Analysis of Algorithm | H.W-5

SUBMITTED BY: ASHTOSH CHAUHAN
UIN: 23200990 24

1.) Exercise 20.2.7.

- **Main Idea:** There are n professional wrestlers and a list of r pairs of wrestlers for which there are rivalries. We have to check possibility of whether it is possible to designate some of the wrestlers as faces and the remainder as heels such that each rivalry is between a face and a heel in $O(n+r)$. We will create a adjacency list ~~such that~~ ~~were~~ based on rivalry list. This problem is similar to two coloring problem. Here we will try to color the vertices of this graph of rivalries by two colors, babyface and heel. We will create a list of colors which we will use to store color of wrestlers. We will use 1 to represent "babyface" and 0 for "heel". Then we will assume any random node in graph as root node (source vertex) (let say node 1) assign it a color (based on wrestler classification (if given)) (let say 0 - heel) and start breadth first search from it. We will use bfs to assign color to every node such that no two alternate nodes get same color. In case two of neighbouring nodes gets same color we break our bfs and conclude that

rivalries can't be separated exclusively to faces and heels. In case we have a pre-wrestler classification data available, we can check after BFS whether colors assigned to every wrestler is right. If our BFS gets complete without breaking in between, we can conclude that we could designate every wrestler as 'babyface' and 'heels'.

#Pseudo Code:

⇒ function Main (n, rivalries):

- create a adjacency list "wrestler" of size n.
- for rival in rivalries:
  wrestler [rival[0]].append(rival[1])
  wrestler [rival[1]].append(rival[0])
  } filling adjacency list.
- create a list to store color of wrestler 1 to n:
  color = [-1] * (n+1). where color[i] is color of wrestle
- for i in range (1, n+1):
  if color[i] == -1:
  if not bfs(i):
  return False // not possible.
  return True. // possible.

⇒ function bfs(node):

```
color[node] = 0
Q = ϕ
Enqueue(Q, node)
while Q ≠ ϕ:
        u = Dequeue(Q)
        for n in wrestler[u]:
                if color[n] == color[u]:
                        return False
                if color[n] == -1:
                        color[n] = 1 - color[u]
                        Enqueue(Q, n).
return True.
```

Time Complexity:

The operation of enqueue and dequeue takes O(1) and so total time devoted to to queue operations is O(V)

Since the procedure scans each adjacency list at most once, since the sum of lengths of all ~~|E|~~ |V| adjacency lists is O(E), the total time spent in scanning adjacency lists is O(V+E).

The overhead for initialization is O(V).

∴ Total time complexity = O(V)+O(V)+O(V+E)

= O(V+E)

Correctness :-

To prove the correctness of algo we need to show two things:

1. If the algorithm terminates without a conflict then the designated is possible
2. If the algo finds a conflict, then graph cannot be designated into two colors.

Proof of 1 by Contradiction :-

Let A be set of nodes colored 'face' and B be the set of nodes colored 'heel'. As algo did not find a conflict, there can be no edge between nodes in the same set.

Suppose there exist an edge between a node in A and a node in B. Let node of A be u and node of B be v. Since u and v have different colors, u must have been enqueued before v. Let w be the node that enqueued v. Then w and u have the same color, which means that algo would have found a conflict when processing edge (u, v). This is a contradiction, so there can be no edges between nodes in different sets. Hence designation is possible.

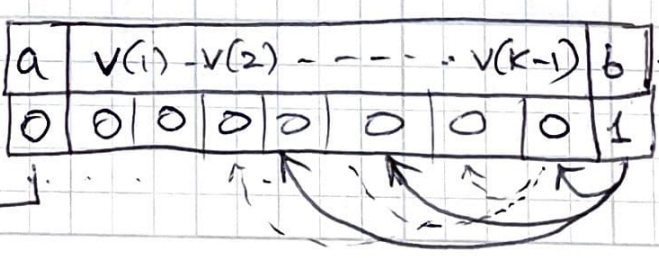
Proof of 2 by contradiction :-

Assume that the algo finds a conflict, which means there exists an edge between nodes of same color. Let the conflict be between node 'u and v both colored 'heel'. Then u and v were enqueued at different times and the node that enqueued v must have already processed u and colored it 'face'. Therefore, there is an edge b/w nodes of diff colors, which means cannot desig

2.) Exercise 20.4-2.

- Main Idea:

To count the number of simple paths from vertex a to vertex b in a directed acyclic graph $G(V, E)$ we can use a bottom-up approach with topological sorting. The basic idea is to first perform topological sorting in a graph. After topological sorting we will iterate over the vertices in reverse order of the topological sort, counting the number of simple paths from vertex to the destination vertex b. The basic idea is to use bottom-up approach (dynamic programming) to calculate the number of simple paths from each vertex to the destination vertex b by calculating the counts for each vertex in reverse order of the topological sort. At each step we calculate the count for a vertex u by summing the counts for each of its neighbours that come later in the topological order. If a vertex do not have any outgoing edges the count of that path remain as 0. Finally we return the count of vertex a.



Pseudo Code:

⇒ Function Path_Counter (G, a, b):

- Topologically sort (G) such that. — O(V+E)

  V(0), V(1), V(2), V(3) ------ V(K-1), V(K)

  ⇓ a ⇓ b.

- Create a list to store count.

  for i = 0 to K-1 — O(K).

  count [i] = ∞

- count [K] = 1. — O(1)
- return DP (G, count, 0)

⇒ Function DP (G, count, i):

- if count [i] != ∞ — O(1)

  return count [i]

- else: count [i] = 0 — O(1)

  for v[node] in G's adjacency [V[i]]: — O(V+E)

  if 0 < node ≤ K:

  count [i] += DP (G, count, node)

  else:

  continue.

  return count [i]

Time Complexity:

The time for topological sort will be O(V+E)
Bottom-up approach require to iterate over the vertices in the graph. which will also take O(V+E).

Total time complexity = O(V+E).

Correctness:

- Proof by contradiction:-

Suppose algo returns an incorrect count of the number of simple paths from vertex a to vertex b. Let c = count returned by algo.
c' = correct count of simple path a → b

Then there is 2 possibility:-

① There must be at least one simple path from a to b that is not counted by the algo.

② Algo must have counted a path which is not valid

⇒ Let x be any simple path from a to b that is not counted by algo. Since x is a simple path, it must include at least one vertex u such that all of u's neighbors that come later in the topological order are not on the path.

Since our algorithm calculate the sum of the counts for each of u's neighbors that come later in the topological order, we can say that it correctly counts the no. of simple paths from u to b. Therefore the algo must also correctly count the number of simple paths from a to b that include vertex u and the rest of the vertices on path x, which is a contradiction.

⇒ If our algo counts an extra path then there must be a vertex u on that path such that there is another path from u to b that is not included. But this contradicts the observation that the algo correctly calculate the count for each vertex.

Therefore our assumption is wrong and the algo correctly computes the number of simple paths from a to b.