**CIS 522: Algorithms and Complexity – HW2**

Anubhav Shankar(01951462)

Q2.) Butterfly Problem: Chapter 3 Ex 4

Ans.)

[1]. Building off the notations provided in the question stem, we have ‘n’ specimens and ‘m’ judgments. Also, we have two probable species ‘A’ and ‘B’. The specimens need to be divided into either ‘A’ or ‘B’. According to the question, it is very difficult to accurately label the species as they all look similar.

To ensure that we have correct judgments, we’ll model this problem using an undirected graph. Let the graph be called ‘B = (V, E)’ where ‘V’ are the vertices and ‘E’ are the edges.

In this case, the specimens are the vertices. An edge (e) exists between the vertices ‘vi’ and ‘vj’ if we can make a judgment.

[2]. The overall idea of the algorithm is to make a judgment for labeling the specimens and then looping through the set of judgments to ensure that it’s correct. This can be ensured by using a Breadth-First Search (BFS) algorithm.

In BFS we explore outward from a selected vertex (v) in all possible directions and add nodes one “layer” at a time. Similarly, in this case, we will arbitrarily designate a starting vertex, ‘vs’, and branch out in all possible directions. It might also be the case that the graph is not completely connected and thus, starting nodes for all the components will be needed.

[3]. Taking [1] and [2] into consideration, we will now formulate pseudocode.

For each element E of B

arbitrarily choose a starting vertex vs  and label it as A

Mark vs as “Visited”

Let set S = vs

Define Layer L(0) = vs

For i = 0 to n{

For each vertex vk in L(i) {

Consider each edge (vk, vj) leading to vj

If vj !== “Visited” // This checks the traversal. Also, if a vertex is not marked visited then it has also not been labeled.

Mark ‘vj‘ as “Visited”

If j(vk , vj) was “same”{

then label vk  same as vj // Here j() indicates judgment

Else ( judgment was “different”){

Label vk as opposite of vj

}

}

Add vk  to S and move it to layer L(i+1)

}

}

}

}

For edge e(vk , vj){

For judgement j(vk , vj){

If (j(vk , vj) == “Same”){

If the vertices have different labels{

then there is an inconsistency

}

ElseIf (j(vk , vj) == “Different”){

If (vk  and vj have the same labels)){

There is an inconsistency

}

}

Else {

The labels are consistent

}

}

}

[4]. Let us now have a general overview of the pseudocode from [3] -> let vs be the starting node. Let vk be the next node visited from vs, consider the judgment made on (vk, vs ). If the judgement was “Same” then we will label vk same as vs. If they are labeled “different” then vk is the opposite of vs.

Example: Let there be a set of specimens – (x, y) which belongs to A. Similarly, set (p, q) belongs to B.

1. Start with vertex, ‘x’ (vs ) and label it as ‘A’
2. Add vs to set S.
3. L(0) = vs
4. Let ‘q’ (vk ) be the unvisited node adjacent to x. Mark ‘q’ as visited
5. Judgment(x, q) = “Same”
6. Label ‘q’ to be the same as ‘x’
7. Add vk to set S and add to layer L(i+1) = L(1) now
8. Repeat the same steps for ‘y’ and ‘q’ till all the vertices have been covered. This happens by the very first For loop which goes through i = 0 to n and creates L(n) layers.

Let us consider the following judgments –

J1: (x,q) -> Same

J2: (p,y) -> Different

J3: (x,y) -> Different

1. Running the last For loop we will come to know that J1 and J3 are inconsistent.

[5]. **Time Complexity**: In BFS, it takes O(v + e) time to construct the graph as there are ‘v’ vertices and ‘e’ edges. Going through the list of judgments, in this case, will take O(e) time. Thus, the time complexity of the algorithm in total is O(v + e). This is also confirmed by the transitive properties of asymptotic growth rates.

Q1.) Stress Test Problem: Chapter 2 Exercise 4

Ans.)

1. The question stem asks us to consider an algorithm that grows slower than a linear time algorithm. The possible candidates are O(log n) or anything O(nk<1). As we are given k = 2 jars, O(log n) is out of the picture because this would entail us dividing the jars into smaller pieces which is not the objective.

So, let us assume that there exists an O(n0.5) algorithm. This would simplify our process as we can assume that ‘n’ i.e. the number of steps is a perfect square.

By making the above supposition, we shall drop the first jar from sqrt(n)th step and then proceed in the multiples of sqrt(n) like 2 \* sqrt(n), 3 \* sqrt(n), and so on. This also has a best-case scenario wherein if we drop the jar from the top-most rung and it survives then the loop terminates, and the algorithm stops.

In another scenario, let the first jar break when it is dropped from height x \* sqrt(n). Then it would mean that the safest rung is (x – 1) \* sqrt(n). Now, the second jar can be dropped from the height of 1 + (x – 1) \* sqrt(n). This way both jars are dropped exactly sqrt(n) times for a total of at most 2 \* sqrt(n).

Let us see the above with the help of an example below:

n = 49 -(1) [Number of steps]

k = 2 -(2) [Number of jars]

Progression of steps from which the jar will be dropped – sqrt(n), 2 \* sqrt(n), 3 \* sqrt(n), and so on -(3)

From (1), these steps will be 7, 14, 21, 28, and so on.

Suppose the first jar breaks at x = 4 then from (3) we know that this will be the 28th step. So, we can conclude that the safest step is between the 21st ((x – 1) \* sqrt(n) = (4 – 1) \* sqrt(49) = 3 \* 7)and 28th step.

Hence, we will now drop the second jar from 1 + (4 – 1) \* sqrt(49)th = 22nd step and upwards to determine the highest safe point.

This way both the jars are dropped at most 14 times.

1. In this case, we have a budget of k > 2 jars and ‘n’ rungs. This problem can be thought of as a combinatorics problem wherein we have to ‘fit’ ‘k’ things in ‘n’ positions. Given that we now have more jars we can take an approach like binary search wherein we will have to throw the jars only (log n) times. Concurrently, the time complexity of Binary Search is O(n log k) which would grow asymptotically slower than other approaches like Dynamic Programming. The binomial coefficient approach coupled with the binary search will get us the minimum number of trials to find the highest safe rung with at most ‘k’ attempts.

**Pseudocode**:

Binomialcoeff(x, n, k){

i //index pointer to move through n rungs

jars = 0 //keep track of the jars being thrown

coeff = 1

while(i <= n and jars < k):

// Argument x is the highest safe rung

Coeff \*= x – i + 1;

Coeff /= i;

Jars += coeff;

i = i + 1;

return jars;

safestrung(n, k):

first = 1; //First rung

last = k; //Last rung

// Binary Search

While(first < last):

Mid = (first + last)/2

If (binomialcoeff(mid,n,k) < k):

First = mid + 1 //First Jar did not break. We can go higher

Else:

Last = mid // Jar broke. Exit loop. We have found the highest safe rung

Return(first);

A walkthrough of the above pseudocode can be found in the JarBreak.py. To execute the driver please run main.py.

**Disclaimer**: The idea with the requisite code was referenced from [here](https://brilliant.org/wiki/egg-dropping/).