CSDS 310 HW 5

1

There are n basketball teams in the world. The ranking of these teams from the previous year is available. This year, some of these n teams played against each other and the winner of each game was determined. There were m games in total. The International Basketball Association wants to introduce a new performance criterion, called “domination factor”, defined as follows: Team i is said to “dominate” team j if we can find a chain of games such that j was beaten by a team that was beaten by a team that was beaten by a team ... that was beaten by i (observe that, according to this definition, domination can be bi-directional, i.e., i and j can dominate each other). Then, for each team i, the domination factor z_i is defined as the rank of the best team (that is, the highest ranked team according to last year’s rankings) that is dominated by team i. Describe an efficient algorithm to compute the domination factor for all the n teams.

✓ Answer

Pseudocode

```
procedure domination_factor(n, E, z)
        SCCs ← Kosaraju(n, E)
        group_max ← empty array of same size as SSCs
        for each scc in SCCs
                calculate the max z for each node in this SCC
                        store it into group_max

        dag_edges ← empty array
        for each edge in E
                convert each node in edge to its group
                if edge is not in dag_edges
                        dag_edges.append(edge)

        sorted_dag ← topological_sort(SCCs, dag_edges)
        for each edge in sorted_dag.reversed()
                if grouping_max[edge.to] > grouping_max[edge.from]
                        grouping_max[edge.from] ←
grouping_max[edge.to]

        node_maxes ← empty array of length n
        for each node
                node_maxes[node] ← group_max of this node
        return node_maxes
```

Example Implementation

```
procedure domination_factor(n, E, z)
        SCCs ← Kosaraju(n, E)
        groups ← length(SCCs)
        grouping ← Array(n)
        group_max ← Array(groups)
        for scc in 1..groups
                for node in SCCs[scc]
                        grouping[node] ← scc
                        val ← z[node]
                        if val > group_max[scc]
                                group_max[scc] ← val
        dag_edges ← []
        node_edges ← Array(groups)
        for edge in E
                a ← grouping[E[1]]
                b ← grouping[E[2]]
                if a ≠ b and [a, b] not in dag_edges
                        dag_edges.append([a, b])
                        node_edges[a].append(b)
        sorted_dag ← topological_sort(groups, dag_edges)
        for edge in sorted_dag.reversed()
                child_max ← grouping_max[edge.to]
                if child_max > grouping_max[edge.from]
                        grouping_max[edge.from] ← child_max
        node_max ← Array(n)
        for node in 1..n
                node_max ← group_max[grouping[node]]
        return node_max
```

Proof

This algorithm functions in four main steps

1. Finding SCCs and the maximum domination factor for each SCC
2. Constructing a DAG out of the SCCs
3. Traversing the DAG in reverse topological order, setting the current domination factor to be the max of itself or its children
4. Setting the domination factor for each node within each SCC to the domination factor of the SCC

All nodes within the same SCC will have the same domination factor. This is because if one node is able to beat a team of rank j, and all teams within the same SCC are able to beat each other, then they are also able to beat a team of rank j.

In the DAG of SCCs, if an SCC is able to beat another SCC, then the parent SCC will be able to beat whatever the child SCC is able to beat, by properties of the DAG. Therefore, the domination factor of an SCC is the max of the domination factors within the SCC and all the SCCs that it can beat.

Runtime

Each loop is of order n, e, e, n
So the runtime is overall $O(n+e)$

2

Prove or disprove the following statements:

a

Let $G=(V,E)$ be a directed graph. For any $uv \in E$, if some run of Depth-First-Search (DFS) on G results in $v.f > u.f$, then uv must be on a cycle.

✓ **Answer**

If for arbitrary nodes v and u with edge uv.
For arbitrary DFS, either u or v could be discovered first.

If u is discovered first, then v will finish before u as v is a child of u.
Therefore $u.f > v.f$.

If v is discovered first, v can either be an ancestor of u or not.
If v is not an ancestor of u, then by the end of the traversal of v, u will not be discovered, so it must be traversed later in DFS.
Therefore $u.f > v.f$.

If v is an ancestor of u, then u must finish before v by the structure of DFS.
Therefore $v.f > u.f$.

By exhaustion, the only possible way to obtain $v.f > u.f$ with edge uv in an arbitrary DFS is if v is discovered before u and u must be an ancestor of v.

If u is an ancestor of v and edge uv exists, then u and v must be on a loop.

b

Consider any run of DFS on a directed graph $G=(V,E)$. For any edge $uv \in E$, if there is a path from v to u in G, then uv cannot be a cross edge

✓ **Answer**

Similar to the previous question, for arbitrary nodes u and v with edge uv where v is also an ancestor of u
For arbitrary DFS, either u or v will be discovered first.

If u is discovered first, then the edge uv must be a tree edge or a forward edge, as we know v must be in u's tree and an ancestor of u.

If v is discovered first, then since u is a descendant of v, it must be part of v's tree. Therefore, the edge uv must be a back edge.

By exhaution, there are no possible ways for uv to be a cross edge given that uv exists and u is also a descendent of v

3

There are two types of professional wrestlers: “babyfaces” (“good guys”) and “heels” (“bad guys”). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an efficient algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it

✓ **Answer**

Where we modify BFS to instead return the depth of all nodes, and a list of all edges not in the BFS tree.

Pseudocode/Example Implementation

```
procedre rivalries(n, E)
        depth, unused_edges ← BFS(n, E)
        for edge in unused_edges
                if edge.from % 2 ≠ edge.to % 2
                        return false
        B ← []
        H ← []
        for node in n
                if depth[node] % 2 == 0
                        B.append(node)
                else
                        H.append(node)
        return B, H
```

Proof

Given an arbitrary BFS, starting at the root, we may assign this node to either group as there are no constraints. The depth one will all have edges connecting to the root. Since the root has been assigned to a group, the first depth must be of the other group. The same may be said about depth two and depth one, where depth two must be of the same group of the root.

More strictly:

Where d_i is defined as the set of nodes at depth i for an arbitrary BFS,

All nodes in d_{j+1} must have an edge connecting it to a node in d_j for any $j \leq l$ where l is max depth for the BFS run.

Therefore, all nodes in d_{j+1} must be in a different group than nodes in d_j, as they all have rivalries with each other.

Therefore, alternating depths in an arbitrary BFS must be in opposite groups.

We may then check all other edges and if they do not go from an even depth to an odd layer then there is no possible arragement such that there are two groups, as it becomes necessary that two people of the same group have a rivalry.

Runtime

Since BFS is $O(n + e)$
And our algorithm additionally utilizes a loop of edges and a loop of nodes, it is also $O(n + e)$.

Therefore, overall, it is also $O(n + e)$

4