PRIORITIZATION OF HOSPITAL ORDERS CONSIDERING CANCELLATION PROBABILITIES

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science, Industrial Engineering

---

by

ARTEM GHAVRISH

Dr. Mustafa Sir, Thesis Supervisor

JUL 2012

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled:

PRIORITIZATION OF HOSPITAL ORDERS CONSIDERING CANCELLATION PROBABILITIES

presented by Artem Ghavrish,

a candidate for the degree of Master of Science and hereby certify that, in their opinion, it is worthy of acceptance.

Dr.Mustafa Sir

Dr. Esra Sisikoglu

Dr. Kalyan Pasupathy

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Mustafa Sir and Dr. Esra Sisikoglu for introducing me to the exciting field of Industrial Engineering and Operations Research, encouraging and guiding me through my graduate studies and this research.

I would also like to thank Dr. Kalyan Pasupathy for formulating the problem discussed in this thesis and providing invaluable support with his expertise in healthcare management and operations.

I am very grateful to Rhonda Thacker, the Lab Supervisor of the University Hospital at University of Missouri-Columbia, for sharing her practical insight on the laboratory operation.

And the last but not the least I would like to thank my beloved wife Ksenia Gavrysh for her support, understanding and inspiration that she is bringing to my life. This would be impossible without you.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

In a hospital laboratory premature cancellations of the test orders are driving up the laboratory running cost as they result in wasted resources and time. A test order that is cancelled during its processing results in waste of laboratory resources (labor, materials, etc.). We develop a queuing heuristic that prioritizes orders based on their cancellation probability and tardiness. The cancellation probability is a function of order attributes (type of test, the doctor requested the test, urgency, etc.) and can be estimated from historical data. We compare the cost and timeliness of lab orders from our prioritization scheme against those from a first-come-first-served queuing system. Having no control over the nature of the tests ordering (arrival) and their cancellations (departure) we model them as random processes that follow certain probability distributions and include them into a Dynamic Programming model that attempts to minimize the expected total running cost by picking the "right" test to process at every decision epoch.

Chapter 1

Introduction

The hospital laboratories are playing a crucial role in healthcare processes. Doctors rely on the timely and accurate results of the laboratory analysis to understand the patient's condition and assign an appropriate treatment. As in the majority of management problems a hospital laboratory problem has the conflict between the quality and the cost. In addition to the quality indicators (the timeless and the accuracy) the running laboratory cost has to be kept to its minimum.

A considerable part of any running costs constitute the inefficiencies connected to the resources spent on the unfinished product. In the case of a hospital laboratory these "unfinished products" are the laboratory test order results that are no longer required or the test orders are lost prior to getting the results due to various technical and human factors. In fact, doctors do cancel the laboratory test orders quite often. Except for the cancellation, the test can be lost in the process because the specimen was of insufficient quality or quantity, labeled incorrectly or lost in transport [1]. In our analysis we do not distinguish between these factors and for the sake of simplicity

call any of such unfinished tests an "order cancellation".

In reality laboratory test orders cancellations pose a serious problem inflating the laboratory running costs and the cost of healthcare services in general. Out of 130,000 test orders coming to the laboratory of the University Hospital at the University of Missouri-Columbia (307 beds) per month, on average about 8000 orders are cancelled.[1] This constitutes about 6.2% as the average cancellation rate to comparing to the industry standard of 5%.

The cost of hospital laboratory varies from about $60 for the cheapest urine test to several thousands of dollars per test if it has to be sent to an external state or national laboratory [2]. Of course, in most of the hospitals the majority of the tests are processed internally but even in this case the average cost of the prematurely cancelled orders can reach $8000 \times \$60 = \$480,000$ in a single hospital per month. As we explain in the Chapter 2 the medical literature in general does not see this as a separate problem and treat it as one of the "laboratory errors".

Of course, due to stochastic nature of many processes that trigger the cancellations it is not possible to completely eliminate the cancellations but every attempt to decrease their rate should be rewarding.

In this research we have attempted to analyze the situation from a hypothetical standpoint of the laboratory manager whose only decision is in what sequence to process the incoming test orders. In other words, every time when the laboratory equipment becomes available for the next test the manager picks a test order out of the tests waiting to be processed (if there is any).

In reality, nowadays the manager is not concerned with such a decision at all. The test orders are coming in at least three different priority classes with the higher pri-

ority order requiring to be processed sooner than the other priorities. Test orders of the same priority are processed in the First-In-First-Out or FIFO sequence (an order that has come first is processed first within the same priority class). This is the operation policy of the laboratory of the University Hospital at the University of Missouri-Columbia [2].

Here we approach the process analysis from several different stand-points. At first, we develop a simple heuristic that gives sub-optimal solutions that are generally better than the performance of a FIFO queue (we have developed and used a simulation tool to demonstrate this). Finally, we model the laboratory operations within a Dynamic Programming (DP) framework and explore several alternative ways to solve it. As we will explain in Chapter 4, the DP model is (as in most cases) affected by the classical "Curse of Dimensionality" and needs some future research on approximation solution methods.

The thesis is organized as follows. Literature review in Chapter 2 describes solution approaches to relevant problems developed in the literature. In Chapter 3 we analyze a case when the laboratory has just one order to process and no time limits. The decision is when to start the processing so that the total cost including possible loss from the order cancellation would be the lowest. Next, we analyze a real queue situation with multiple test orders waiting for processing. In this case, we decide what test order to process from those waiting in the queue to minimize total running cost. This results in a simple decision rule. We have also developed a simulation model to demonstrate that the decision algorithm indeed provides better solutions comparing to the conventional FIFO approach.

In Chapter 4, we develop a Dynamic Programming model that incorporates two ran-

dom processes (arrival and cancellation) in a finite planning horizon formulation. We explore two conventional methods of the model solution and demonstrate that development of a comprehensive solution *approximation* method is crucial for this type of problems.

Chapter 2

Literature Review

The question of the healthcare inefficiency has been widely discussed in the medical and healthcare management literature. In their work Hamid and Pasupathy [1] (this thesis was inspired by that paper) have formulated the problem of a laboratory test orders premature cancellations. They have mined a large data set which resulted in several classifications of the test order cancellations by the reasons, doctors, patients and elapsed times.

Many other authors in the medical literature were looking at the reasons of the cancellations presumably because it is related to the quality of healthcare. Except for a hospital laboratory, cancellations can happen in pharmacies, during surgeries, various non-intrusive therapies and other medical procedures. For example in the paper of Wilton *et.al* [3] a decrease in cancellation rate of surgeries was demonstrated as a result of implementation of certain pre-surgery procedures.

In most of the medical literature the cancellations are not viewed separately from the other medical errors. In this manner Bonini *et.al* [11] and Carraro and Plebani [12]

classify the laboratory errors by the phases they were made (pre-analytical, analytical and post-analytical). The paper of Lippi *et.al*[13] emphasize a patient identification error as one, that can have very dangerous consequences (wrong blood transfusion or drug administration) but still happening in the every day hospital operation despite all the modern technologies such as bar-code wrist identifiers, etc. According to their research majority of the laboratory errors (62-68%) are happening in the pre-analytical phase (patient misidentification, hemolyzed sample, etc.) or, in other words, before the samples enter the laboratory. About 23% [12] are happening during the post-analytical phase and only 15% belong to the analytical phase (the actual result of the laboratory misconduct). In this classification the test orders cancellations could be seen as a special category of the pre-analytical laboratory errors. According to the Table 1 of the Carraro and Plebani paper [12] two of the subcategories in the pre-analytical phase could include the test order cancellations: the "Request procedure error" (7.5% of errors) and the "Physician request order missed" (1.9% or errors). If this is true then the cancellations cold be responsible for up to 10% of the laboratory errors.

Unfortunately, except the Hamid and Pasupathy paper [1] we could not find medical literature that would look specifically at the laboratory test orders cancellations. Their results suggested that there could be some differences in the cancellation frequency and timing that might be related to the other factors, than the random errors in a hospital operation. They have found that, for example, some doctors are more likely to cancel their test orders prematurely than their colleagues others or sometimes, the difference in the cancellation frequency was correlated with the patient floors (*e.g.* pediatric floor clearly had more cancellations than adult patient floors),

type of test ordered, time of the day, etc.

It is yet to be clarified whether the differences in the cancellation are *caused* by the different doctors and patient floors or it is just a coincidence that the cancellation rates sometimes *correlates* with them. Therefore, to avoid the classical confusion between the correlation and causation we took the reasons for cancellations out of the scope of this thesis and assumed the cancellations to be homogenized events that are happening because of some unknown reasons. At the same time we had to keep diversity of the cancellation probability distribution parameter over the time because of the two reasons. First, it helps to better represent the real hospital operation and, second, it could be shown that if all the test orders had the same cancellation probability distribution, the First-In-First-Out processing would be optimal. In effect, the reservation that we want to make here is just that the different test orders types are, at this stage, meaningless and cannot be linked to individual doctor performances or any other factors (although this could be the goal of some future investigations).

The question of optimal procession prioritization or scheduling is also one of the central to Industrial Engineering and Operations Research literature. Some first deterministic scheduling problem have been already described by G.B Dantzig [14] as an illustration for his Simplex method performance.

Soon, purely deterministic scheduling formulations have been found insufficient for real-life applications and somehow "uninteresting". A number of researchers have introduced various levels of stochastic processes, which gave rise to such developments as Markov Decision Making, Queuing Theory and Brownian Control [15].

Due to their sequential decision making the prioritization (scheduling) problems in

the relevant literature are mostly formulated as Dynamic Programming models. The introduction of random variables made the scheduling optimization formulation the problems harder to solve because of the exponential increase in the state-action space also known as "Curse of Dimensionality".

We can separate two major approaches to solving such problems. Both of them include approximation of the DP model using different simplifying assumptions and relaxations. The solutions of these approximations could be used directly (as the DP solution) or indirectly (as upper or lower bounds to measure the gap in optimality). The first approach utilizes discrete models and some of the recent examples of that could be found in the papers of Patric *et.al* [6] and Erdelyi and Topaloglu [16]. The second approach is using continuous approximation models (mostly Brownian Control) and is used in works of Rubino and Ata [4] and Kim and Ward [17].

Patric *et.al* [6] investigates a problem of "dynamically scheduling multi-priority patients to a diagnostic facility". They were able to approximate the Dynamic Programming model by a related Linear program and come up with simple scheduling policies. In their working example they have used three different patient types but the formulation allows for increasing that number to much bigger values. Unlike in our problem of the test orders cancellations, the only random variable was the number and the time of the patient arrivals. Besides, the Linear program that they have used in the DP approximation, was found empirically as a result of rigorous experimentation with solutions, which makes it difficult to use their findings directly.

Erdelyi and Topaloglu [16] have based their research on the previously mentioned paper but have slightly changed the formulation of the DP state to allow for easier Linear Programming approximation. They have developed an interesting approxi-

mation technique that we are going to investigate in our future research. As in the previous paper the limitation of only one random process (arrival) was critical to a direct application of their methods to our test order cancellations problem.

In their paper Rubino and Ata [4] developed and tested a Dynamic Control policy for a make-to-order production system. In a very similar manner to our problem they model their system to include three random processes: arrival time, service time and cancellation(abandonment) time. All there processes were assumed to have exponential probability distribution and the authors formulated a continuous-time Bellman equation to solve it through an approximation of a Brownian control problem reduced to a workload formulation.

Kim and Ward [17] have chosen an similar approach and in addition payed specific attention to the abandonment (cancellation) process. Unlike the Rubino and Ata, they did not make the assumption of constant abandonment (cancellation) rate which makes it closer to reality. It is somehow intuitive to see that the cancellation probability of the test orders is changing throughout the order's life.

To the best of our knowledge the last two papers are the most relevant to modeling of the test order cancellation problem. Although, the complexity of the continuous-time formulation and solution of the differential equations that result from the Brownian control formulation, seems to be not an ordinary obstacle. Besides the limited number of order-types is oppositional to our view on the test order cancellations problem. The formulations and solution methods of Rubino and Ata [4] and Kim and Ward [17] suggest that increase of the test order types number will potentially increase the complexity of the resulting solution methods up to the point, where no solution could be found.

Contribution of this thesis is analysis of the laboratory test order cancellation problem and development of the model that includes at least two random processes (arrival and cancellation) and could accommodate a big number of order types without necessarily exploding the solution complexity. We are not focusing on finding an exact solution method for our model, which is going to be the goal of our future research. We have chosen to first develop a discrete-time Dynamic Programming model (presented in details in the Chapter 4) and then look at two conventional methods to solve it. We also demonstrate that the conventional solution method are not efficient for any realistic-size problem instance and a further research is needed to find a practically usable solution method.

Chapter 3

Hospital laboratory operation analysis and a simple prioritization heuristic

3.1 Single Order case analysis

In this section we analyze a case with only one test order in the system (waiting to be processed). Particularly, here we are trying to answer several important questions about the test order characteristics:

1. How to interpret and model the probability of the cancellation throughout the lifetime of an order?

2. How to interpret and model the cost of overdue laboratory results and their (negative) consequences on the healthcare processes?

3. When is the optimal time to start processing an order in the system depending

on the order's characteristic?

To answer the first question let us remember the original idea that the probability of cancellation of a test order is changing over its life-time with its minimum when it is issued first and its maximum at a certain time right before at or right after its due-time. The real cancellation probability (CP) distribution function over time is believed to be of a very complex nature and therefore difficult to model. Therefore we had used a simple distribution function as an approximation to the cancellation probability distribution function (PDF).

A simple analysis of the "elapsed time before cancellation" data [1] reveals that about 75% of the test orders are cancelled in the first six hours after their issue (see Fig.1). This might be explained by various factors but in this paper we simply assume that doctors normally are more concerned about the test immediately after issuing the order. Therefore, they are more likely to change their minds and cancel the order soon after placing it. Given this observation (the frequency of the cancellations decreases exponentially with the time elapsed but never reaches 0) we figured that the best approximation function for the cancellation PDF would be an Exponential probability distribution that is widely used for similar purposes due to its simplicity and a relatively good fit.

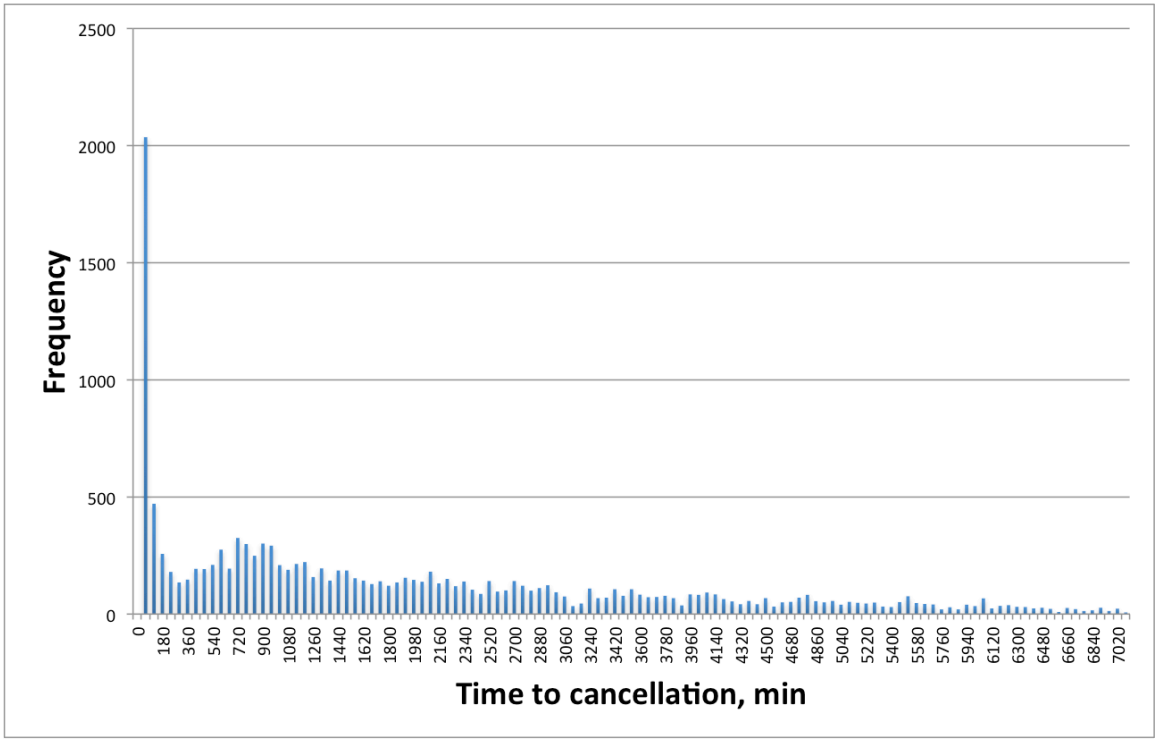

Figure 3.1: The cancellation times frequency distribution

There are also some other probability distribution functions that can be used to model the cancellations. For example Weibull PDF[7] was shown to be one of the best known approximation for failure processes in general according to the research of Zhang *et.al*[8]. Indeed, Weibull might be a better probability distribution function to use in the cancellations model but for simplicity we decided to stay with Exponential PDF. The data distribution shape also suggested the Exponential PDF. Besides there is a lot of literature demonstrating subtle relation between Exponential and Weibull PDFs and the fact the the former might be just a realization of the later.

The second question is more challenging. How do we model or even quantify the possible negative consequences of a delay in delivery of the test results? Intuitively, the larger the delay time is, the higher the expected "cost" in terms of the patient

health. As it says the paper of Lippi *et.al*[13]: "There is an evidence that laboratory errors, in general, might impact on patient care producing serious harm, with a risk of inappropriate care and adverse events ranging from 6.4% to 12% of total errors."(page 146). However, it must also depend on many other individual factors (e.g. - how fast this late cost is growing, etc.) which cannot be generalized by a simple function. While this is a very important question, it was not the focus of this thesis, therefore, for illustration purposes we choose a simple cost function to account the cost of tardiness. This simple function approximated the "late cost" or "Tardiness" as a squared time-period of the delay - difference between the results delivery time and the due-time of the order. Of course, only positive difference is considered as the Tardiness = 0 for any time $\leq$ due-time.

Finally, the answer to the last question of what is the optimal time to start processing a test order we used the assumptions that the cancellation distribution is approximated as an Exponential probability distribution and the Tardiness is represented by a squared positive difference of the results delivery time and the order due time.

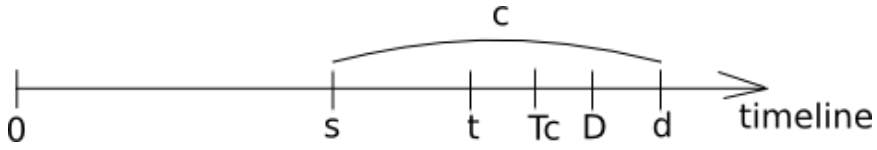

Figure 3.2: A single test order time line

In general a test order has the following parameters:

- s variable starting time of the test order processing;
- t current system time;
- T_c random variable representing the time the test order is cancelled;

- D due-time of the order;
- c - the order's processing time (elapsed);
- d the time of the order results delivery;
- λ - the parameter of the order's cancellation pdf (Exponential);

As we are minimizing the running cost of the laboratory we have identified two sources of potential "cost" – Tardiness and the resources wasted on a cancelled order. We call the later cost Wasted Work and model it as an expected time of processing the order until the cancellation multiplied by a unit-time cost of laboratory resources. Thus, the two objective functions are:

- Tardiness:

$$g(s) = \begin{cases} (s + c - d)^2, & \text{if} s + c \geq d \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

- Wasted Work:

$$h(s) = E[T_c - s] = E[T_c] - s = \Big(\int_c^{s+c} t \times p(t), dt \Big) - s$$

where $p(t)$, the order's probability of being cancelled is given by the Exponential probability distribution function:

$$p(t) = \lambda e^{-\lambda t}$$

After integrating the Wasted Work becomes:

$$h(s) = \frac{(e^{\lambda c} - \lambda c - 1)e^{-\lambda(s+c)}}{\lambda} \tag{3.2}$$

To handle multiple objectives, we use a pre-specified coefficient

$\alpha \in [0 : 1]$ to combine them into one objective function. To get the cost values consistent in both objective functions we also have to multiply their values by unit costs of Tardiness and Unnecessary Work respectively. So the final objective function is:

$$\min_s \alpha g(s)k_t + (1 - \alpha)h(s)k_w \tag{3.3}$$

where $g(s)$ and $h(s)$ are defined above and k_t and k_w are unit costs of the Tardiness and the Wasted Work respectively.

Here α represents a trade-off between the laboratory's economic efficiency and the timeliness of the results. We do not know what is the best value for α and let it vary according to individual circumstances. We analyze the effect of changing α on the optimal processing starting time graphically. Here we present graphical charts of the objective function in (3.3) for different values of α while keeping the rest of parameters fixed.

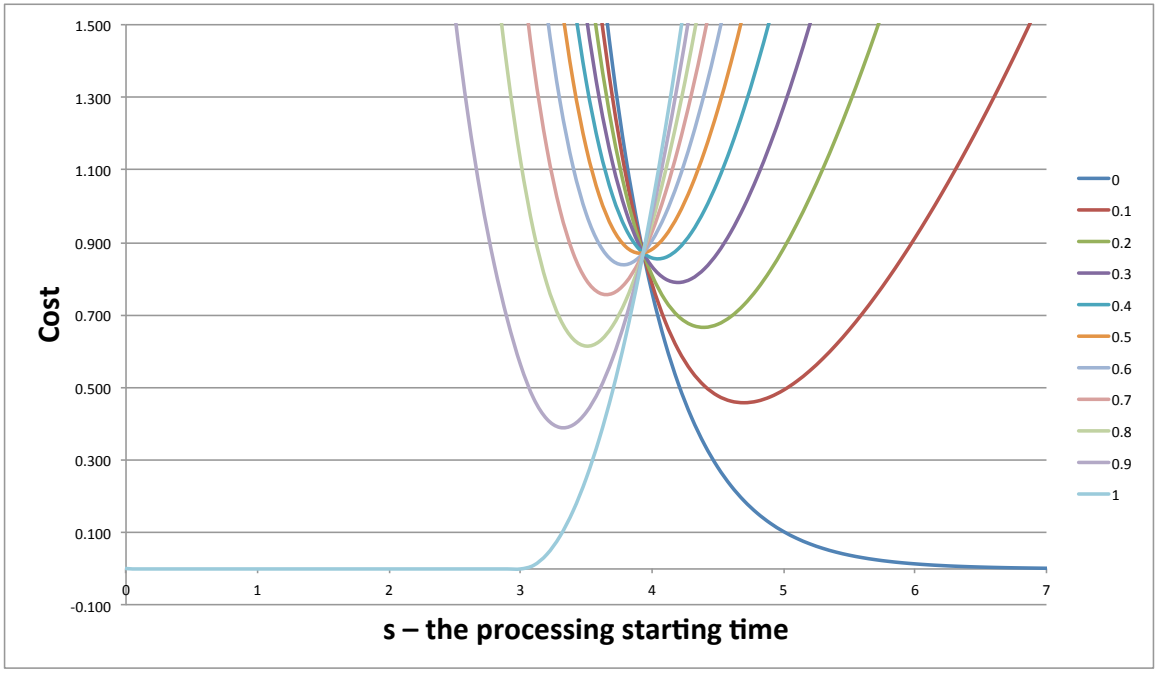

Figure 3.3: Objective function for different α in a single order case

As we can see, for the two extreme values of α (0 and 1) the cost function is determined only by the Wasted Work and the Tardiness, respectively. We can also see, that Wasted Work function is decreasing with time and the Tardiness is, in contrary, increasing. The values of α between the two extremes give the objective function some input from both cost sources, which gives it a nice convex shape with clearly identifiable minimums. The highest minimum has the objective function that corresponds to $\alpha = 0.5$ that could be interpreted as the notion that our running cost will be at maximum whenever we equally focus on minimization of Wasted Work and Tardiness. This is true for at least those model parameters, for which the objective function has a clear minimum in terms of the starting processing time s.

3.2 Multiple Orders case analysis

We now consider more realistic situation where we have many orders with different parameters waiting to be processed by the laboratory. Here we have to make a statement about the time horizon of our problem. Although many hospital laboratories are working non-stop on a 24/7 basis, we are modeling a laboratory operation working 12 hours per day from 8.00 am to 8.00 pm 7 days per week. This is the current schedule of the laboratory of the University Hospital in Columbia, Missouri. The time horizon is more important for the model we develop in Chapter 4 and we just mention it here for clarity.

We assume all the test orders to be issued (arrived to the system) randomly. As in many similar problems we use Exponential probability distribution to model the inter-arrival time of the test orders that belong to several different categories (we explained this in Chapter 2). In our model every category has its own arrival distribution rate and cancellation distribution rate (explained later in this section).

Clearly the main question in this situation is not when to start processing the order but rather how to arrange the waiting queue so that the total processing cost by the end of the planning time-horizon would be minimized.

Another formulation of the optimization decision in the multi-order environment could be: "Which order to pick up for processing every time the server becomes available in order to finish the day with the lowest possible incurred cost?"

This is a more difficult question to answer then the one we considered in the single order case. First of all, it is clear that for some of the orders the starting time of process would not be optimal with respect to the single order case. Any processing arrangement may be sub-optimal for an individual test orders in it but here we are

interested in global optimality. Furthermore, we have to consider the probability of test order cancellations as well as probability of new test order arrivals at every time-step until the end of the time horizon. This considerations make the problem intractable.

While we develop a comprehensive Dynamic Programming model in Chapter 4 that addresses these considerations, in this chapter we describe a more simplistic approach that would allow us to develop an algorithm shown to produce sub-optimal solutions which are generally better than a FIFO queueing mode.

The simplification we are to make here (and relax in Chapter 4) is an assumption about the future arrivals. We no longer consider them when deciding where to insert a new order arrival. In other words, whenever the server becomes available we are looking for the best processing arrangement assuming that no more test orders will come into the system. In this way, the only random process in the system we have to deal with, is test order cancellation process.

Various techniques can be used to solve this simplified problem. For its simplicity we have chosen the Monte-Carlo simulation [10]. First formulated by nuclear physicist in 1930th-40th, this simple method allows to solve for a value of function of a large number of stochastic variables without having to solve complex equations that sometimes do not have a close-form solutions.

Recall, that the main question is how to arrange the processing sequence optimally. In this case the queue is rearranged each time a new order arrives (issued) by deciding on where in the existing queue to insert the new arrival. Algorithmically this is described as follows:

1. At the time of new arrival duplicate the existing queue.

2. Temporally insert the new arrival at the first position.

3. Simulate the queue process until it is empty (without any new arrivals). Record the incurred cost from cancellations and late processing.

4. Repeat the previous step (3) a large number of times (we used 100 times in the actual algorithm). Take the average of the cost incurred in over the number of the repetitions.

5. Duplicate the existing queue again and insert the new arrival into the second position in the duplication. Repeat steps 3 and 4 for this insertions.

6. Repeat this (step 5) for every position in the queue until its end.

7. Find the position that has with the lowest cost and actually insert the new arrival into that position.

A flow-chart of this algorithm is shown on the Figure 3.4 below.

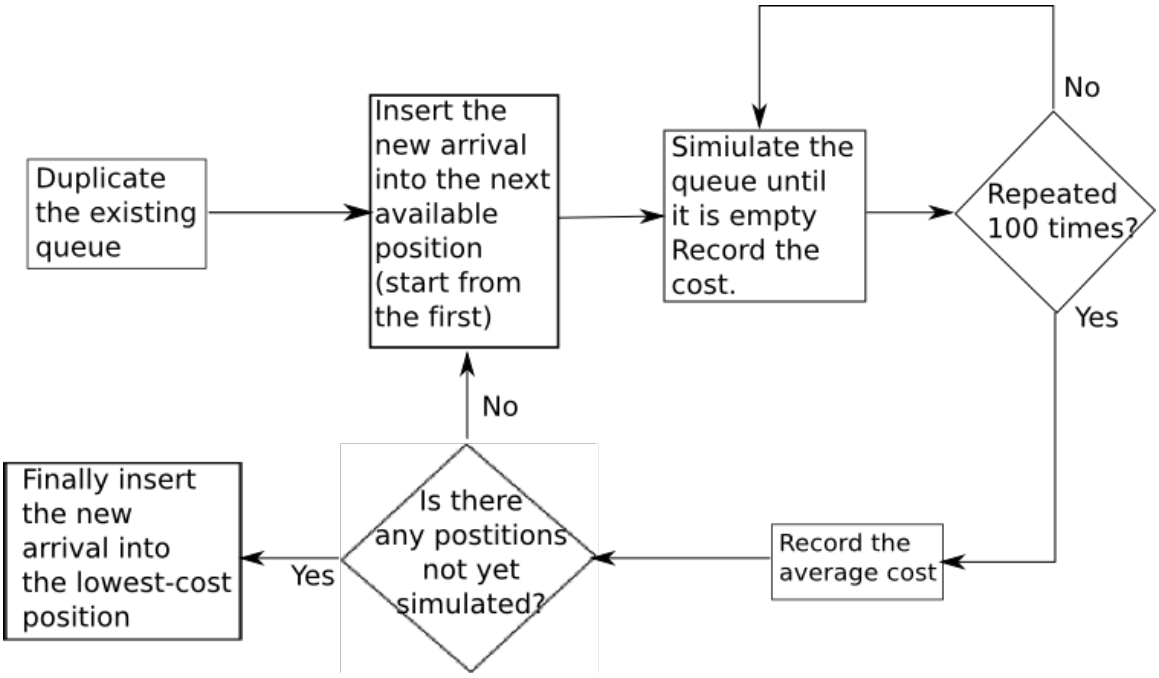

Figure 3.4: The simple heuristic flow-chart.

3.3 Simulation description

To compare the performance of the heuristic described in the previous section with a FIFO queue we have developed a simulation model. The simulation models consists of the three modules:

- Generator that generates random test orders from a pre-defined set of order-types. This is done to mimic the random arrival process of test orders that are independently issued by different doctors (order-types).
- FIFO queue processes the orders in a First-In-First-Out manner (this is the state-of-practice at the hospital of the University of Missouri-Columbia). Specifically it adds all the newly generated arrivals to the beginning of the queue.

- SMART queue processes the orders in a sequence formed by the heuristic described in the previous section.

During the simulation the Generator generates orders from a pre-determined (arbitrary) set of order-types that represent test orders issued at different circumstances (*e.g.* issued by different doctors, hospital floors, shifts, etc).

In theory the only limitation of the order type number is the system's ability to estimate their parameters (the parameters of the cancellation probability distribution) accurately. Liu et al. [9] found that a three-layer neural network can be used to safely estimate parameters of a failure (cancellation) probability distribution (Weibull in that case) after 300,000 training iterations. Given the average 130,000 of test orders being issued in a mid-size hospital (the University Hospital in Columbia, MO) this number seems to be realistic. Here for the illustrating purposes we have used only nine order types, parameters of which are given in the following table.

Table 3.1: The arbitrary order data types used for the random orders generation.

Number	Arrival rate	Due time	Completion time	Cancellation rate (λ)
1	0.4	1.00	0.3	0.95
2	0.3	0.75	0.2	0.8
3	0.2	0.50	0.1	0.55
4	0.1	1.00	0.3	0.1
5	0.5	0.75	0.2	0.95
6	0.4	0.50	0.1	0.8
7	0.3	1.00	0.3	0.55
8	0.2	0.75	0.2	0.1
9	0.1	0.50	0.1	0.95

The costs accrued during the simulation by both queues are recorded and the difference is what can be used to define the relative advantages of the heuristic algorithm. A flow-chart of the simulation model is shown on the following figure. The heuristic is represented by the small loop inside the "SMART" simulation loop. The simulation code itself is given in the Appendix A.

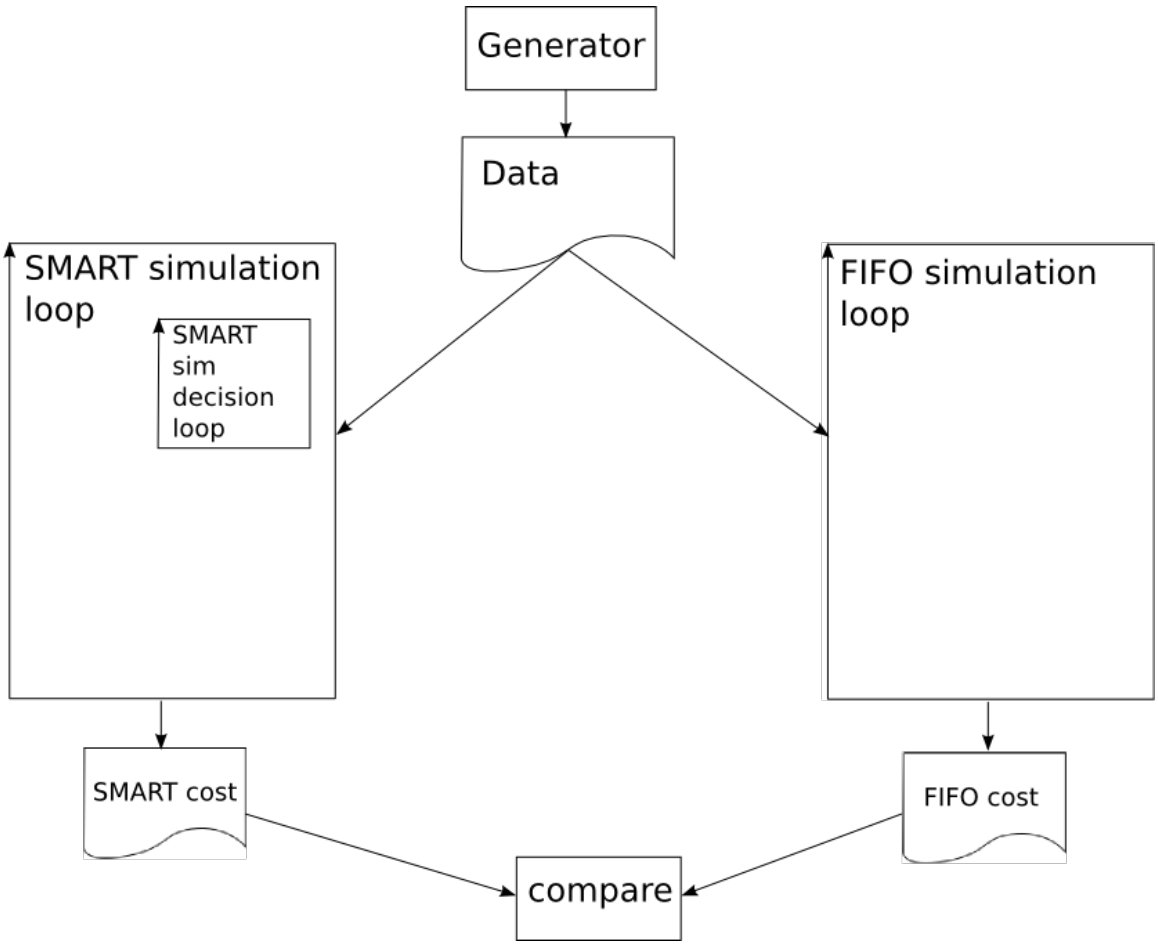

Figure 3.5: The comparative Simulation algorithm Flow-chart.

To make the comparison meaningful, we used 100 repetitions of simulation runs for FIFO and SMART queues, with some random seeds. The resulting distribution

of cumulative costs of FIFO and SMART queues looked as follows.

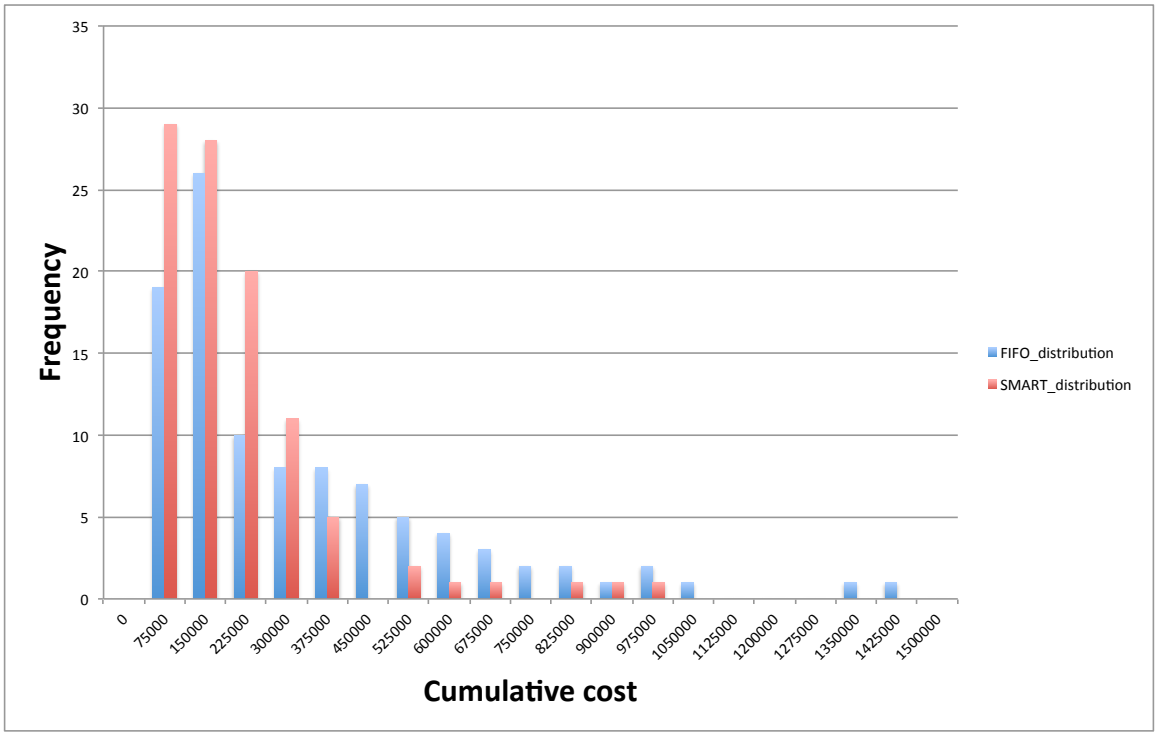

Figure 3.6: The running cost frequency distribution for the SMART and the FIFO queues.

As can be seen in Figure 3.6, although it is not always the case (due to the random processes included in the models), SMART queue generally results in a lower than FIFO queue.

The average costs over the simulation runs are shown in Figure 3.7 below. Based on this graph it is more evident that SMART queue outperforms FIFO queue.

On a Windows-operated computer workstation that operates 2 Intel Xeon e5530@2.40GHz processors (8 cores totally) the simulation has taken 11 hours 55 minutes and 56 seconds (note that this included all 100 repetitions of the 12 hours simulations). A single actual estimation of the lowest-expected cost processing sequence inside the SMART

queue takes less then a second.

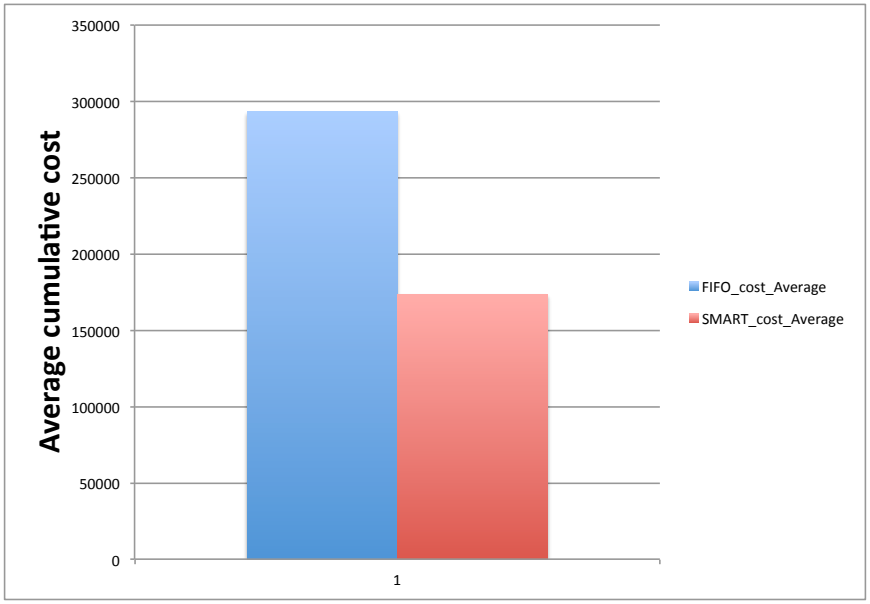

Figure 3.7: The average running cost for the SMART and the FIFO queue simulations.

3.4 Discussion

In this chapter we describe the problem of inflated running cost of a hospital laboratory as a result of premature test order cancellations. We first consider the simplified case of a single order and then argued that the decision-making for the case of multiple orders can be formulated in two ways: we have found that the decision-making in a multiple orders situation can be formulated in two different ways:

1. At every new order arrival, find a position in the existing orders queue, which would result in the lowest cost of processing this queue.

2. Every time the server becomes available (departure or cancellation of the order in service) choose an order to process next (from the ones waiting in the queue)

so that the final accumulated cost would be minimized.

We have developed a simple heuristic, utilizing the first of the formulations above along with an important simplification that at the time of the decision making we do not expect any new orders to arrive.

Even with this unrealistic assumption, the numerical experiment show that the hospital laboratory running cost was improved with the simple heuristic compared with the state-of-practice FIFO system.

Chapter 4

Dynamic Programming model development and solution

4.1 Problem Description as a Dynamic Program.

Let us once again go over the details of the problem to provide some background for the DP model we describe in the next section.

As we want to minimize total laboratory running cost we need to localize the potential sources of the inefficient cost or the cost that would be avoided in an idealized world. As such we define the two most important in our opinion cost sources - the Wasted Work cost and the Tardiness cost.

The Waisted Work cost is directly defined by the premature cancellations. This is simply the amount of time every test order spends in the laboratory before cancellation times the unit cost of the server. The unknown component here is, of course, the time spent in the server before cancellation (if it happens) and this determines

the stochastic nature of the DP model we describe in the next section.

The Tardiness cost is our attempt to introduce certain punishment for the late processing of a test order. Tardiness is a critical component of the objective function since solely considering the Wasted Work will result in a solution that keeps the test orders waiting in the queue infinitely long in this way preventing the Wasted Work accumulation. Ideally Tardiness cost should reflect potentially negative consequences of late processing of test orders on the patient health. We have already discussed that how to quantify the negative health consequences of procrastination is itself an active research field and is outside the scope of our work. We simply model the Tardiness as a squared over-due time multiplied by a predefined unit cost. If a test order completed before its due time, then Tardiness of this order set to be 0.

Considering both the Wasted Work and the Tardiness costs we are at identifying the solution that minimizes the waste and not compromising patient's health. These are two opposite-dynamic function and, as we have shown in Chapter 3 for any test order the Wasted Work cost s decreasing with time and the Tardiness cost is increasing. Another difference between the two is that the Wasted Work cost reaches it's limit when the probability of the cancellation reaches 1 whereas the Tardiness cost has no theoretical limit.

For the multiple orders queue we minimize the total cumulative cost, which consist of the Wasted Work and the Tardiness costs of all orders. Unlike the First-In-First-Out prioritization, which is currently in use in many hospital laboratories (for the same priority tests), we formulate this problem as a sequential decision-making problem using Dynamic Programming model. In this model we determine the next test order to pick up for processing once the laboratory server becomes available so that the

total expected cost (coming from the two sources described above) is minimized considering all the possible arrivals and cancellations scenarios that might happen in the future.

4.2 Dynamic Programming Model.

Presenting the Dynamic Programming model, we will clarify certain important assumption we used in our model.

4.3 Assumptions.

- The server processes the orders in batches, meaning that at a time we can process a number of orders between 0 and the maximum batch size.
- The working time of the laboratory is limited by a normal 12-hour day shift that starts at 8 am and concludes at 8 pm (week-days only). This is the real schedule of the laboratory of the University Hospital in Columbia, MO. Based on this assumption, we formulated a finite horizon model for our problem.
- We discretize the planning horizon by equal-length time-periods and assume that all the service time for all the test orders equals to one such time-period (in all experiment we used 15 minutes time-periods).
- We also assume that a queue test order type can be cancelled during a time period with a certain probability and this probability increases in subsequent time periods. Note that the discretization at time horizon is not limit ing for

our purposes since we can easily use shorter time periods and capture several test order types and properties without loss of generality. However, the discretization significantly simplifies the model and improve that tractability of our problem.

- We assume that there is a finite number of fixed due-hours throughout the planning horizon that are chosen by the doctor as a due date at the test ordering. In our experiments we use 6 of such due-hours: 10.00 am, 12.00 pm, 2.00 pm, 4.00 pm, 6.00 pm and 8 pm. The ordering process would work as follows: a test ordered at 11.00 am would be assigned the 12.00 pm or any later due-hour as its due time. This approach is different then the real laboratory operation where the due time of the test order is defined as a specific time after the test ordering. However, the actual system can be easily captured by increasing the set of due-hours. For example if we define a 15-minutes period to be a particular due-hour then our model would almost exactly reflect the actual system.

- After the end of the planning time (8.00 pm) there are no more orders generated and none of the waiting orders could be cancelled.

- All the test orders left in the system after the end of the planning time are processed one by one. No Wasted Work cost could be incurred and all the orders are late until 8.00 am of the next day a corresponding Tardiness cost is recorded (therefore the model does not want to delay the process until after 8.00 pm as expected Tardiness cost is huge). This assumption might seem a little artificial since a test order can be cancelled any time during its life but we make it here because of the following two reasons. First, it is implying that

the doctors are not in the hospital between 8.00 pm and 8.00 am (and cannot cancel their test orders), which by itself is not very unreasonable. Second, this assumption greatly simplifies our DP model giving it a fixed and easily determined Boundary Condition.

- During the time between the end of the shift (8.00 pm.) and the beginning of the new shift (8.00 am) we must be able to process ALL orders that are left in the system. This could be controlled by making the arrival rate of the test-types generating the number of new orders which is less or equal than the processing capacities of the laboratory defined by the server maximum batch-size and the processing time-period.

- At the beginning of the shift (*i.e.* 8.00 am) the system is assumed to be empty.

- There is a finite number of test-types with each test-type having its own cancellation probability parameter and the arrival rate parameter.

- The arrival of new test orders follows Poisson distribution and the arrival rate of each test-type is predeterminded.

- Every test order can be cancelled at any time-period while it is in the system and the number of cancelled orders of the same test-type is Binomial, defined by the cancellation probability for that test-type at the corresponding time-period.

Now let us specify some data sets and parameters we will be operating with in the DP model.

4.4 Notation and Parameters.

- $t \in \mathcal{T}$: set of the time periods (or decision epochs) during laboratory's working hours. The length of these time periods (defined by Δt) is equal to the processing time of lab orders. $t = \{1, 2, ...T\}$, where $T = \frac{|8.00\text{pm}-8.00\text{am}|}{\Delta t} = \frac{12\text{hours}}{\Delta t}$.
- $d \in \mathcal{D}$: set of due-hours at which the laboratory is supposed to return the results of the tests.
- $c \in \mathcal{C}$: set of test-types. The test-types are identified based on the cancellation distribution parameter (rate) and the arrival rate.
- η : the server batch-size – maximum number of orders that can be processed in one time-period.
- c_w : a unit cost of the Wasted Work for a canceled test order in one time-period.
- c_t : a unit cost of Tardiness for a test order in one time-period.
- $p_{dc}(t)$: probability that a test order with due date $d \in \mathcal{D}$ and cancellation distribution $c \in \mathcal{C}$ will be canceled in time period $t \in \mathcal{T}$.
- $\lambda_{dc}(t)$: arrival rate of a lab order with due date $d \in \mathcal{D}$ and cancellation distribution $c \in \mathcal{C}$ in time period $t \in \mathcal{T}$.

4.5 DP Formulation.

State: Let n_{dc} be the number of laboratory orders with due date $d \in \mathcal{D}$ and cancellation distribution $c \in \mathcal{C}$ currently waiting in the queue at the beginning of time period

t. The state is at the beginning of time period t is then given by $(t, [n_{dc}])$ where $[n_{dc}]$ is a $|\mathcal{D}| \times |\mathcal{C}|$ matrix.

Actions: Let the action be the $[a_{dc}]$, which is a $|\mathcal{D}| \times |\mathcal{C}|$ matrix and every a_{dc} is:

$$a_{dc} = \begin{cases} 1 & \text{if we choose to process lab orders from category (c, d),} \\ 0 & \text{otherwise,} \end{cases}$$

where $\sum_{d \in \mathcal{D}} \sum_{c \in \mathcal{C}} a_{dc} = 1$. (A pointing matrix with all the elements = 0 except one of them = 1, which correspond to the one whose orders we pick for processing.)

Random Disturbance and System Dynamics:

- Let $X_{dc}(t)$ be the number of lab orders arrive during time period t. $X_{dc}(t) \sim \text{Poisson}(\lambda_{dc}(t))$.
- Let $Y^{\text{q}}_{dc}(t, a_{dc})$ be the number of lab orders cancelled while waiting in the queue during time period t. $Y^{\text{q}}_{dc}(t, a_{dc}) \sim \text{Binomial}(n_{dc} - a_{dc} \min\{n_{dc}, \eta\}, p_{dc}(t))$.
- Let $Y^{\text{p}}_{dc}(t, a_{dc})$ be the number of lab orders cancelled while being processed during time period t. $Y^{\text{p}}_{dc}(t, a_{dc}) \sim \text{Binomial}(a_{dc} \min\{n_{dc}, \eta\}, p_{dc}(t))$.

Where $\min\{n_{dc}, \eta\}$ – minimum of n_{dc} and η.

The state of the system evolved as follows:

$$\begin{aligned} t &\leftarrow t + 1 \\ n_{dc} &\leftarrow n_{dc} + X_{dc}(t) - Y^{\text{q}}_{dc}(t, a_{dc}) - a_{dc} \min\{n_{dc}, \eta\} \end{aligned}$$

DP Recursion: The Value Function can be written as:

$$J(t, [n_{dc}]) = \min_{[a_{dc}]} E\left[c(t, [n_{dc}], [a_{dc}]) + J(t+1, [n_{dc} + X_{dc}(t) - Y^{\text{q}}_{dc}(t, a_{dc}) - a_{dc} \min\{n_{dc}, \eta\}]\right] \tag{4.1}$$

where

$$c(t,[n_{dc}],[a_{dc}]) = \sum_{d\in\mathcal{D}}\sum_{c\in\mathcal{C}} \overbrace{c_t a_{dc}\min\{n_{dc},\eta\}(\max\{t-d,0\})^2}^{\text{Tardiness}} + \overbrace{c_w Y^{\mathrm{p}}_{dc}(t,a_{dc})}^{\text{Wasted Work}}. \qquad (4.2)$$

Here the first element of the immediate cost equation (2) (that describes the Tardiness cost) does not depend on any stochastic process and could be brought out of the expectation symbol. Also note that Tardiness cost is accrues only is a test order is processed late (*i.e.* $a_{dc} = 1$ and $t - d > 0$). The Tardiness cost is not accrued if the order is cancelled (before or after the due time).

The second element $c_w Y^{\mathrm{p}}_{dc}(t, a_{dc})$ describes expected Wasted Work cost and is the expectation of a linear combination of a Binomial random variable ($Y^{\mathrm{p}}_{dc}(t, a_{dc})$), therefore can be as follows:

$$E[c_w Y^{\mathrm{p}}_{dc}(t,a_{dc})] = c_w E[Y^{\mathrm{p}}_{dc}(t,a_{dc}] = a_{dc}\min\{n_{dc},\eta\}c_w p_{dc}(t)$$

where $a_{dc}\min\{n_{dc},\eta\}p_{dc}(t))$ - expectation of the Binomial random variable $Y^{\mathrm{p}}_{dc}(t, a_{dc})$.

In this case the **Value Function** can be reformulated as:

$$\begin{aligned} J(t,[n_{dc}]) = \min_{[a_{dc}]} a\Bigg[& \sum_{d\in\mathcal{D}}\sum_{c\in\mathcal{C}} a_{dc}\min\{n_{dc},\eta\}[c_t(max\{(t-d),0\})^2 + a_{dc}\min\{n_{dc},\eta\}c_w p_{dc}(t)] \\ & + \sum_{d\in\mathcal{D}}\sum_{c\in\mathcal{C}}\sum_{y^q_{dc}=0}^{n_{dc}}\sum_{x_{dc}=0}^{N-n_{dc}} \binom{n}{y^q_{dc}} p_{dc}(t)^{y^q_{dc}}(1-p_{dc}(t))^{1-y^q_{dc}} \frac{e^{\lambda_{dc}(t)}\lambda_{dc}(t)^x_{dc}}{x_{dc}!} \\ & \times J\big(t+1, \underbrace{[n_{dc} - a_{dc}min\{n_d c,\eta\} + x_{dc} - y^q_{dc}]}_{n_{dc}+X_{dc}(t)-Y^{\mathrm{q}}_{dc}(t,a_{dc})-a_{dc}\min\{n_{dc},\eta\}}\big)\Bigg] \end{aligned} \qquad (4.3)$$

where N - maximum number of orders we can accept in any of the due-hour$\times$cancellation parameter categories ($0 \leq n_{dc} \leq N$). To obtain a finite state space of the model we limit the number of arrivals (and make it a truncated Poisson variable). Note that this assumption is not very living as N can be made significantly large to capture the real system. The terms x_{dc} and y^q_{dc} in equation (4.3) represent the realization of random variables $X_{dc}(t)$ and $Y^{\mathrm{q}}_{dc}(t, a_{dc})$, respectively.

Boundary condition:

$$J(T, [n_{dc}]) = \sum_{d \in \mathcal{D}} \sum_{c \in \mathcal{C}} c_t n_{dc} (T + \frac{12}{t} - d)^2 \tag{4.4}$$

where t - length of one decision epoch. This formulation of the Boundary Condition is result of two of the assumptions the we have made in the previous section. In simple words we assume that there are no more arrivals and cancellations after the 8.00 pm ($X_{dc}(T) = Y^{\mathrm{q}}_{dc}(T, a_{dc}) = 0$), we can process the orders left in the system over night and deliver the results on the next morning by 8.00 am. In this way the only cost to the system is from Tardiness of $T + \frac{12}{\Delta t}$.

4.6 Model Analysis and Solution Suggestions.

For the DP model developed in the previous section, we made several assumptions for problem of the running cost minimization of a hospital laboratory such as discretization the time horizon and limiting the possible number of arrivals, etc. Given the discretized structure of the model the quality of the approximation directly depends on the size of the parameters: length of a decision-epoch time-period t, number of due-hours d and the test-types c as well as the maximum allowed value of every n_{dc}

(N).

Note that increasing the size of $\mathcal{C}, \mathcal{D}$ and N increases the state space exponentially. In the current state definition the size of the state space can be calculated:

$$SS = \mathcal{T} N^{\mathcal{C} \times \mathcal{D}}$$

It is easy to calculate that for $|\mathcal{C}| = 4$, $|\mathcal{D}| = 6$ and $N = 20$ (the $|\mathcal{T}|$ is given by $= \frac{12 \text{ hours}}{15\text{minutes}} = 48$, for the 15-minute long time period) the size of the state space is:

$$SS = 48(20)^{4 \times 6} = 48 \times 20^{24} = 8.05 \times 10^{32}$$

In addition to that, we have up to $|\mathcal{C}| \times |\mathcal{D}|$ actions available in every state and, when computing expectations in the Value Function (4.3), we need to consider all the realizations for the two random variables $X_{dc}(t)$ and $Y^{\mathrm{q}}_{dc}(t, a_{dc})$ for every element of every state-action combination. In effect, we are facing the main problem with Dynamic Programming models – the so called "Curse of Dimensionality". Before coming up with approximations and simplifications , to solve a large-scale instances of this problem, we have implemented Backward recursion on the actual problem and also solved a Linear Programming version of our DP formulation. Our goal here is to better understand the structure of the optimal solution of the original problem and estimate the largest problem scale that can be handled in this way. This can be very important for defining approximation strategies which is proposed as future work.

4.7 A Potential Solution Method: Dijkstra and Backward recursion.

Dijkstra method has been proven to be the most efficient known method for solving deterministic finite-horizon Dynamic Programming problems. The idea here is that any Dynamic Program can be represented as an acyclic directed graph (network), where the states represent the nodes and the actions represent the arcs of the graph. Following the same logic the cost of every action represents the arc length. Then, finding the least cost solution of the DP problem is equivalent to finding the shortest path from the root node to the sink node on the corresponding acyclic directed graph. This shortest path would be the solution to our problem as it would tell us what action to choose in every state to minimize the total cost.

The outline of Dijkstra's algorithm is as follows:

- Label the source node with value 0.
- Label all the rest of nodes with value ∞.
- Label all the non-source nodes as "pending".
- Label all the non-source nodes predecessors as 0.
- Pick the "pending" node with the smallest label: if the node is terminal - stop: we have found the shortest path. If not, for every arc coming to this node: if the cost (length) of arc + the value of the node where the arc is coming from is less than the value of the node - update the value of the node and label the node that the arc is coming from as predecessor. Then label this node as "not pending".

This works very well for deterministic problems. Our problem includes two stochastic processes (the orders arrival and cancellation) that creates certain difficulties in a straight-forward implementation of the Dijkstra's algorithm. Under the cancellation and arrival uncertainty each action taken might lead us to several different future states with certain probabilities. This means that instead of a separate deterministic state (node) we have to con sider a set of potential stated we can reach and therefore to calculate an *expected* "cost to go" based on the probability associated with easy state that we might go to. In this situation the following modifications need to be done to the classical Dijkstra's algorithm:

- Proceed from the terminal (known) state towards the current state
- Calculate the cost of every action as an *expected* cost of continuing to the future states.
- For the optimal path pick the states with *the best expected* cost of continuing optimally.

Under the above modifications, the solution algorithm is known as Backward recursion. We have developed and tested a Python script that finds the solutions to the problem (4.3) using the Backward recursions (please see it in Appendix B.1).
Unfortunately the Backward Induction algorithm for our problem has suffered from Curse of Dimensionality. As we are increasing the number of elements in the Cancellation rates set $|\mathcal{C}|$ and due-hours set $|\mathcal{D}|$ the computational time increases exponentially. Here we provide the computational time needed to find the optimal path for the following state-matrix dimensions (given as $|\mathcal{C}| \times |\mathcal{D}|$):

- 1×2 - less than a second;

- 2×2 - 55 minutes;
- 3×2 - could not be solved after 24 hours of computation.

This makes a straight-forward Backward recursion method very inefficient and incapable of solving our problem for any realistic data-set in a reasonable amount of time. Therefore we decided to take a look at the other popular method - LP transformation, as explained in the next section.

4.8 A Potential Solution Method: LP-transformation.

The second method explores the notion that any Dynamic Program can be expressed as a Linear Program. In this section we use a method given in the text book by Puterman [5] describing an equivalent Linear Programming model of the DP model developed in Section 4.5. After implementing this method and writing the DP model in Linear Programming terms we used Gurobi 5.0 [18] solver to solve it.

A lot of efficient LP solvers were developed in the recent times. They use some preprocessing to decrease the solution time, such as Branch-and-Bound and Cutting Plane algorithms. Moreover, the LP solvers are usually written on C-based languages that are somehow "closer to metal" (do not require an interpretation phase and consume less memory) than Python that we have used for the Backward recursion script. As the numerical experiment show (please see the end of this section), the corresponding LP model was more efficient in solving the problem.

Corresponding LP formulation of the DP model can be written as follows:

$$\max c^T J(t, [n_{dc}])$$

subject to:

$$E\left[c(t,[n_{dc}],[a_{dc}])+J(t+1,[n_{dc}+X_{dc}(t)-Y^{\mathrm{q}}_{dc}(t,a_{dc})-a_{dc}\min\{n_{dc},\eta\}]\right]\geq J(t,[n_{dc}])$$

$\forall[a_{dc}]\in A(n_{dc})$(the state-dependent action space) and $J(t,[n_{dc}])\in SS$ except $t=T$

and

$$J(T,[n_{dc}])=\sum_{d\in\mathcal{D}}\sum_{c\in\mathcal{C}}c_t n_{dc}(T+\frac{12}{t}-d)^2$$

where $A(n_{dc})$ is the set of feasible actions and c is a vector of arbitrary positive numbers.

We have programmed and solved this problem using Gurobi 5.0 LP solver [18] (please see it in Appendix B.2) on a Windows-operated computer workstation with two Intel Xeon e5530@2.40GHz processors (8 cores totally).

A major issue with using this model was the result interpretation and the backward translation of the LP-solver results into the DP optimal path.

The LP solver returns the optimized values for each state $J(t,n_{dc})$. Using these values, then the corresponding optimal action for each state can be easily calculated.

The solution time for different state-matrix (given as $|\mathcal{C}|\times|\mathcal{D}|$) was as follows:

- 1×2 - less than a second;
- 2×2 - 1 minute and 15 seconds;
- 3×2 - 2 hours 05 minutes 32 seconds;

Even though we were able to handle larger matrices with the LP model, we still could not find the optimal solution for a realistic problem size. These results suggest that approximation schemes are critical to solve larger instances of this problem as

explained in Section 5.2 of this thesis.

4.9 Discussion of the DP model and its solution.

The Dynamic Programming model developed in Chapter 4 is obtained by using the "sequential decision making" nature of the original problem. For such problems DP is one of the most powerful modeling techniques, however, because of the Curse of Dimensionality (discussed in Section 4.6) it fails to handle real life problems due to their large sizes. The results we present here confirm this fact and emphasizes the need for approximation strategies. In addition it provides intuition for the structure of optimal solutions on the smaller scale problems, which would be useful for the future directions of our research.

We experimented with various arrangements of a state-matrix $[n_{dc}] = \big\{[n_{dc}] : |\mathcal{C}| = 2, |\mathcal{D}| = 2\big\}$, where the cancellation parameters set is $|\mathcal{C}$ and the due times set is $|\mathcal{D}|$. The corresponding solution (optimal action) is given by a $|\mathcal{C}| \times |\mathcal{D}| = (2 \times 2)$ binary matrix the non-zero element of which points to the element of the state-matrix that has to be processed next. We have observed the following features of the solutions:

- The optimal solution tends to process the element the matrix that corresponds to the first row (cancellation rate) and the first column (due time) (n_{11}). In other words the orders with lower cancellation rate the closer due times are given priorities over the others even if there is just one order with these parameters.

- If the first element is empty ($n_{11} = 0$) the preference is always given to the next element in the same column n_{21}. This can be interpreted as, with the current set of unit-cost parameters of Tardiness and Wasted Work ($k_t = 1, k_w = 1$),

the due date has more effect on selecting the next order to process, than the cancellation rate.

In our experiments we arranged the cancellation rates of the elements of the state-matrix from the lowest to the highest and the due-times of the state-matrix element from the closest to the farthest.

We have also experimented with various combinations of zeros and ones in a 2×2 matrix. The most interesting results are given in the following Figure:

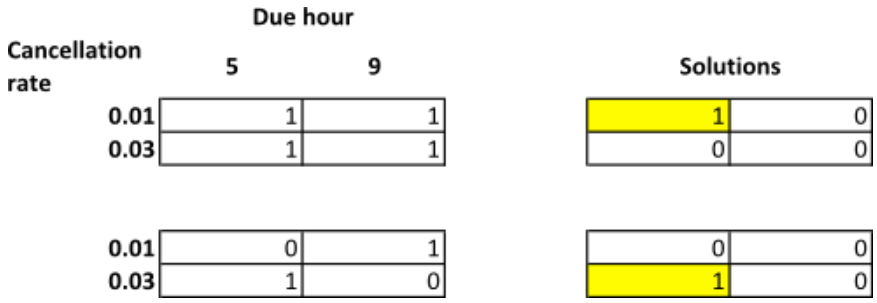

Figure 4.1: The result of a sensitivity experiments of the Dynamic Programming solutions of 2×2 matrix.

The first line instance illustrates the notion that the most urgent orders with lower cancellation rate is processed first. The second instance of on Figure 4.1 demonstrated that the preference was given to the more urgent order even if its more likely to be cancelled.

In general, a solution tends to be affected by both the due time and cancellation rate. These observations also support the findings we presented in Chapter 3 that demonstrate that the original FIFO processing approach is not the most efficient method and can be easily over-performed by optimization algorithms or optimization-based heuristic.

Ranging the elements of the state matrix from the highest to the lowest by the cancel-

lation rate in the rows and from the earliest to the latest by the due-time in columns should theoretically create an arrangement that executing the orders from the top left corner of the matrix and then moving down and right could turn out to be a descent approximation. This requires a great deal of additional investigation which we are planning to carry out in our future research (please see Section 5.2).

Chapter 5

Conclusions and future work

5.1 Conclusions

In this thesis, we described the problem of prioritizing (scheduling) the hospital laboratory test orders considering cancellations that result in significant increase of the laboratory running cost. We first analyzed the case of a single test order and developed two (conflicting) objective functions: Wasted Work and Tardiness, where the former is dominated by the latter as the life-time of the test order exceeds its due-time.

In the general case of multiple tests orders in the laboratory queue, the problem of minimizing the running cost becomes a problem of prioritization or finding the right processing sequence of the test orders. With the assumption that there will be no future arrivals, it is possible to identify the best test order to process through a brute-force approach which simulates every possible insertion of the new arrival into the

current queue. Although such an approach cannot be guaranteed to yield the optimal solutions, numerical experiments have shown encouraging results when compared to the state-of-practice First-In-First-Out queueing system.

In the second part of this work we develop a Dynamic Programming model that captures the most of the relevant (random) processes of the laboratory operation. In other words, although we had to make some simplifications, the DP model incorporates all important details of the process including the random cancellation and arrival processes. We describe the states of the DP model as time-specific matrices with columns corresponding to certain due-times and rows to certain cancellation and arrival parameters. The elements of this matrix represent the numbers of orders in the queue with the corresponding due-times and cancellation rates.

We also tested two alternative methods for solving the DP model, the Backward recursion and the Linear Programming transformation. The Backward recursion method appears to be the most straight-forward approach to solving similar problems (finite-time stochastic DP) but it did not perform well and quickly became unable to solve the problem instances as they get larger. The time to solve a 2×2-matrix problem was almost 1 hour on a powerful 8-core workstation (2 processors Intel Xeon e5530@2.40GHz).

Our second approach was to develop the corresponding Linear program of the original DP model and to solve it using an LP solver. We used Gurobi 5.0 commercial LP solver[18]. The performance of the LP model was better compared to the Backward recursion and for a 2×2-matrix problem it took only 1 minute and 15 seconds. We were able to get solutions for up to 3×2 matrix problems, however, this method became also intractable for larger problem instances.

5.2 Future work

The findings of this thesis confirm importance of development of approximation methods, that could be used for solving realistic-size instances of this problem. As we discussed in our Literature Review (Chapter 2) there could be two main future research directions.

The first one is what we call "discrete" methods. They could be the more logical continuation of the research presented in this thesis as we have already used a planning horizon discretization to develop our DP model. There are several important papers, that pursued similar approach. Patric *et.al* [6] and Erdelyi and Topaloglu [16] are looking at very similar problems and at first formulate them as Dynamic Programming models. The later paper takes on an interesting development that we are currently trying to adapt to our formulation. The authors are also expressing their DP model as a Linear program and then assume the random variable (in their problem they only deal with random arrival times) to be equal to its expected value. This assumption, together with relaxation of limited processing capacity (the batch size in our case) constraint for all periods except the current one, allowed them to find a very efficient approximation of their original DP model solution. Unfortunately, we were not able to apply the same methods without modification to our problem, mainly because of the fact that we have two random variables in the system (arrivals and cancellations) instead of one.

Another approach, is what we may call "continuous" and it is represented by papers

of Rubino and Ata [4] and Kim and Ward [17]. These methods consist a departure from the discretizing logic on which we have developed our DP model in Chapter 4. It utilizes a continuous-time models that are eventually approximated by Brownian control problem. Although both papers found good solutions to their problems, Brownian control problems in general do not have closed-form solution methods as they may consist of complex differential equations.

In general, the "discrete" approach is more flexible in terms of large number of test orders in the system, than the "continuos" methods. Instead, the later methods more are easier in accommodation of several random processes and in addition to the random arrival and cancellation times we might incorporate a random processing time of the test orders.

In summary, as the future direction we are going to pursue the "discrete" approach together with looking into the development of the "continuous" methods. As soon as we will be able to solve our original DP model for some realistic scale we may think of making it more thorough by incorporating possibility for batch processing of several different tests that includes similar operations (*i.e* centrifuge) or an on-line coordination of several laboratories operation in a hospital network.

5.3 Bibliography

1. Hamid Z., Pasupathy K., (2010) DM-HI Identification of causes of laboratory order cancellations using association rules and clustering. Proceeding the 5^{th} INFORMS Workshop on Data Mining and Health Information.

2. Thacker, R. (2011, 09 16). MT (AMIT) Lab Supervisor, IR Processing. (A. Ghavrish, Interviewer) Columbia, MO.

3. Wilton A. van Klei, Moons K, Charles L., Rutten G., Schuurhuis A., Johannes T, Knape, Kalkman C., Grobbee D.,(2002) The Effect of Outpatient Preoperative Evaluation of Hospital Inpatients on Cancellation of Surgery and Length of Hospital Stay. Anesth. Analg.

4. Rubino M., & Ata B. (2009). Dynamic Control of a Make-to-Order, Parallel-Server System with Cancellations. Operations Research.

5. Puterman, M. L. (1994). Markov Decision Processes. New York: Wiley-Interscience Publication.

6. Patric J., Puterman M., Queyranne M., (2006) Dynamic Multi-Priority Patient Scheduling for a Diagnostic Resource. Operations Research.

7. Weibull, W. (1951). A Statistical Distribution Function of Wide Applicability. Annual Meeting of the American Society of Mechanical Engineers.

8. Tieling Zhang, M. X. (2010). On the upper truncated Weibull distribution and its reliability implications. Reliability Engineering and System Safety: 7.

9. Ming C.Liu, W. K., Tep Sastri (1995). An exploratory study of a neural network approach for reliability data analysis. Quality and Reliability Engineering International 11: 107-112.

10. Wikipedia. The Free Encyclopedia. "Monte Carlo method." from http://en.wikipedia.org/wiki/Monte-Carlo method.

11. Bonini, P., Mario P., Ferruccio C., and Rubboli, F. (2002). Errors in Laboratory Medicine. Clinical Chemistry, 48(5), 691698.

12. Carraro P. and Plebani M. (2007). Errors in a Stat Laboratory: Types and Frequencies 10 Years Later. Clinical Chemistry, 53: 1338-1342.

13. Lippi, G., Blanckaert, N., Bonini, et. al. (2009). Causes, consequences, detection, and prevention of identification errors in laboratory diagnostics. Clinical Chemistry and Laboratory Medicine, 47(2), 143-153.

14. Dantzig G.B, Fulkerson D.R.,(1953). Minimizing the number of tankers to meet a fixed schedule. The basic George B. Dantzig. Stanford, California. Stanford University Press.

15. Ross S.M. (2007). Introduction to Probability Models. Berkley, California. Elsevier Publication.

16. Erdelyi A., Topaloglu H., (2011) Approximate dynamic programming for dynamic capacity allocation with multiple priority levels. IIE Transactions 43,129-142.

17. Kim J., Ward A. (2012) Dynamic Scheduling of a GI/GI/1+GI Queue with Two

Customer Classes. Information and Operation Department, Marshall School of Business, USC, Los Angeles, California.

18. Gurobi Optimization. http://www.gurobi.com/

Appendix A

Python Code: The simulation algorithm code used in the Chapter 3.

A.1 The Main Simulation loop

```
from Generator import Generator
from Test import Test
from FIFO import FIFO
from SMART import SMART
import random
from numpy import*
from xlrd import open_workbook
from xlwt import easyxf
from xlutils.copy import copy
import time
# ====================================================================
#The simulation parameters
alpha = 0.5
un_w_unit_cost = 50
num_sim = 100 #number of main simulation loops
sim_time = 12 #length of a single simulation in hours
#====================================================================
```

```
#The simulation starts
#start the time count
start = time.clock()
#define the simulation length in minutes
sim_time_min = 60*sim_time
#create the two lists to record the outcomes of the simulations
F_cost = []
S_cost = []
#=========================================================================
#open the excel book and sheet with results
book = open_workbook('results.xls', formatting_info=True)
rs = book.sheet_by_index(0)
#copy them to a new book
wb = copy(book)
ws = wb.get_sheet(0)
#set formatting preferences
plain = easyxf('')
#=========================================================================
#start the simulation loop
for i in range(num_sim):
    #generate the table of random orders arrivals
    #at least the same number as minutes in the simulations
    table = Generator.Generate(Generator(), sim_time_min)
    #create the two parallel queues
    f_line = FIFO(table, sim_time_min)
    s_line = SMART(table, sim_time_min, alpha, un_w_unit_cost)
    #let the both queues run their simulation cycle
    f_line.Revolver()
    s_line.Revolver()
    #record the cost accumulated in both queues in this run
    f_cost = alpha*f_line.Tardiness + (1-alpha)*un_w_unit_cost* \
    f_line.Unness_Work
    F_cost.append(f_cost)
    #and save them in the excel results book
    ws.write(i+1,0,f_cost,plain) #row index first
    s_cost = alpha*s_line.Tardiness + (1-alpha)*un_w_unit_cost* \
    s_line.Unness_Work
    S_cost.append(s_cost)
    ws.write(i+1,1,s_cost,plain) #row index first
    #this concludes the simulation runs
#=========================================================================
#calculate the time taken for the simulation
finish = time.clock()
comp_time = finish - start
#convert the time in second into hours and minutes
m, s = divmod(comp_time, 60)
h, m = divmod(m, 60)
print "The simulation took: %d:%02d:%02d" % (h, m, s)
```

```
#=========================================================================
#calculate and record the average cost
#over the simulations loops and write the result into the excel file.
f_av = float(sum(F_cost))/len(F_cost)
ws.write(1,2,f_av,plain)
s_av = float(sum(S_cost))/len(S_cost)
ws.write(1,3,s_av,plain)
#replace the old data sheet by the newly created sheet
wb.save('results.xls')
print 'The end of the simulation!'
```

A.2 The Generator

```
from xlrd import open_workbook
from Test import Test
import numpy as np
import random

class Generator(object):
    '''
    This generator randomly generates the list of
    the orders to arrive and their respective arrival times
    '''

    def __init__(self):
        '''
        Constructor
        '''
        #open the excel book from the sheets
        book = open_workbook('orders_data.xls', formatting_info = True)
        sheet = book.sheet_by_index(0)

        #read the arrival rates of the orders
        self.arr_rates = sheet.col_values(0,1)

        #get the number of test orders' types
        self.types = len(self.arr_rates)

    def Generate(self, number):

        #create the empty dictionary of two linked values -
        #the order type and it's next generation time.
        #We assume that the first generation time is 0
        arr_times = []
```

```
        #set up the initial last arrival time
        last_arrival = 0

        #fill up the generation table with random
        #orders generated at random consecutive times
        for x in range(number):
            #order to generate
            next_test = random.randint(0,self.types-1)
            #generate the next arrival time
            last_arrival += random.expovariate(self.arr_rates[next_test])
            #add this order to the array
            arr_times.append(Test(next_test,last_arrival))

        #return the generated dictionary
        return arr_times
```

A.3 The Test order object

```
from xlrd import open_workbook
import random

class Test(object):
    '''
    classdocs
    '''
    numTests = 0

    def __init__(self, next_test, arr_time):
        '''
        Creates a new test order
        '''
        self.numTests += 1
        #Use the current time as an arrival time of the test
        self.arr_time = arr_time
        #Create a new test order of a needed type
        #Read the data from the excel-sheet
        book = open_workbook('orders_data.xls', formatting_info = True)
        sheet = book.sheet_by_index(0)
        cols = int(sheet.ncols)


        #read the parameters of the order
        self.due_time = self.arr_time + sheet.cell_value(next_test+1,cols-4)
        self.comp_time = random.expovariate(sheet.cell_value(next_test+1,cols-3))
```

```
        self.canc_rate = sheet.cell_value(next_test+1,cols-1)

        self.canc_time = self.arr_time +\
        random.expovariate(self.canc_rate)

    def __str__(self):
        '''
        Defines what would be printed about the test order
        '''
        info = 'Arrived at: '+str(self.arr_time)
        return info
```

A.4 The FIFO object

```
from numpy import *
from scipy.integrate import quad
from math import exp
import random
from Test import Test

class FIFO(object):
    '''
    This is the FIFO simulation line itself. It has to be self-contained.
    This is a process-oriented simulation. The following part consist
    definition of several processes that start at the initialization of
    the SMART object and live until the end of the simulation.
    '''

    def __init__(self, table, end):
        '''
        Constructor
        how do we identify the orders in the queue?
        if by their arrival rate, what happens if two orders
        have arrived at exact same time?
        If we consider the probability of it to be really low
        (we are using the 'float' values for the arrival times)
        we can still identify the orders by their arrival times
        '''
        #make the random table the class parameter
        self.table = [order for order in table]

        #the end time
        self.End = end
```

```
        #create the Event_Calendar
        self.Event_Calendar = zeros((3,2), dtype = 'f4')
        # the first field the arrival time of the order, the second - the event time

        #populate the array with infinitely large values
        self.Event_Calendar[:] = float('inf')
        #the simulation time of the line
        self.Sim_time = 0

        #create the line queue which is an array that contains all the numbers in the
        self.queue = []
        #create the records for Unnecessary Work and Tardiness
        self.Tardiness = 0
        self.Unness_Work = 0
        '''
        Initialization
        '''
        if self.table:
            #initialization of the line: the first order arrived, it has seized the server
            #and it's departure time has been calculated.
            #remove the first line from the random table
            self.queue += [self.table.pop(0)]

            #record the next arrival event
            self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)
            #calculate the departure time of
            #the initial arrival and record it in the calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time +self.queue[0].comp_time)
            #random.expovariate(self.queue[0].comp_rate))
            #record the cancellation time of the arrival into the calendar
            self.Event_Calendar[2] = (0, self.queue[0].canc_time)

        else:
            print'The simulation table is not long enough!'


    def Arrival(self):
        '''
        #This function manages the arrival processes.
        When we call the arrival function it simply calls
        the next arrival to come and
        #if the server is available it takes it, if not it goes to the queue.
        '''
        #check if there is at least one order in the queue
        #it must be in service
        if self.queue:
            #execute the arrival
```

```
            #remove the first row from the random table
            self.queue += [self.table.pop(0)]

            #record the next arrival to the Calendar
            self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)

            #find and record the next cancellation
            next_cancel = min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

        else:
            #execute the arrival
            self.queue += [self.table.pop(0)]
            #remove the first row from the random table

            #record the next arrival to the Calendar
            self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)

            #record the next departure and cancellation into the Calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)
            self.Event_Calendar[2] = (0, self.queue[0].canc_time)

    def Departure(self):
        '''
        #Manages the departure processes
        '''
        #calculate the accrued tardiness if any
        if self.Sim_time > self.queue[0].due_time:
            self.Tardiness += (self.Sim_time - self.queue[0].due_time)**2

        #see if anything will left after the departure
        if len(self.queue) > 1:
            #execute the departure
            self.queue.pop(0)

            #record the new departure and cancellation to the Calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)

            #find and record the next cancellation
            next_cancel = min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
```

```
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

        elif len(self.queue) == 1:
            #execute the departure
            self.queue.pop(0)

            #record the dummy departure and cancellation to the Calendar
            self.Event_Calendar[1:3] = (inf,inf)
        else:
            print 'Error in Departure'

    def Cancellation(self, where):
        '''
        #Manages the cancellation process
        '''
        #the where parameter tells where the cancellation is has to happen
        #where == index means cancellation of
        #the order with the index in the self.queue
        if where == 0:
            self.Unness_Work += self.Sim_time - \
            self.Event_Calendar[1,0]
            #time between the start of process and the cancellation

            #if there are orders waiting
            if len(self.queue) > 1:
                #drop the InService
                self.queue.pop(0)
                #update the departure record in the Calendar
                self.Event_Calendar[1] = \
                (self.Sim_time, self.Sim_time + self.queue[0].comp_time)

                #find and record the next cancellation
                next_cancel = min(self.queue, key=lambda order: order.canc_time)
                #find the index of the next cancellation
                index = self.queue.index(next_cancel)
                #update the cancellation in the Calendar
                self.Event_Calendar[2] = (index, next_cancel.canc_time)

            elif len(self.queue) == 1:
                #drop the inService
                self.queue.pop(0)

                #as we have no other orders waiting we record
                #a dummy departure and cancellation to the Calendar
                self.Event_Calendar[1:3] = (inf, inf)
            else:
                print 'Cancellation Error!'
        else:
```

```
            #drop the order from the queue
            self.queue.pop(where)

            #find and record the next cancellation
            next_cancel = min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

    def Revolver(self):
        '''
        #Manages all the rest of the SMART line function
        '''
        while self.Sim_time <= self.End:
            if self.table:
                #create a 1-dimension array from
                #the second column of the Calendar
                next_times = self.Event_Calendar[0:,1]
                #see what time is sooner
                if next_times.argmin() == 0:
                    #append the simulation time
                    self.Sim_time = next_times.min()
                    self.Arrival()
                elif next_times.argmin() == 1:
                    self.Sim_time = next_times.min()
                    self.Departure()
                elif next_times.argmin() == 2:
                    self.Sim_time = next_times.min()
                    self.Cancellation(int(self.Event_Calendar[2,0]))
            else:
                print 'Out of table'
                break
```

A.5 The SMART object

```
from numpy import *
from scipy.integrate import quad
from math import exp
import random
from Test import Test


class SMART(object):
```

```
'''
This is the SMART simulation line itself. It has to be self-contained.
This is a process-oriented simulation. The following part consist
definition of several processes that start at the initialization of
the SMART object and live until the end of the simulation.
'''

def __init__(self, table, end, alpha, un_cost):
    '''
    Constructor
    If we consider the probability of it to be
    really low (we are using the 'float' values for the arrival times)
    we can still identify the orders by their arrival times
    '''
    self.alpha = alpha
    self.un_cost = un_cost

    #make the random table the class parameter
    self.table = [order for order in table]

    #the end time
    self.End = end
    #create the Event_Calendar
    self.Event_Calendar = zeros((3,2), dtype = 'f4')
    #the first field the arrival time of the order, the second - the event time
    #populate the array with infinitely large values
    self.Event_Calendar[:] = float('inf')
    #the simulation time of the line
    self.Sim_time = float(0)

    #create the line queue which is an array that contains all the numbers in the
    self.queue = []
    #create the records for Unnecessary Work and Tardiness
    self.Tardiness = 0
    self.Unness_Work = 0
    '''
    Initialization
    '''
    if self.table:
        #initialization of the line: the first order arrived,
        #it has seized the server and it's departure time has been calculated.
        self.queue += [self.table.pop(0)]

        #record the next arrival event
        self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)

        #calculate the departure time of
        #the initial arrival and record it in the calendar
```

```
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)

            #record the cancellation time of the arrival into the calendar
            self.Event_Calendar[2] = (0, self.queue[0].canc_time)
        else:
            print'The simulation table is not long enough!'

    def Decision_Rule(self, queue, MC_number):
        '''
        This function gets an arrangement proposed by
        the Arrival insertion process, runs the Monte-Carlo simulation,
        and returns the averaged cost of the arrangement
        '''

        #for the number of simulations given by the MC_number,
        #simulate the emptying process of the given queue
        #and then calculate the averaged cost

        Costs = zeros((MC_number,2))

        for n in range(MC_number):

            #create the event calendar
            Calendar = zeros((2,2), dtype = 'f4')
            Calendar[:] = float('inf')
            #copy the current time into the local simulation loop
            sim_time = self.Sim_time
            #create the tardiness and unnecessary work records
            tardiness = 0
            unnecessary_work = 0

            #copy the simulation queue from the initial queue
            sim_queue = [test for test in queue]

            ##INITIALIZATION
            #for every test in the simulation queue we
            #create its random cancellation time
            for test in sim_queue:
                self.canc_time = \
                test.arr_time + random.expovariate(test.canc_rate)

            #populate the Event Calendar
            Calendar[0] = (sim_time, sim_time + sim_queue[0].comp_time)

            #find and record the next cancellation
            n_canc = min(sim_queue, key=lambda test: test.canc_time)
            #find the index of the next cancellation
```

```
ind = sim_queue.index(n_canc)
#update the cancellation in the Calendar
Calendar[1] = (ind, n_canc.canc_time)

#run the simulation until the queue is empty
while sim_queue:
    #create a 1-dimensional array of the next times
    n_times = Calendar[0:,1]

    #if the DEPRATURE time is sooner - execute the departure
    if n_times.argmin() == 0:
        #bring the simulation time to the departure event time
        sim_time = n_times.min()
        #record the accrued tardiness if any
        if sim_time > self.queue[0].due_time:
            tardiness += 1

        #execute the departure
        sim_queue.pop(0)

        #record the new departure and
        #cancellation time if any orders left in the queue
        if sim_queue:
            Calendar[0] = (sim_time, sim_time + sim_queue[0].comp_time)

            #find and record the next cancellation
            n_canc = min(sim_queue, key=lambda test: test.canc_time)
            #find the index of the next cancellation
            ind = sim_queue.index(n_canc)
            #update the cancellation in the Calendar
            Calendar[1] = (ind, n_canc.canc_time)
        else:
            Calendar[:] = float('inf')


    #if the CANCELLATION time is sooner -
    #execute the cancellation
    elif n_times.argmin() == 1:
        #bring the simulation time to the cancellation event time
        sim_time = n_times.min()

        #record the accrued unnecessary work if any
        if int(Calendar[1,0]) == 0:
            unnecessary_work += sim_time - Calendar[0,0]

            #execute the cancellation
            sim_queue.pop(0)
```

```
                        #record the new departure and
                        #cancellation time if any orders left in the queue
                        if sim_queue:
                            Calendar[0] = (sim_time, sim_time + sim_queue[0].comp_time)

                            #find and record the next cancellation
                            n_canc = min(sim_queue, key=lambda test: test.canc_time)
                            #find the index of the next cancellation
                            ind = sim_queue.index(n_canc)
                            #update the cancellation in the Calendar
                            Calendar[1] = (ind, n_canc.canc_time)
                        else:
                            Calendar[:] = float('inf')

                    #if the cancellation is not from InService -
                    #just drop the test and find a new cancellation time
                    else:
                        sim_queue.pop(int(Calendar[1,0]))

                        #find and record the next cancellation
                        n_canc = min(sim_queue, key=lambda test: test.canc_time)
                        #find the index of the next cancellation
                        ind = sim_queue.index(n_canc)
                        #update the cancellation in the Calendar
                        Calendar[1] = (ind, n_canc.canc_time)

                else:
                    print 'In the small loop both departure and cancellation times are equal!'

            #this concludes the small simulation.
            #Now we just calculate the total cost
            cost = self.un_cost*unnecessary_work
            Costs[n] = (cost, tardiness)

        #average the costs from the simulations
        av_cost = Costs[0:,0].mean()

        #if tardiness rating is higher than the alpha - report is as an unacceptable
        sum_tard = Costs[0:,1].sum()
        if sum_tard >= self.alpha*MC_number:
            tard = True
        else:
            tard = False

        return (av_cost,tard)

    def Arrival(self):
        '''
```

```
#This function manages the arrival processes.
#When we call the arrival function it simply calls
#the next arrival to come and
#if the server is available it takes it, if not it goes to the queue.
'''
#check if there is at least one order in the queue - it must be in service
if self.queue:
    '''
    Unlike the FIFO queue here we will run
    an algorithm to find the optimal insertion point for the new arrival
    '''
    #create the arrival
    arrival = self.table.pop(0)

    #create a test queue - copy of the self queue
    init_ord = [order for order in self.queue]
    #remove the first order, which is in service
    init_ord.pop(0)
    n = len(init_ord) #get the length the queue

    #set an infinitely large cost to hypothetical insert
    #position of n+1 (two orders after the last position)
    opt_dict = {float('inf'):n+1}
    #insert the new arrival at the testing position and calculate the cost
    for i in range(n+1):
        pos = n+1-i
        init_ord.insert(pos, arrival)

        #implement the Decision Rule function
        result = self.Decision_Rule(init_ord,100)
        #the cost is a result of 100 simulations

        #if the Tardiness constraint is met
        if result[1] == True:
            #record the position and the cost to the optimal dictionary
            opt_dict[result[0]] = pos

    #find the minimum cost insertion position
    opt_pos = opt_dict[min(opt_dict)]

    #insert the new arrival into the queue
    self.queue.insert(opt_pos+1,arrival)

    #-----------------------------------------------
    #The rest of the code in the same as in FIFO

    #record the next arrival to the Calendar
    self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)
```

```
            #find and record the next cancellation
            next_cancel = min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

        else:
            #execute the arrival
            self.queue += [self.table.pop(0)]

            #record the next arrival to the Calendar
            self.Event_Calendar[0] = (self.Sim_time, self.table[0].arr_time)

            #record the next departure and cancellation into the Calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)
            self.Event_Calendar[2] = (0, self.queue[0].canc_time)


    def Departure(self):
        '''
        #Manages the departure processes
        '''
        #calculate the accrued tardiness if any
        if self.Sim_time > self.queue[0].due_time:
            self.Tardiness += (self.Sim_time - self.queue[0].due_time)**2

        #see if anything will left after the departure
        if len(self.queue) > 1:
            #execute the departure
            self.queue.pop(0)

            #record the new departure and cancellation to the Calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)

            #find and record the next cancellation
            next_cancel = min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

        elif len(self.queue) == 1:
            #execute the departure
            self.queue.pop(0)
```

```
        #record the dummy departure and cancellation to the Calendar
        self.Event_Calendar[1:3] = (inf,inf)
    else:
        print 'Error in Departure'


def Cancellation(self, where):
    '''
    #Manages the cancellation process
    '''
    #the where parameter tells where the cancellation is has to happen
    #where == index means cancellation of
    #the order with the index in the self.queue
    if where == 0:

        #record the accrued unnecessary work if any
        self.Unness_Work += self.Sim_time - self.Event_Calendar[1,0]

        #if there are orders waiting
        if len(self.queue) > 1:
            #drop the InService
            self.queue.pop(0)
            #update the departure record in the Calendar
            self.Event_Calendar[1] = \
            (self.Sim_time, self.Sim_time + self.queue[0].comp_time)

            #find and record the next cancellation
            next_cancel = \
            min(self.queue, key=lambda order: order.canc_time)
            #find the index of the next cancellation
            index = self.queue.index(next_cancel)
            #update the cancellation in the Calendar
            self.Event_Calendar[2] = (index, next_cancel.canc_time)

        elif len(self.queue) == 1:
            #drop the inService
            self.queue.pop(0)

            #as we have no other orders waiting
            #we record dummy departure and cancellation to the Calendar
            self.Event_Calendar[1:3] = (inf, inf)
        else:
            print 'Cancellation Error!'
    else:
        #print 'Queue cancellation!'
        #drop the order from the queue
        self.queue.pop(where)
```

```
        #find and record the next cancellation
        next_cancel = \
        min(self.queue, key=lambda order: order.canc_time)
        #find the index of the next cancellation
        index = self.queue.index(next_cancel)
        #update the cancellation in the Calendar
        self.Event_Calendar[2] = (index, next_cancel.canc_time)


def Revolver(self):
    '''
    #Manages all the rest of the SMART line function
    '''
    while self.Sim_time <= self.End:
        if self.table:
            #create a 1-dimension array from
            #the second column of the Calendar
            next_times = self.Event_Calendar[0:,1]
            #see what time is sooner
            if next_times.argmin() == 0:
                #append the simulation time
                self.Sim_time = next_times.min()
                self.Arrival()
            elif next_times.argmin() == 1:
                self.Sim_time = next_times.min()
                self.Departure()
            elif next_times.argmin() == 2:
                self.Sim_time = next_times.min()
                self.Cancellation(int(self.Event_Calendar[2,0]))
        else:
            print 'Out of table'
            break
```

Appendix B

Python Code: The Dynamic Programming solution codes used in the Chapter 4.

B.1 The Backward recursion (Dijkstra).

```
import numpy as np
from math import exp, factorial as fc
from itertools import product
import time
'''
Here we attempt to solve the problem
using Dijkstra algorithm.
'''
#-----------------------------------------------------------------------------------------------
'''
Now we define the sets specific to the
data mentioned above - the state space and the action space
'''
class State(object):
    '''
    This class defines the object of state.
    The definition of state here is the current period -
```

```
t and the current matrix where rows correspond to
the cancellation parameters and the
columns correspond to the the due time
'''
def __init__(self, t, tuple, C, D, k_t, k_w):
    '''
    Constructor - all the information about the State
    that is created when the State-class is created
    '''
    self.t = t #current time period
    self.matrix = np.array(tuple).reshape(len(C),len(D))

    #create the empty action lists for the state
    self.Actions = []

    #the matrix of immediate cost is also
    #a property of the state - we calculate it here
    #create a CxD matrix with elements = t
    t_m = np.ones((len(C),len(D)), dtype='i4')
    t_m = t*t_m

    #create a CxD matrix with column elements = d - due times
    d_m = np.zeros((len(C),len(D)), dtype='i4')
    for i_d in range(len(D)):
        d_m[0:,i_d] = D[i_d]

    #create CxD zero matrix
    z_m = np.zeros((len(C),len(D)), dtype='i4')
    #create CxD 0.9 matrix
    o_m = np.ones((len(C),len(D)), dtype='i4')
    o_m = o_m*0.9

    #calculate the immediate cost matrix
    # the following expression simplifies to c(t) = \
    k_t*max{(t-d),0}^2 + k_w*(0.9 - C*max{(d-t),0}
    #if t > d: d-t < 0, Prob_T > 0, Prob_W = 0.9 - overdue
    #if t < d: t-d < 0, Prob_T = 0, Prob_W < 0.9 - before due
    self.im_cost = k_t*(np.maximum((t_m - d_m),z_m))**2 + \
    k_w*(o_m - (np.array(C).reshape(len(C),1))*(np.maximum((d_m-t_m),z_m)))

    #check it out!
    if self.im_cost.any() == 0:
        print self.im_cost


    #We will need this for the Dijkstra:
    self.value = float("inf")
    self.opt_act = None
```

```
    def __str__(self):
        '''
        Special function that returns data about
        the class when the object is called for printing
        '''
        info = str(self.t) + str(np.matrix(self.matrix))
        return info


class Action(object):
    '''
    This class defines the object of action
    Every state has an individual state of actions because
    actions depend on the numbers in the state matrix
    '''
    def __init__(self, State, a, AR):
        '''
        Constructor:
        The properties of the Action: FromState,
        ToState, Probability, immediate cost
        '''
    #Every action is defined by From State, To State and probability

        self.State = State
        #-----------------------------------------------------------------
        #Calculate the cost of the action
        #calculate the effective decision matrix -
        #only one non-zero element corresponding to the action
        self.a = a
        #here we simply filter the im_cost matrix of
        #the FromState object by the effective action
        self.cost = np.sum(State.im_cost*self.a)
        #-----------------------------------------------------------------
        #record the action into the lists of the State it belongs to
        self.State.Actions.append(self)

    def Probability(self,FromState,ToState):
        '''
        calculate the probability of action a
        from the FromState to the ToState
        '''
        #get the after-decision matrix at t
        Init_mat = FromState.matrix-self.a

        #separate and prepare the matrix at t+1
        Fin_mat = ToState.matrix
```

```
        #find out the element-wise difference
        #between the matrix in period t and t+1
        Dif_mat = Fin_mat - Init_mat

        #initialize the probability matrix
        P_mat = np.zeros((len(C),len(D)),dtype='f4')

        #populate the probability matrices
        for c in range(len(C)):
            for d in range(len(D)):
                #calculate the cancellation probability of an element
                c_p = 0.9 - C[c]*max([(D[d]-FromState.t),0])

                #calculate the probability of the element cancellation
                p_y = []
                for y in range(max(0,-Dif_mat[c,d]),Init_mat[c,d]+1):
                    p_y.append(fc(Init_mat[c,d])/(fc(y)*fc(Init_mat[c,d]-y))*\
                    (c_p**y)*((1-c_p)**(Init_mat[c,d]-y)))

                #and arrival
                p_x = []
                for x in range(max(0,Dif_mat[c,d]),(Fin_mat[c,d])+1):

                    #the new arrival possible only in time before the due
                    if D[d] >= FromState.t:
                        p_x.append(exp(-AR[c])*(AR[c]**x)/(fc(x)))
                    else:
                        p_x.append(1.0)

                #multiply element-wise and sum up
                #the elements of the resulting list
                P_mat[c,d] += (sum([a*b for a,b in zip(p_y,p_x)]))

        #multiply all the elements of the matrix to get the final
        #probability score and record it as an object property
        return np.prod(P_mat)


    def __str__(self):
        '''
        Special function that returns data about
        the class when the object is called for printing
        '''
        info = str(self.a)
        return info


class Problem(object):
```

```
'''
The problem class is similar to the Network class
in a classical Dijkstra. It is used to generate all
the states and actions objects and tight them
together and solve for the best actions (cheapest route)
'''
def __init__(self, C, D, N, T, AR, un_p, b_size, k_w, k_t):
    '''
    Constructor: just get a hold of the Problem parameters
    '''
    self.C = C
    self.D = D
    self.N = N
    self.T = T
    self.AR = AR
    self.un_p = un_p
    self.b_size = b_size
    self.k_w = k_w
    self.k_t = k_t

    #create a 2-dimensional list of states
    self.M = [[] for t in range(self.T)]
    self.GenerateProblem()

def GenerateProblem(self):
    '''
    This method generates the State and
    the Actions objects and ties them together
    '''
    #First generate the States
    for t in range(self.T):
        #create a Cartesian product of CxD-length
        #where every element can get up to N
        for combo in product(range(self.N+1), repeat = len(self.C)*len(self.D)):
            self.M[t].append(State(t, tuple(reversed(combo)) ,self.C,self.D, self.k_w, self.k
            #create a State object from the every realization

    #Then generate the Actions
    for t in range(self.T-1):
        #there is an action-link between all states
        #in the t period and all states in the t+1 period?
        #No! Some of the states are unreachable
        #in the natural way (through any actions)
        for state in self.M[t]:
            if np.sum(state.matrix) != 0:
                #define a set of possible actions
                for c in range(len(self.C)):
                    for d in range(len(self.D)):
```

```
                        #create the Action object
                        if state.matrix[c,d] > 0:
                            a = np.zeros((len(self.C),len(self.D)),dtype='i4')
                            a[c,d] = min(state.matrix[c,d], self.b_size)
                            tempAct = Action(state, a, self.AR)
            else:
                a = np.zeros((len(self.C),len(self.D)),dtype='i4')
                tempAct = Action(state, a, self.AR)


def Backward(self, cur_matrix):
    '''
    This method find the lowest cost action
    for any state realization
    '''
    #calculate the minimum expected value for every state
    #find the optimal action in the current state
    #-----------------------------------------------------------------

    #calculate values of the T-epoch
    for state in self.M[self.T-1]:
        #create a CxD matrix with elements = t
        T_m = np.ones((len(C),len(D)), dtype='i4')
        T_m = (T+(12/un_p))*T_m

        #create a CxD matrix with column elements = d - due times
        d_m = np.zeros((len(C),len(D)), dtype='i4')
        for i_d in range(len(D)):
            d_m[0:,i_d] = D[i_d]

        #calculate the final cost of processing
        # all the orders overnight
        state.value = k_t*np.sum(state.matrix*((T_m-d_m)**2))

    #start the vicious circle
    t = self.T-2
    while t != 0:
        for state in self.M[t]:
            #create an empty list of expected state
            #value due to the action
            state_action_val = []
            for action in state.Actions:
                #calculate the expected value
                #(plus the cost of action) of the state
                #is the action is taken
                act_val = action.cost
                for next_state in self.M[t+1]:
                    act_val += \
```

```
                        action.Probability(state,next_state)*next_state.value
                    #append the calculated value into the list
                    state_action_val.append(act_val)

                #find the lowest expected-value action for
                #the state and assign it as a value for the state
                state.value = min(state_action_val)
                #find the action corresponding to
                #the lowest value and record it as its optimal action

            #step down to the cur_time
            t -= 1

        #now we have calculated the values of states at all
        #periods down to cur_t+1. all we have
        #left to do is to find the action
        #minimizing the total expected cost of continuing
        #optimally from the the current state

        #First we find the state in the 0-time epoch
        for state in self.M[0]:
            if (state.matrix == cur_matrix).all():
                Cur_State = state
                #Then find the lowest expected cost action at this state
                state_action_val = []
                for action in state.Actions:
                    #calculate the expected value
                    #(plus the cost of action) of the state
                    #is the action is taken
                    act_val = action.cost
                    for next_state in self.M[1]:
                        act_val += \
                        action.Probability(Cur_State,next_state)*next_state.value
                    #append the calculated value into the list
                    state_action_val.append(act_val)

                #find the lowest expected-value action for
                #the state and assign it as a value for the state
                Cur_State.value = min(state_action_val)
                #find the action corresponding to
                #the lowest value and record it as its optimal action
                Cur_State.opt_act = \
                Cur_State.Actions[state_action_val.index(Cur_State.value)]

        #return the optimal action
        return Cur_State.opt_act

#=============================================================================================
```

```
#NOW THE ACTUAL PROBLEM RUN

#PARAMETERS
#Define the set of the due times.
D = [5,7]

#Define the set of the cancellation parameters
C = [0.01,0.04,0.03]

#Define the set of arrival rates
AR = [0.1,0.3,0.2]

#Define the maximum number of the test
#orders of a specific cancellation-due combination
N = 5

#define the length of the unit time-period
un_p = 0.25 # in portions of an hour

#Define the length of the planning horizon - the time left in the day
T = 7   #in unit periods

#Define the batch-size
b_size = 3

#define the unit-costs of the wasted
#work and tardiness respectively
k_w = 1
k_t = 1


#THE PROBLEM RUN

cur_t = 3 #set the current time period

cur_matrix = np.array((5,0,3,4,1,0)).reshape(len(C),len(D))
#set the current state-matrix

print cur_matrix

Horizon = T-cur_t #set the time-horizon - what's left until the end of the shift

MY_PROBLEM = Problem(C, D, N, Horizon, AR, un_p, b_size, k_w, k_t)
print 'Done setting up!'
start = time.clock()

next_action = MY_PROBLEM.Backward(cur_matrix)
```

```
finish = time.clock()
comp_time = finish - start
#convert the time in second into hours and minutes
m, s = divmod(comp_time, 60)
h, m = divmod(m, 60)
print "The calculation took: %d:%02d:%02d" % (h, m, s)
print ''

print next_action
```

B.2 The Linear Programming transformation.

```
import numpy as np
from gurobipy import *
from itertools import product
from math import exp, factorial as fc
import time
from hashlib import sha1
'''
The following class was developed and
posted by Perroni Filho in his blog at:
http://machineawakening.blogspot.com/2011/03/
making-numpy-ndarrays-hashable.html
I have just copied it here.
'''
class hashable(object):
    r'''
        Hashable wrapper for ndarray objects.

        Instances of ndarray are not hashable,
        meaning they cannot be added to
        sets, nor used as keys in dictionaries.
        This is by design - ndarray
        objects are mutable, and therefore cannot
        reliably implement the
        __hash__() method.

        The hashable class allows a way around
        this limitation. It implements
        the required methods for hashable objects
        in terms of an encapsulated
        ndarray object. This can be either a copied
        instance (which is safer)
```

```
        or the original object (which requires the user
        to be careful enough
        not to modify it).
    '''
    def __init__(self, wrapped, tight=False):
        r'''Creates a new hashable object encapsulating an ndarray.

            wrapped
                The wrapped ndarray.

            tight
                Optional. If True, a copy of the input ndaray is created.
                Defaults to False.
        '''
        self.__tight = tight
        self.__wrapped = np.array(wrapped) if tight else wrapped
        self.__hash = int(sha1(wrapped.view(np.uint8)).hexdigest(), 16)

    def __eq__(self, other):
        return np.all(self.__wrapped == other.__wrapped)

    def __hash__(self):
        return self.__hash

    def unwrap(self):
        r'''Returns the encapsulated ndarray.

            If the wrapper is "tight", a copy of
            the encapsulated ndarray is
            returned. Otherwise, the encapsulated
            ndarray itself is returned.
        '''
        if self.__tight:
            return np.array(self.__wrapped)

        return self.__wrapped

#=======================================================================================================
#Here we start the LP_model.

#C - the set of the cancellation probabilities parameters.
#C <= 1/T because otherwise the value of the immediate
#wasted work would be negative
#D - the set of the due times
#N - max number of orders on a particular position
#Ar - the set of the arrival time. The same length as C.
#cur_matrix - the current matrix.
#u_t - unit time
```

```
#T - length of the planning horizon - the time left in u_t
#k_t - unit cost of tardiness
#k_w - unit cost of wasted work
#b_size - batch size
#=============================================================================================

def LP_opt(C,D,N,Ar,cur_matrix,T,u_t,k_t,k_w,b_size):
    '''
    This function takes the values of the cancellation
    parameters and the due times
    find the optimal values of the J(value)-functions in
    every state using Gurobi.
    '''
    #create and populate the action space

    #initialize the action space list
    A = []

    #initialize a dummy zero-array of size CxD
    Dummy = np.zeros((len(C),len(D)),dtype='i4')

    #create all the possible decisions a and
    #add them to the list A
    for c in range(len(C)):
        for d in range(len(D)):
            a = Dummy.copy()
            a[c,d] = 1
            A.append(hashable(a))

        ###########################################################################
        #Optimize using Gurobi
        ###########################################################################

    #Create the Gurobi model
    m = Model('Lab_Processing_LP')

    #-------------------------------------------------------------------------------------
    #Create and integrate the variables

    #the Python dictionaries can hold any type of data:
    #e.g. the Gurobi variable.

    #let me start all over again.
    J = {}
    J[0,hashable(cur_matrix)] = \
    m.addVar(0.0, GRB.INFINITY, 1.0, GRB.CONTINUOUS,
    name='%s_%s' % (0, cur_matrix))
    for t in range(1,T+1):
```

```
    #for each decision epoch in the planning horizon
    #we create the Cartesian product of the matrices CxD
    mat_size = len(C)*len(D)
    for combo in product(range(N+1), repeat = mat_size):
    #first create a range of numbers or order C*D with
    #every element taking from 0 to N
        matr = \
        np.array(tuple(reversed(combo))).reshape(len(C),len(D))
        #than we create an array from that number
        #make the matrix hashable - to be able to
        #use it as a dictionary key
        matr = hashable(matr)
        #assign a variable to this state
        J[t,matr] = m.addVar(0.0, GRB.INFINITY, 1.0,
        GRB.CONTINUOUS, name='%s_%s' % (t, matr.unwrap()))

#integrate the new variables
m.update()

#-------------------------------------------------------------------------------
#Set the objective function
#In this case the objective function is defined
#while adding the variables.
#-------------------------------------------------------------------------------
#Create and integrate the constraints
#The first is the constraint made from
#the Bellman equation
#-------------------------------------------------------------------------------
#before writing the constraint for every decision
#we calculate the components that do not depend
#on the decision:

for t in range(T):

    #the c(t) - matrix of the immediate costs and

    #create a CxD matrix with elements = t
    t_m = np.ones((len(C),len(D)), dtype='i4')
    t_m = t*t_m

    #create a CxD matrix with column
    #elements = d - due times
    d_m = Dummy.copy()
    for i_d in range(len(D)):
        d_m[0:,i_d] = D[i_d]


    #create CxD zero matrix
```

```
z_m = Dummy.copy()

#create CxD 0.9 matrix
o_m = np.ones((len(C),len(D)), dtype='i4')
o_m = o_m*0.9

#calculate the c_t - the immediate cost matrix
# the following expression simplifies to
#c(t) = k_t*max{(t-d),0}^2 + k_w*(0.9 - C*max{(d-t),0}
#if t > d: d-t < 0, Prob_T > 0, Prob_W = 0.9 - overdue
#if t < d: t-d < 0, Prob_T = 0, Prob_W < 0.9 - before due
c_t = k_t*(np.maximum((t_m - d_m),z_m))**2 + \
k_w*(o_m - (np.array(C).reshape(len(C),1))*(np.maximum((d_m-t_m),z_m)))

#-------------------------------------------------------------------------------------
#first, we prepare the components of the J(t+1) matrix:
# the matrices of departures and arrivals.

#get a hold of the state matrix at time t
for key in J.iterkeys():
    if key[0] == t:

        #copy its matrix-tuple
        j_matr = key[1].unwrap()

        #write the constraint for every
        #decision made on the matrix
        for a in A:
            #The following block of code is to
            #calculate the immediate cost of the action a
            #we form a CxD matrix with one non-zero
            #entry equal to the minimum of
            #the batch size or the j_matr element
            a_dec = \
            np.minimum(b_size*a.unwrap(),j_matr*a.unwrap())

            #calculate the actual cost by 'filtering'
            #the cost_matrix with the minimum
            #of a_dec and j_dec
            im_cost = np.sum(c_t*a_dec)

            #-------------------------------------------------
            #initialize the linear expression for the constraint
            Constr_J = LinExpr()

            #add the first unique term to the constraint
            Constr_J.addTerms(1.0, J[key])
```

```
#get the after-action matrix
j_matr = j_matr-a_dec

#a safety check
if np.prod(j_matr) < 0:
    print 'Bad action '+str(a_dec)+str(j_matr)

P = 0
for c in range(len(C)):
    for d in range(len(D)):
        #find the probability of the cancellation
        #and the arrival rate of the elements
        #considering the arrival time
        c_p = 0.9-C[c]*max((D[d]-t),0)
        #the cancellation probability
        if D[d] > t:
            # the arrival rate from the
            #list of arrival rates
            ar_r = Ar[c]
        else:
            ar_r = 0

        #get the number of the order
        #in the c,d element of the matrix
        element = j_matr[c,d]


        for y in range(element+1):
            #calculate the binomial probability
            #of cancellation y orders out of n
            #orders in the element
            prob_y =  \
            fc(element)/(fc(y)*fc(element-y))* \
            (c_p**y)*((1-c_p)**(element-y))

            #a safety check
            if prob_y>1:
                print fc(element)/(fc(y)*fc(element-y))

            for x in range(N-element+y+1):
                #calculate the Poisson
                #probability of arrival of x new orders
                prob_x = exp(-ar_r)*(ar_r**x)/fc(x)

                #form the J_t+1 matrix
                #copy the j_matr
                j_matr_new = j_matr.copy()
                #append the element of
```

```
                                    #the new j_matrix
                                    j_matr_new[c,d] = element-y+x
                                    #make it a tuple
                                    j_matr_new = hashable(j_matr_new)
                                    #a safety check
                                    if prob_y*prob_x>1:
                                        print prob_y*prob_x

                                    #add the constraint
                                    Constr_J.addTerms(-(prob_y*prob_x),
                                    J[t+1,j_matr_new])

                    #write the actual constraint
                    m.addConstr(Constr_J, GRB.LESS_EQUAL,
                    im_cost, 'Constraint_J_' + str(t))


#-------------------------------------------------------------------------------------------
#The second is the constraint of
#the Boundary condition

for key in J.iterkeys():
    if key[0] == T:
        #for all the actions in the final decision epoch
        #the result depend only on the state
        for a in A:
            #create a CxD matrix with elements = t
            T_m = np.ones((len(C),len(D)), dtype='i4')
            T_m = (T+(12/u_t))*T_m

            #create a CxD matrix with column
            #elements = d - due times
            d_m = Dummy.copy()
            for i_d in range(len(D)):
                d_m[0:,i_d] = D[i_d]

            #separate and prepare the matrix
            matr = key[1].unwrap()

            #calculate the final cost of processing
            #all the orders overnight
            fin_val = k_t*np.sum(matr*((T_m-d_m)**2))

            m.addConstr(LinExpr(1.0, J[key]),
            GRB.EQUAL, fin_val, 'Constraint_F_' +str(T))

#-------------------------------------------------------------------------------
#Solve the model
```

```
m.ModelSense = -1 #model sense = +1 to minimize,
#-1 to maximize
m.optimize()
#-------------------------------------------------------------------------------
#Display the results

#For every action available in the current state
    #Calculate the expected value of that action
        #Calculate a probability of finding ourselves
        #in a particular state realization of the next
        #period after taking the action
        #Multiply that probability with the state
        #value (from the optimized set)
#pick up the action with the lowest expected action

#First of all we form a dictionary of
#the current+1 time states values
OptMat_dict = {}
next_time = {}
if m.status == GRB.status.OPTIMAL:
    #find up all the optimized variable of time 1
    #(current time == 0) and add them to the dictionary
    for v in m.getVars():
        if str(1)+"_" in v.VarName:
            #strip the name of the 1_
            name = v.VarName.lstrip('1_')
            next_time[name] = (v.X)


    #recreate the matrices for the next_time values
    #and add them as keys to the OptMat dictionary
    for combo in product(range(N+1), repeat = mat_size):
        matr = \
        np.array(tuple(reversed(combo))).reshape(len(C),len(D))
        OptMat_dict[hashable(matr)] = (next_time[str(matr)])


#the following block we're calculating the expected
#value from every valid action
Exp_score = {}
#create an empty dictionary for the expected values

for a in A:
    if (cur_matrix*a.unwrap() != 0).any():
        #calculate the effective action on the current state
        a_dec = np.minimum(b_size*a.unwrap(),j_matr*a.unwrap())

        #calculate the matrix after the action
```

```
Init_mat = cur_matrix-a_dec

a_score = 0 #initialize the expectation score
#for every state-matrix in the next time
#calculate the probability of reaching it
#from the initial matrix
for state in OptMat_dict.iterkeys():
    #assign it as the final matrix
    Fin_mat = state.unwrap()

    #find the element-wise difference between
    #the matrices in the periods t and t+1
    Dif_mat = Fin_mat - Init_mat

    #initialize the probability matrix
    P_mat = Dummy.copy()

    #populate the probability matrices
    for c in range(len(C)):
        for d in range(len(D)):
            #calculate the cancellation
            #probability of an element
            c_p = 0.9 - C[c]*max([(D[d]-t),0])

            #calculate the probability of
            #the element cancellation
            p_y = []
            for y in range(max(0,-Dif_mat[c,d]),Init_mat[c,d]+1):
                p_y.append(fc(Init_mat[c,d])/(fc(y)*\
                fc(Init_mat[c,d]-y))*(c_p**y)*((1-c_p)**(Init_mat[c,d]-y)))

            #and arrival
            p_x = []
            for x in range(max(0,Dif_mat[c,d]),(Fin_mat[c,d])+1):
                #the new arrival possible only
                #in time before the due
                if D[d] >= t:
                    p_x.append(exp(-AR[c])*(AR[c]**x)/(fc(x)))
                else:
                    p_x.append(1.0)
                    #if the time is past due the only
                    #probability of transition is
                    #the cancellation probability

        #multiply element-wise and sum up
        #the elements of the resulting list
        P_mat[c,d] += (sum([x*y for x,y in zip(p_y,p_x)]))
```

```
                #to calculate the probability score of this action
                #into the dictionary multiply the elements of
                #the P_mat - independent events
                Prob_score = np.prod(P_mat)

                #multiply the probability with the state value
                #and add to the general score
                a_score += Prob_score*OptMat_dict[state]

            #record the total score as the value into
            #the Expectation dictionary
            Exp_score[a] = (a_score)

    #find the minimum expected value action and
    #return it as an optimal
    action = min(Exp_score, key=Exp_score.get)
    return action.unwrap()

#=====================================================================

#NOW RUN THE PROGRAM

C = [0.01,0.03] #the set of the cancellation probabilities parameters.
C <= 0.9/(T=Dmax) because otherwise the value of
#the immediate wasted work would be negative
D = [5,9] #the set of the due times
N = 5 #max number of orders on a particular position
AR = [0.1,0.2] #the set of the arrival time. The same length as C

T = 9 #length of the planning horizon - the time left in u_t
u_t = 0.25 #unit time-period

k_t = 1 #unit cost of tardiness
k_w = 1 #unit cost of wasted work

b_size = 3 #batch size
#-----------------------------------------------------
cur_matrix = np.array((0,1,1,0)).reshape(len(C),len(D))
start = time.clock()
optimal_decision = LP_opt(C,D,N,AR,cur_matrix,T,u_t,k_t,k_w,b_size)
finish = time.clock()
comp_time = finish - start
#convert the time in second into hours and minutes
m, s = divmod(comp_time, 60)
h, m = divmod(m, 60)
print "The calculation took: %d:%02d:%02d" % (h, m, s)
print cur_matrix
print optimal_decision
```