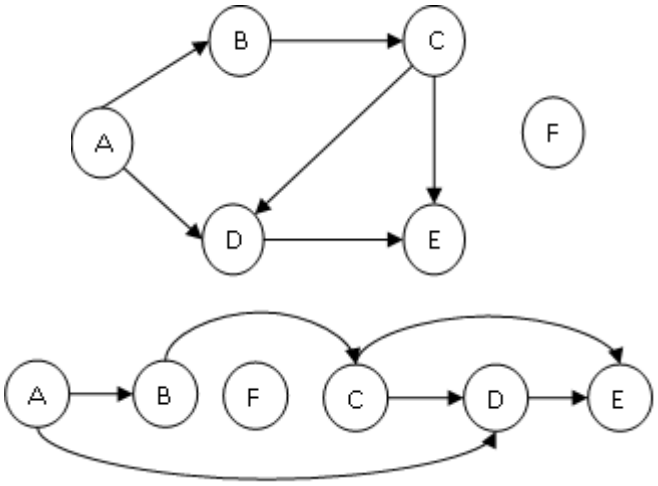
Directed acyclic graph (DAG) to be a directed graph containing no cycle (a cycle is a set of edges forming a loop, and all pointing the same way around the loop.

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

Suppose G is not a DAG, so it has a cycle. In any ordering of G, one vertex of the cycle has to come first but then one of the two cycle edges at that vertex would point the wrong way for the ordering to be topological. In the other direction we have to prove that every graph without a topological ordering contains a cycle.

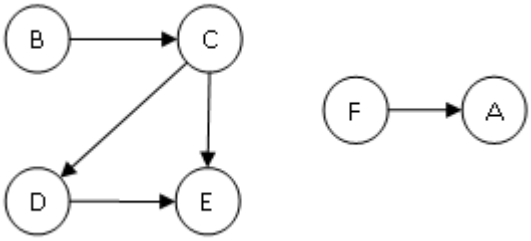
F. Example

G = (E, V) find a linear ordering of vertices such that for all edges (v, w) in E, V precedes in the ordering



Step-1: Identify vertices that have no incoming edge the in-degree of these vertices is zero.

Step-2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



Repeat step 1 and step 2 until graph is empty.

II. TOTAL PATHS IN DIRECTED ACYCLIC GRAPH

A. Pre-algorithm

Obviously the graph has to be acyclic, as otherwise there can be infinite paths. (circling the cycle infinite times).
Also, since the graph is acyclic and finite, there has to be atleast one exit and entry nodes. To prove that there will be an exit point, pick any node and keep on following any edge from it until you find a node that has no outgoing edge. Since, the number of nodes are finite and there are no cycles, you are bound to bump into a node that has no outgoing edge. (hence, the exit node). Similarly, there has to be exactly on entry node.

B. Algorithm

Let us first go over the algorithm, then I will run through it using an example.

Addition Data to be stored

1. A queue of nodes having no out edge.
2. Meta-data with each node storing the total number of paths from this node to exit points initialized to zero for all nodes.

Let us call this path

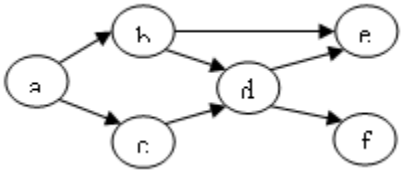
- Insert all the exit nodes into the queue and initialize path[] to 1 for them.
- Iterate over queue until queue is not empty.
- Iterate over incoming edges of current node.
- Increment the path[other end of edge] by path[current node].

- Delete current edge.
- If the other end of edge has no more outgoing edge insert it into the queue.

The path[i] now stores the total number of paths from node I to any exit.

C. Run through

Consider the following graph:



Paths:
The total number of paths is=14.

Paths:

a	b	c	d	e	f
0	0	0	0	0	0

Queue:

Paths:

a	b	c	d	e	f
0	0	0	0	1	1

Queue:
f e
Paths:

a	b	c	d	e	f
0	0	0	1	1	1

Queue:
e
Paths:

a	b	c	d	e	f
0	1	0	2	1	1

Queue:
d
Paths:

a	b	c	d	e	f
0	3	2	2	1	1

Queue:
b c
Paths:

a	b	c	d	e	f
3	3	2	2	1	1

Queue:
c
Paths:

a	b	c	d	e	f
5	3	2	2	1	1

Queue:
a

D. The Algorithm

TotalPths_DAG

```
1)  Exit_Queue = Ø
2)  foreach vertex u ∈ G.V
3)  {
4)          u.paths = 0
5)          u_Parent_Queue = Ø
6)  }
7)  foreach vertex u ∈ G.V
8)  {
9)          Degree = 0
10)         foreach v ∈G.Adj[u]
```



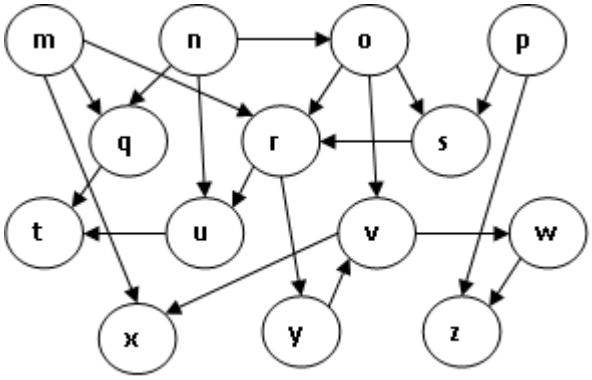
```
11)         {
12)                 Degree = Degree + 1
13)                 ENQUEUE ( v_Parent_Queue, u)
14)         }
15)         u.degree = Degree
16)         if(u.degree==0)
17)         {
18)                 u.paths=1
19)                 ENQUEUE(Exit_Queue, u)
20)         }
21) }
22) while(Exit_Queue≠ Ø)
23) {
24)         u=Dequeue(Exit_Queue)
25)         while(u_Parent_Queue≠ Ø)
26)         {
27)                 v = Dequeue(u_Parent_Queue)
28)                 v.paths = v.paths + u.paths
29)                 v.degree = v.degree – 1
30)                 if ( v.degree == 0)
31)                 {
32)                 ENQUEUE (Exit_Queue, v)
33)                 }
34)         }
35) }
36) Total_Paths = 0
37) foreach vertex u ∈ G.V
38) {
39)         Total_Paths = Total_Paths + u.paths
40) }
```

E. Working of the Algorithm TotalPths_DAG

- A Queue called 'Exit_Queue' is declared and initialized in line 1. It is used to hold the exit nodes.
- The for loop in lines 2-6 is used to initialize the 'paths' attribute of all the vertices along with their respective 'Parent_Queue'. The 'Paths' attribute is initialized with 0 in line 4 and the 'Parent_Queue' of all the vertices is initialized with null in line 5.
- The for loop in lines 7-21 is used to perform all the pre-requisite operations before we actually start to count the total number of paths in the given DAG.
  - A variable 'Degree' is declared and initialized with 0 in line 9.
  - The for loop in lines 10-14 scans the adjacency list of every vertex 'u' ∈ G.V increments the value of 'Degree' variable accordingly in line 12 and the parent queue is formed in line 13.
  - The value of 'Degree' variable is assigned to the 'degree' attribute of respective variable in line 15.
  - The if condition in line 16 checks whether the 'degree' of the vertex 'u' is equal to 0, if so then the 'paths' attribute is set to 1 for the vertex 'u' in line 18 and the vertex 'u' is enqueued into the 'Exit_Queue' in line 19.

- The While loop in line 22-35 iterates through every vertex present in the 'Exit_Queue' and calculates the number of paths from each vertex u ∈ G.V in the given graph.
- The vertex 'u' is Dequeued from the 'Exit_Queue' in line 24 and the while loop in lines 25-34 iterates through each vertex present in the 'Parent_Queue' of the vertex 'u' dequeued in line 24.
  - In line 27, a vertex 'v' is dequeued from the 'Parent_Queue' of vertex 'u'.
  - In line 28, the v.paths attribute is incremented with the value of u.paths attribute.
  - In line 29, the Degree of vertex 'v' is decremented by 1.
  - The if condition in line 30 checks whether the degree of vertex 'v' is equal to 0, if so then the vertex 'v' will be dequeued into the 'Exit_Queue' in line 32.
- A variable called 'Total_Paths' is declared and initialized to 0 in line 35. It will be used to hold the total number of paths in the given DAG.
- The for loop in lines 37-40 iterates through all the vertices in the given graph and stores the total number of paths in the variable 'Total_Paths'.An easy way to comply with the journal paper formatting requirements is to use this document as a template and simply type your text into it.

F. For Example: Consider the following (DAG)



According to the above algorithm the total number of paths is = 51.

G. Time Complexity

- The for loop in lines 2-6 runs once for every vertex u ∈ G.V, hence it takes O (V) time.
- The for loop in lines 7-21 scans all the vertices and edges once, hence it takes O (V+E) time.
- The while loop in lines 22-35 scans all the vertices and edges once, hence it takes O (V+E) time.
- The for loop in lines 37-40 runs once for every vertex u ∈ G.V, hence it takes O (V) time.

Hence the above algorithm takes O (V+E) running time.

REFERENCES

[1] 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)
[2] A Brandstadt, V.B.LE and J.P.Spinrad, Graph classes: A survey, SIAM, Philadelphia, 1999.
[3] 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.
[4] 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.
[5] 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