HW 5

#CSDS310

1 - Basketball Teams

> There are n teams ranked from $\{1, 2.. . , n\}$.
> Since there are m games that means there are m directed edges.
> Since teams are able to dominate each other we should use Strongly Connected Components.
> The highest rank in that component is the value of that component.
> Apply DFS to determine highest factor of each component.
> This algorithm checks for the highest ranked team domination factor (so 1 is highest ranked)

Pseudo Code

```
def DominationFactor(Teams, Games):
        n = len(Teams)
        m = len(Games)
        groups = SCC(Teams, Games)
        g = len(groups)
        groupings = [None] * g
        maxs = [None] * g
        # Find max node in a group
        for group in groups do
                mins = group[0].val
                for node in group do
                        groupings[node] = group
                        if mins > node.val then mins = node.val
                maxs[group] = mins
        scc_edges = [None] * g
        # Find edges for SCC
        for edge in Games do
                g1 = groupings[edge[0]]
                g2 = groupings[edge[1]]
                if g1 != g2 then
                        if scc_edges[g1] is None then
                                scc_edges[g1] = [g2]
                        else if g2 not in scc_edges[g1] then
                                scc_edges[g1].append(g2)
        # Find Domination Factor of a group
```

```
        bests = [None] * g
        for group in groups do
                if bests[group] is None then
                        modifiedDFS(scc_edges, group, bests, maxs)
        # Find Domination Factor of nodes
        z = [None] * Teams
        for team in Teams do
                z[team] = maxs[groupings[team]]
        return z

def modifiedDFS(edges, start, bests, max_group):
        if bests[start] is None then bests[start] = max_group(start)
        for v in edges[start] do
                if bests[v] is None then
                        modifiedDFS(edges, v, bests, max_group)
                bests[start] = min(bests[start], bests[v])
```

Proof

Create a graph $G = (Teams, Games)$ where $Teams$ is the different teams and $Teams[0]$ is the best team. $Games$ is the different games between teams where $Games[0]$ is the winning team of that game and $Games[1]$ is the team that lost. G is a directed graph and the edges are unweighted.

Given this setup, we know that the SCC algorithm will return groups of nodes where each group contains nodes able to reach every other node in the group. Therefore the domination factor of nodes in a group is at least the highest ranked team in a given group.

We can then run DFS on these SCC groups using the edges that connect the groups. Because DFS searches all the children nodes before it completes the ancestors, we can modify it to find the largest domination factor of its children and return the largest domination factor between a nodes children and the domination factor of the group. Recursively finding the domination of a nodes children works inductively as base case is trivially if a node has no children then itself is its greatest domination factor. Then the inductive step is using DFS to find the domination factor of its children and then the domination factor of all nodes in the group is the greatest domination factor of its children and itself.

This approach for DFS works as long as the graph DFS is being applied to has no loops. Since we are applying DFS to our SCC there cannot be any loops otherwise any looped groups would make their own larger component.

> Thus applying a modified DFS to our SCC will return the true greatest domination factor for all groups, then the domination factor of any given node is the true domination factor of the group that it is in.

Runtime

> SCC is runtime of $O(n+m)$, finding max of groups is $O(n)$, finding the edges of SCC's is $O(m)$, DFS is worst case $O(n+m)$ (assuming each node is its own SCC). So total runtime is $O(n+m)$

2 - Proofs

a

> True, there are 3 possible cases for a DFS search with $uv \in E$ and directed graph G.
>
> Case 1, the DFS reaches u before v. When the DFS reaches u it then searches through its neighbours and finishes u's neighbours first. Since v is a neighbour of u, it will finish before u is finished. Therefore $u.f > v.f$.
>
> Case 2, is DFS reaches v before u, and v is an ancestor of u. Since DFS finishes children before ancestors, u will finish before v. Therefore $v.f > u.f$.
>
> Case 3, is DFS reaches v before u and v is not an ancestor of u. This means that u will not be traversed in this DFS so v will finish before u. Therefore $u.f > v.f$.
>
> Therefore the only way for $v.f > u.f$ is if v is an ancestor of u and v is discovered before u.
>
> Since v is an ancestor of u then $\exists$ a path p from v to u. Since we have an edge uv which connects u to v, then we have a cycle with edge uv.
>
> $\therefore$ If $v.f > u.f$, then uv must be on a cycle.

b

> True, proof by contradiction.

> Assume uv is a cross edge and there is a path from v to u.
>
> If there is a path from v to u then v is an ancestor of u. If v is an ancestor of u then v cannot be finished until u has finished.
>
> However if uv is a cross edge then v must have discovered and finished before u was discovered.
>
> This is creates a contradiction. Therefore if there is a path from v to u in G, then uv is not a cross edge.

3 - Wrestler Rivalries

> Apply BFS, if node has already been visited, check if the depth is of different parity. If they are same parity return false. Otherwise assign baby faces to odd parity nodes and heels to even nodes and return these assignments.

Pseudo Code

```
def WrestlerRivalries(W, R):
        n = len(W)
        Assignments = [None] * n
        for i in n:
                if Assignments[i] is None then
                        Queue = new Queue
                        Queue.insert((W[i],0))
                        # Queue format: (Node, Heel (0)/ Baby Face
(1))
                        while Queue is not empty do
                                node, parity = Queue.pop()
                                for v in R[node] do
                                        if Assignments[v] is None
then
                                                Queue.insert((v,
(parity+1)%2))
                                        else if Assignments[v] !=
(parity+1)%2 then
                                                return False
                                Assignments[node] = parity
        return Assignments
```

Proof

> Create a graph $G = (W, R)$ where W represents the different wrestlers and R represents the rivalries between them. Let the edges be undirected and unweighted (weight = 1).
>
> Given this setup, we know that BFS for discontinuous graphs will correctly reach all other nodes, and be able to determine the distance from the start node. We assign nodes an even distance from our start Heel and nodes with an odd distance as Baby Face. However this is not enough to guarantee that no same types have rivalries because some wrestlers can have rivalries with other established type of wrestlers. To ensure this situation is handled correctly, instead of not inserting a node into the queue if it has been visited, check the parity of the node to determine if it will not result in a rivalry between two wrestlers of the same type. If it will then return false, otherwise keep going in the BFS. Inductively this works because the parities is an optimal way of assigning types to wrestlers.
>
> Each node that is reachable from an arbitrary source node will be placed in queue only once during the BFS because we only add nodes into the queue if they have not been visited and we only update the parity of a node once it has been visited. Therefore, once the parity of a node is calculated it does not change by the BFS and our algorithm returns false if we have parities that are not alternating.

Runtime

> BFS is runtime $O(n + r)$. All other parts of the code are constant time. So total runtime is $O(n + r)$

4 - Course Evaluation

> | | | |
> |---|---|---|
> | CSDS 310: Algorithms (100/6869) | Nov 25 - 11:59 PM Dec 18 | Yes |