Assignment -4

Advanced Algorithms (CPT_S 515)

Name: Coral Jain
WSU ID: 011632780

1. I have k, for some k, water tanks, $T_1, \cdots, T_k$ (which are identical in size and shape), whose water levels are respectively denoted by nonnegative real variables $x_1, \cdots, x_k$. Without loss of generality, we assume that x_i equals the amount of water that is currently in T_i. Initially, all the tanks are empty; i.e., $x_i = 0$; $1 \le i \le k$. I have m pumps $p_1,\cdots,p_m$, that pump water into tanks. More precisely, a pump instruction, say, P_{A,c_1,c_2}, where $A \subseteq \{T_1, \cdots, T_k\}$, is to pump the same amount of water to each of the tank T_i with $i \in A$ (so water levels on other tanks not in A will not change), where the amount is anywhere between c_1 and c_2 (including c_1 and c_2, of course we have assumed $0 \le c_1 \le c_2$). For instance, $P_{\{T_2,T_5\},1.5,2.4}$ means to pump simultaneously to T_2 and T_5 the same amount of water. However, the amount can be anywhere between 1.5 and 2.4. Suppose that we execute the instruction twice, say:

$P_{\{T_2,T_5\},1.5,2.4}$;
$P_{\{T_2,T_5\},1.5,2.4}$.
The first $P_{\{T_2,T_5\},1.5,2.4}$ can result in 1.8 amount of water pumped into T_2

and T_5, respectively, and the second $P_{\{T_2, T_5\},1.5,2.4}$ can result in 2.15 amount of water pumped into T_2 and T_5, respectively. That is, the amount of water can be arbitrary chosen inside the range specified in the instruction, while the choice is independent between instructions. Now, let M be a finite state controller which is specified by a directed graph where each edge is labeled with a pump instruction. Different edges may be labelled with the same pump instruction and may also be labeled with different pump instructions. There is an initial node and a final node in M. Consider the following condition Bad $(x_1, \cdots, x_k)$:

$x_1 = x_2 + 1 = x_3 + 2 \wedge x_3 > x_4 + 0.26$.

A walk in M is a path from the initial to the final. I collect the sequence of pump instructions on the walk. If I carefully assign an amount (of water pumped) for each such pump instruction and, as a result, the water levels $x_1, \cdots, x_k$ at the end of the sequence of pump instruction satisfy Bad$(x_1, \cdots, x_k)$, then I call the walk is a bad walk. Such a walk intuitively says that there is an undesired execution of M. Design an algorithm that decides whether M has a bad walk. (Hint: first draw an example M where there is no loop and see what you can get. Then, draw an M that is with a loop and see what you get. Then, draw an M that is with two nested loops and see what you get, and so on.)

Solution:

This problem can be considered to be similar to pigeon and holes problem, where more than one pigeon can choose the same hole and may collide. For this question, we can solve this by using the DFS traversal method considering the graph for M, given in the question. It can be used in the similar fashion like we did in the other graphs. While traversing the graph, the concept of universal hashing can be used to give the tanks a set of values. The set of values can help us define the hash functions for the different tanks that is pumped with the help of pumps.

However, this is to be realized that there are a number of hash functions, but they are not necessarily unique. This is because of the reason that if same instructions are given to more than one pump, it leads to the concept of collision. For example,

For two pump instruction, same tanks let's say T_1 may cause collision. In order to avoid this, we can use the concepts of collision handling such as linear probing or double hashing and get away from this problem.

Therefore, first we start with traversal of the graph with initial node. Then we move along edge to edge as per the pump instruction since every edge has a pump instruction and all the edges have pump instructions. Therefore, after we have traversed the whole graph and reached the end node, at the end we need to determine whether the walk that we have considered is a *Bad* walk or not. Also, in our case if we are using the concept of hashing, it becomes easier to identify a particular and this will lead our program to work fast because of better time complexity.

Now, for determining the walk is *Bad* or not, we can use dynamic programming. For dynamic programming, we can use the concepts of optimal substructure or overlapping subproblems in order to evaluate the *Bad* condition to be true or false.

2. The word bit comes from Shannon's work in measuring the randomness in a fair coin. However, such randomness measurement requires a probability distribution of the random variable in consideration. Suppose that a kid tosses a dice for 1000 times and hence he obtains a sequence of 1000 outcomes $a_1, a_2, \cdots, a_{1000}$ where each a_i is one of the six possible outcomes. Notice that a dice may not be fair at all; i.e., the probability of each outcome is not necessarily 1/6. Based on the sequence only, can you design an algorithm to decide how "unfair" the dice that the kid tosses is?

Solution:

Method 1: <u>Using Standard Deviation</u>

One of the best ways that I think the problem can be solved is by using the concepts of standard definition. We should agree upon the fact that for an unfair dice will have non uniform distribution of outcomes. In other words, this means that there is in no way a probability of 1/6 possible for each side coming up when rolling the dice. If we calculate the average/mean of the distribution of outcomes of a dice comes out to be 3.5. As per the Central Limit Theorem, the sample mean of the set of outcomes of the dice should converge towards this value.

The mean, ideally, for a given number of samples should turn out to be 3.5 which will be the mean of the sample means and the standard deviation of 1.707 is obtained in ideal case which will be the true standard deviation.

Now in our experiment, are given the we have following number of samples defined by:

$a_1, a_2, \cdots, a_{1000}$

Hence, we will structure these 1000 samples in normal distribution space. Thus, if we call the distribution as x', it can be expressed as $N(m, \frac{\sigma}{\sqrt{n}})$ where m is the mean that we discussed above, σ is the standard deviation and n is the number of samples that we are considering. In our case,

m = 3.5

$\sigma = \sqrt{35/12}$

n = 1000

Therefore, standard deviation is defined as a quantity calculated to indicate the extent of deviation for the group as a whole, we shall consider that the unfairness of the dice would be based on the value of standard deviation obtained from the 1000 samples taken for our experiment. Now, since we have the value of m as 3.5, we shall expect that m', the mean of our experiment.

Now, we must know how to calculate this new mean m'. If we observe carefully, the mean is dependent upon the sum of the outcomes (in terms of positive integers) obtained after rolling a dice. Since the m is 3.5, it can be considered on an average that the sum of the values of the outcomes is 3500, which is difficult for any sample element say a_i to have an average of 3.5 in an event. Thus, for an unfair dice, we can consider that the values of this mean do not vary a lot from the mean of a fair dice. Therefore, we shall consider an unfair dice only if the value is 2σ away from the mean. That is, for a normal distribution curve, the values lying between $m + 2\sigma$ and $m - 2\sigma$ are not considered for an unfair dice. Hence, if the sum in our case 3661, the value does not lie within the specified range and the dice is unfair.

Thus, while writing the algorithm, we can calculate the sum from the total number of samples and calculate the standard deviation. If the standard deviation lies within the specified range, we can say that out dice is fair or unfair based on normal distribution curve to determine on which side the sample has majority.

Method 2: <u>Using Shannon Entropy principle</u>

In this method, we can calculate the entropy of an event. For example, the formula for Shannon entropy is given by $\sum -plogp$ which can be used to calculate the amount of entropy. The entropy gives us the number of *bits* that will be used for storing the information. The entropy for fair dice is given by

$$H(x) = -\sum_{i=1}^{6}(1/6)\log_2(1/6)$$

which is the amount of entropy that is contained within a dice for which the probability of any outcome from 1 to 6 comes out to be 1/6.

Therefore, the above value comes out to be 2.59. As per the Shannon entropy knowledge, it can be said that for the values of H(x) less than 2.59, the dice should be considered to be a fair dice. Otherwise, for the values of H(x) greater than 2.52, the dice should be considered to be an unfair dice. This can be verified in the similar fashion as in out Method 1 and can be used in the algorithm to check whether the dice that we are using is fair or not.

One of the algorithms that we can use in this case is the Decision tree algorithm. This algorithm works in a way in which we draw the trees based on different outcomes and questions are asked in a classification manner. Each question is represented in a way which is stored in *bits* which we talked about earlier in this method. Every *bit* stores a message or a question that we ask which going down every tree. The average of these questions or bits is nothing but the entropy that we obtain for a dice that we calculated above. As said above, the value of that entropy will determine unfairness of the dice. Hence, based on the questions asked, we can calculate the average number of bits and hence justify unfairness.

3. In below, a sequence is a sequence of event symbols where each symbol is drawn from a known finite alphabet. For a sequence $\alpha = a_1 \cdots a_k$ that is drawn from a known finite set S of sequences, one may think it as a sequence of random variables $x_1 \cdots x_k$ taking values $x_i = a_i$, for each i. We assume that the lengths of the sequences in the set S are the same, say n. In mathematics, the sequence of random variables is called a stochastic process and the process may not be IID at all (independent and identical distribution). Design an algorithm that takes input S and outputs the likelihood on the process being IID.

Solution:

In this question we are given that a sequence given by $\alpha = a_1, a_2, a_3 \ldots, a_k$ which is drawn from a finite set S of n sequences which has variables $x_1, x_2 \ldots\ldots, x_k$. After we have drawn a sequence from a set S, the question says that the process may or may not be IID at all. This means that for our question, we need to determine with the help of an algorithm whether the variables x (1), x (2), ……..., x (n) is an IID or not. For this we need to know the properties of an IID. The identically distributed and independent variables are the variables which are identically distributed and therefore have similar probability distribution and are independent of the probability of other variables. For example, in our example, the sequence can be of an event x (0) of flipping a coin. When you flip a coin again for event x (1), the probability of getting a head or a tail is 0.5 and if you flip the same coin again, the probability is 0.5 again and therefore x (0) and x (1) are IID.

It can also be derived from the same probability distributions of two events that both of them will have same variance or standard deviation and mean. Therefore, for a sequence α retrieved from S, we can verify whether it follows the properties of IID or not and justify this in our algorithm.

However, it is also to be noted that for a process to be an IID if every random sequence in that stochastic process is an IID. If we check this question through the lens of Markov chain, we need to check first that whether this stochastic process is a Markov chain or not. If it is a Markov chain, for a process to be IID is not possible since in a Markov chain the probability of a random variable X depends on the previous time and is therefore is dependent. This is given by $P (X_{n+1} \mid X_n, X_{n-1}, X_{n-2} \ldots..) = P (X_{n+1} | X_n)$. However, for an IID this is impossible as all the variables for current time are independent of their values in the previous time.

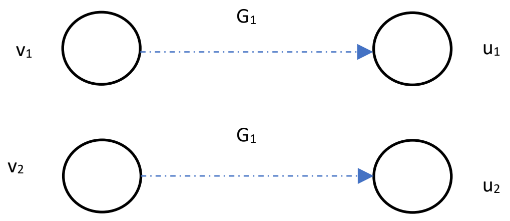
Therefore, first need to determine whether the given stochastic process is a Markovian chain or not. If it is, IID is not possible. If it is not, we need to check the other properties in order to determine the same.

4. Let G_1 and G_2 be two directed graphs and v_1, u_1 be two nodes in G_1 and v_2, u_2 be two nodes in G_2. Suppose that from v_1 to u_1, there are infinitely many paths in G_1 and that from v_2 to u_2, there are infinitely many paths in G_2 as well. Design an algorithm deciding that the number of paths from v_1 to u_1 in G_1 is "more than" the number of paths from v_2 to u_2 in G_2, even though both numbers are infinite (but countable).

Solution:

Method 1: **Backtracking**

One of the methods for this type of problem could be basic Backtracking method. Since it is given that there are infinitely but countable paths in the graph for G_1, we can traverse from one node v_1 to another node u_1 by DFS traversal method in order to count the number of paths from one node to another. In the same way, we can also use the same technique and traverse along the two nodes v_2 and u_2 in order to count the number of paths in G_2.



In the algorithm, we can write a function which will do DFS traversal. It will start traversing from the source towards the destination. If one node is reached from a source node along one simple path, it is checked whether it is the desired node or not. If it is the desired node, it is counted as 1 path. There will be a counter that will store the number of paths in a graph. If it is not the desired path, there is a recursion step that will let the graph traverse along different path. This will be done until a node is reached. The counter will keep track of the number paths. The time complexity.

The problem with this solution is that, since it is given that there are infinitely many paths and since the time complexity of such an algorithm is O(V+E), where V is the number of vertices and E is the number of edges. Since, a path can be of any length, the number of vertices along which the traversing is done will lead to huge amount of time being taken to run this algorithm. Therefore, this approach seems to be very Naïve for our problem.

Method 2: Using Perron Frobenius theorem

In this another method, Perron Frobenius theorem can be used to calculate the number of paths between two nodes in a graph. The Perron's theorem can be quickly defined as follows:

It basically talks about the matrices that have positive or non-negative values in a system. It also says that if you come across a matrix that has all the values of its elements as positive, then there exists a unique real eigenvalue that is greater than 0. This term has been derived from Perron number of mathematics. It also justifies that the eigenvectors for that matrix need to necessarily have positive values. In our case, we define the eigenvalue as λ_p. Every eigenvalue here would be less than λ_p.

Now when we apply this theorem to our program, we can get the desired result. For example, we have one graph G_1 which consists of n number of nodes. Now when we draw the adjacency matrix of that graph, it will look something like this (example).

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

The adjacency matrix is made on the basis of the connectivity of nodes between the graph. If two nodes i and j are connected, we call that particular element as A_{ij}. Now if we multiply A with A, just like we do in a Markov chain, we get A^2. Similarly, when if we do this k times, we'll have A^k matrix. Here $(A^k)_{ij}$ will give us the number of paths of the graph with the length k.

Since in our case value of k could be large, A_k can be expressed in terms of the eigenvectors and λ_p as:

$A^k = \lambda_p(1^T y)\ x(y/1^T y)^T$

where λ_p is the eigenvalue and x and y are defined as right and left eigenvectors.

The above term gives us a lot of information regarding the graph. Some of the information that we need is as follows:

Number of paths of length k α $\lambda_p(1^T y)$

Number of paths of length k α $xy/1^T y$

Similarly, we can count the total number of paths that are not specifically of length k and for other for graph G_2 as well.

From the above terms, we know the proportionality of the number of paths. Therefore, we can say that for a given directed graph, if we identify the adjacency matrix and the corresponding eigenvalues and eigenvectors, it is easier to compare the number of paths in the graph without having to traverse along the graphs and waste time and resources. With the introduction of this method, it is more of a mathematical computation in order to compare which graph has a greater number of paths.