Anuj Panwar
Varun Goyal

Solution 1

DFS is run picking up any vertex keeping track of the edges and vertices being seen.

The main idea is when running DFS, if we encounter a vertex that has been seen but the edge connecting to the present vertex and that vertex is not yet traversed there exists a loop or cycle and that edge is a back edge.

The algorithm exists if we encounter such an edge otherwise DFS is run normally until all vertices have been traversed.

Pseudo code:

DFS-RUN-cycle (G= (V, E), s)

1. seen[v]=false for every vertex v
2. DFS(s)
3. If all vertices have been traversed ,Print("No cycle found");

DFS-cycle (v)

1. seen[v]=true
2. for every neighbor u of v
3. if not seen[u] then

    DFS-cycle (u)

    edge (u, v) = true

  else

  If seen [u] and edge(u,v) = false

    Print("Cycle found"), exit.

The algorithm will run in O(m+n) time as the algorithm is same as running the DFS(runs in O(m+n)) .In addition, a condition is checked if already seen vertex is found.

Anuj Panwar
Varun Goyal

Solution 2

First the vertices are sorted in Topological order and stored in a reversed order.

(actually tolopological sort is not fully implemented as we needed the vertices in reverse order)

or the edges now go from right to left in the array obtained v[n] after reverse of actual Topological sort.

Then for every i-th element if neighbours exist in (i-1) elements then the (max value of a neighbor of i) +1 is assigned to element i.

Heart of the solution:

S[i] = value of longest path considering the graph obtained by vertices from v[0] to v[i]

S[i] = max (j) + 1 such that j is neighbor of i (0<j<=i-1)

=0 (if no such j is found)

Return max(S[i])

Algorithm:

Longest-path(G=(V,E))

TopSort (G=(V,E))

```
1.  for every vertex v
2.    seen[v]=false
3.    fin[v]=1
4.  time=0
5.  for every vertex s
6.    if not seen[s] then
7.      DFS(s)
```

DFS(v)

```
1.  seen[v]=true
2.  for every neighbor u of v
```

Anuj Panwar
Varun Goyal

```
3.      if not seen[u] then
4.         DFS(u)
5.    time++
6.    fin[v]=time (and output v)


//v1,v2….vn are vertices in topological reverse order
8. s[0]=0;
9. for k=1 to n{
10.    for j =0 to (k-1)
11.   { s[k]=0
12.   if edge exists from v(k) to v(j)
13.    then( if S[k]< S[j])
 14.          S[k] = S[j]+1 }}
Return max{S[k]}
```

Topological Sort takes O(m+n) time . the steps 9 to 14 take O(n^2) time as the inner loop will check k-1 values for every k if they are adjacent.

- ⇨ Time complexity = O(m+n) + O(n^2) = O(n^2).

We traverse the array in reverse topological order, so we know that if an edge exists it would be from i^{th} to one or more of the (i-1) vertices and we take the maximum value of those (i-1) +1 is assigned. The maximum value ensures that we are taking the longest path and we add 1 to increase the count as the present vertex would also be connected to all vertices who were used to compute the s[ ]value of the neighbour.

Anuj Panwar
Varun Goyal

Solution 3

First all the edges of the F subset are considered and if they form a loop , -1 is output and the program exists.

Otherwise , all edges of F are included and then Kruskal's algorithm is run.

Edges are considered in increasing order of weight and corresponding edges are included,if they form a loop they are not included.Program ends if all the vertices have been included

Min-cost-F(V,E,w)

1. If F contains a cycle
2. Print("-1")

   exit.

   Else T =F

   Sort the edges in increasing order of weight
3. For edge e do
4. If union(T , e) does not contain a cycle then
5. Add e to T
6. Return T

Union (u,v)

1. if size[boss[u]]>size[boss[v]] then
2. for every z in set[boss[v]] do
3. boss[z]=boss[u]
4. set[boss[u]=set[boss[u]] union set[boss[v]]
5. size[boss[u]]+=size[boss[v]]
6. else do steps 2.-5. with u,v switched

Anuj Panwar
Varun Goyal

The running time is time for sorting that is O(m logm) where m is the number of edges plus running time of unioin that is O(nlogn)

Step 1,4,5 overall executes n-1 times

Every vertex changes its boss less than logn times(whenever the vertex changes size it doubles or more)

Overall step2,3 takes O(nlogn)

Time complexity => O(mlogm) + O(nlogn) = O(mn)