MODELING AND ANALYZING SPREAD OF EPIDEMIC DISEASES: CASE STUDY BASED ON CERVICAL CANCER

A Thesis

by

HODA PARVIN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Industrial Engineering

MODELING AND ANALYZING SPREAD OF EPIDEMIC DISEASES: CASE STUDY BASED ON CERVICAL CANCER

A Thesis

by

HODA PARVIN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Natarajan Gautam
Committee Members,	Eylem Tekin
	Daren B. H. Cline
Head of Department,	Brett A. Peters

May 2007

Major Subject: Industrial Engineering

ABSTRACT

Modeling and Analyzing Spread of Epidemic Diseases:

Case Study Based on Cervical Cancer. (May 2007)

Hoda Parvin, B.S., Sharif University of Technology

Chair of Advisory Committee: Dr. Natarajan Gautam

In this thesis, health care policy issues for prevention and cure of cervical cancer have been considered. The cancer is typically caused by Human Papilloma Virus (HPV) for which individuals can be tested and also given vaccinations. Policymakers are faced with the decision of how many cancer treatments to subsidize, how many vaccinations to give and how many tests to be performed in each period of a given time horizon. To aid this decision-making exercise, a stochastic dynamic optimal control problem with feedback was formulated, which can be modeled as a Markov decision process (MDP). Solving the MDP is, however, computationally intractable because of the large state space as the embedded stochastic network cannot be decomposed. Hence, an algorithm was proposed that initially ignores the feedback and later incorporates it heuristically. As part of the algorithm, alternate methodologies, based on deterministic analysis, were developed, Markov chains and simulations to approximately evaluate the objective function.

Upon implementing the algorithm using a meta-heuristic for a case study of the population in the United States, several measures were calculated to observe the behavior of the system through the course of time, based on the different proposed policies. The policies compared were static, dynamic without feedback and dynamic with feedback. It was found that the dynamic policy without feedback performs almost as well as the dynamic policy with feedback, both of them outperforming the static policy. All these policies are applicable and fast for easy what-if analysis for the policymakers.

Dedicated to: *You*

For always being there for me...

ACKNOWLEDGEMENTS

In the first place, I would like to express my gratitude to my mother, Faranak Moradhamzeh, my father, Mohammadnabi Parvin, and my brother, Amin, for their support and encouragement because without them, I would have never been able to make it this far.

I would also like to express my deepest thanks to my grandmother, Giti Bazyaran, and my grandfather, Ghasem Moradhamzeh, who always valued and motivated my academic pursuits.

I offer my sincerest gratitude to my advisor, Dr. N. Gautam, who has supported me in my thesis with his patience and knowledge, while giving me leeway to work independently.

Many thanks to Dr. E. Tekin and Dr. D. B. Cline for serving in my committee and guiding me through this research work.

I would like to express thanks to my friend, Piyush Goel, for all his help in this research work and his friendship, which is priceless to me.

I really want to express my appreciation for my friends: Elnaz Jalili, Amir Paknejad, Arash Haghshenas, Nima Zonoozi, Shahram Amini, Sahar Gargari, Danial Kaviani, Reza Sharifi, Young Myong Ko, Sahar Faraji, Mehdi Madooliat, Ali Eskandari, Arash Dabirzadeh, and all the other good friends, who always supported me with their kindness and cordiality.

TABLE OF CONTENTS

Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Cervical cancer, as the name suggests, is the cancer of the cervix which is the lower part of the uterus. This cancer affects women, and is the second most prevalent cancer in women all around the world; the first being breast cancer. According to an estimate by the American Cancer Society, about 9710 cases of invasive cervical cancer would have been diagnosed in the United States in the year 2006, and about 3700 women would unfortunately die from the disease [12]. The situation is much worse in other parts of the world, as shown in Figure 1 (Source: Katz and Wright [20]). The risk of getting cancer is elevated due to smoking, abnormal diet, genetic reasons, etc. Once a person gets cervical cancer, there are several options for her treatment: local surgery, radiation therapy and chemotherapy [13]. The good news is that treatments for cervical cancer are fairly successful [1]. However the cost of treatment is significantly higher than, say in the case of breast cancer. Hence not all patients can afford the treatment, as mentioned by Wolstenholme and Whynes [30].

As a result, cost-effective treatment of cervical cancer is extremely important. Another approach is to control Human Papilloma Virus (HPV), which is the primary agent causing the cancer. HPV is a sexually transmitted virus, which includes more than 100 different stains or types. However, a person with HPV (but not cancer) does not show any symptoms. It takes several years for a person affected by HPV to get cervical cancer [11]. In addition, not all persons with HPV will develop cervical cancer, and in fact, all the HPV in a person's body can disappear, like other viruses such as influenza. Furthermore, there are vaccinations that people who do not have HPV can take to prevent an attack, however

---

This thesis follows the style of Queueing Systems.

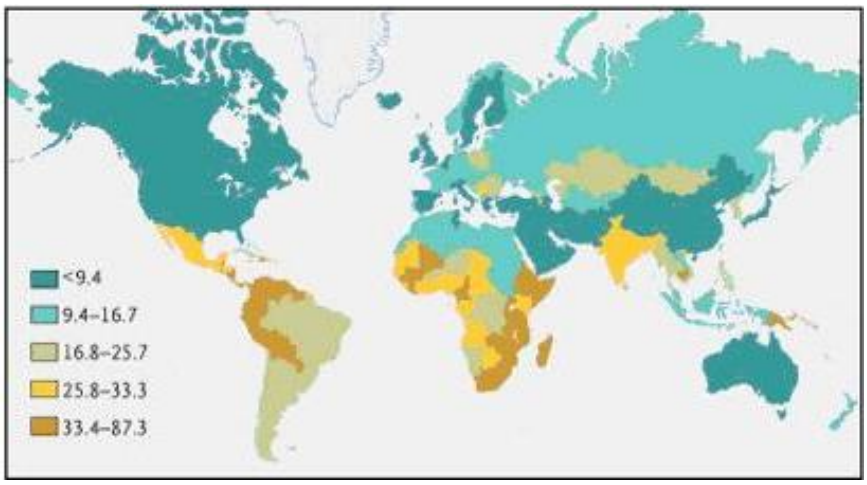

Figure 1. Age-Standardized Incidence Rates of Cervical Cancer per 100,000

the vaccinations are not effective for people who already have HPV. The vaccination effect lasts for about 4 years, and the individual has to retake it periodically. Thereby, in order to prevent the spread of HPV, it would be wise to get vaccinated as well as tested to see if one has HPV.

There are several options for policymakers to prevent and cure cervical cancer [9]. For example, the policymakers could allocate funds for: vaccination, testing patients and subsidizing treatment. Generally, the biggest constraint for prevention and treatment plans is the limited budget. It is usually a difficult decision to allocate budgets for different phases of the plan, viz. vaccinations, tests and treatments. If surplus money is allocated to treatments, it enhances the chances for people to get the virus, which would in turn cause more cancer patients and deaths. On the other hand, if superfluous money is allocated to vaccinations, it would become the cause for more deaths due to the unavailability of subsidized treatments. Examination or testing is the means of educating and notifying people of the disease and their individual condition. So, when a person is affected, she can help in reducing the spread of the disease by reducing the contact with other people.

The main objective of this research is to aid the policymaker to decide the allocation of budget for vaccination, testing and treatment subsidy. For this purpose, a methodology for budget allocation to optimize an objective specified by the policymaker has been developed. Since we are dealing with multi-dimensional decision variables under a stochastic-network environment, modeling and analyzing this problem is non-trivial, even when using dynamic programming. The number of states in the system is very large. The transition rates in this case are also dependent upon the state of the system. In essence the budget allocation problem over a finite horizon is a stochastic dynamic program where the objective is to determine the optimal set of actions in each stage of the horizon, which is a dynamic control problem with feedback. The contribution of this research is not only to provide a tool for policymakers to effectively deal with a serious health care issue, but also to develop a methodology to handle complex dynamic control problems effectively.

Following is the organization of this thesis. In Chapter II, a review of the literature pertinent to the application of mathematical tools to healthcare and to the budget-allocation problem described above, is presented. The problem is presented in detail, along with notations used, in Chapter III. In Chapter IV, the system is characterized as a stochastic network under dynamic control with feedback and modeled as a Markov decision process (MDP). However, the MDP is intractable due to the curse of dimensionality. Therefore, approximations to solve the MDP are suggested in Chapter V. Some numerical results on a case study in the United States are presented in Chapter VI. Finally, findings are discussed along with the concluding remarks in Chapter VII.

CHAPTER II

LITERATURE REVIEW

In this chapter a review of the literature relevant to the problem at hand is presented by categorizing it into three phases: historical perspective, mathematical modeling and optimization.

II.1. A Brief History

Mathematical modeling of epidemic diseases has been an active area of research since the 1920s. A number of statistical models have been developed in order to be able to predict various factors, such as the rate of the spread of disease, the number of infected people, mortality rate, etc. within a given time horizon. Early research was mainly devoted to developing deterministic models. This is mainly due to the fact that these models are simpler. Anderson and May [4] presented the various existing deterministic models, along with some applications based on real data, in their book. However, the spread of such a disease should only be defined by specifying a probability of disease transmission. Hence, whenever possible, a stochastic model must be implemented. Stochastic epidemic models started to come around with the one proposed by McKendrick [25]. This work was a stochastic continuous time version of the deterministic model proposed by Kermack and McKendrick [21]. However, this model did not gain as much popularity as the one proposed by Reed and Frost (but never published) which is known as the chain-binomial model (Anderson and Britton [5]). Stochastic continuous time epidemic models began to gain much more attention when Barlett [7] studied the stochastic version of the Kermack-McKendrick model. Since then, a substantial amount of research has been done in this field. Gabriel et al. [23], Bailey [6] and Anderson and May [4] are excellent resources that covered stochastic as well

as deterministic models. These sources also discussed about the statistical inference and a large number of applications to real data. More recently, Daley and Ganni [14] reviewed the existing stochastic and deterministic models and include statistical inference and several historical remarks. Andersson and Britton [5] also provided an excellent summary of the stochastic models that are being used in the area. In addition, Diekmann and [15] were mainly concerned with mathematical epidemiology of infectious diseases and used real data for illustrations.

II.2. Mathematical Models

Mathematical models have become important tools in analyzing the spread and control of infectious diseases. The model formulation process clarifies assumptions, variables, and parameters; moreover, models provide conceptual results such as thresholds, basic reproduction numbers, contact numbers, and replacement numbers. Mathematical models and computer simulations are useful experimental tools for building and testing theories, assessing quantitative conjectures, answering specific questions, determining sensitivities to changes in parameter values, and estimating key parameters from data. Understanding the transmission characteristics of infectious diseases in communities, regions, and countries can lead to better approaches to decreasing the transmission of these diseases. Mathematical models are used in comparing, planning, implementing, evaluating, and optimizing various detection, prevention, therapy, and control programs. Epidemiology modeling can contribute to the design and analysis of epidemiological surveys, suggest crucial data that should be collected, identify trends, make general forecasts, and estimate the uncertainty in forecasts.

Mathematical epidemiology seems to have grown exponentially starting in 1950's so that a tremendous variety of models have now been formulated, mathematically analyzed,

and applied to infectious diseases.

The most common type of models used for infectious diseases are commonly termed as SIR (Susceptible - Infected - Removed) models, first proposed by Reed and Frost [5]. Here, it is assumed that initially, there is a mixed population. That is, there are some infectious individuals (I) and some susceptible individuals (S). During her infectious period, an infective person makes contact with a susceptible individual with a given probability. Different researchers used different methods to come up with this probability, like defining the contact number by Hethcote [19], etc. If a susceptible person comes in contact with the infectious person, he/she also becomes infectious and is immediately able to infect other individuals. There is usually a distribution assigned to the duration of the infectious period. An individual is considered removed (R) when he/she becomes immune to the disease after his infectious period is over, and plays no further part in the spread of the disease.

Many models for the spread of infectious diseases in populations have been analyzed mathematically and applied to specific diseases. A very valuable source in the area of mathematical treatment of infectious diseases is attributed to Hethcote [19]. Due to the relevance of this source to research work at hand, a rather comprehensive review of this work is in order. Hethcote reviewed threshold theorems involving basic reproduction number R_0 , the contact number σ, and the replacement number R for the classic SIR epidemic and endemic models. He also obtained similar results with new expressions for R_0 in MSEIR and SEIR endemic models with either continuous age or age groups. He provided estimated values for R_0 and σ for various diseases including measles in Niger and pertussis in the United States. He also surveyed the then existing models with age structure, heterogeneity, and spatial structure. To grasp a better understanding of the subject matter, widely accepted nomenclature in this area has been discussed ahead.

Compartments with labels such as M, S, E, I, and R are often used for the epidemiological classes as shown in Figure 2 [19]. This model has the class M, with passive

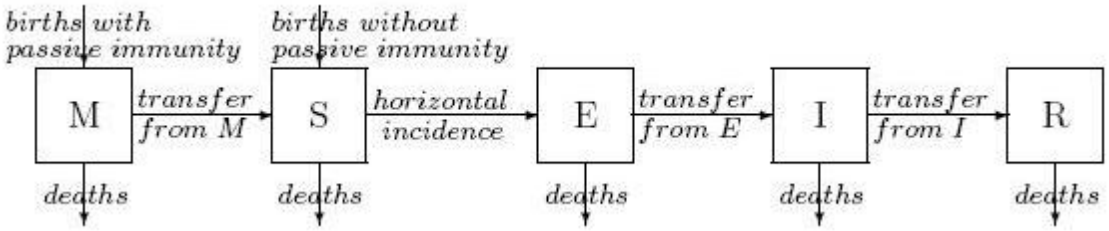

Figure 2. The General Transfer Diagram for the MSEIR Model

immunity, the susceptible class S, the exposed class E, the infective class I, and the recovered class R. If a mother has been infected with a disease, her newborn infant develops temporary passive immunity to that infection. The class M contains these infants with passive immunity. After the maternal antibodies disappear from the body, the infant moves to the susceptible class S. Infants who do not have any passive immunity, because their mothers were never infected, also enter the class S of susceptible individuals; that is, those who can become infected. When there is an adequate contact of a susceptible with an infective so that transmission occurs, then the susceptible enters the exposed class E of those in the latent period, who are infected but not yet infectious. After the latent period ends, the individual enters the class I of infected people, who are infectious in the sense that they are capable of transmitting the infection. When the infectious period ends, the individual enters the recovered class R consisting of those with permanent infection-acquired immunity.

The choice of which compartments to include in a model depends on the characteristics of the particular disease being modeled and the purpose of the model. The passively immune class M and the latent period class E are often omitted, because they are not crucial for the susceptible-infective interaction. Acronyms for epidemiology models are often based on the flow patterns between the compartments such as MSEIR, MSEIRS, SEIR, SEIRS, SIR, SIRS, SEI, SEIS, SI, and SIS. For example, in the MSEIR model shown in

Figure 2, passively immune newborns first become susceptible, then exposed in the latent period, then infectious, and then removed with permanent immunity. An MSEIRS model would be similar, but the immunity in the R class would be temporary, so that individuals would regain their susceptibility when the temporary immunity ended.

Allen [2] considered discrete-time models for single and multi-population, discrete-time epidemic models, or difference equations, of some well-known SI, SIR, and SIS epidemic models, which give rise to systems of nonlinear difference equations that are similar in behavior to their continuous analogues under the natural restriction that solutions to the discrete-time models be positive. She considered the entire system since the difference equation for infected people I in an SI model has a logistic form which can exhibit period-doubling and chaos for certain parameter values. She excluded these parameter values under the restriction that S and I are positive. She concluded that in the case of a discrete SIS model, positivity of solutions is not enough to guarantee asymptotic convergence to an equilibrium value (as in the case of the continuous model) and the positive feedback from the infective class to the susceptible class allows for more diverse behavior in the discrete model. She also argued that period-doubling and chaotic behavior is possible for some parameter values. In addition, the paper concluded that if births and deaths are included in the SI and SIR models (positive feedback due to births) the discrete models exhibit periodicity and chaos for some parameter values.

Allen and Burgin [3] analyzed and compared the dynamics of deterministic and stochastic discrete-time epidemic models. The discrete-time stochastic models they considered are Markov chains, approximations to the continuous-time models. They analyzed models of SIS and SIR type with constant population size and general force of infection, and then provided an analysis of a more general SIS model with variable population size. They concluded that in the deterministic models, the value of the basic reproductive number R_0 determines persistence or extinction of the disease. If $R_0 < 1$, the disease is eliminated,

whereas if $R_0 > 1$, the disease persists in the population. Since all stochastic models they consider have finite state spaces with at least one absorbing state, they argue that ultimate disease extinction is certain regardless of the value of R_0 . However, in some cases, they add, the time until disease extinction may be very long. In these cases, if the probability distribution is conditioned on non-extinction, then when $R_0 > 1$, there exists a quasi-stationary probability distribution whose mean agrees with deterministic endemic equilibrium. They also investigated the expected duration of the epidemic.

In the study of social networks, and in particular the spread of disease on networks, Newman [26] showed that a large class of standard epidemiological models, the so-called susceptible/infective/removed (SIR) models can be solved exactly on a wide variety of networks. In addition to the standard but unrealistic case of fixed infectiveness time and fixed and uncorrelated probability of transmission between all pairs of individuals, he solved cases in which times and probabilities are nonuniform and correlated. He also considered one simple case of an epidemic in a structured population of a sexually transmitted disease in a population divided into men and women. He confirmed the correctness of his exact solutions with numerical simulations of SIR epidemics on networks.

Guardiola and Vecchio [16] considered an Ordinary Differential Equation (ODE) system representing a large class of mathematical models concerning the dynamics of an infection in an organism or in a population and show that the study of the linearized stability leads to some conditions on the basic reproduction number which are sufficient, not only for the local asymptotic stability of a stationary point, but also for its global asymptotic stability. Even for a more restricted class of problems, they proved the necessity and described the limiting behavior of the model.

II.3. Optimization

Most of the models found in the literature dealt with obtaining the measures, such as the rate of transmission, basic reproduction number, etc., for the disease. To the best of our knowledge, not many researchers have tried to use the models to optimize the system by changing the different probabilities and rates. This is because, in a society, it is generally not easy to change these values. Researchers have used these models to study many diseases, such as Malaria, Influenza and more recently, AIDS. The main components of the cost, when trying to curtail a disease are the costs for treatment, examination, vaccination and the cost for advertising to educate people about the disease. Some researchers have suggested strategies to minimize the costs at different stages, for example, in developing vaccinations (Wu et al. [31]), or to minimize the cost of examinations (Wein and Zenios [29]), etc. Li et al. [24] have proposed a vaccination program when the total population size is not constant. But none of the researchers to the best of our knowledge have looked at the problem as a whole (i.e. treatment, vaccination and testing) and tried to come up with a resource-allocation strategy.

CHAPTER III

PROBLEM DESCRIPTION

III.1. Introduction of the Problem

As we have seen, mathematical treatment of problems arising in the area of healthcare has a very long history. The relevant literature in the field attests to several successful applications of different tools, usually borrowed from the realm of operations research to tackle problems in healthcare. Some of these efforts have been documented in literature review (See Chapter II.2). Specifically, some applications of mathematical tools in dealing with problems related to transmission, control, and optimal management of infectious diseases have been cited. Such efforts fall in the category of the so called epidemiologic models.

As far as the nature of the mathematical tools is concerned, tools of purely mathematical nature (differential equations, difference equations, etc.) to tools of operations research nature (stochastic processes, simulation, queueing theory, etc.) have been reported by researchers as means of mathematical treatment in the area of healthcare problem analysis.

An infectious disease, cervical cancer, has been chosen as the area of concentration in this research effort. The objective of this study is to develop guidelines that can help policymakers in deciding how to allocate budget spending for vaccination, testing, and treatment subsidy. Stochastic dynamic programming has been initially chosen as the mathematical tool to analyze this problem. The formulation of the problem along these lines is amongst the achievements of this effort.

The main purpose of this research is to model spread of cervical cancer caused by HPV in a society and develop a methodology (or set of methodologies) to control the epidemic of disease by vaccinating, treating and examining people under the constraint of limited

budget. For modeling this problem, the main idea has been borrowed from the SIR model, which has been well studied in the literature.

In order to solve the problem, we need to be able to divide the population into the different states of the SIR model. We also need to define the conditions under which each individual can move from one state to the other and assign proper probabilities to these transitions. To propose a model that takes care of the intricacies of a real-life situation, while incorporating the mathematical aspect of the problem is not a trivial task. Moreover, the model for the problem should be built in a way that it is solvable. Theoretical models are sometimes useful to gain insights into a complicated problem, by studying the intertwined relationships between participating variables. However, to come up with a practical approach to confront a real-life problem, it is vital to be able to solve the model numerically.

As a first step to model the epidemiology, the problem formulation (Chapter III.2) is presented. The importance of coming up with a tractable formulation of the problem cannot be emphasized enough. In order to do this, some simplifying assumptions were needed that have been noted in Chapter III.3. The solution methodology and the numerical results will be discussed in subsequent chapters.

III.2. Problem Formulation

Consider a large population of individuals. At the beginning, the individuals can be assumed to be in one of the following different states: u^1: Healthy individuals (susceptible), u^2: Individuals having HPV but are unaware of it, u^3: Individuals having the virus and are conscious of that, and u^4: People who have cervical cancer due to HPV. Now, consider a fairly long time horizon (say, 20 years) and define each year as a period where a decision can be made regarding the number of vaccines, examinations and treatments to be administered. This decision is made at the beginning of the period based on the state of the system

(i.e. all individuals taken together). In order to make such a decision, the policymaker must specify an objective that needs to be maximized subject to satisfying some constraints, such as the budget. The objective function can be a function of number of people in different states of the system at different time periods. It could also be based on the number of total effective vaccinations, probability of an individual getting the virus at the end of the time horizon, etc.

In order to solve this control problem of deciding the number of vaccines, treatments and testing in each period, it is critical to characterize the dynamics of the system using a stochastic network. Consider different states in the system, where apart from the aforementioned states, we also have states for people who died due to cancer, and states for people in the different stages of vaccinations: namely v^1, v^2 and v^3 denoting vaccinations done 1, 2 and 3 years ago respectively. In order to keep track of the system dynamics, birth process, death process, vaccinations, etc. also need to be accounted for.

An important consideration is that states u^1 and u^2 are indistinguishable. This is because both types of people have not been diagnosed with the virus or cancer. Both groups would appear healthy until an examination is carried out. For any policy that provides for vaccination, certain number of vaccinations are going to be given to some people in both of these groups. An effective vaccination is the one where the individual getting the vaccination is not affected by the virus (i.e., state u^1), and hence, would not get the virus for 4 years, due to the effect of the vaccination. The vaccination could also be given to people who do not know that they have the virus (i.e., state u^2), but it would be ineffective since they already have the virus. Similarly, examinations can be termed effective only if the person actually had the virus (i.e., state u^2), and was not a healthy individual in the susceptible state (i.e., state u^1). It should be noted that while treating a cancer patient, it is known that the person indeed has the disease. Hence, no such issue arises. When a person is treated, she can move to one of the three states: Healthy and susceptible (u^1), having virus (u^3),

and having virus and not knowing about it (u^2).

Aside from the transitions based on our decisions, the 'natural' flows in the system also need to be accounted for. Since the problem is being solved for a fairly long time horizon, it is imperative to keep track of the births and deaths. It is known that the chance of the disease being passed on from the mother to the child is very small. Hence, all the newborns can be assumed to go to the healthy and susceptible state. Irrespective of the state an individual is in, there is a chance that the person dies of natural causes. These are also taken into consideration.

The probability that a healthy individual gets the virus is dependent upon the number of individuals who actually have the virus, and their awareness level. The people in u^2 are more likely to spread the virus as compared to the ones in u^3 and u^4. Hence, probability a healthy individual gets the virus can be given by a function of number of individuals in these three different states. When an individual has the virus, the chance of that person getting the cancer are dependent upon the person's awareness level. If the individual knows that she has the virus, she can take precaution in diet, etc. to reduce the risk of her getting cancer. Hence, the probability of a person getting the cancer is different dependent upon the person's knowledge of their condition. The nature of this virus is such that in many of the cases, the virus just goes away after a certain period of time. Hence, the person just moves from the u^2 and u^3 states to the u^1 state, with a certain probability.

III.3. Assumptions

In order to come up with a model based on the problem description presented above, we have to make some simplifying assumptions. These assumptions help in defining the nature and the boundaries of the system.

- The budget constraint cannot be violated in any period. In addition any portion of

the budget amount not used in a period cannot be carried over to any other period. The consequence of this assumption is that for most objective functions, the budget constraint would be binding in every period.

- All treatments will be successful, but a person who moves out of the cancer state may still have the virus.
- All tests for viruses are perfect (no false positives or false negatives).
- Chronological order of events in a period is: Vaccination, spread of virus, birth, getting cancer, test, cancer treatment, death due to cancer, and natural death (Figure 3).
- All newborns are healthy and in susceptible state.
- Only one kind of treatment is modeled, as opposed to three kinds that are commonly used.
- A person having the virus and knowing about it is less likely to spread it as compared to the person who has the virus, but does not know that the person is not healthy.
- The persons who know that they have the virus are less likely to get cancer as compared to the persons who do not know about their having the virus, since the former is better equipped with the knowledge of how to avoid getting the disease.
- The only way a person can come to know that she has the virus is through examination. That is, a person cannot go directly from being healthy in one state to having virus and knowing about it in the next.
- Only the people who have the virus can get cancer.

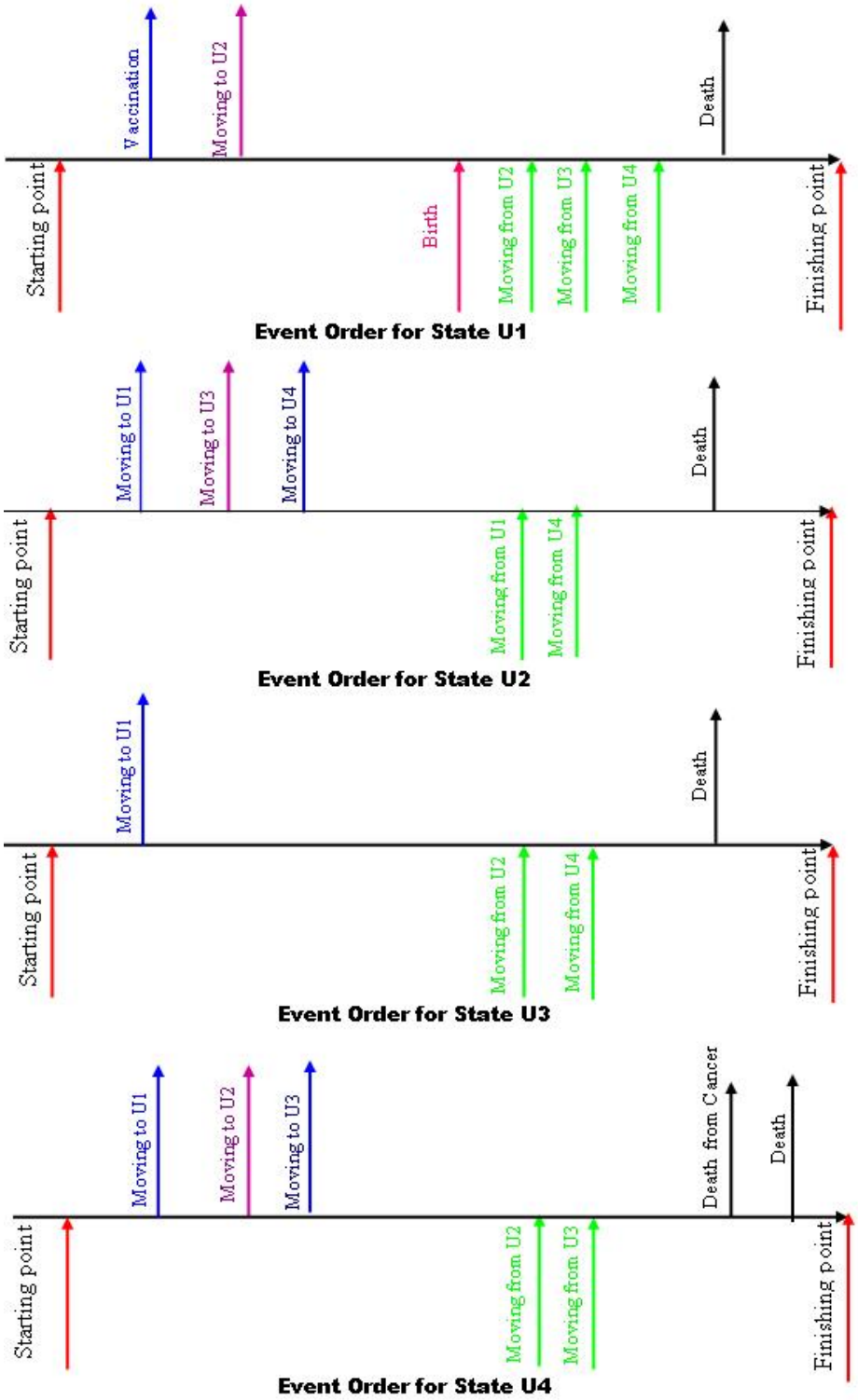

Figure 3. Event Order at Different States

CHAPTER IV

MDP MODEL FOR DYNAMIC CONTROL

The problem formulation in Chapter III.2 shows that the system under investigation changes over time. People go from one state to another with the progress in time and these transitions also depend upon the external stimulus provided in terms of the vaccinations, etc. Our objective is to come up with a control policy that is best for the society, given the budget constraint. In order to do this, we would use all the available information, and come up with a dynamic policy. In this chapter, a methodology for solving the problem has been developed.

IV.1. Markov Decision Processes: Background

Some of the terms that are essential in understanding the methodology are described below.

A stochastic process is a random process evolving with time [22]. This evolving property is referred to as the state of the random process. An appropriate definition of the state of a system is required when one decides to model the system for optimization purposes.

Markov process is an important stochastic process because of its widespread application to analysis of real world systems. In Markov processes, the state transitions are dependent only upon the current state of the system, not on the history. In other words, a significant characteristic of a Markov process that distinguishes it from other random processes is its memoryless property; which means that the probability that the process moves from state i to state j does not depend on the states visited by the system before coming to state i. Due to this property, a probability matrix, called the transition probability matrix (TPM) can be obtained, which contains the probabilities of moving from any state to any other state.

Calculating the elements of the TPM can be challenging due to the large size and the complicated nature of the required calculations. However, once the matrix is obtained, n-step transition probability of going from state i to state j, can be obtained using the Chapman-Kolmogorov theorem [27]: by raising the one-step TPM to the n^{th} power. For ergodic systems, as n increases, the elements of this matrix approach to a limit, defining the limiting probabilities of the system.

In this research work, the central idea is to have a control optimization. That is, we should be able to exert control to change the path taken by the random process. Markov Decision Process (MDP) is a methodology that helps in selecting the optimal decision when faced with multiple control options, in order to optimize an objective function. There have been a lot of developments in the theory of Stochastic Dynamic Programming since the pioneering work if Richard Bellman [8], which is a critical tool in solving the MDP.

IV.2. System Definition

The system under consideration could be analyzed using a Markov decision process where W_n is the state of the system at the n^{th} period. W_n is a vector of the number of individuals in various states and will be described in detail in Chapter IV.3. The decision variable, i.e. action to take in the n^{th} period is to determine how many vaccinations (denoted by x_n), how many treatments (denoted by y_n), and how many tests (denoted by z_n) to administer during that period. Let X, Y and Z be T-dimensional vectors denoting the action during T periods.

The objective is to optimize a function specified by the policymaker in terms of the expected value of the state of the system at the end of each period in the horizon (consisting of T periods): $f(E[W_1], E[W_2], \ldots, E[W_T])$ with the understanding that the expectation of a vector of random variables is the expected value of the individual elements of the vector.

The optimization is subject to

$$c_x^n x_n + c_y^n y_n + c_z^n z_n \leq R_n$$

where R_n, c_x^n, c_y^n and c_z^n are respectively the total budget, cost of vaccines, cost of treatment and cost of testing in period n.

The main problem is to determine an optimal control $A = (X, Y, Z)$ that would optimize the specified objective function subject to budget (and possibly other) constraints. The optimal control is sequential with feedback, i.e. for $n = 0, 1, 2, \ldots, T-1$ the problem is to determine the action in the $(n+1)^{st}$ period $A_{n+1} = (x_{n+1}, y_{n+1}, z_{n+1})$ given W_n (and history which can be dropped if the stochastic process $\{W_n, n \geq 0\}$ can be suitably modeled as Markovian) for the overall objective across all periods.

IV.3. Markov Decision Process Model

Define the following notation in order to keep track of the state of each person in the population under consideration. Let U_n^1 be the number of healthy persons that are not vaccinated in the n^{th} period (corresponding to being in state u^1). Similarly, define U_n^2, U_n^3, U_n^4, U_n^5 V_n^1, V_n^2 and V_n^3, respectively, as the number of people in the n^{th} period with virus and are unaware of it, with virus and are aware of it, with cancer, dead due to cervical cancer, healthy and vaccinated in the previous period, healthy and vaccinated two periods ago, and healthy and vaccinated three periods ago. Therefore, the system state in the n^{th} period, W_n is an 8-dimensional vector

$$W_n = (U_n^1, U_n^2, U_n^3, U_n^4, U_n^5, V_n^1, V_n^2, V_n^3).$$

By modeling W_n in this manner one can ensure the Markov property. Therefore in order to determine A_{n+1}, the action in the $(n+1)^{st}$ stage, all that is needed is W_n and

not the entire history. In addition, to determine W_{n+1} only W_n and A_{n+1} would suffice. Furthermore, the transition probability function is given by:

$$P\{W_{n+1} = (i_1 + i_8 - \ell - m + b + t_2 + t_3 + y_{n+1} - \alpha_2 - \alpha_3 - d_1 - d_8,$$

$$i_2 - k - t_2 + m - j_2 + \alpha_2 - d_2, i_3 + k - t_3 + \alpha_3 - j_3 - d_3,$$

$$i_4 - d_c - y_{n+1} + j_2 + j_3 - d_4, i_5 + d_c, \ell, i_6 - d_6, i_7 - d_7)$$

$$|W_n = (i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8), A_{n+1} = (x_{n+1}, y_{n+1}, z_{n+1})\}$$

$$= q_b(b)q_t(k; z_{n+1}, i_1, i_2)q_v(\ell; x_{n+1}, i_1 + i_8)q_s(m; i_1 + i_8 - \ell)q_c(j_2, j_3; i_2, i_3)$$

$$q_r(t_2, t_3; i_2 - k, i_3 + k)q_h(y_{n+1} - \alpha_2 - \alpha_3, \alpha_2, \alpha_3; y_{n+1}, i_4)q_n(d_c; i_4)$$

$$q_d(d_1, d_2, d_3, d_4, d_6, d_7, d_8; i_1 + i_8 - \ell - m + b + t_2 + t_3 + y_{n+1} - \alpha_2 - \alpha_3,$$

$$i_2 - k - t_2 + m - j_2 + \alpha_2, i_3 + k - t_3 + \alpha_3 - j_3, i_4 - y_{n+1} + j_2 + j_3, \ell, i_6, i_7)$$

where $q_b(b)$ is the probability that there are b births (or new entrants), $q_t(k; z_{n+1}, i_1, i_2)$ is the probability that k out of the z_{n+1} tests were administered on the i_2 individuals, $q_v(\ell; x_{n+1}, i_1 + i_8)$ is the probability that ℓ of the x_{n+1} vaccines were given to the $i_1 + i_8$ healthy individuals, $q_s(m; i_1 + i_8 - \ell, i_2 - k)$ is the probability that out of the $i_1 + i_8 - \ell$ healthy individuals m get virus, $q_r(t_2, t_3; i_2 - k, i_3 + k)$ is the probability that t_2 people get removed from having virus out of the $i_2 - k$ individuals who do not know that they have virus, and t_3 people get removed from having virus from $i_3 + k$ individuals who know that they have the virus, $q_h(y_{n+1} - \alpha_2 - \alpha_3, \alpha_2, \alpha_3; y_{n+1}, i_4)$ is the probability that of the y_{n+1} treatments, α_2 people still have the virus and don't know about it and α_3 people know that they still have the virus, $q_n(d_c; i_4)$ is the probability that d_c people die of cancer of the i_4 number of cancer patients, $q_d(\cdot)$ is the probability of the number of deaths in each state given the number of people in the respective state, and, $q_c(j_2, j_3; i_2, i_3)$ is the probability that j_2 people got cancer among the i_2 people who do not know they have HPV and j_3 people got

cancer from the i_3 people that know they have HPV. It is possible to obtain formulae for the probabilities: $q_b(b)$, $q_t(k; z_{n+1}, i_2)$, $q_v(\ell; x_{n+1}, i_1 + i_7)$, $q_s(m_1, m_2; i_1 + i_7 - \ell, i_3 + k, i_2 - k)$, and $q_c(j_1, j_2; i_2, i_3)$ in terms of the following parameters: $p_v = a_2 i_2 + a_3 i_3 + a_4 i_4$ which is the probability of getting virus, pc_2 which is the probability of getting cancer when the person has virus but does not know about it, pc_3 which is the probability of getting cancer when the person has virus and knows about it, pd_{cancer} which is the probability of death due to cancer, pd_{nat} which is the probability of natural death, p_{birth} which is the probability of birth, p_{r1} which is the probability of recovery from virus for a person that does not know they have virus, p_{r2} which is the probability of recovery from virus for a person that knows they have virus, p_{y2} which is the probability of being treated for cancer and move to having virus but not knowing, and, p_{y3} which is the probability of being treated for cancer and move to having virus while being aware.

Remark 1 *Although it is possible to formulate the problem as an MDP, the difficulty is in solving it. Due to the curse of dimensionality, this MDP is indeed intractable. None of the standard techniques such as value iteration or policy iteration (see Puterman [28]) can be used to obtain the optimal actions for any given state vector in a reasonable amount of time.*

Therefore, some approximations are described to solve such MDPs in the next chapter.

CHAPTER V

APPROXIMATE ANALYSIS

As an approximation to the dynamic control with feedback problem that was modeled using an MDP in Chapter IV.3, first, the "feedback" aspect was relaxed (which is sometimes appropriate in this and other health care applications where only an estimate of the system state can be made) and just considered as an improvement step. Specifically, an equivalent problem was considered where the aim was to obtain a dynamic control *without* feedback. In other words, the set of control actions $A = (X, Y, Z)$ were obtained and set a-priori, which depended only on the knowledge of the initial system state W_0 (and other states W_n for $n = 1, 2, \ldots, T$ are known only probabilistically). A major portion of the approximate analysis description revolves around this dynamic control problem without feedback. However towards the end of this chapter, this approach has been extended to the case of dynamic control with feedback as a sequential approximation (see Chapter V.4).

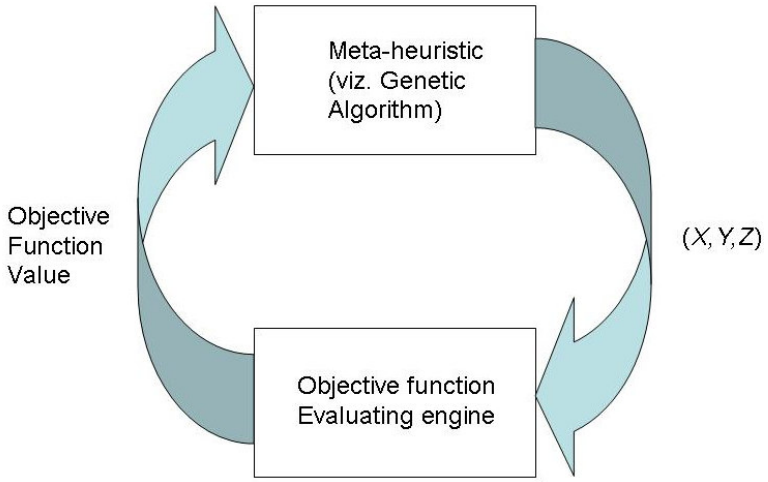

Figure 4. Meta-heuristic and Objective Function Evaluation Engine

Even for the optimal control without feedback, it is extremely difficult to write the objective function as a closed-form algebraic expression in terms of the decision variables, i.e. X, Y and Z. However for a given set of X, Y and Z values, it is possible (although not straightforward) to obtain the objective function. An approximate algorithm was developed taking advantage of this. In particular, a meta-heuristic was selected to search through the space of all possible values of X, Y and Z. One of the key requirements for the meta-heuristic is an engine that can evaluate the objective function for a given (X, Y, Z). This is depicted in Figure 4. The objectives in this research include selecting an appropriate meta-heuristic and to develop alternative methodologies to evaluate the objective function (i.e. objective function evaluating engine) for a given set of X, Y and Z values. These tasks are addressed in the subsequent parts of this chapter.

V.1. Genetic Algorithm: Background

A meta-heuristic is an algorithmic approach to approximate optimal solutions for problems in combinatorial optimization. These are generally applied to problems with either no satisfactory problem-specific algorithm or heuristic, or where it is not practical to implement such an algorithm. Most sophisticated meta-heuristics maintain a pool of several candidate solutions rather than just a single solution. The idea is then to improve on the current pool by adding and deleting from this pool of solutions. This is done by discarding solutions defined by a user-given procedure, and adding new ones by a combination or crossover of two or more states from the pool. A meta-heuristic may also keep track of the current optimum, the optimum solution among those already evaluated so far. Genetic Algorithm is one of the most popular meta-heuristic. Genetic Algorithm is one of the most popular meta-heuristic. In this, and the forthcoming chapters, an overview of genetic algorithm has been presented. This information has been obtained from various websites, but predominantly

[17] and [18].

V.1.1. Outline of the Basic Genetic Algorithm

As adaptive heuristic search algorithms, genetic algorithms are based on the evolutionary ideas of genetics and natural selection. That is, GA's are intended to simulate processes in natural systems dealing with evolution, specifically the principles first laid down by Charles Darwin. Because of this reason, it is said that the solution to a problem solved by genetic algorithms is evolved.

A genetic algorithm begins with a set of solutions (represented by chromosomes) called population. In biology, chromosomes are strings of DNA, which define the characteristics of a living organism. Solutions from one population are taken and used to form a new population. This is motivated by the hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more the chances they are going to be reproduced.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied. The basic outline of how a genetic algorithm can be implemented is presented ahead.

1. **Start**: Generate random population of n chromosomes (suitable solutions).

2. **Fitness**: Evaluate the fitness of each chromosome in the population.

3. **New population**: Create a new population by repeating the following steps until the new population is complete:

   (a) *Selection*: Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)

(b) *Crossover*: With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

(c) *Mutation*: With a mutation probability mutate new offspring at each locus (position in chromosome).

(d) *Accepting*: Place new offspring in a new population

4. **Replace**: Use new generated population for a further run of algorithm

5. **Test**: If the end condition is satisfied, stop, and return the best solution in current population

6. **Loop**

   Go to step 2

As the outline reveals, the mutation and crossover are the most important parts of the genetic algorithm. The performance is influenced mainly by these two operators.

V.1.2. Parameters of GA

Today there is no general theory which would describe parameters of GA for any problem. Recommendations are often results of some empirical studies of GA's, which were often performed only on binary encoding. Following is what has been recommended in the literature on this topic (See [17] and [18]).

Crossover rate: Crossover rate generally should be high, about 80%-95%. (However some results show that for some problems crossover rate about 60% is the best.)

Mutation rate: Mutation rate should be very low. Best rates reported are about 0.5%-1%.

Population size: It may be surprising, that very big population size usually does not improve performance of GA (in terms of speed of finding the solution). Good population size is about 20-30, however sometimes sizes 50-100 are reported as well. A number of research efforts indicate that the best population size depends on encoding, on the size of encoded string.

Selection: Basic roulette wheel selection can be used, but sometimes rank selection works better. There are also some more sophisticated methods, which change parameters of selection during the run of GA. Basically they behave like simulated annealing. But surely elitism should be used (if you do not use other method for saving the best found solution).

Encoding: Encoding depends on the problem and also on the size of instance of the problem.

Crossover and mutation type: Operators depend on encoding and on the problem.

V.1.3. Advantages and Disadvantages of GA

The advantage of GAs comes from its parallelism. GA travels in a search space with more individuals (and with the entire genetic constitution (genotype) rather than the observable characteristics (phenotype)) so it is less likely to get stuck in a local extreme like some other methods.

They are also easy to implement. For a new GA, one just has to write new chromosome (just one object) to solve the problem. With the same encoding one just changes the fitness function. On the other hand, choosing encoding and fitness function can be difficult. Nearly everyone can gain benefits from Genetic Algorithms, once he can encode solutions of a given problem to chromosomes in GA, and compare the relative performance (fitness) of solutions. An effective GA representation and meaningful fitness evaluation are the keys to success in GA applications. The appeal of GA's comes from their simplicity and elegance

as robust search algorithms as well as from their power to discover good solutions rapidly for difficult high-dimensional problems. GA's are most useful and efficient when:

- The search space is large, complex or poorly understood.
- Domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space.
- No mathematical analysis is available.
- Traditional search methods fail.

Another advantage of the GA approach is the ease with which it can handle arbitrary kinds of constraints and objectives; all such things can be handled as weighted components of the fitness function, making it easy to adapt the GA scheduler to the particular requirements of a very wide range of possible overall objectives.

Disadvantage of GAs is in their computational time. They can be slower than some other methods. But with computers which are nowadays around this is not that big a problem.

V.1.4. Applications

Genetic algorithms have been applied to difficult problems (such as NP-hard problems), and also for evolving simple programs. GAs have been applied to solve optimization problems, such as the Traveling Salesman Problem (TSP) and its variations, and in a variety of business problems in functional areas such as finance, marketing, information systems, production etc.

V.2. Implementation of Genetic Algorithm

In this specific implementation of genetic algorithm, a population of chromosomes, or solutions, consisting of values for our three T-dimensional decision variables: Number of vaccinations (X), Number of treatments (Y) and Number of examinations (Z), are considered. Each chromosome can be evaluated for its 'fitness' by evaluating the objective value using any one of the techniques discussed. This objective value can also be stored as a part of the solution. The number of solutions that are stored is termed as the population size in the genetic algorithm, and has been kept constant in this implementation. This is done in order to increase the tractability of the population.

To start the algorithm, in the initiation phase, a whole population of chromosomes is randomly generated. Specifically, in order to obtain the three decision variables for a particular period, three random numbers were generated and normalized so that they add to 1. Then, budget was allocated to each of the decision variables in proportion to their associated random number to give the values for the particular period. Once a population of chromosomes is at hand, the 'fitness' of each of them needs to be computed by evaluating their objective value. In order to generate the chromosomes for the next generation, healthy chromosomes from this generation are picked: i.e., the ones with higher objective values for this maximization problem.

There are three ways chromosomes for subsequent generations are generated: Crossover, Mutations and Immigration. For the crossover state of the genetic algorithm, two healthy chromosomes need to be picked. A variable *separation* has been defined in this implementation, which is essentially the difference in the ranks of the two parent chromosomes. For example, a *separation* of 10 means that crossover will take place between the best and the 11th best chromosomes, 2nd best and the 12th best, etc. From each set of parents, two offsprings are generated. The first one is obtained by using the means of X, Y and Z values

from the two parents for each period. This offspring is bound to satisfy the constraints of non-negativity and being within the budget. The other offspring is the prodigy. The value of each of the decision variables of this offspring can be obtained by extrapolating the line joining the decision variables of the parents away from the weaker parent, multiplying the difference in the values of the decision variable in the two parents by a factor defined as the *Distance Factor*.

In order to satisfy the non-negativity and budget constraints, the above approach can be used for two of the three decision variables. If either of the variables turns out to be negative, it can just be set to zero. In case they exceed the total budget, they are pushed back to satisfy the budget. Then, the binding budget constraint is used to obtain the third decision variable. Half of the population in the next generation would be generated using the crossovers from the previous generation.

Mutations is the second way of generating chromosomes. Here, a healthy chromosome is picked and its decision variable values are tweaked a bit to obtain a new solution. In this implementation, the top one-fifth of the chromosomes, based on their fitness level, have been picked for mutation. Each of these chromosomes has been mutated in two ways, and hence, two-fifths of the chromosomes for the new generation are generated. Once a chromosome is picked, the first way to mutate it is by increasing the value of the number of vaccinations in period n (x_n). This value is randomly increased up to somewhere between 1 and 20 percent, matching it by a corresponding reduction in y_n and z_n values, making sure that none of the constraints are being violated. The other way is to increase the value of the number of treatments (y_n) and matching it with a corresponding decrease in the values of x_n and z_n.

The crossovers and mutations account for 90% of the chromosomes for the new generation. In order to imbibe more randomness into the chromosomes pool, some new chromosomes are also introduced, in the process called Immigration. These chromosomes are

created exactly the way the solutions were created for the very first generation, in the initiation phase. These account for all of the rest of chromosomes in the population.

In this way, the whole population for each of the generations was generated. Each chromosome is evaluated for its fitness and subsequent generation obtained from this generation. This process is repeated for a large, pre-specified number of generations. From the first generation, the best solution is stored as the incumbent solution and at the end of the run, obtained as the best solution. This is expected to be close to the optimal solution. For this implementation, the following values have been used: Population size is 100, number of generations is 1000, separation is 10, distance factor is 0.1, and number of periods is 20.

V.3. Determining the Objective Function Value

The crucial item required for the genetic algorithm in Chapter V.1 is an engine for evaluating the objective function value for a given action (X, Y, Z) as described in Figure 4. Given the initial state W_0 and action (X, Y, Z), it is required to obtain the values of $E[W_1]$, $E[W_2]$, ..., $E[W_T]$, in order to evaluate the objective function value $f(E[W_1], E[W_2], \ldots, E[W_T])$. Since the approximation does not use any feedback, it is possible to evaluate the objective function with just the initial state and action which is generated by the genetic algorithm. Three techniques are described ahead to evaluate $E[W_1]$, $E[W_2]$, ..., $E[W_T]$: Deterministic Analysis (Chapter V.3.1), Individual Markov Chains (Chapter V.3.2), and Simulation (Chapter V.3.3). The three techniques would converge asymptotically as the number of individuals in each state approaches infinity for the first two and the number of replications besides the number of individuals approaches infinity for simulation. Therefore, although the techniques are indeed approximations, the conditions for the problem domain are conducive for asymptotic analysis as the population is large and simulation replications for large populations (due to central limit theorem and strong law of large numbers) are

doable. However, the main reason these techniques are presented is for broader considerations (such as for different population numbers, other diseases, other types of stochastic networks, generic MDPs, etc.). In addition to the three techniques, present bounds (best case and worst case) are also presented for the objective function value in Chapter V.3.4.

V.3.1. Deterministic Analysis

The deterministic analysis is an approximation where the state of the system is known deterministically at the beginning of each period. In particular, given the state at the beginning of a period and the action during that period, the state of the system at the beginning of the next period is approximated as its expected value (rounded off to the nearest integer). As a result, the analysis is reduced to a deterministic dynamic programming problem. However, genetic algorithm is still used to generate candidate solutions and search through the solution space.

Mathematically, the deterministic analysis is explained as follows. The state of the system at the beginning of the $(n+1)^{st}$ period, W_{n+1}, given W_n and the action during the $n+1^{st}$ period $A_{n+1} = (x_{n+1}, y_{n+1}, z_{n+1})$ is approximated as

$$W_{n+1} = (E[U^1_{n+1}|W_n, A_{n+1}], E[U^2_{n+1}|W_n, A_{n+1}], E[U^3_{n+1}|W_n, A_{n+1}], E[U^4_{n+1}|W_n, A_{n+1}],$$

$$E[U^5_{n+1}|W_n, A_{n+1}], E[V^1_{n+1}|W_n, A_{n+1}], E[V^2_{n+1}|W_n, A_{n+1}], E[V^3_{n+1}|W_n, A_{n+1}]).$$

Therefore for a given action set (X, Y, Z) and initial state W_0, $E[W_1]$ is obtained. Then to obtain $E[W_2]$ given W_1 and the action during that period, approximate by using $E[W_1]$ instead of all possible values of W_1. In this manner, $E[W_2]$, $E[W_3]$, ..., $E[W_T]$ can be recursively obtained. Therefore using the deterministic approximation, the objective function $f(E[W_1], E[W_2], \ldots, E[W_T])$ is evaluated.

V.3.2. Individual Markov Chains

Instead of looking at the system as a whole, each individual can be modeled as though the individual is going over various possible states: u^1, u^2, u^3, u^4, u^5, v^1, v^2 and v^3. The state an individual is within a particular period can be modeled as a Markov chain with 8 elements (i.e. the 8 states mentioned above) in its state-space. The transition diagram is described in Figure 5. Let P_n be the one-step transition probability matrix for the n^{th} stage. Since it is straightforward to write down P_n from the transition diagram, it is not explicitly presented here.

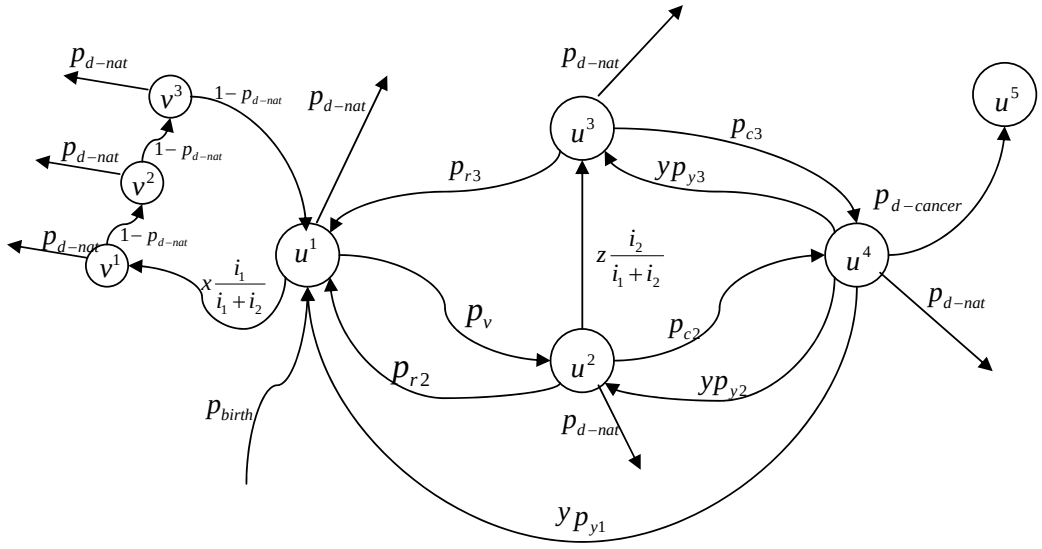

Figure 5. Transition Diagram for an Individual

Note that the Markov chain is not time-homogeneous and that is why there is a subscript n in P_n. This is because some of the transition probabilities depend on the number of people that are present in each state and as this number changes from stage to stage, and so does the transition probability matrix. Two approximations to estimate P_n are considered, given the action (X, Y, Z). The first approximation uses the initial number of individuals

in each state across the horizon which can be computed easily and thereby P_n becomes independent of n. The second approximation uses the deterministic analysis in Chapter V.3.1 to estimate the average number of individuals in each state in each stage.

Then using the P_n matrices and knowing the number of individuals in each state initially, the average number of individuals in each state in every stage can be obtained. For the first method, P^{20} can be used, and for the second method, $P_1 \times P_2 \times \ldots P_{20}$ is needed. Also, using the number of new individuals that enter the system at stages other than the first stage, they also can be included in the computation (by considering the right stages in the analysis). Thereby, $E[W_1]$, $E[W_2]$, ..., $E[W_T]$ is obtained, which can be used in the objective function.

V.3.3. Simulations

Simulation is a powerful tool for analysis of complicated dynamic systems. There are numerous applications for this tool ranging from optimization to what-if games. At times simulation is used to evaluate, compare and caliber the merits of other solution techniques. In such applications, one chooses problems with some known standard of performance and compares the simulated results against the standard and ranks the solution techniques. In another kind of application, one can simulate problems which can be solved analytically but the exact method is complicated and time consuming.

Hence, as an alternative for an engine to evaluate the objective function, Monte Carlo simulation is considered. In particular, using the initial state W_0 and action (X, Y, Z), sample realizations of W_1, W_2, ..., W_T are generated and the values of $E[W_1]$, $E[W_2]$, ..., $E[W_T]$ are statistically estimated; thereby obtaining an estimate of the objective function $f(E[W_1], E[W_2], \ldots, E[W_T])$. The simulation also helps to get estimates of $Var[W_1]$, $Var[W_2]$, ..., $Var[W_T]$.

From an implementation standpoint, 1000 replications (i.e. sample realizations) of

$W_1, W_2, \ldots, W_T$ were performed. In order to do that, a fast way to simulate the transitions is needed, since Bernoulli trials for each individual in the population are computationally tedious. For the purpose of the simulation study in this chapter, it was determined to resort to the central limit theorem and generate random deviates for a normal random variable. In fact, since the original random variable in our study follows a Bernoulli probability distribution and the sum of a large number of independent and identically distributed Bernoullies must be the subject of further studies, by reasoning that the distribution of the sum approaches a normal density function as the sample size grows larger, the normal density function was assumed for the sum and in the following step generated random deviates for this density. To generate random deviates for a standard normal density, one can choose a method from a rather long list of possibilities. One of the most easy to understand and easy to apply methods has been proposed by Box and Müller [10]. The basis of this method is the following problem in probability:

Let U_1 and U_2 be independent random variables distributed as $Uniform[0,1]$ and the random variables X_1 and X_2 are defined as $X_1 = \sqrt{-2ln(U_1)}Cos(2\pi U_2)$ and $X_2 = \sqrt{-2ln(U_1)}Sin(2\pi U_2)$, then, X_1 and X_2 form a random sample from $N[0,1]$. Once a random deviate is generated for the standard normal, it will be a straightforward task to transform it to any arbitrary normal.

V.3.4. Bounds

As noted earlier, it is not possible to distinguish between people in states u^1 (healthy and no vaccination) and u^2 (have virus but do not know). Vaccinations and tests are provided to people in both states. For a given policy (X, Y, Z) the best possible scenario (vaccines are administered to people in u^1 and tests to people in u^2) and the worst possible scenario (vaccines are administered to people in u^2 and tests to people in u^1) is studied. By comparing against the best-case and the worst-case scenarios, the impact of having complete informa-

tion and wrong information can be assessed. However, it must be noted that because of the random nature of the model, a sample simulation may violate the bounds that have been obtained.

V.4. Incorporating Feedback

So far in the approximation, it has been assumed there is no feedback. In other words, the genetic algorithm together with the objective function evaluation engine will produce an optimal action (X, Y, Z) for a given initial state W_0 and no other information (such as feedback). However, consistent with standard MDPs, it is possible to obtain W_1, W_2, ..., W_T values by sampling the population at the end of each period. A naive algorithm is suggested here, to incorporate the feedback that is obtained periodically over time.

In order to determine the optimal action in the $(n+1)^{st}$ period (A_{n+1}), given the state at the beginning of that period (i.e. W_n), the approximation *without* feedback was solved for the remaining $T - n$ periods and the optimal action for the first of those periods was picked, namely, A_{n+1}. Therefore at the beginning of each period the approximation without feedback is performed as though the current state is the initial state and the horizon is the number of periods that are remaining.

CHAPTER VI

NUMERICAL EVALUATIONS

Several numerical evaluations were performed to (a) implement the approximation schemes suggested to solve the MDP in Chapter V for a specific case study, (b) analyze the resulting control policy, (c) compare the different engines for objective function evaluation, (d) study the effect of different budgets, (e) investigate the effect of additional constraints, (f) contrast the policies obtained by considering feedback against those that do not consider feedback, and, (g) understand requirements in terms of computational time and effort for the various approximation schemes. In order not to clutter graphs and also to avoid being repetitive, the evaluations were divided up into categories and only a subset of the results is presented.

As an example case study, consider the population in the United States with the understanding that it is purely for illustration purposes and not a suggestion for the level at which policymaking must take place. Furthermore, since it is desired to study a real population and data was available only for the entire United States, it has been chosen in our case study. For this case study, the numerical values for the parameters described in Chapters III and IV are: $a_2 = 7 \times 10^{-7}$, $a_3 = 7 \times 10^{-8}$, $a_4 = 7 \times 10^{-8}$, $pc_2 = 0.00006$, $pc_3 = 0.00004$, $pd_{cancer} = 0.03$, $pd_{nat} = 0.006$, $p_{birth} = 0.012$, $p_{r1} = 0.02$, $p_{r2} = 0.04$, $p_{y2} = 0.3$, $p_{y3} = 0.1$, $i_1 = 100000000$, $i_2 = 10000000$, $i_3 = 8000000$, $i_4 = 500000$, $i_5 = 0$, $i_6 = 0$, $i_7 = 0$, $i_8 = 0$, $c_x = 90$, $c_y = 8000$, and $c_z = 40$. In addition, the following numerical values are used for the coefficients which would be defined later: $w_0 = 0.25$, $w_1 = 0.1$, $w_2 = -0.2$, $w_3 = 0$, $w_4 = -0.5$, $w_5 = -8$. The coefficients w_i are chosen arbitrarily and will be used in the objective function. A program was written in language C to run the above experiments, and has been presented in Appendix A.

The results of the numerical evaluations are presented by separating them into three

categories: a static policy is considered in Chapter VI.1, results for dynamic policy without feedback are described in Chapter VI.2, and finally, results for dynamic policy with feedback are presented in Chapter VI.3. While details of the above three categories are described later, it is worthwhile to point out here that the objective functions for the three categories are not chosen to be the same. This is done intentionally to clarify that our contribution is not in selecting an objective function, but if the policymaker provides an objective function of the format $f(E[W_1], E[W_2], \ldots, E[W_T])$ and other input data, then our tool will prescribe a policy (or control action) that should be taken.

VI.1. Static Policy Evaluations

As described above, the first of the three categories considered is the static policy. Here, a constraint has been imposed on the problem described in Chapter III. In particular, it is requireed that the control policy in each period be the same. The motivation for this comes from the fact that the policymaker may be inclined to announce publicly what control action he/she proposes (and many times it appears reasonable if the action is uniform in each year). It is important to note that this restriction automatically implies that there is no feedback and the policy is made upfront. Although this is significantly different from what is described in Chapter III, the reason it is presented before the other categories is that our intention is to describe results in an order where the objective function value improves with the categories.

For this set of experiments, the objective function that has been used is:

$$f(E[W_1], E[W_2], \ldots, E[W_T]) = (E[W_T] - E[W'_T]) \cdot [w_1\ w_2\ w_3\ w_4\ w_5\ w_0\ w_0\ w_0]$$

for $T = 20$ years, where w_i are given weights and $E[W'_T]$ is the state of the system under the "do nothing" policy. The "do nothing" policy corresponds to giving zero vaccinations,

zero treatments and zero testing (in other words zero budget). Essentially this objective is a weighted function of the improvement in each state between the do nothing policy and the static policy. The objective function translates to the following (for $T = 20$)

$$\sum_{i=1}^{5} w_i(E[U_T^i] - E[U_T'^i]) + \sum_{t=1}^{T} w_0 E[V_t^1].$$

Table 1. Objective Function for Static Control for Various Budgets

Budget (million)	Simulation	Deterministic	Markov P^{20}	Markov P_n
1	6.5328	6.6886	6.5148	6.5515
5	33.1604	33.1194	31.7414	32.3034
10	65.6323	65.6794	60.9582	63.2792
50	324.3613	324.2647	229.6935	287.2886
100	647.3186	647.4737	411.4088	566.3500
500	3175.1503	3175.1937	2833.7934	3085.1242
1000	6351.7723	6351.8246	5328.7201	6309.7490

Objective function obtained using the three methods: deterministic, individual Markov chains (using both P^{20} as well as the product of P_n values), and simulation as described in Chapter V.3 is studied first. In Table 1, comparison of the objective function values for various budgets has been made using the three methods. Note that these are indeed the optimum objective function values generated by the genetic algorithm. From the analysis, it is evident that the objective value obtained using simulation is always close to the that using the deterministic approximation. The individual Markov chain methodology produces results that are also reasonably close to the simulation values (but not as close as the deterministic analysis). Although the simplest to implement and the fastest, the individ-

ual Markov chain model with the approximate P matrix which is raised to higher powers was not as close as the others but reasonably good. It should be pointed out that similar results in terms of the performance of the simulation, deterministic analysis and individual Markov chains were obtained for the dynamic policies both with or without feedback (these corresponding results are not presented in the results chapter for the dynamic policies).

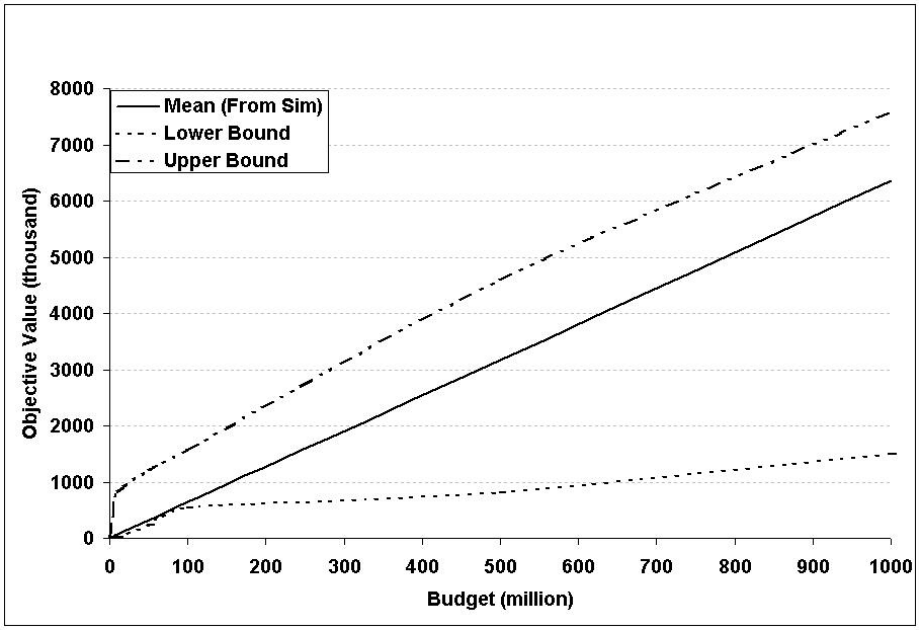

Figure 6. Best Case, Worst Case and Mean Objective Values for Static Policy

Next, the results for the bounds described in Chapter V.3.4 are presented in Figure 6. In particular, the bounds are obtained using the worst case and best case of administering of vaccines as well as tests. Clearly, significant benefits can be obtained with additional information (which is hidden in terms of who is susceptible and who already has the virus and does not know) and significant losses would be incurred if erroneous information is used. The results are similar for the dynamic policies as well and hence are not presented in that chapter.

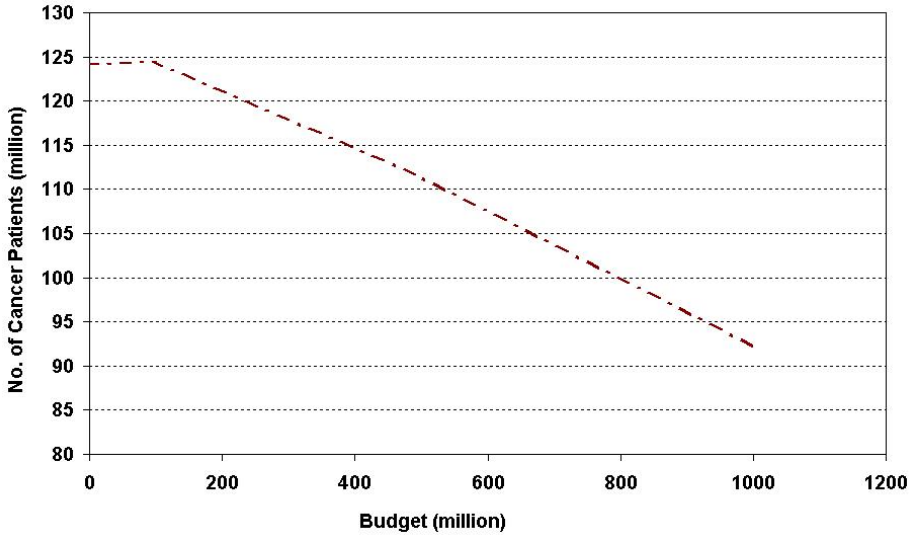

Figure 7. Number of Susceptible People in the Case of Static Policy

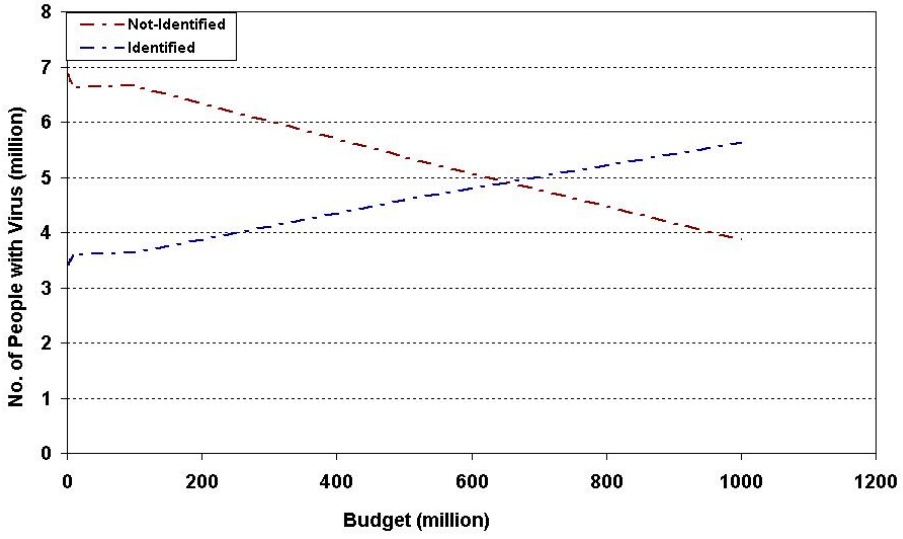

Figure 8. Number of People Having Virus when Implementing Static Policy

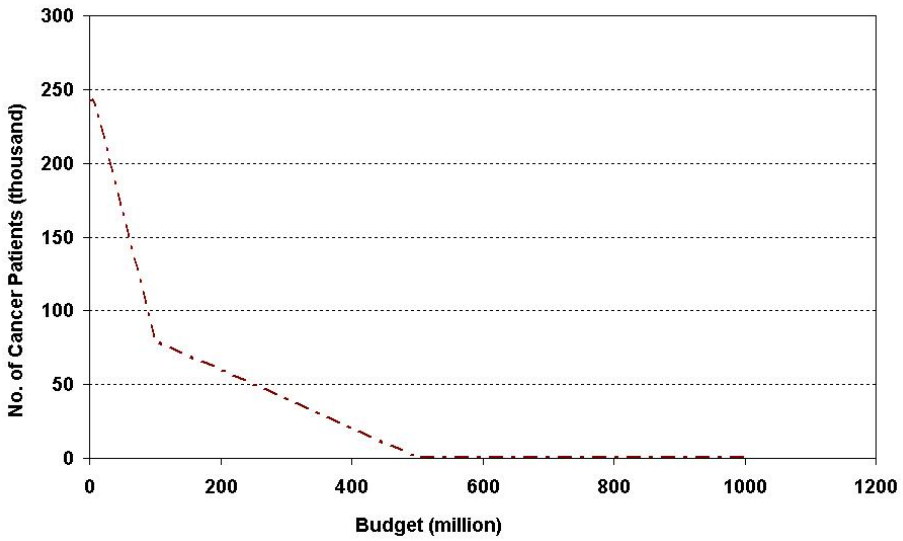

Figure 9. Number of Cancer Patients when Implementing Static Policy

Under a static policy, as the budget increases, an incremental benefit can be observed. Firstly, note that the number of people that are susceptible to catch the virus goes on reducing as budget increases (Figure 7). In Figure 8, it can be observed that as the total budget increases, the number of people who know that they have the virus increases. But it should be noted that the total number of people who have the virus is actually decreasing, since the number of people who have the virus but are unaware of it is reducing significantly. Both the above trends are expected and desired. Moreover, a reduction in the number of people affected by virus can also be observed, as shown in Figure 9.

Finally, for the static policy where the control action is identical in every period, it is easy to tabulate the policy for various budget values. The results are presented in Table 2. In that table, note that X, Y and Z respectively denote the number of vaccines, number of treatments and number of tests in every period for each budget value. It is encouraging

Table 2. Policies for Static Control for Various Budgets

Budget (million)	X	Y	Z
1	172	5	23613
5	170	16	121417
10	164	156	218431
50	164	5048	240031
100	65	11254	249053

from a fairness standpoint to note that the optimal policy spreads out the action over all three domains (and not just one, although such all-or-none policies are fairly common in other applications).

VI.2. Dynamic Policy without Feedback

The second of the three categories that is considered is the dynamic policy without feedback. Here no constraints are imposed, as was the case in the static policy. However feedback is not being used. Therefore this does not describe the final step of the approximation when using feedback. For this set of experiments (and those in the next chapter which deals with dynamic policy with feedback), the objective function that is used is

$$f(E[W_1], E[W_2], \ldots, E[W_T]) = \frac{1}{T}\sum_{n=1}^{T}(E[W_n] - E[W'_n]) \cdot [w_1\ w_2\ w_3\ w_4\ w_5\ w_0\ w_0\ w_0]$$

for $T = 20$ years where w_i are given weights and $E[W'_T]$ is the state of the system under the do nothing policy. Essentially this objective is a weighted function of the improvement in each state between the do nothing policy and the dynamic policy. In terms of the number

of people in various stages, the objective function translates to:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{5} w_i(E[U_t^i] - E[U_t^{'i}]) + \sum_{t=1}^{T} w_0 E[V_t^1].$$

Effect of dynamic policy on the action space over time has been analyzed first. A budget value of 1 Billion units was arbitrarily picked for the analysis (with the understanding that for other budget values the results would follow a predictable trend). In Table 3, the vector of X, Y and Z values is illustrated for this dynamic policy without feedback. The policy is indeed dynamic and the results appear to be different at the extremes (than in the middle periods). This is due to the fact that the horizon is finite and the policy could potentially be different near the start and finish.

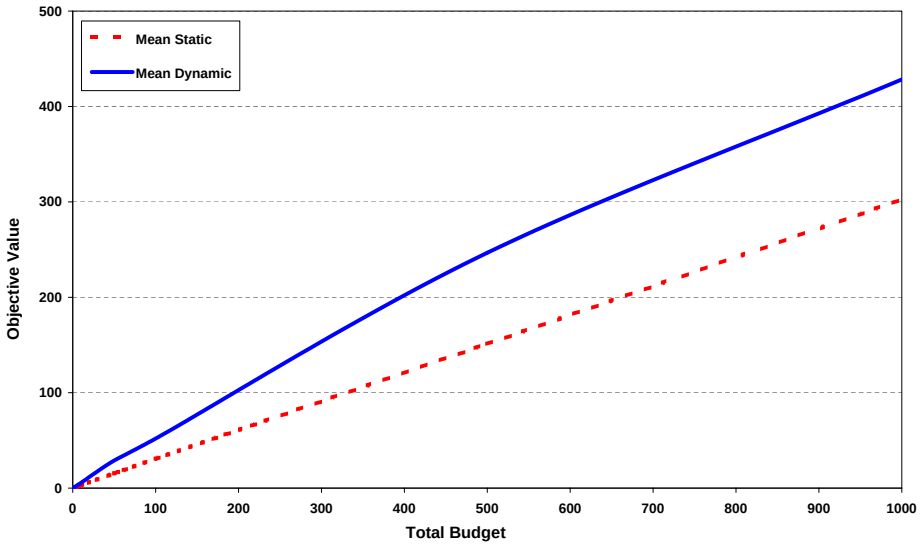

Figure 10. Comparing Objective Values in Static and Dynamic Policies

Next, the dynamic policy without feedback is compared against the static policy. Although there is merit in announcing a "static" policy for vaccination, treatment and tests, the dynamic policy possibly produces better results. For this purpose, comparison is made between the objective function for the dynamic policy against the static policy for various

Table 3. Dynamic Policy without Feedback Vectors for 1 Billion Budget

Period	X (million)	Y (thousand)	Z (million)
1	0.000	124.995	0.001
2	0.000	76.765	9.647
3	0.000	124.995	0.001
4	1.303	86.493	4.769
5	2.773	53.351	8.089
6	7.123	7.578	7.456
7	7.760	3.612	6.817
8	7.931	4.396	6.274
9	6.010	3.981	10.680
10	8.270	1.285	6.133
11	8.536	3.580	5.077
12	5.825	5.471	10.797
13	6.118	18.673	7.499
14	9.446	0.259	3.692
15	5.961	40.545	3.478
16	3.318	71.113	3.309
17	11.111	0.000	0.000
18	11.111	0.000	0.000
19	11.106	0.000	0.011
20	11.111	0.000	0.000

budget values as shown in Figure 10. Clearly, there is a significant improvement in the objective (higher value is better) when the static policy constraint is removed. Therefore this makes a clear case for considering dynamic policies.

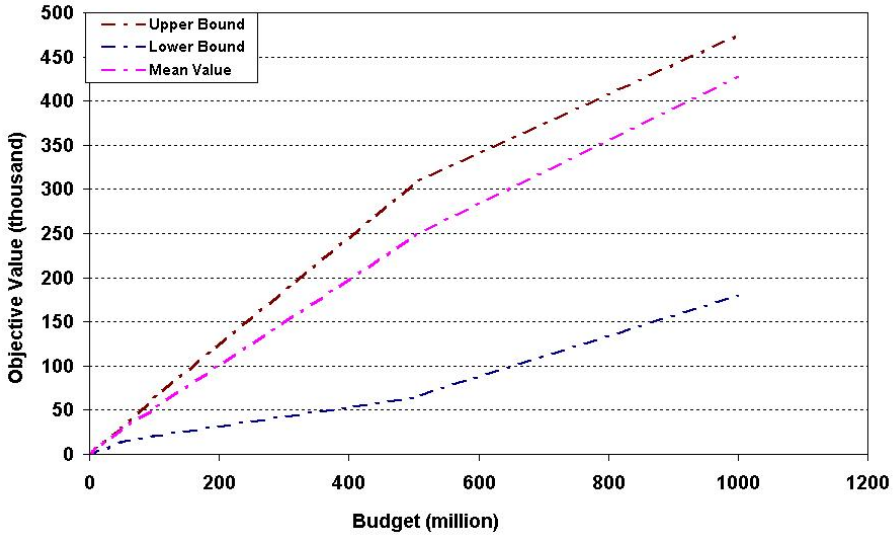

Figure 11. Objective Value in Case of Dynamic Policy Without Feedback

It can be observed from the Figure 11, that having the information as to whether the person already has the virus or not can make a significant difference in the objective case. This is similar to the case seen in the static policy. Notice that the mean objective value for random samples is much closer to the upper bound as compared to the lower. This is another reason why a large number of patients being examined even at very low budget values can be seen (Table 3).

As the budget increases when implementing Dynamic Policy without any feedback, as expected, the total number of people having virus reduces. However, as one would hope, at the end of time horizon, there are much more number of people who know that they

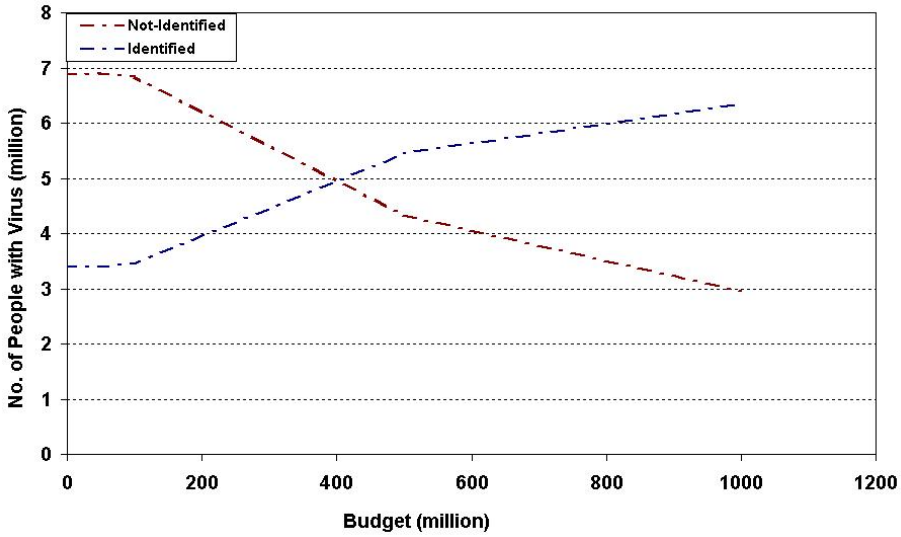

Figure 12. Number of People with Virus in Case of DP Without Feedback

have the virus as compared to the number of people who are not aware (Figure 12). This relative change in the number of people in the two states is not present at lower budget, but progresses as the budget increases, and as the budget goes to very high values, the number of people who have the virus but don't know about it practically goes to zero.

Finally, comparison is made between the optimal actions for the dynamic policy in each period and the static policy. Across different periods, the number of vaccinations (X), number of treatments (Y) and number of tests (Z) are plotted in Figures 13, 14 and 15. The figures present the policies for both static as well as dynamic policy without feedback.

VI.3. Dynamic Policy with Feedback

The last of the three categories that is considered is the dynamic policy with feedback. This essentially is the complete approximation to the MDP described in Chapter V. For this set

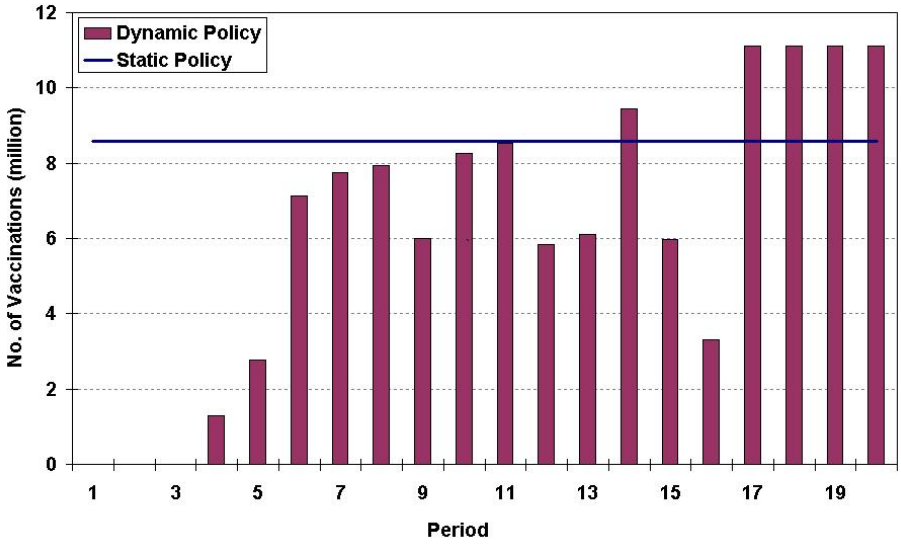

Figure 13. Number of Vaccines in Each Period, Static vs. Dynamic Policies

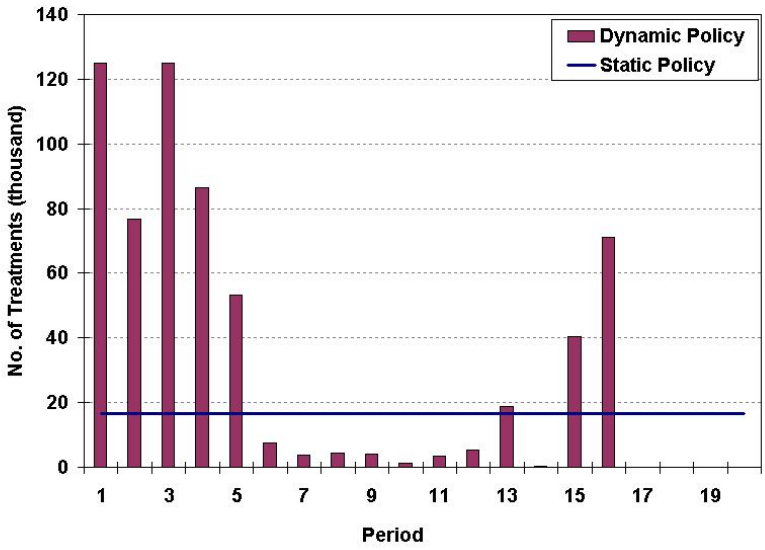

Figure 14. Number of Treatments in Each Period, Static vs. Dynamic Policies

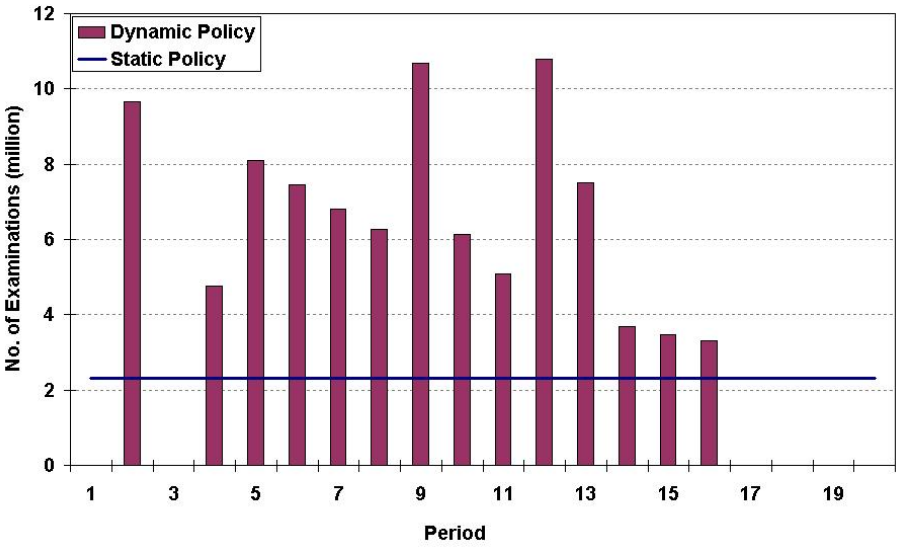

Figure 15. Comparing Number of Tests in Static and Dynamic Policies

of experiments, the same objective function is used as in ChapterVI.2.

Effect of dynamic policy with feedback on the objective function for various budgets is analyzed first. Table 4 contains the objective values (in 1000's) for the static policy, dynamic policy without feedback and dynamic policy with feedback. The dynamic policy with feedback is indeed better than without feedback but not considerably. Especially at larger budgets the results are comparable between the two dynamic policies, however the dynamic policies are lot better than static case. It does not appear like making the extra effort in obtaining state information is worthwhile according to this numerical example.

Next comparison is made between the control policy of the various policies. In particular, the average number of vaccinations, average number of treatments and average number of tests was plotted in each period. This is done in Figure 16. The control actions even after averaging are reasonably different for each policy. It is also interesting to com-

Table 4. Objective Function for Various Policies for Different Budgets

Budget (million)	Static Policy	Dynamic Policy without Feedback	Dynamic Policy with Feedback
1	0.32	0.51	0.61
5	1.58	2.64	2.76
10	3.13	5.58	5.53
50	15.44	28.93	29.01
100	30.83	51.98	56.21
500	151.19	246.69	248.45
1000	302.47	428.27	430.15

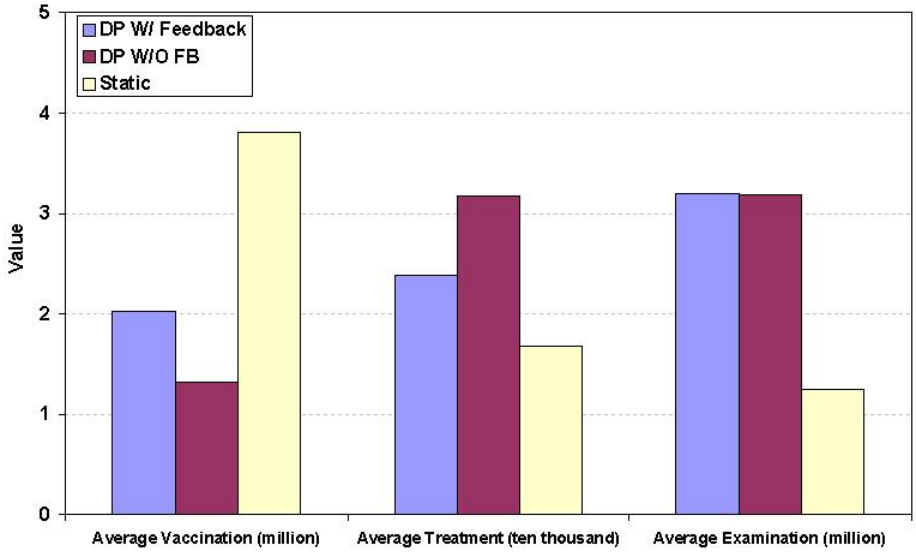

Figure 16. Comparing Decision Variables in Different Policies

Table 5. Dynamic Policy with Feedback Vectors for 1 Billion Budget

Period	X (thousand)	Y (thousand)	Z (million)
1	0	125	0
2	0	125	0
3	0	125	0
4	0	125	0
5	0	125	0
6	0	125	0
7	0	125	0
8	0	125	0
9	48	105	3892
10	0	125	0
11	0	125	0
12	11111	0	0
13	0	125	0
14	0	69	11200
15	0	125	0
16	0	69	11200
17	11111	0	0
18	11111	0	0
19	11111	0	0
20	11111	0	0

pare the control policy of one random sample from dynamic policy with feedback (Table 5) to that of one without feedback (Table 3).

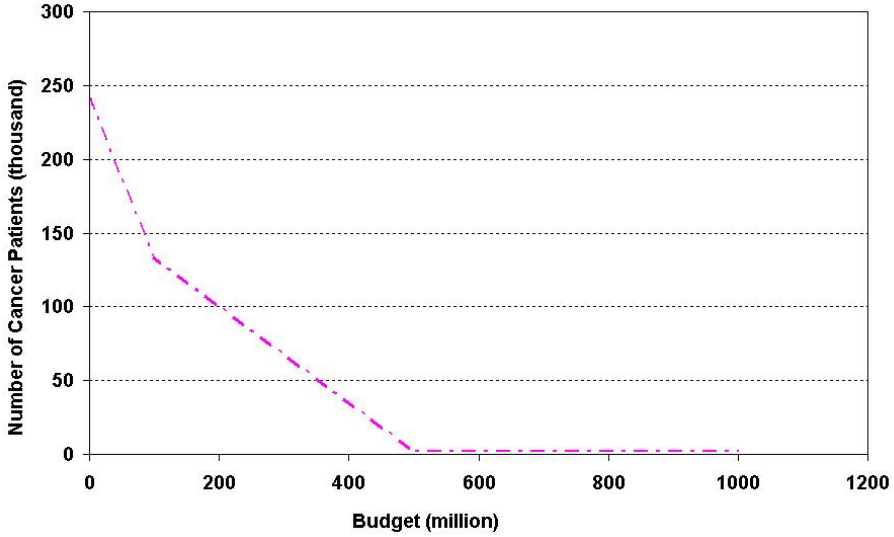

Figure 17. Number of Cancer Patients in Case of DP with Feedback

In the case of Dynamic Policy with Feedback, it can be seen that the budget is allocated smartly based on the current state of the system, rather than the expected state of the system. This means that the policy performs much better than any of the other policies discussed earlier. If we look at the number of people having cancer at the end of time horizon when using this policy (Figure 17), we see that it starts reducing drastically even for lower values of budget. It goes down to almost zero at moderately low budgets. Moreover, the total number of people vaccinated at the end of time horizon increases linearly with the budget (Figure 18). Thus, the number of people not susceptible to catch the virus increases with the increase in the allocated budget.

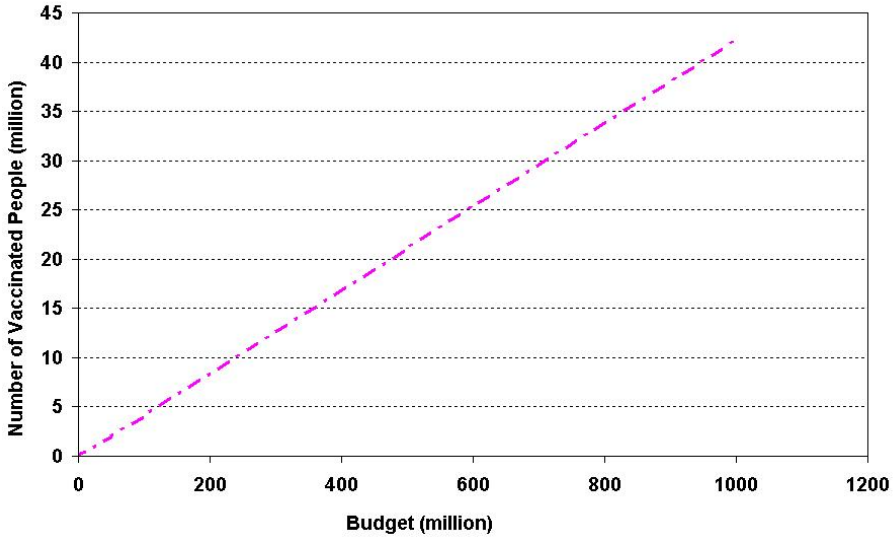

Figure 18. Number of Vaccinated People in Case of DP with Feedback

CHAPTER VII

SUMMARY AND CONCLUSIONS

In this research, policies for prevention and cure of cervical cancer, caused by Human Papilloma Virus, have been studied. The main contribution of this work is that it provides mathematical model for a real-life problem, which can be solved numerically to come up with an optimal strategy for the society. Specifically, if a policymaker provides us a budget and an objective, recommendations regarding the number of vaccinations, number of treatments and number of tests in each period within a finite horizon can be made using the proposed methodology. The methodology developed is very general. As a result, it can be used with any form of objective function, which is dependent upon the state of the system or any of the input parameters.

In this research, first, a Markov Decision Process (MDP) model has been formulated to obtain the optimal dynamic control with feedback. However since solving the MDP is computationally intractable, approximations have been suggested by considering static policies and dynamic policies without feedback and iteratively solving for the feedback case as information rolls in. In this work, a genetic algorithm has been developed, which searches through the entire space of possible actions and evaluates the objective function value for each candidate action. One of the other contributions of this work is the development of methods to approximately characterize the objective function value for a given action vector. These methods span deterministic analysis, Markov models and simulations.

Several experiments were performed that resulted in the findings mentioned ahead. The approximation methods to determine the optimal action in each period can be implemented and are computationally tractable. The resulting control policy is not easy to characterize. However, in case of the static policy, the optimal solution spreads across the

action space. That is, the same policy has to be used in all the periods. As for the dynamic policies, the optimal solutions look different near the boundaries as compared to the center of the time horizon. Based on the case study that has been presented, we can say that the different engines for objective function evaluation: namely simulation, individual Markov chains and deterministic analysis, produce reasonably close solutions. There was a noticeable difference in the control policy as the budgets changed, both in the case of static and dynamic policies. Clearly, the static policy was found to be inferior to the dynamic policies. For the case study presented, the dynamic policy without feedback was found to be close to the case with feedback, as far as the objective values are concerned. It must be noted however that for different objective functions and under different conditions, the DP with feedback may perform much better than either of the two policies. The dynamic policy without feedback is not only just marginally worse off than the case with feedback, the computation time is much lower (few seconds) as compared to the case with feedback (about a day). It is important to realize that the results do depend on the choice of objective function. In conclusion, the technique described here is reasonable to solve MDPs approximately when they are computationally intractable.

REFERENCES

[1] AICR, *http://www.aicr.org.uk*, 2006.

[2] L. J. S. Allen, Some discrete time SI, SIR and SIS epidemic models, Mathematical Biosciences, 124 (1994) 83–105.

[3] L. J. S. Allen and A. M. Burgin, Comparison of deterministic and stochastic sis and sir models in discrete time, Mathematical Biosciences, 163 (2000) 1–33.

[4] R. M. Anderson and R. M. May, *Infectious diseases of humans; dynamics and control* (Oxford University Press, Oxford, 1991).

[5] H. Andersson and T. Britton, *Stochastic Epidemic Models and their Statistical Analysis* (Springer, New York, NY, 2000).

[6] N. T. J. Bailey, *The Mathematical Theory of Infectious Disease and its Applications* (Griffin, London, 1975).

[7] M. S. Barlett, Some evolutionary stochastic processes, J. Roy. Statist. Soc. B, 11 (1949) 211–229.

[8] R. Bellman, *Dynamic Programming* (Dover Publications, 2003).

[9] Cancer-Research-UK, *http://www.cancerhelp.org.uk/*, 2006.

[10] J. S. Carson J. Banks, B. L. Nelson, and D. M. Nicol, *Discrete-Event System Simulation* (Prentice-Hall, Inc., Upper Saddle River, New Jersey, 2005).

[11] Cervical-Cancer, *http://www.cnn.com/HEALTH/library/DS/00167.html*, 2006.

[12] Cervical-Cancer, *http://www.healthywomen.org/healthtopics/cervicalcancer*, 2006.

[13] Cervical-Cancer-Center, *http://www.cancercenter.com/cervical-cancer.cfm*, 2006.

[14] D. J. Daley and J. Ganni, *Epidemic Modeling: An Introduction* (Cambridge University Press, Cambridge, 1999).

[15] O. Diekmann and J. A. P. Heesterbeek, *Mathematical Epidemiology of Infectious Diseases: Model Building, Analysis and Interpretation* (Wiley Series in Mathematical and Computational Biology, New York, NY, 2000).

[16] J. Guardiola and A. Vecchio. The basic reproduction number for infections dynamics models and the global stability of stationary points, WSEAS Transactions on Biology and Biomedicine, 2 (2005) 337–349.

[17] Genetic Algorithms, *http://cs.felk.cvut.cz/ xobitko/ga/*, 2006.

[18] Genetic Algorithms, *http://www.doc.ic.ac.uk/ñd/surprise_96/journal/vol4/tcw2/ report.html/*, 2006.

[19] H. W. Hethcote, The mathematics of infectious diseases, SIAM Review, 42(4) (2000) 599–653.

[20] I. T. Katz and A. A. Wright, Preventing cervical cancer in developing world, The New England Journal of Medicine, 354(11) (2006) 1110.

[21] W. O. Kermack and A. G. McKendrick, A contribution to the mathematical theory of epidemics, Proc. Roy. Soc. Lond. A, 115 (1927) 700–721.

[22] G. F. Lawler, *Introduction to Stochastic Processes* (Chapman and Hall/CRC, 1995).

[23] C. Lefvre J. P. Gabriel and P. Picard, *Stochastic Processes in Epidemic Theory*. (Springer Lecture Notes in Biomath, 1990).

[24] X. Z. Li, G. Gupur, and G. T. Zhu, Analysis of an age structured seirs epidemic model with varying total population size and vaccination, Acti Mathematicae Applicatae Sinica, 20(1) (2004) 25–36.

[25] A. G. McKendrick, Applications of mathematics to medical problems, Proc. Edinburgh Math. Soc., 14 (1926) 98–130.

[26] M. E. J. Newmann, Spread of epidemic disease on networks, Physical Review E, 66(1) (2002).

[27] Athanasios Papoulis, *Probability, Random Variables, and Stochastic Processes* (McGraw-Hill Companies, 1984).

[28] M. L. Puterman, *Markov Decision Processes–Discrete Stochastic Dynamic Programming* (John Wiley & Sons, Inc., New York, NY, 1994).

[29] L. M. Wein and S. A. Zenios, Pooled testing for hiv screening: Capturing the dilution effect, Operations Research, 44(4) (1996) 543–569.

[30] J. L. Wolstenholme and D. K. Whynes, Stage-specific treatment costs for cervical cancer in the united kingdom, European Journal of Cancer, 34(12) (1998) 1889–1893.

[31] J. T. Wu, L. M. Wein, and A. S. Perelson, Optimization of influenza vaccine selection, Operations Research, 53(3) (2005) 456–476.

APPENDIX A

C-PROGRAM FOR GENETIC ALGORITHM

```
/*****************************************************************************************
 The program to use genetic algorithm to evaluate a good policy to maximize the objective
for the HPV problem. This program tries to evaluate dynamic policies that need not
necessarily have the same x, y and z value over the time horizon, with or without using
the feedback. This program can also be used to evaluate the bounds for a given policy.
When the bounds are being evaluated, a different simulation is run, which is the most
pessimistic and most optimistic view of the situation…
The program has been modified to as to have changing values of Pv. Now, the probability
of a person getting a virus is dependent upon the number of people in N2, N3 and N4.
Program modified to include the dynamic policies with feedback.

  Created By:   Hoda Parvin
  Dated:        10/24/06

*****************************************************************************************/
#include <stdio.h>
#include "library.c"
/*****************************************************************************************/
struct str_solution
{
  long * x;
  long * y;
  long * z;
  double Objective;
};
/*****************************************************************************************/
struct str_state
{ /* Can be used to store the state of the system at the end of any run. Being used to
store the state of the system at the end of the run with zero budget, as comparing with
the one with budget to evaluate the budget         */
  long * N1;
  long * N2;
  long * N3;
  long * N4;
  long * N5;

  long * V1;
  long * V2;
  long * V3;
  long * V4;
};
/*****************************************************************************************/
/*    Global variables being used in the program       */
long seed;
int Deterministic;  // 1 for deterministic model, 0 otherwise
const double DistFactor = 0.1;
const double PI = 3.14159265358979323846;
int STATICPOLICY;   // Set the value to 1 for static policy, 0 for dynamic
int EVALPOLICY;     // Set to 1 to not run the GA, just evaluate a given policy
int FEEDBACK;
long MaxX;
long MaxY;
long MaxZ;
struct str_state * ZeroExpState;
struct str_state * FinalState;
struct str_state * OrigState;
FILE * fpOutput;
```

```
/*****************************************************************************/
/* Following variables correspond to the system parameters & the policy being developed*/

double Pv = 0.0007; // Initial value
const double Pc1 = 0.00006;
const double Pc2 = 0.00004;
const double Pd_cancer = 0.03;   // Death rate due to cancer
const double Pd_nat = 0.006;   // Death rate due to natural cause
const double Pbirth = 0.012;   //birth + Immigration rate
const double Pr1 = 0.02;
const double Pr2 = 0.04;
const double P_imm= 0.006;
const double Py2 = 0.3;
const double Py3 = 0.1;

long POPSIZE = 100;
const long TotalPeriods = 21;
const long NoOfGenerations = 1000;
const long CrossoverDist = 10;        // The difference in the array position of the
chromosomes    //    that are being crossed over to get a new solution

// Contact probabilities for the evaluation of Pv. These are independent of N1
const double Alpha2 = 0.000000000007;
const double Alpha3 = 0.000000000007;
const double Alpha4 = 0.000000000007;

const long Init_N1 = 100000000;
const long Init_N2 = 10000000;
const long Init_N3 = 8000000;
const long Init_N4 = 500000;
const long Init_N5 = 0;
const long Init_V1 = 0;
const long Init_V2 = 0;
const long Init_V3 = 0;
const long Init_V4 = 0;

unsigned long TotBudget = 500000000;

const long CostX=90;
const long CostY=8000;
const long CostZ=40;

/* Following are the weights for the objective function. The higher the weight, more
important and better it is for the population to be in this state. C1 is the weight for
healthy people (N1 and V1, V2, V3 and V4) whereas Ci is the weight for people in state i
in general, including that for the dead people!!            */
const double C0 = 0.25;
const double C1 = 0.1;
const double C2 = -0.2;
const double C3 = 0;
const double C4 = -0.5;
const double C5 = -8;
/*****************************************************************************/
void AllocateStrState(struct str_state * FinalState, long Periods);
void AllocInitStrState(struct str_state * InitialState);
long AssignInitSolutions(struct str_solution ** Solution, long NumPeriods, struct
str_state * InitiState);
long Bin(long n,double p);
void CheckPopSize(void);
void Crossover(struct str_solution ParentSoln1, struct str_solution ParentSoln2, struct
str_solution * OffSpring1, struct str_solution * OffSpring2, long NumPeriods, struct
str_state * InitState);
void DuplParentSolns(struct str_solution * Solution, struct str_solution ** ParentSolns,
long NumPeriods);
double EvalObjective(struct str_state * FinalState, struct str_solution Solution, long
NumPeriods);
double EvalSolution(struct str_state * InitState, struct str_solution Solution, struct
str_state * FinalState, int EvalBound, long NumPeriods);
void EvaluateBounds(struct str_solution OrigSolution);
void EvaluatePolicy(struct str_solution OrigSolution, long NoOfReplicates);
void freeSolnStr (struct str_solution * Solution, long NoOfSolutions);
```

```
void freeStrState(struct str_state FinalState);
void fPrintSolution (struct str_solution Solution);
void GenCrossovers(struct str_solution * ParentSolns, struct str_solution ** Solutions,
long NumPeriods, struct str_state * InitState);
void GenMutations(struct str_solution * ParentSolns, struct str_solution ** Solutions,
long NumPeriods, struct str_state * InitState);
long GetBudgetUsed(struct str_solution Solution);
void GetNextState(long x, long y, long z, struct str_state * SystemState, int EvalBound);
void GetOneInitSoln(struct str_solution * Solution, long NumPeriods, struct str_state *
InitState);
void GetZeroExpState(struct str_state * InitState, long NumPeriods);
long MaxIndex(double * Numbers, long NoOfNumbers);
void Mutate(struct str_solution ParentSoln, struct str_solution * OffSpring1, struct
str_solution * OffSpring2, long NumPeriods, struct str_state * InitState);
double Normal(double Mean, double StdDev);
void ObtainDPWithFeedBack(void);
void ObtainPolicyNoFeedBack(struct str_state * InitState, long NumPeriods, struct
str_solution * OptSoln);
void PrintSolution (struct str_solution Solution, long NumPeriods);
void PrintState(struct str_state State);
long SortSolutions(struct str_solution ** Solution, long NumPeriods);
void StrSolnCopy(struct str_solution * Dest, struct str_solution Source, long
NumPeriods);
/*************************************************************************************/

int main(void)
{
 char OutputFile[31];
 long j;
 long NoOfReplicates;
 double ObjValue;
 struct str_solution EvalSoln;
 struct str_solution OptSoln;

 programTimer(0);
 seed = time(NULL);

 FEEDBACK = 0; // Initializing

 printf("Enter the type of policy you want to use:\n");
 STATICPOLICY = GetUserLong("\tDynamic Policy:\t0\n\tStatic Policy:\t1\nEnter your
 Choice: ");
 if (!STATICPOLICY)
 {
     printf("\nDo you want to include feedback in the dynamic policy?\n");
     FEEDBACK = GetUserLong("\tNo Feedback:\t0\n\tWith Feedback:\t1\nEnter your Choice:
     ");
 }
 if (!FEEDBACK)
 {
     printf("\nEnter the type of model you want to use: \n");
     Deterministic = GetUserLong("\tSimulation:\t\t0\n\tDeterministic Model:\t1\nEnter
     your Choice: ");
     printf("\nWhat do you want to do?: \n");
     EVALPOLICY = GetUserLong("\tRun Genetic Algorithm:\t0\n\tEvaluate a
     policy:\t1\nEnter your Choice: ");
 }

 FinalState = (struct str_state *) memAlloc (1, sizeof(struct str_state), "FinalState");
 ZeroExpState = (struct str_state *) memAlloc (1, sizeof(struct str_state),
 "ZeroExpState");
 OrigState = (struct str_state *) memAlloc (1, sizeof(struct str_state), "OrigState");
 AllocateStrState(FinalState, TotalPeriods+1);
 AllocateStrState(ZeroExpState, TotalPeriods+1);
 if (STATICPOLICY)
     sprintf(OutputFile, "Out_%d_Static.xls", TotBudget);
 else
 {
     if (FEEDBACK)
         sprintf(OutputFile, "Out_%d_DynWithFB.xls", TotBudget);
     else
```

```
          sprintf(OutputFile, "Out_%d_DynNoFB.xls", TotBudget);
}
AllocInitStrState(OrigState);
if (EVALPOLICY)
{
     if (Deterministic)
          NoOfReplicates = 1;
     else
          NoOfReplicates = GetUserLong("Enter the number of replications: ");
     EvalSoln.x = (long *) memAlloc (TotalPeriods+1, sizeof(long), "EvalSoln.x");
     EvalSoln.y = (long *) memAlloc (TotalPeriods+1, sizeof(long), "EvalSoln.y");
     EvalSoln.z = (long *) memAlloc (TotalPeriods+1, sizeof(long), "EvalSoln.z");

     for (j=1; j<=TotalPeriods; j++)
     {
          if (STATICPOLICY && j > 1)
          {
               EvalSoln.x[j] = EvalSoln.x[1];
               EvalSoln.y[j] = EvalSoln.y[1];
               EvalSoln.z[j] = EvalSoln.z[1];
          }
          else
          {
               printf("Inputing the values for Period: %d\n", j);
               EvalSoln.x[j] = GetUserLong("Enter the value of x: ");
               EvalSoln.y[j] = GetUserLong("Enter the value of y: ");
               EvalSoln.z[j] = GetUserLong("Enter the value of z: ");
          }
     }
     fpOutput = fopen(OutputFile, "w");
     fprintf(fpOutput, "Total Budget\t%d\n", TotBudget);

     GetZeroExpState(OrigState, TotalPeriods);
     EvaluatePolicy(EvalSoln,NoOfReplicates);
     Deterministic = 1;
     ObjValue = EvalSolution(OrigState, EvalSoln,FinalState, 0, TotalPeriods);
     printf("Expected Objective Value = %lf\n", ObjValue);
     fprintf(fpOutput, "Expected Objective Value\t%lf\n", ObjValue);
     PrintState(*FinalState);
     freeMem(EvalSoln.x);
     freeMem(EvalSoln.y);
     freeMem(EvalSoln.z);
     fclose(fpOutput);
}
else
{
     CheckPopSize();
     if (!FEEDBACK)
     {
          OptSoln.x = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptSoln.x");
          OptSoln.y = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptSoln.y");
          OptSoln.z = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptSoln.z");
          fpOutput = fopen(OutputFile, "w");
          fprintf(fpOutput, "Total Budget\t%d\n", TotBudget);

          ObtainPolicyNoFeedBack(OrigState, TotalPeriods, &OptSoln);
          j = Deterministic;
          Deterministic = 0;
          EvaluatePolicy(OptSoln, 1000);
          Deterministic = 1;
          ObjValue = EvalSolution(OrigState, OptSoln,FinalState, 0, TotalPeriods);
          Deterministic = j;
          printf("Expected Objective Value = %lf\n", ObjValue);
          fprintf(fpOutput, "Expected Objective Value\t%lf\n", ObjValue);
          fPrintSolution(OptSoln);
          PrintState(*FinalState);
          freeMem(OptSoln.x);
          freeMem(OptSoln.y);
          freeMem(OptSoln.z);
          fclose(fpOutput);
     }
```

```
     else
          ObtainDPWithFeedBack();
  }
  freeStrState(*FinalState);
  freeStrState(*ZeroExpState);
  freeStrState(*OrigState);
  freeMem(FinalState);
  freeMem(ZeroExpState);
  freeMem(OrigState);
  printMemoryUsageStatistics();
  programTimer(1);
  return 0;
}
/*****************************************************************************/
void ObtainPolicyNoFeedBack(struct str_state * InitState, long NumPeriods, struct
str_solution * OptSoln)
{
  /* OptSoln is the optimal policy obtained by the GA, and is the return value */
  struct str_solution * ParentSolns;
  struct str_solution * Solution;
  struct str_solution IncSoln;
  long i;
  long CurrGen;

  IncSoln.x = (long *) memAlloc (NumPeriods+1, sizeof(long), "IncSolution.x");
  IncSoln.y = (long *) memAlloc (NumPeriods+1, sizeof(long), "IncSolution.y");
  IncSoln.z = (long *) memAlloc (NumPeriods+1, sizeof(long), "IncSolution.z");

  printf("Total Budget: %d\n", TotBudget);
  MaxX = TotBudget/CostX;
  MaxY = TotBudget/CostY;
  MaxZ = TotBudget/CostZ;
  Solution = (struct str_solution *) memAlloc (POPSIZE+1, sizeof(struct str_solution),
  "Solution");
  ParentSolns = (struct str_solution *) memAlloc (POPSIZE+1, sizeof(struct str_solution),
  "ParentSolns");
  for (i=1; i<=POPSIZE; i++)
  {
      Solution[i].x = (long *) memAlloc (NumPeriods+1, sizeof(long), "Solution.x");
      ParentSolns[i].x = (long *) memAlloc (NumPeriods+1, sizeof(long), "ParentSolns.x");
      Solution[i].y = (long *) memAlloc (NumPeriods+1, sizeof(long), "Solution.y");
      ParentSolns[i].y = (long *) memAlloc (NumPeriods+1, sizeof(long), "ParentSolns.y");
      Solution[i].z = (long *) memAlloc (NumPeriods+1, sizeof(long), "Solution.z");
      ParentSolns[i].z = (long *) memAlloc (NumPeriods+1, sizeof(long), "ParentSolns.z");
  }
  GetZeroExpState(InitState, NumPeriods);
  AssignInitSolutions(&Solution, NumPeriods, InitState);
  SortSolutions(&Solution, NumPeriods);
  StrSolnCopy(&IncSoln, Solution[1], NumPeriods);

  //   printf("%lf\n", IncSoln.x);
  for (CurrGen=2; CurrGen<=NoOfGenerations; CurrGen++)
  {
      if (CurrGen%100 == 0)
          printf("Working on generation number %d...\n", CurrGen);
      DuplParentSolns(Solution, &ParentSolns, NumPeriods);
      GenCrossovers(ParentSolns, &Solution, NumPeriods, InitState);
      GenMutations(ParentSolns, &Solution, NumPeriods, InitState);
      SortSolutions(&Solution, NumPeriods);
      if (IncSoln.Objective < Solution[1].Objective)
      {
          StrSolnCopy(&IncSoln, Solution[1], NumPeriods);
          PrintSolution(IncSoln, NumPeriods);
          GetBudgetUsed(IncSoln);
      }
  }
  StrSolnCopy(OptSoln, IncSoln, NumPeriods);
  freeMem(IncSoln.x);
  freeMem(IncSoln.y);
  freeMem(IncSoln.z);
  freeSolnStr(ParentSolns, POPSIZE);
```

```
  freeMem(ParentSolns);
  freeSolnStr(Solution, POPSIZE);
  freeMem(Solution);

  return;
}
/*****************************************************************************************/
void ObtainDPWithFeedBack(void)
{
  long CurrPeriod;
  long CurrPolicy;
  long NoOfPolicies;
  double * ObjValues;
  double Mean;
  double Variance;
  double CoV;
  char OutputFile[31];
  struct str_state * InitState;
  struct str_state * SolnState;
  struct str_solution * OptPolicy;
  struct str_solution * DynPolicy;

  NoOfPolicies = GetUserLong("Enter the number of policies you want to obtain: ");
  ObjValues = (double *) memAlloc (NoOfPolicies+1, sizeof(double), "ObjValues");
  InitState = (struct str_state *) memAlloc (1, sizeof(struct str_state), "InitState");
  SolnState = (struct str_state *) memAlloc (1, sizeof(struct str_state), "SolnState");
  OptPolicy = (struct str_solution *) memAlloc (1, sizeof(struct str_solution),
  "OptPolicy");
  DynPolicy = (struct str_solution *) memAlloc (1, sizeof(struct str_solution),
  "DynPolicy");
  AllocateStrState(InitState,1);
  OptPolicy->x = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptPolicy->x");
  OptPolicy->y = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptPolicy->y");
  OptPolicy->z = (long *) memAlloc (TotalPeriods+1, sizeof(long), "OptPolicy->z");
  DynPolicy->x = (long *) memAlloc (TotalPeriods+1, sizeof(long), "DynPolicy->x");
  DynPolicy->y = (long *) memAlloc (TotalPeriods+1, sizeof(long), "DynPolicy->y");
  DynPolicy->z = (long *) memAlloc (TotalPeriods+1, sizeof(long), "DynPolicy->z");
  AllocateStrState(SolnState,TotalPeriods+1);

  for (CurrPolicy=1; CurrPolicy <= NoOfPolicies; CurrPolicy++)
  {
       sprintf(OutputFile, "Out_%d_DynWFB_%d.xls", TotBudget, CurrPolicy);
       fpOutput = fopen(OutputFile, "w");
       fprintf(fpOutput, "Total Budget\t%d\n", TotBudget);

       SolnState->N1[1] = OrigState->N1[1];
       SolnState->N2[1] = OrigState->N2[1];
       SolnState->N3[1] = OrigState->N3[1];
       SolnState->N4[1] = OrigState->N4[1];
       SolnState->N5[1] = OrigState->N5[1];
       SolnState->V1[1] = OrigState->V1[1];
       SolnState->V2[1] = OrigState->V2[1];
       SolnState->V3[1] = OrigState->V3[1];
       SolnState->V4[1] = OrigState->V4[1];

       for (CurrPeriod=1; CurrPeriod <= TotalPeriods; CurrPeriod++)
       {
            InitState->N1[0] = InitState->N1[1] = SolnState->N1[CurrPeriod];
            InitState->N2[0] = InitState->N2[1] = SolnState->N2[CurrPeriod];
            InitState->N3[0] = InitState->N3[1] = SolnState->N3[CurrPeriod];
            InitState->N4[0] = InitState->N4[1] = SolnState->N4[CurrPeriod];
            InitState->N5[0] = InitState->N5[1] = SolnState->N5[CurrPeriod];
            InitState->V1[0] = InitState->V1[1] = SolnState->V1[CurrPeriod];
            InitState->V2[0] = InitState->V2[1] = SolnState->V2[CurrPeriod];
            InitState->V3[0] = InitState->V3[1] = SolnState->V3[CurrPeriod];
            InitState->V4[0] = InitState->V4[1] = SolnState->V4[CurrPeriod];

            Deterministic = 1;
            ObtainPolicyNoFeedBack(InitState, 1+TotalPeriods-CurrPeriod, OptPolicy);

            DynPolicy->x[CurrPeriod] = OptPolicy->x[1];
```

```
            DynPolicy->y[CurrPeriod] = OptPolicy->y[1];
            DynPolicy->z[CurrPeriod] = OptPolicy->z[1];

            Deterministic = 0;

            GetNextState(OptPolicy->x[1],OptPolicy->y[1],OptPolicy->z[1],InitState,0);

            SolnState->N1[CurrPeriod+1] = InitState->N1[1];
            SolnState->N2[CurrPeriod+1] = InitState->N2[1];
            SolnState->N3[CurrPeriod+1] = InitState->N3[1];
            SolnState->N4[CurrPeriod+1] = InitState->N4[1];
            SolnState->N5[CurrPeriod+1] = InitState->N5[1];
            SolnState->V1[CurrPeriod+1] = InitState->V1[1];
            SolnState->V2[CurrPeriod+1] = InitState->V2[1];
            SolnState->V3[CurrPeriod+1] = InitState->V3[1];
            SolnState->V4[CurrPeriod+1] = InitState->V4[1];

        }
        ObjValues[CurrPolicy] = EvalObjective(SolnState, * DynPolicy, TotalPeriods);

        fprintf(fpOutput, "Objective Value\t%lf\n", ObjValues[CurrPolicy]);
        fPrintSolution(*DynPolicy);
        PrintState(*SolnState);
        fclose(fpOutput);

        sprintf(OutputFile, "Out_%d_DynWFB_Summary.xls", TotBudget);
        fpOutput = fopen(OutputFile, "w");
        fprintf(fpOutput, "Total Budget\t%d\n", TotBudget);

        Mean = GetMean(ObjValues, NoOfPolicies);
        printf("\n\nMean Objective Value: %lf\n", Mean);
        fprintf(fpOutput, "Mean Objective\t%lf\n", Mean);
        Variance = GetVariance(ObjValues, NoOfPolicies);
        printf("Variance in Objective Values: %lf\n", Variance);
        fprintf(fpOutput, "Variance\t%lf\n", Variance);
        CoV = fabs(sqrt(Variance)/Mean);
        printf("Coeff. Of Variation = %lf\n", CoV);
        printf("Highest Value: %lf\n", GetMax(ObjValues, NoOfPolicies));
        printf("Lowest Value: %lf\n", GetMin(ObjValues, NoOfPolicies));
        fprintf(fpOutput, "Coeff. Of Variation\t%lf\n", CoV);
        fprintf(fpOutput, "Highest Value\t%lf\n", GetMax(ObjValues, NoOfPolicies));
        fprintf(fpOutput, "Lowest Value\t%lf\n", GetMin(ObjValues, NoOfPolicies));
        fclose(fpOutput);
    }
    freeStrState(*InitState);
    freeStrState(*SolnState);
    freeMem(OptPolicy->x);
    freeMem(OptPolicy->y);
    freeMem(OptPolicy->z);
    freeMem(DynPolicy->x);
    freeMem(DynPolicy->y);
    freeMem(DynPolicy->z);
    freeMem(InitState);
    freeMem(ObjValues);
    freeMem(SolnState);
    freeMem(OptPolicy);
    freeMem(DynPolicy);

    return;
}
/*****************************************************************************************/
void EvaluateBounds(struct str_solution OrigSolution)
{
    double UpperBound, LowerBound;
    int Det;

    Det = Deterministic;
    Deterministic = 1;

    UpperBound = EvalSolution(OrigState, OrigSolution, FinalState, 1, TotalPeriods);
    LowerBound = EvalSolution(OrigState, OrigSolution, FinalState, -1, TotalPeriods);
```

```
  printf("Upper Bound for the policy: %lf\n", UpperBound);
  printf("Lower Bound for the policy: %lf\n", LowerBound);
  fprintf(fpOutput, "Upper Bound\t%lf\n", UpperBound);
  fprintf(fpOutput, "Lower Bound\t%lf\n", LowerBound);
  Deterministic = Det;

  return;
}
/*****************************************************************************************/
void GetZeroExpState(struct str_state * InitState, long NumPeriods)
{ /* This function obtains and stores the state of the system had the budget been zero.
  This is used to evaluate the objective function, since the objective is to capture the
  affect of putting in money in the system.         */
  struct str_solution ZeroSoln;
  long i;
  int OrigDeterministic;

  ZeroSoln.x = (long *) memAlloc (NumPeriods+1, sizeof(long), "ZeroSoln.x");
  ZeroSoln.y = (long *) memAlloc (NumPeriods+1, sizeof(long), "ZeroSoln.y");
  ZeroSoln.z = (long *) memAlloc (NumPeriods+1, sizeof(long), "ZeroSoln.z");

  for (i=1; i<=NumPeriods; i++)
  {
       ZeroSoln.x[i] = 0;
       ZeroSoln.y[i] = 0;
       ZeroSoln.z[i] = 0;
  }
  OrigDeterministic = Deterministic;
  Deterministic = 1;     // To compute the expected value for the zero budget  */
  EvalSolution(InitState, ZeroSoln, ZeroExpState, 0, NumPeriods);
//    PrintState(*ZeroExpState, "ZeroState.xls", 0);

  Deterministic = OrigDeterministic;
  freeMem(ZeroSoln.x);
  freeMem(ZeroSoln.y);
  freeMem(ZeroSoln.z);
}
/*****************************************************************************************/
void PrintState(struct str_state State)
{
  long CurrPeriod;

  fprintf(fpOutput, "N1\tN2\tN3\tN4\tN5\tV1\tV2\tV3\tV4\n");
  for (CurrPeriod=1; CurrPeriod<=TotalPeriods+1; CurrPeriod++)
       fprintf(fpOutput, "%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", State.N1[CurrPeriod],
                State.N2[CurrPeriod], State.N3[CurrPeriod], State.N4[CurrPeriod],
                State.N5[CurrPeriod], State.V1[CurrPeriod], State.V2[CurrPeriod],
                State.V3[CurrPeriod], State.V4[CurrPeriod]);
  return;
}
/*****************************************************************************************/
void EvaluatePolicy(struct str_solution OrigSolution, long NoOfReplicates)
{
  long i, j;
  double * ObjValues;
  double Mean, Variance, CoV;
  struct str_solution * Solution;

  Solution = (struct str_solution *) memAlloc (NoOfReplicates+1, sizeof(struct
  str_solution), "Solution");
  for (i=1; i<=NoOfReplicates; i++)
  {
       Solution[i].x = (long *) memAlloc (TotalPeriods+1, sizeof(long), "Solution.x");
       Solution[i].y = (long *) memAlloc (TotalPeriods+1, sizeof(long), "Solution.y");
       Solution[i].z = (long *) memAlloc (TotalPeriods+1, sizeof(long), "Solution.z");
  }
  for(i=1; i<=NoOfReplicates; i++)
  {
       for(j=1; j<=TotalPeriods; j++)
       {
```

```
          Solution[i].x[j] = OrigSolution.x[j];
          Solution[i].y[j] = OrigSolution.y[j];
          Solution[i].z[j] = OrigSolution.z[j];
     }
 }
 ObjValues = (double *) memAlloc (NoOfReplicates+1, sizeof(double), "ObjValues");
 printf("\n\nPerforming replications to evaluate the policy...\n");
 for (i=1; i<=NoOfReplicates; i++)
 {
     if(i%1000 == 0)
          printf("Evaluating replication number %d\n", i);
     Solution[i].Objective = EvalSolution(OrigState, Solution[i], FinalState, 0,
     TotalPeriods);
     ObjValues[i] = Solution[i].Objective;
 }
 Mean = GetMean(ObjValues, NoOfReplicates);
 printf("\n\nMean Objective Value: %lf\n", Mean);
 fprintf(fpOutput, "Mean Objective Value\t%lf\n", Mean);
 Variance = GetVariance(ObjValues, NoOfReplicates);
 printf("Variance in Objective Values: %lf\n", Variance);
 fprintf(fpOutput, "Variance\t%lf\n", Variance);
 CoV = fabs(sqrt(Variance)/Mean);
 printf("Coeff. Of Variation = %lf\n", CoV);
 printf("Highest Value: %lf\n", GetMax(ObjValues, NoOfReplicates));
 printf("Lowest Value: %lf\n", GetMin(ObjValues, NoOfReplicates));
 fprintf(fpOutput, "Coeff. Of Variation\t%lf\n", CoV);
 fprintf(fpOutput, "Highest Value\t%lf\n", GetMax(ObjValues, NoOfReplicates));
 fprintf(fpOutput, "Lowest Value\t%lf\n", GetMin(ObjValues, NoOfReplicates));

 freeMem(ObjValues);
 EvaluateBounds(OrigSolution);

 freeSolnStr(Solution, NoOfReplicates);
 freeMem(Solution);

 return;
}
/*****************************************************************************************/
void StrSolnCopy(struct str_solution * Dest, struct str_solution Source, long NumPeriods)
{
 long i;

 for(i=1; i<=NumPeriods; i++)
 {
     Dest->x[i] = Source.x[i];
     Dest->y[i] = Source.y[i];
     Dest->z[i] = Source.z[i];
 }
 Dest->Objective = Source.Objective;

 return;
}
/*****************************************************************************************/
void PrintSolution (struct str_solution Solution, long NumPeriods)
{
 long i;

 printf("Period\tX\tY\tZ\n");
 for(i=1; i<=NumPeriods; i++)
     printf("%d\t%d\t%d\t%d\n",i, Solution.x[i], Solution.y[i], Solution.z[i]);
 printf("Objective Value: %lf\n\n\n", Solution.Objective);

 return;
}
/*****************************************************************************************/
void fPrintSolution (struct str_solution Solution)
{
 long i;

 fprintf(fpOutput, "Period\tX\tY\tZ\n");
 for(i=1; i<=TotalPeriods; i++)
```

```
      fprintf(fpOutput, "%d\t%d\t%d\t%d\n", i, Solution.x[i], Solution.y[i],
      Solution.z[i]);
  return;
}
/*****************************************************************************/
void freeSolnStr (struct str_solution * Solution, long NoOfSolutions)
{
  long i;

  for (i=1; i<=NoOfSolutions; i++)
  {
      freeMem(Solution[i].x);
      freeMem(Solution[i].y);
      freeMem(Solution[i].z);
  }
  return;
}
/*****************************************************************************/
void CheckPopSize(void)
{
  if(POPSIZE%100 != 0 || POPSIZE <= 0)
  {
      printf("Population size assigned is %d. It must be a multiple of 100.\n", POPSIZE);
      if(POPSIZE%100 < 50)
          POPSIZE = POPSIZE - POPSIZE%100;
      else
          POPSIZE = 100 + POPSIZE - POPSIZE%100;
      if (POPSIZE < 100)
          POPSIZE = 100;
      printf("Changing it to %d\n", POPSIZE);
  }
  return;
}
/*****************************************************************************/
void GenMutations(struct str_solution * ParentSolns, struct str_solution ** Solutions,
long NumPeriods, struct str_state * InitState)
{
  /*  We  will  generate  mutations  from  parent  solutions  1  to  POPSIZE/5  and  from
0.6*POPSIZE+1 to 0.65*POPSIZE. This would give us a total of POPSIZE/5 + 0.05POPSIZE =
POPSIZE/4 offsprings. We will have mutations down and up, so that will double the number
of offsprings, bringing it to exactly POPSIZE/2             */
  /* We know that POPSIZE is an exact multiple of 100       */
  long i;
  long j;

  for(i=1, j=POPSIZE/2+1; i<=POPSIZE/5; i++, j+=2)
      Mutate(ParentSolns[i],&((*Solutions)[j]),&((*Solutions)[j+1]), NumPeriods,
      InitState);

  for(i=(long)(0.6*POPSIZE)+1, j=(long)(0.9*POPSIZE)+1; i<=0.65*POPSIZE; i++, j+=2)
  {
  /* Rather than mutating weaker solutions, generate new solutions from scratch */
      GetOneInitSoln(&((*Solutions)[j]), NumPeriods, InitState);
      GetOneInitSoln(&((*Solutions)[j+1]), NumPeriods, InitState);
//    Mutate(ParentSolns[i],&((*Solutions)[j]),&((*Solutions)[j+1]));
  }

  return;
}
/*****************************************************************************/
void Mutate(struct str_solution ParentSoln, struct str_solution * OffSpring1, struct
str_solution * OffSpring2, long NumPeriods, struct str_state * InitState)
{
  long i;
  double Rand;
  double BudgetUsed;

  for (i=1; i<=NumPeriods; i++)
  {
      if (STATICPOLICY && i>1)
      {
```

```
            OffSpring1->x[i] = OffSpring1->x[i-1];
            OffSpring1->y[i] = OffSpring1->y[i-1];
            OffSpring1->z[i] = OffSpring1->z[i-1];

            OffSpring2->x[i] = OffSpring2->x[i-1];
            OffSpring2->y[i] = OffSpring2->y[i-1];
            OffSpring2->z[i] = OffSpring2->z[i-1];
        }
        else
        {
/* First, we increase x by somewhere between 5 and 20%, and reduce y proportionally */
            Rand = ran2(&seed);
            Rand = (1+20*Rand)/100;
            BudgetUsed = Rand*ParentSoln.x[i] * CostX;
            OffSpring1->x[i] = (long)((1+Rand)*ParentSoln.x[i]);

            if (ParentSoln.z[i] < (long)(BudgetUsed/(2*CostZ)))
            {   /* Half because we are allocating equal budgets to Y and Z */
                OffSpring1->z[i] = 0;
                OffSpring1->y[i] = (long)(TotBudget - (OffSpring1->x[i])*CostX)/CostY;
            }
            else if(ParentSoln.y[i] < (long)(BudgetUsed/(2*CostY)))
            {
                OffSpring1->y[i] = 0;
                OffSpring1->z[i] = (long)(TotBudget - (OffSpring1->x[i])*CostX)/CostZ;
            }
            else
            {
                OffSpring1->y[i] = ParentSoln.y[i] - (long) (BudgetUsed/(2*CostY));
                if (TotBudget > (OffSpring1->x[i])*CostX + (OffSpring1->y[i])*CostY)
                {
                    BudgetUsed = TotBudget - (OffSpring1->x[i])*CostX - (OffSpring1-
                    >y[i])*CostY;
                    OffSpring1->z[i] = (long)(BudgetUsed/CostZ);
                }
                else
                {
                    BudgetUsed = (OffSpring1->x[i])*CostX + (OffSpring1->y[i])*CostY -
                    TotBudget;
                    OffSpring1->z[i] = 0;
                    if (OffSpring1->y[i] > (long)(BudgetUsed / CostY))
                    {
                        OffSpring1->y[i] = OffSpring1->y[i] - (long)(BudgetUsed / CostY);
                        BudgetUsed -= CostY * (long)(BudgetUsed/CostY);
                    }
                    else
                    {
                        BudgetUsed -= CostY * OffSpring1->y[i];
                        OffSpring1->y[i] = 0;
                    }
                    OffSpring1->x[i] = OffSpring1->x[i] - (long)(BudgetUsed / CostX);
                }
            }
            if (OffSpring1->x[i] > MaxX)
            {
                OffSpring1->x[i] = MaxX;
                OffSpring1->y[i] = 0;
                OffSpring1->z[i] = 0;
            }
/* Then, we increase y by somewhere between 5 and 20%, and reduce x proportionally */
            Rand = ran2(&seed);
            Rand = (1+20*Rand)/100;
            BudgetUsed = Rand*ParentSoln.y[i] * CostY;
            OffSpring2->y[i] = (long)((1+Rand)*ParentSoln.y[i]);
            if (ParentSoln.z[i] < BudgetUsed/(2*CostZ))
            {
                OffSpring2->z[i] = 0;
                OffSpring2->x[i] = (long)(TotBudget - (OffSpring2->y[i])*CostY)/CostX;
            }
            else if (ParentSoln.x[i] < (long)BudgetUsed/(2*CostX))
            {
```

```
                OffSpring2->x[i] = 0;
                OffSpring2->z[i] = (long)(TotBudget - (OffSpring2->y[i])*CostY)/CostZ;
           }
           else
           {
                OffSpring2->x[i] = ParentSoln.x[i] - (long) (BudgetUsed/(2*CostX));

                if (TotBudget > (OffSpring2->x[i])*CostX + (OffSpring2->y[i])*CostY)
                {
                     BudgetUsed = TotBudget - (OffSpring2->x[i])*CostX - (OffSpring2-
                     >y[i])*CostY;
                     OffSpring2->z[i] = (long)(BudgetUsed/CostZ);
                }
                else
                {
                     BudgetUsed = (OffSpring2->x[i])*CostX + (OffSpring2->y[i])*CostY -
                     TotBudget;
                     OffSpring2->z[i] = 0;
                     if (OffSpring2->y[i] > (long)(BudgetUsed / CostY))
                     {
                          OffSpring2->y[i] = OffSpring2->y[i] - (long)(BudgetUsed / CostY);
                          BudgetUsed -= CostY * (long)(BudgetUsed/CostY);
                     }
                     else
                     {
                          BudgetUsed -= CostY * OffSpring2->y[i];
                          OffSpring2->y[i] = 0;
                     }
                     OffSpring2->x[i] = OffSpring2->x[i] - (long)(BudgetUsed / CostX);
                }
           }
           if (OffSpring2->y[i] > MaxY)
           {
                OffSpring2->y[i] = MaxY;
                OffSpring2->x[i] = 0;
                OffSpring2->z[i] = 0;
           }
      }
  }
  GetBudgetUsed(*OffSpring1);
  GetBudgetUsed(*OffSpring2);
//    printf("Budget = %d\n", GetBudgetUsed(*OffSpring1));
//    printf("Budget = %d\n", GetBudgetUsed(*OffSpring2));
  OffSpring1->Objective = EvalSolution(InitState, *OffSpring1, FinalState, 0, NumPeriods);
  OffSpring2->Objective = EvalSolution(InitState, *OffSpring2, FinalState, 0, NumPeriods);
  return;
}
/*****************************************************************************/
long GetBudgetUsed(struct str_solution Solution)
{
  unsigned long BudgetUsed;
  double Excess;

  BudgetUsed = CostZ*(Solution.z[1]) + CostX*(Solution.x[1]) + CostY*(Solution.y[1]);
  if (BudgetUsed > TotBudget)
  {
      Excess = (double)BudgetUsed/TotBudget - 1;
      if (Excess > 0.0001)
           printf("Budget Exceeded by %lf!!\n", Excess*100);
  }
  return BudgetUsed;
}
/*****************************************************************************/
void GenCrossovers(struct str_solution * ParentSolns, struct str_solution ** Solutions,
long NumPeriods, struct str_state * InitState)
{
  long i;
  long ItrNum;
  long NoOfCrossovers;

  ItrNum = 1;
```

```
  NoOfCrossovers = 0;
  while(NoOfCrossovers < POPSIZE/2)
  {
       for (i=1+2*CrossoverDist*(ItrNum-1); i<1+2*CrossoverDist*(ItrNum-1)+CrossoverDist &&
       NoOfCrossovers<POPSIZE/2; i++)
       {
//          printf("%d\n%d\n", ParentSolns[i].x, ParentSolns[i+CrossoverDist].x);
            Crossover(ParentSolns[i], ParentSolns[i+CrossoverDist],
            &((*Solutions)[NoOfCrossovers+1]), &((*Solutions)[NoOfCrossovers+2]),
            NumPeriods, InitState);
            NoOfCrossovers+=2;
       }
       ItrNum++;
  }
  return;
}
/*****************************************************************************************/
void Crossover(struct str_solution ParentSoln1, struct str_solution ParentSoln2, struct
str_solution * OffSpring1, struct str_solution * OffSpring2, long NumPeriods, struct
str_state * InitState)
{
  unsigned long BudgetUsed;
  long i;

  for (i=1; i<=NumPeriods; i++)
  {
       if (STATICPOLICY && i>1)
       {
            OffSpring1->x[i] = OffSpring1->x[i-1];
            OffSpring1->y[i] = OffSpring1->y[i-1];
            OffSpring1->z[i] = OffSpring1->z[i-1];

            OffSpring2->x[i] = OffSpring2->x[i-1];
            OffSpring2->y[i] = OffSpring2->y[i-1];
            OffSpring2->z[i] = OffSpring2->z[i-1];
       }
       else
       {
            OffSpring1->x[i] = (ParentSoln1.x[i] +ParentSoln2.x[i])/2;
            OffSpring1->y[i] = (ParentSoln1.y[i] +ParentSoln2.y[i])/2;
//          OffSpring1->z[i] = (ParentSoln1.z[i] +ParentSoln2.z[i])/2;

            BudgetUsed = CostX * (OffSpring1->x[i]);
            BudgetUsed += CostY * (OffSpring1->y[i]);
            if (BudgetUsed > TotBudget)
            {
                 OffSpring1->z[i] = 0;
                 BudgetUsed = BudgetUsed - TotBudget;
                 if (OffSpring1->y[i] > (long)(BudgetUsed / CostY))
                 {
                      OffSpring1->y[i] = OffSpring1->y[i] - (long)(BudgetUsed / CostY);
                      BudgetUsed -= CostY * (long)(BudgetUsed/CostY);
                 }
                 else
                 {
                      BudgetUsed -= CostY * OffSpring1->y[i];
                      OffSpring1->y[i] = 0;
                 }
                 OffSpring1->x[i] = OffSpring1->x[i] - (long)(BudgetUsed / CostX);
            }
            else
                 OffSpring1->z[i] = (TotBudget - BudgetUsed)/CostZ;
            // We know that the parent solution 1 is better than the parent solution 2
            OffSpring2->x[i] = ParentSoln1.x[i] + (long) (DistFactor*(ParentSoln1.x[i] -
            ParentSoln2.x[i]));
            if (OffSpring2->x[i] < 0)
                 OffSpring2->x[i] = 0;
            OffSpring2->y[i] = ParentSoln1.y[i] + (long) (DistFactor*(ParentSoln1.y[i]-
            ParentSoln2.y[i]));
            if (OffSpring2->y[i] < 0)
                 OffSpring2->y[i] = 0;
```

```
          BudgetUsed = CostX * (OffSpring2->x[i]);
          BudgetUsed += CostY * (OffSpring2->y[i]);

          if (OffSpring2->x[i] > MaxX)
          {
               OffSpring2->x[i] = MaxX;
               OffSpring2->y[i] = 0;
               OffSpring2->z[i] = 0;
          }
          if (OffSpring2->y[i] > MaxY)
          {
               OffSpring2->x[i] = 0;
               OffSpring2->y[i] = MaxY;
               OffSpring2->z[i] = 0;
          }
          if (BudgetUsed > TotBudget)
          {
               OffSpring2->z[i] = 0;
               BudgetUsed = BudgetUsed - TotBudget;
               if (OffSpring2->y[i] > (long)(BudgetUsed / CostY))
               {
                    OffSpring2->y[i] = OffSpring2->y[i] - (long)(BudgetUsed / CostY);
                    BudgetUsed -= CostY * (long)(BudgetUsed/CostY);
               }
               else
               {
                    BudgetUsed -= CostY * OffSpring2->y[i];
                    OffSpring2->y[i] = 0;
               }
               OffSpring2->x[i] = OffSpring2->x[i] - (long)(BudgetUsed / CostX);
          }
          else
               OffSpring2->z[i] = (TotBudget - BudgetUsed)/CostZ;

     }
  }
  OffSpring1->Objective = EvalSolution(InitState, *OffSpring1, FinalState, 0, NumPeriods);

  OffSpring2->Objective = EvalSolution(InitState, *OffSpring2, FinalState, 0, NumPeriods);

  GetBudgetUsed(*OffSpring1);
  GetBudgetUsed(*OffSpring2);
  return;
}
/*****************************************************************************/
void DuplParentSolns(struct str_solution * Solution, struct str_solution ** ParentSolns,
long NumPeriods)
{
  long i;

  for(i=1; i<=POPSIZE; i++)
       StrSolnCopy(&((*ParentSolns)[i]), Solution[i], NumPeriods);
  return;
}
/*****************************************************************************/
long AssignInitSolutions(struct str_solution ** Solution, long NumPeriods, struct
str_state * InitState)
{
  long i;

  for (i=1; i<=POPSIZE; i++)
       GetOneInitSoln(&((*Solution)[i]), NumPeriods, InitState);
  return 0;
}
/*****************************************************************************/
void GetOneInitSoln(struct str_solution * Solution, long NumPeriods, struct str_state *
InitState)
{
  long j;
  double FractForX;
  double FractForY;
```

```
 double FractForZ;
 double Sum;

 for (j=1; j<=NumPeriods; j++)
 {
     if(STATICPOLICY && j>1)
     {
          Solution->x[j] = Solution->x[j-1];
          Solution->y[j] = Solution->y[j-1];
          Solution->z[j] = Solution->z[j-1];
     }
     else
     {

          FractForX = ran2(&seed);
          FractForY = ran2(&seed);
          FractForZ = ran2(&seed);
          Sum = FractForX + FractForY + FractForZ;

          FractForX = FractForX/Sum;
          FractForY = FractForY/Sum;
          FractForZ = FractForZ/Sum;

          Solution->x[j] = (long)(MaxX * FractForX);
          Solution->y[j] = (long)(MaxY * FractForY);
          Solution->z[j] = (TotBudget - CostX*Solution->x[j] - CostY*Solution-
          >y[j])/CostZ;
     }
 }

 Solution->Objective = EvalSolution(InitState, *Solution, FinalState, 0, NumPeriods);
 GetBudgetUsed(*Solution);
 return;
}
/*****************************************************************************/
long SortSolutions(struct str_solution ** Solution, long NumPeriods)
{
 struct str_solution TempSoln;
 long i, j;

 TempSoln.x = (long *) memAlloc (NumPeriods+1, sizeof(long), "TempSoln.x");
 TempSoln.y = (long *) memAlloc (NumPeriods+1, sizeof(long), "TempSoln.y");
 TempSoln.z = (long *) memAlloc (NumPeriods+1, sizeof(long), "TempSoln.z");

 for (j=1; j<POPSIZE; j++)
 {
     for(i=1; i<=POPSIZE-j; i++)
     {
          if((*Solution)[i].Objective < (*Solution)[i+1].Objective)
          {
               StrSolnCopy(&TempSoln, (*Solution)[i], NumPeriods);
               StrSolnCopy(&((*Solution)[i]), (*Solution)[i+1], NumPeriods);
               StrSolnCopy(&((*Solution)[i+1]), TempSoln, NumPeriods);
          }
     }
 }
 freeMem(TempSoln.x);
 freeMem(TempSoln.y);
 freeMem(TempSoln.z);
 return 0;
}
/*****************************************************************************/
void AllocateStrState(struct str_state * FinalState, long Periods)
{
 FinalState->N1 = (long *) memAlloc (Periods+1, sizeof(long), "N1");
 FinalState->N2 = (long *) memAlloc (Periods+1, sizeof(long), "N2");
 FinalState->N3 = (long *) memAlloc (Periods+1, sizeof(long), "N3");
 FinalState->N4 = (long *) memAlloc (Periods+1, sizeof(long), "N4");
 FinalState->N5 = (long *) memAlloc (Periods+1, sizeof(long), "N5");
 FinalState->V1 = (long *) memAlloc (Periods+1, sizeof(long), "V1");
 FinalState->V2 = (long *) memAlloc (Periods+1, sizeof(long), "V2");
```

```
 FinalState->V3 = (long *) memAlloc (Periods+1, sizeof(long), "V3");
 FinalState->V4 = (long *) memAlloc (Periods+1, sizeof(long), "V4");
 return;
}
/*****************************************************************************/
void AllocInitStrState(struct str_state * InitialState)
{
 InitialState->N1 = (long *) memAlloc (2, sizeof(long), "N1");
 InitialState->N2 = (long *) memAlloc (2, sizeof(long), "N2");
 InitialState->N3 = (long *) memAlloc (2, sizeof(long), "N3");
 InitialState->N4 = (long *) memAlloc (2, sizeof(long), "N4");
 InitialState->N5 = (long *) memAlloc (2, sizeof(long), "N5");
 InitialState->V1 = (long *) memAlloc (2, sizeof(long), "V1");
 InitialState->V2 = (long *) memAlloc (2, sizeof(long), "V2");
 InitialState->V3 = (long *) memAlloc (2, sizeof(long), "V3");
 InitialState->V4 = (long *) memAlloc (2, sizeof(long), "V4");

 InitialState->N1[1] = Init_N1;
 InitialState->N2[1] = Init_N2;
 InitialState->N3[1] = Init_N3;
 InitialState->N4[1] = Init_N4;
 InitialState->N5[1] = Init_N5;
 InitialState->V1[1] = Init_V1;
 InitialState->V2[1] = Init_V2;
 InitialState->V3[1] = Init_V3;
 InitialState->V4[1] = Init_V4;

 return;
}
/*****************************************************************************/
void freeStrState(struct str_state FinalState)
{
 freeMem(FinalState.N1);
 freeMem(FinalState.N2);
 freeMem(FinalState.N3);
 freeMem(FinalState.N4);
 freeMem(FinalState.N5);
 freeMem(FinalState.V1);
 freeMem(FinalState.V2);
 freeMem(FinalState.V3);
 freeMem(FinalState.V4);
 return;
}
/*****************************************************************************/
double EvalSolution(struct str_state * InitState, struct str_solution Solution, struct
str_state * FinalState, int EvalBound, long NumPeriods)
{
 /* The function takes in the 'Solution' that need to be evaluated. 'FinalState'
 is the state of the system at the end of all the periods, and is returned as a
 pointer to the structure.
 The final argument is whether we want to evaluate the bounds, or just evaluate the
 policy normally.
      0: Normal Evaluation
      1: Evaluate Upper Bound
      -1: Evaluate Lower Bound       */
 int i;
 long x;
 long y;
 long z;

 double ObjValue;
 struct str_state * SystemState;

 SystemState = (struct str_state *) memAlloc (1, sizeof(struct str_state),
 "SystemState");
 AllocateStrState(SystemState, 1);

 FinalState->N1[1] = InitState->N1[1];
 FinalState->N2[1] = InitState->N2[1];
 FinalState->N3[1] = InitState->N3[1];
 FinalState->N4[1] = InitState->N4[1];
```

```
  FinalState->N5[1] = InitState->N5[1];
  FinalState->V1[1] = InitState->V1[1];
  FinalState->V2[1] = InitState->V2[1];
  FinalState->V3[1] = InitState->V3[1];
  FinalState->V4[1] = InitState->V4[1];

  for(i=1;i<=NumPeriods;i++)
  {
       x = Solution.x[i];
       y = Solution.y[i];
       z = Solution.z[i];

       SystemState->N1[0] = FinalState->N1[i];
       SystemState->N2[0] = FinalState->N2[i];
       SystemState->N3[0] = FinalState->N3[i];
       SystemState->N4[0] = FinalState->N4[i];
       SystemState->N5[0] = FinalState->N5[i];
       SystemState->V1[0] = FinalState->V1[i];
       SystemState->V2[0] = FinalState->V2[i];
       SystemState->V3[0] = FinalState->V3[i];
       SystemState->V4[0] = FinalState->V4[i];

       GetNextState(x, y, z, SystemState, EvalBound);

       FinalState->N1[i+1] = SystemState->N1[1];
       FinalState->N2[i+1] = SystemState->N2[1];
       FinalState->N3[i+1] = SystemState->N3[1];
       FinalState->N4[i+1] = SystemState->N4[1];
       FinalState->N5[i+1] = SystemState->N5[1];
       FinalState->V1[i+1] = SystemState->V1[1];
       FinalState->V2[i+1] = SystemState->V2[1];
       FinalState->V3[i+1] = SystemState->V3[1];
       FinalState->V4[i+1] = SystemState->V4[1];
  }
  ObjValue = EvalObjective(FinalState, Solution, NumPeriods);
//    printf("N4[1] = %d\t N4[%d] = %d\n", N4[1], NoOfPeriods+1, N4[NoOfPeriods+1]);
  freeStrState(* SystemState);
  freeMem(SystemState);
  return ObjValue;
}
/*****************************************************************************************/
void GetNextState(long x, long y, long z, struct str_state * SystemState, int EvalBound)
{
  /* EvalBound = 1 for UpperBound, -1 for LowerBound and 0 for normal simulation   */
  long NumBorn;
  long NumSusceptible;
  long NumGetVirus;
  long NumEffTest;
  long NumEffVacc;
  long NumN4ToN2;
  long NumN4ToN3;
  long NumN1Dead;
  long NumN2Dead;
  long NumN2Rec;
  long NumN2ToCancer;
  long NumN3ToCancer;
  long NumN3Rec;
  long NumN4Rec;
  long NumN3Dead;
  long NumN4NatDead;
  long NumN4CanDead;
  long NumV1Dead;
  long NumV2Dead;
  long NumV3Dead;
  long NumV4Dead;
  long TotalPop;
  double Ratio;

  const long N1 = SystemState->N1[0];
  const long N2 = SystemState->N2[0];
  const long N3 = SystemState->N3[0];
```

```
const long N4 = SystemState->N4[0];
const long N5 = SystemState->N5[0];
const long V1 = SystemState->V1[0];
const long V2 = SystemState->V2[0];
const long V3 = SystemState->V3[0];
const long V4 = SystemState->V4[0];

if (y > N4)
   y = N4;

Pv = Alpha2*N2 + Alpha3*N3 + Alpha4*N4;
if (EvalBound == 1)
{
     if (N1 > x)
          NumEffVacc = x;
     else
          NumEffVacc = N1;
}
else if (EvalBound == -1)
{
     if (x < N2)
          NumEffVacc = 0;
     else
     {
          NumEffVacc = x - N2;
          if (NumEffVacc > N1)
               NumEffVacc = N1;
     }
}
else
{
     if (x >= N1+N2)
          NumEffVacc = N1;
     else
     {
          NumEffVacc = Bin(x,(double) N1/(N1+N2));
          if (NumEffVacc > N1)
               NumEffVacc = N1;
     }
}
NumSusceptible = N1 - NumEffVacc;
NumGetVirus = Bin(NumSusceptible,Pv);
TotalPop = N1+N2+N3+N4+V1+V2+V3+V4;
NumBorn = (long) (Pbirth*TotalPop);
NumN2Rec = Bin(N2,Pr1);
NumN3Rec = Bin(N3,Pr2);

if (z > N2-NumN2Rec)
   z = N2-NumN2Rec;

if (EvalBound == 1)
{
     NumN4ToN2 = 0;
     NumN4ToN3 = 0;
     NumN4Rec = y;
}
else if (EvalBound == -1)
{
     NumN4ToN2 = y;
     NumN4ToN3 = 0;
     NumN4Rec = 0;
}
else
{
     NumN4ToN2 = Bin(y,Py2);
     NumN4ToN3 = Bin(y,Py3);
     NumN4Rec = y - NumN4ToN2 - NumN4ToN3;
}
NumV2Dead = Bin(V1,Pd_nat);
NumV3Dead = Bin(V2,Pd_nat);
NumV4Dead = Bin(V3,Pd_nat);
```

```
 NumN1Dead = Bin(N1-NumEffVacc-NumGetVirus+NumN2Rec+NumN3Rec+NumN4Rec+NumBorn+V4-
 NumV4Dead,Pd_nat);

 if (EvalBound == 1)
      NumEffTest = z;
 else if (EvalBound == -1)
 {
      if (z < N1)
           NumEffTest = 0;
      else
           NumEffTest = z - N1;
 }
 else
 {
      Ratio = (double)(N2-NumN2Rec)/(N1+N2);
      NumEffTest = Bin(z,Ratio);
 }
 NumN2ToCancer = Bin(N2-NumN2Rec-NumEffTest,Pc1);
 NumN2Dead = Bin(N2+NumGetVirus+NumN4ToN2-NumEffTest-NumN2Rec-NumN2ToCancer,Pd_nat);
 NumN3ToCancer = Bin(N3-NumN3Rec,Pc2);
 NumN3Dead = Bin(N3-NumN3Rec-NumN3ToCancer+NumEffTest+NumN4ToN3,Pd_nat);
 NumN4CanDead = Bin(N4-NumN4Rec-NumN4ToN2-
 NumN4ToN3+NumN2ToCancer+NumN3ToCancer,Pd_cancer);
 NumN4NatDead = Bin(N4-NumN4Rec-NumN4ToN2-NumN4ToN3+NumN2ToCancer+NumN3ToCancer-
 NumN4CanDead,Pd_nat);
 NumV1Dead = Bin(NumEffVacc,Pd_nat);

 SystemState->N1[1]= N1 - NumEffVacc - NumGetVirus + NumBorn + NumN2Rec + NumN3Rec +
 NumN4Rec - NumN1Dead + V4 - NumV4Dead;
 SystemState->N2[1] = N2 + NumGetVirus + NumN4ToN2 - NumEffTest - NumN2Rec -
 NumN2ToCancer - NumN2Dead;
 SystemState->N3[1] = N3 - NumN3Rec - NumN3ToCancer + NumEffTest + NumN4ToN3;
 SystemState->N4[1] = N4-NumN4Rec-NumN4ToN2-NumN4ToN3+NumN2ToCancer+NumN3ToCancer-
 NumN4CanDead-NumN4NatDead;
 SystemState->N5[1] = N5+NumN4CanDead;
 SystemState->V1[1] = NumEffVacc - NumV1Dead;
 SystemState->V2[1] = V1-NumV2Dead;
 SystemState->V3[1] = V2-NumV3Dead;
 SystemState->V4[1] = V3-NumV4Dead;

 if (SystemState->N1[1] < 0)
      printf("ERROR!! N1 Became negative!! \n\a");
 if (SystemState->N2[1] < 0)
      printf("ERROR!! N2 Became negative!! \n\a");
 if (SystemState->N3[1] < 0)
      printf("ERROR!! N3 Became negative!! \n\a");
 if (SystemState->N4[1] < 0)
      printf("ERROR!! N4 Became negative!! \n\a");

 return;
}
/*****************************************************************************/
long MaxIndex(double * Numbers, long NoOfNumbers)
{
 long i;
 double MaxValue = 0;
 long Index;

 for (i=1; i<=NoOfNumbers; i++)
      if (Numbers[i] > MaxValue)
      {
           MaxValue = Numbers[i];
           Index = i;
      }
 return Index;
}
/*****************************************************************************/
double EvalObjective(struct str_state * CurrState, struct str_solution Solution, long
NumPeriods)
{
 /* Also available is ZeroExpState, with values for N1, N2 etc. */
```

```
  double ObjValue;
  long CurrPeriod;
  double IncVacc;
  double IncN1;
  double IncN2;
  double IncN3;
  double IncN4;
  double IncN5;

  IncVacc = 0;
  IncN1 = 0;
  IncN2 = 0;
  IncN3 = 0;
  IncN4 = 0;
  IncN5 = 0;
  for (CurrPeriod=2; CurrPeriod <= NumPeriods+1; CurrPeriod++)
  {    /* The difference in the first period is going to be zero  */
       IncVacc = (CurrState->V1[CurrPeriod]-ZeroExpState->V1[CurrPeriod])+
            (CurrState->V2[CurrPeriod]-ZeroExpState->V2[CurrPeriod])+
            (CurrState->V3[CurrPeriod]-ZeroExpState->V3[CurrPeriod])+
            (CurrState->V4[CurrPeriod]-ZeroExpState->V4[CurrPeriod]);
       IncN1 = CurrState->N1[CurrPeriod]-ZeroExpState->N1[CurrPeriod];
       IncN2 = CurrState->N2[CurrPeriod]-ZeroExpState->N2[CurrPeriod];
       IncN3 = CurrState->N3[CurrPeriod]-ZeroExpState->N3[CurrPeriod];
       IncN4 = CurrState->N4[CurrPeriod]-ZeroExpState->N4[CurrPeriod];
  }
  IncN5 = CurrState->N5[NumPeriods+1]-ZeroExpState->N5[NumPeriods+1];

  IncVacc = IncVacc / NumPeriods;
  IncN1 = IncN1 / NumPeriods ;
  IncN2 = IncN2 / NumPeriods ;
  IncN3 = IncN3 / NumPeriods ;
  IncN4 = IncN4 / NumPeriods ;
  IncN5 = IncN5 / NumPeriods ;

  ObjValue = C0*IncVacc + C1*IncN1 + C2*IncN2 + C3*IncN3 + C4*IncN4 + C5*IncN5;

/*    double ObjValue;
  long SigmaVacc;
  double VirusKnow;
  long CancerChange;
  long i;

  SigmaVacc=0;
  VirusKnow=0;
  CancerChange=0;
  for (i=1;i<= NumPeriods;i++)
  {
       SigmaVacc+=CurrState->V1[i];
       VirusKnow+= (((double)Solution.z[i])*(CurrState->N2[i]))/(CurrState-
       >N1[i]+CurrState->N2[i]);
  }

  CancerChange =CurrState->N4[NumPeriods+1]-CurrState->N4[1]-CurrState->N5[NumPeriods+1];


  ObjValue = C1*SigmaVacc+C2*VirusKnow+C3*CancerChange;
*/
  return ObjValue;
}
/*****************************************************************************/
long Bin(long n, double p)
{
  // The function is supposed to return a binomial(n,p), but returns a normal(np,np(1-p))
  long Rand;

  if (p>=1)
       return n;
  if (Deterministic)
       Rand = (long)(n*p);
  else
```

```
  {
       Rand = (long) Normal(n*p, sqrt(n*p*(1-p)));
       if (Rand < 0)
            Rand = 0;
       if (Rand > n)
            Rand = n;
  }
  return Rand;
}
/*****************************************************************************************/
double Normal(double Mean, double StdDev)
{
  /*
  Box-Muller Transformation:
       http://mathworld.wolfram.com/Box-MullerTransformation.html
  z_1  =    sqrt(-2lnx_1)*cos(2*pi*x_2)     (1)
  z_2  =    sqrt(-2lnx_1)*sin(2*pi*x_2)     (2)
  */
  /*The function generates two random numbers, normally distributed, using Box-Muller
  transformation. The function first generates standard normals, and then converts
  them to normals with mean Mean and variance Variance. The two random numbers are
  returned as Rand1 and Rand2         */
  double rndno1;
  double rndno2;
  double Rand;

  rndno1 = ran2(&seed);
  rndno2 = ran2(&seed);
  Rand = sqrt(-2*log(rndno1))*cos(2*PI*rndno2);
  Rand = StdDev*Rand + Mean;

  return Rand;
}
/*****************************************************************************************/
```

VITA

Hoda Parvin received her Bachelor of Science degree in Industrial Engineering from Sharif University of Technology, Tehran, Iran, in January 2005. She pursued her Master of Science degree in the Industrial and Systems Engineering Department at the Texas A&M University, during May 2005 to May 2007. Her research interests include Markovian Processes, Applied Probability, Probabilistic Optimization and Queueing Theory.

Hoda Parvin may be reached at Department of Industrial and Systems Engineering, c/o Dr. Natarajan Gautam, Texas A&M University, College Station, TX 77843. Her email address is hoda@tamu.edu.