ECE374 Assignment 7

Due 04/03/2023

Group & netid

Chen Si *chensi3*

Jie Wang *jiew5*

Shitian Yang *sy39*

Problem 3

3. Your job is to arrange n ill-behaved children in a straight line, facing front. You are given a list of m statements of the form "i hates j." If i hates j, then you do not want to put i somewhere behind j, because then i is capable of throwing something at j.

   (a) Give an algorithm that orders the line (or says that it is not possible) in $O(m+n)$ time.

   (b) Suppose instead you want to arrange the children in rows such that if i hates j, then i must be in a lower numbered row than j. Give an efficient algorithm to find the minimum number of rows needed, if it is possible.

(a) Solution:

Intuition: We could transform this problem as a graph, where each child is a node in the graph, and we mark the hate relationships like "i hates j" as an edge that point from i to j. Then, with performing a topological sort of the graph from left to right, we could ensure that for each hate relationship "i hates j", we would sort the node i ahead of node j in the topologically sorted sequence, and therefore meets the requirement of the question.

If the graph that we constructed could be topologically sorted (it's a DAG), we could have an order of the line successfully. Otherwise we may not have such an ordered line.

Therefore, the algorithm is:

```
GetOrder(n, Hates[1,…,m]):
    // Initialize graph
    G = Graph()
    for i ← 1 to n:
        G.addVertex(i)    // Add vertices
    for k ← 1 to m:
        (i, j) = Hates[k] // “i hates j”
        G.addEdge(i, j)   // Add edge from i to j

    if isCyclic(G):       // Check if we could topological sort
        return False      // Can’t arrange

    sorted = TopSort(G)   // Get topological sort results

    return sorted
```

isCyclic is a helper function that checks if a graph has cycles with DFS (run time O(m+n)):

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

isCyclic( G(V,E) ) {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Topological Sort Algorithm cited from lecture 16:

```
TopSort(G):
    Sorted ← NULL
    deg_in[1 .. n] ← −1
    Tdeg_in[1 .. n] ← NULL
    Generate in-degree for each vertex
    for each edge xy in G do
        deg_in[y] + +
    for each vertex v in G do
        Tdeg_in[deg_in[v]].append(v)
    Next we recursively add vertices
     with in-degree = 0 to the sort list
    while (Tdeg_in[0] is non-empty) do
        Remove node x from Tdeg_in[0]
        Sorted.append(x)
        for each edge xy in Adj(x) do
            deg_in[y] − −
            move y to Tdeg_in[deg_in[y]]
    Output Sorted
```

Run Time Analysis:

(1) Initialize empty graph: O(1)

Insert n nodes (n child): O(n)

Insert m edges (m hates relationships): O(m)

(2) Check cycle with DFS: O(m+n)

(2) Topological Sort: O(m+n)

Therefore, the runtime of this algorithm is O(m+n).

(b) Solution:

Intuition: We could modify the code in (a) to transform the order to the form of sitting in rows.

Base Case: For each node, if it doesn't have a "parent node", meaning that there's no incoming edge, it could be placed at the first row – (row number = 1).

General Case: For each node, we could obtain a list of its "parent node", the row number of the current node should be 1 + the max row number of its parents, as this child must be placed at least one row after the last one it would bully.

Therefore, it's turned into a dynamic programming problem and the following algorithm is purposed. We could use a n length table to store the row values, and the minimum number of rows is the maximum value stored in the table after iterating through all nodes.

```
GetNRows(n, Hates[1,…,m]):
    G = Graph()
    parent_dict = {}      // Dictionary of Lists
    for i ← 1 to n:
        G.addVertex(i)    // Add vertices
    for k ← 1 to m:
        (i, j) = Hates[k] // "i hates j"
        G.addEdge(i, j)   // Add edge from i to j
        parent_dict[j].add(i)     // mark j as one of i's parents

    if isCyclic(G):
        return False      // Can't arrange

    sorted = TopSort(G)   // Get topological sort results

    T = Table(1,n)    // Initialize a 1×n table to store the values
    for i ← 1 to n:
        parents = parent_dict[sorted[i]]
        if len(parents) == 0:
            T[i] = 1      // If no parents occur, sit in row 1
        else:
            T[i] = 1 + max(T[p] for p in parents)

    return max(T)
```

This algorithm has a run time of O(m+n), as the topological sorting part from (a) is O(m+n), and the for loop for dynamic programming has n loops with nearly constant time for each loop. We have the total time cost to be O(m+n).