A Problem in Optimization

Summer Project

Formal Problem Statement

Given K bitonic discrete functions $f_1(\tau_1), \ldots, f_k(\tau_k)$; $\tau_i \in \mathbb{W}$, we need to find the sequence $A = \{a_1, a_2, \ldots, a_T\}$, where $a_i \in \{0, 1, 2, \ldots, K\}$, which maximizes the function

$$R = f_{a_1}(\tau_{a_1,1}) + f_{a_2}(\tau_{a_2,2}) + \ldots + f_{a_T}(\tau_{a_T,T}) = \sum_{a_i \in A} f_{a_i}(\tau_{a_i,i})$$

where $\tau_{a_i,i}$ denotes the value of τ_{a_i} at the i^{th} time instant.
$f_0(\tau_0)$ is trivially defined to be zero for all τ_0.

The above is equivalent to sampling one of the K given functions at every instant of time, for T instants of time. Consider the t^{th} time instant. Let t_i be the largest number such that $t_i < t$ and $a_{t_i} = i$, and zero if no such t_i exists. Then, the parameter τ_i corresponding to each f_i at the time instant t is given by:

$$\tau_i = t - t_i - 1$$

Intuition

The problem statement can be visualized in the following manner:

- There are K arms present. Pulling an arm results in a reward, which is a function of τ. The rewards associated with each arm are a known function of the parameter τ.
- There are T time instants, in each of which we may choose to pull any one of the k arms or none at all.
- The parameter τ associated with a particular arm is representative of the time elapsed since the arm was last pulled. In other words, pulling the i^{th} arm resets τ_i to 0, while increasing the τ values of all the other arms by one.
- We need to find the sequence in which the arms must be pulled to obtain the maximum total reward.

Variations

Possible variations in the problem statement are:

- The maximum delay between two pulls of a particular arm is given. Thus, there is an upper bound on the value of τ.
- The maximum number of times an arm may be pulled in the T time instants is given.

Potential problem-solving strategies

The following are potential strategies to devise an algorithm to solve the problem:

- Brute Force
- Dynamic Programming
- Divide-and-Conquer: If the problem can be solved by finding solutions to a number of subproblems, a divide and conquer algorithm can be devised.
- Incremental solution: This can be applied in one of two ways. The first is if we are given the optimum sequence A for K arms, which we are able to use to obtain the optimum sequence for k+1 arms. An alternative approach would be to increment the time instead of the number of arms.
- Linear programming

Brute Force Algorithm

- Each of the T time slots can be filled in (K+1) ways, i.e. either one of the K arms is pulled or no arm is pulled.
- The algorithm exhaustively checks the reward obtained by each of the $(K+1)^T$ possible sequences, and returns the one(s) with the greatest reward.
- Its time complexity is $O((K+1)^T)$.
- In the small-K, large-T domain, the time complexity of the algorithm becomes enormous. The brute force algorithm implemented in python returns in satisfactory time for small k(less than 6) and small T(less than 15).

Dynamic Programming: Introduction

Dynamic programming can be used to obtain the optimum sequence, by using a combination of the bottom-up approach and by storing information for future use.

State and Action:

- Define *state* to be a K-dimensional vector containing the current τ values of each of the K functions.
- Each time instant has an initial *state*, before an *action* is taken in that time slot. This *action* will determine the final *state* for this time instant, which is also the initial *state* for the next time instant.
- The problem is viewed as maps between the initial *state* and the best course of *action*, at each time instant.

Dynamic Programming: Procedure

- Start building the sequence from the last time slot.
- Find all possible initial states before the T^{th} time instant. The best *action* to take corresponding to each state is the one which gives the maximum reward on that pull.
- Store the above information as ordered triplets of the form (initial state, sequence(s) to play, maximum reward) for each of the initial states possible.
- Find all possibles initial states before the $(T-1)^{th}$ time instant. Each course of *action* that can be taken now will lead to an initial *state* for the T^{th} time instant. The best course of action that needs to be taken for this time instant is already known and stored.

Dynamic Programming: Procedure(cont.)

- For each of these initial states, find the course of *action* that will give maximum reward, which is the sum of the reward obtained in this time instant and the maximum reward that can be obtained from the resulting final state.
- Store this information as above.
- Perform the above procedure for the $(T-i)^{th}$ time instant, using the data stored about the $(T-i+1)^{th}$ time instant, for $i = 1, 2, .., T$.
- The initial state before the first time slot is the K-dimensional zero vector. When the procedure is complete, we obtain all the optimum sequences and the reward they give.

Dynamic Programming: Finding valid states

There are a total of t^K states possible before the t^{th} time instant. However a large number of these states are not attainable due to the nature of the problem. Finding only the attained states will save a significant amount of computation and space.
The vectors in the attainable state space are constrained by the following rule:

- For valid initial states before the t^{th} time instant, no two elements in the state vector can be equal unless they are both equal to $t-1$. For example, before the 5^{th} time slot, (3,3,4) is not a valid initial state whereas (4,4,3) is.(attained by playing the sequence $\{3,0,0,0\}$).

Dynamic Programming: Finding valid states(cont.)

Computational savings:
Upon applying the above constraint, the number of states for which computation needs to be performed for the t^{th} time slot reduces from t^K to:

$$\sum_{n=0}^{K} \frac{{}^{t-1}C_{k-n} k!}{n!}$$

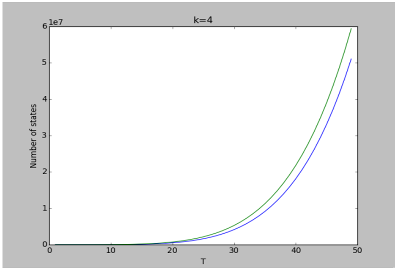
Thus the total number of states for computation needs to be performed for the a sequence of with T time slots reduces from $\sum_{t=1}^{T} t^K$ to:

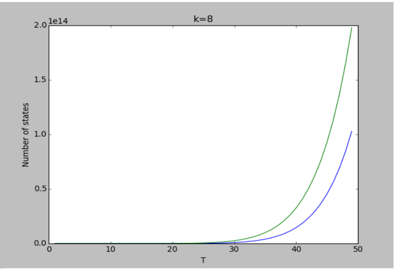
$$\sum_{t=1}^{T} \sum_{n=0}^{K} \frac{{}^{t-1}C_{k-n} k!}{n!}$$

Note: ${}^{n}C_{m}$ is defined to be zero for $m > n$

Dynamic Programming: Finding valid states(cont.)

This expression also increases exponentially with T, but nevertheless represents computational savings. The savings achieved become more significant with larger k, as shown in the graphs below.





For k=8 and T=50, the computation required is almost halved!

Dynamic Programming: Problem Variations

Both of the aforementioned variations to the problem can be implemented in the dynamic programming approach.

- Maximum Delay Constraint: If the maximum permissible delay between two arm pulls of Arm i is D_i, then redefine $f_i(\tau_i)$ to be $0 \; \forall \; \tau_i \geq D_i$.
- Maximum Number of Pulls Constraint: If there is a constraint on the maximum number of times an arm can be pulled, then there comes a need to store more data about each state. It would be necessary to store the *action* that should be taken an arm cannot be played. Thus, the (K+1) alternatives will be stored for each initial state. When using this data, we need to check which arms are available for use to us when adding up the rewards.

Dynamic Programming: Limitations

- The number of states for which computation needs to be done becomes very large at large T, even for a small value of K such as 5. The space requirements become very large. For example, for K=5 and T=100, nearly 1.5 billion states are considered.
- The time complexity of the algorithm is $O(T^{K+1})$. In the small-K, large-T domain this is significantly better than the Brute force approach. But it is still exponential in K. Even for relatively small K(5 or 6), the time taken for algorithm run increases very quickly. The program written works in reasonable time for $T < 100$ and $K < 5$.