2011-12-05

Dynamic Appointment Scheduling in Healthcare

McKay N. Heasley
Brigham Young University - Provo

Dynamic Appointment Scheduling for Healthcare

McKay Heasley

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jeffrey Humpherys, Chair
Shane Reese
Robin Roundy

Department of Mathematics

Brigham Young University

December 2011

Abstract

Dynamic Appointment Scheduling for Healthcare

McKay Heasley

Department of Mathematics

Master of Science

In recent years, healthcare management has become fertile ground for the scheduling theory community. In addition to an extensive academic literature on this subject, there has also been a proliferation of healthcare scheduling software companies in the marketplace. Typical scheduling systems use rule-based analytics that give schedulers advisory information from programable heuristics such as the Bailey-Welch rule [1, 2], which recommends overbooking early in the day to fill-in potential no-shows later on. We propose a dynamic programming problem formulation to the scheduling problem that maximizes revenue. We formulate the problem and discuss the effectiveness of 3 different algorithms that solve the problem. We find that the 3rd algorithm, which has smallest amount of nodes in the decision tree, has an upper bound given by the Bell numbers. We then present an alternative problem formulation that includes stochastic appointment lengths and no shows.
Keywords: dynamic programming, appointment scheduling, health care, Bell numbers

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Scheduling theory pertains to the allocation and management of time and operational resources within an organization, spanning myriad business operations from factory design to airline routing. Generally, scheduling algorithms are formulated as optimization problems, with multiple, and often conflicting, objectives such as the efficient utilization of labor, space, equipment, and inventories.

In recent years, healthcare management has become fertile ground for the scheduling theory community. In addition to an extensive academic literature on this subject, there has also been a proliferation of healthcare scheduling software companies in the marketplace. Principle areas of focus include the scheduling of emergency-rooms, laboratory services, hospital beds, nurse and physician staffing, and outpatient scheduling [3, 4]. Our primary interest is outpatient scheduling, but with an eye toward demand management.

Most of the commercial outpatient scheduling systems on the market today seem to focus on calendaring, thereby relegating all of the actual decision making to the scheduler. While there are a few outpatient scheduling systems that do provide decision support in addition to calendaring, these solutions typically use rule-based analytics that give schedulers advisory information from programable heuristics such as the Bailey-Welch rule [1, 2], which recommends overbooking early in the day to fill-in potential no-shows later on.

Although there are a few optimization-based decision algorithms in both the marketplace and in the academic literature that manage outpatient scheduling, the solutions we observed primarily use tactical ad-hoc methods in an attempt to simultaneously satisfy multiple disparate objectives. For example, one system focuses on minimizing a weighted sum of physician idle time, patient mean-waiting time, and the lateness of the doctors to the scheduled appointment [6].

In Chapter 1 we give a review of the literature on appointment scheduling and discuss our unique approach to the scheduling problem. Chapter 2 gives a brief overview of dynamic programming with stochastic and deterministic examples. In Chapter 3 we formulate the appointment scheduling problem in terms of a finite horizon dynamic program. We consider 3 different algorithms that each solve the case when every appointment scheduled is the same length. Each algorithm differs in how the decision tree is organized, which is used as a means of finding the optimal decisions. We find that the 3rd algorithm, which has the fewest number of nodes, has an upper bound which is the same as the number of partitions of n elements. The sequence of numbers is commonly referred to as Bell's numbers. We then consider stochastic appointment lengths in Chapter 4. One of the goals of this paper is to formulate adaptive and profit-maximizing decision support into a scheduling system. In addition, we also propose the use of an outpatient scheduling system as the launching point for demand management and clinic utilization. As the scheduling system is the primary tool used to interface with patients outside of the examination room, we suggest that it also be the data-collection engine that links together all of the diverse information systems used to manage a practice. Along these lines, in Chapter 5 we also consider the use of radio-frequency identification (RFID) technology in the clinic as a way to capture the various stages of a visit. With this additional information, a scheduling system could be made to adapt to the particular clinic, thus providing tailored decision support as conditions change over time. In Chapter 6 we draw conclusions and discuss directions for further research.

1.1 Literature Review

The literature on appointment scheduling can be separated into two categories: static and dynamic. Static scheduling is where all decisions are made before the clinical session begins. This is the most common scheduling system in health care. Consequently most of the literature is for static schedules. In static scheduling it is assumed that the number of patients

to be scheduled has already been determined. This may seem limiting but often general dynamic strategies can be inferred by static solutions. Dynamic scheduling is where future decisions are changed to adapt to the current state of the schedule. For a comprehensive review of both types of problems see Cayirli and Veral [5].

The seminal papers by Bailey and Welch [1, 2] showed that patient waiting time and doctor idle time can be simultaneously reduced by scheduling each appointment to last the length of the average appointment time and by overbooking at the beginning of a clinical session. They suggest moderation in overbooking but recommend it in order to make up for patients that do not show up for their appointments. Their work has been the foundation of both static and dynamic appointment scheduling literature. Since then most of the literature focuses on developing algorithms or implementing heuristics that minimize patient waiting time, doctor idle time, some convex combination of both, or some other time-based measure. They often seek to show that the Bailey-Welch rule is optimal under certain conditions [6, 7]. Occasionally these algorithms will maximize profit where the costs come in the form of missed sales or waiting time [7].

Klassen and Rohleder are two of the lead researchers in dynamic appointment scheduling. In [8] they consider the impact of various overloading rules and rule delays on different patient and server measures. Overloading rules are heuristics that take affect when a schedule is full such as double booking and overtime. Rule delays are conditions that must be met before the overloading rules are implemented. They conclude that the combination of rules that a clinic should apply is highly dependent on the performance measure chosen and the demand rate. They comment that a performance measure ought to be determined by what is most important to the clinic. It is implicit that most clinics are first concerned about revenue. Hence, our research will be aimed at maximizing revenue. The same authors also considered what rules to implement in a multi-period environment [9]. That is if a patient calls in to schedule an appointment and the current scheduling period is full, then they are placed into a later period or an overtime slot. While there research is insightful and practical we will

divert from the usual way of approaching dynamic appointment scheduling by formulating it as an optimization problem.

Chapter 2. Dynamic Programming

Before we state the scheduling problem we give a brief introduction to dynamic programming with some examples. Dynamic programming is a method for solving problems where decisions are made in successive discrete time periods. Decisions are made by maximizing (or minimizing) a profit (cost) function for each period while simultaneously optimizing over future profits. This yields solutions with foresight in that short term profit is sometimes sacrificed for long term profit. This is accomplished by optimizing, in each period, the sum of the profit function and the expected future profits.

The problem formulation consists of two parts; the profit function and a dynamic system. The dynamic system shows how the state being observed changes over time with respect to decisions and random outcomes. We assume that after a decision and random outcome we can observe the evolution of the state. Let x_k, d_k, and w_k represent respectively the state, decision and random variable at period k. The system is described by

$$x_{k+1} = f_k(x_k, d_k, w_k), \qquad k \in \{0, 1, \ldots, T-1\} \tag{2.1}$$

where T is the number of periods.

To provide spaces for our variables to live in we let $x_k \in X_k$ be the state space, $d_k \in \Gamma_k(x_k, w_k)$ be the decision space, and the $w_k \in W_k$ the space of random variables. The set $\Gamma_k(x_k, w_k)$ is dependent on x_k and w_k since in many problems the current state and the realization of w_k determine the admissible decisions.

The profit function $R_k(x_k, d_k, w_k)$ gives the profit for a single period. The sum of the

profit equations

$$R_T(x_T) + \sum_{k=0}^{T-1} R_k(x_k, d_k, w_k) \tag{2.2}$$

gives the profit over all time periods. We omit d_T and w_T from $R_T(x_T)$ meaning that there is no decision or random outcome at the end. $R_T(x_T)$ is sometimes referred to as the terminal profit. Since w_k represents a random variable then it is more fitting to say that the profit over all time periods is the expected value

$$\mathbb{E}\left[R_T(x_T) + \sum_{k=0}^{T-1} R_k(x_k, d_k, w_k)\right].$$

Let x_0 be the initial state and $\Pi = \{(d_0, d_1, \ldots, d_{T-1}) | d_k \in \Gamma(x_k, w_k)\}$ for $k = 1, \ldots, T-1$ be the set of sequences of decisions. Let

$$V_\pi(x_0) = \mathbb{E}\left[R_T(x_T) + \sum_{k=0}^{T-1} R_k(x_k, d_k, w_k)\right]. \tag{2.3}$$

The dynamic programming algorithm, which we will describe in section 2.1, finds $\pi^\star \in \Pi$ so that

$$V_{\pi^\star}(x_0) = \max_{\pi \in \Pi} V_\pi(x_0) \tag{2.4}$$

2.1 Principle of Optimality

Theorem 2.1 (Principle of optimality)**.** *Let x_0 be the initial state and $\pi^\star = (d_0^\star, d_1^\star, \ldots, d_{T-1}^\star)$ be a solution given by (2.4). Suppose that while using $\pi^\star$ we arrive at state x_i then $\pi_i^\star = (d_i^\star, d_{i+1}^\star, \ldots, d_{T-1}^\star)$ is a minimizing policy to the truncated subproblem*

$$\mathbb{E}\left[R_T(x_T) + \sum_{k=i}^{T-1} R_k(x_k, d_k, w_k)\right] \tag{2.5}$$

The principle of optimality implies that if we solve the subproblem (2.5) we will obtain a

portion of the solution given by (2.4). This is the foundation of the dynamic programming algorithm. We break up the problem into subproblems and solve each piece. Since each individual solution is a piece of the solution to a larger problem then a recursive problem formulation of (2.5) would be useful. Let

$$V_k(x_k) = \max_{d_k,\ldots,d_{T-1}} \mathbb{E}\left[R_T(x_T) + \sum_{i=k}^{T-1} R_i(x_i, d_i, w_i)\right]. \tag{2.6}$$

We can restate this recursively as

$$V_k(x_k) = \max_{d_k \in \Gamma(x_k, w_k)} \mathbb{E}\left[R_k(x_k, d_k, w_k) + V_{k+1}(f_k(x_k, d_k, w_k))\right]. \tag{2.7}$$

where

$$V_T(x_T) = R_T(x_T). \tag{2.8}$$

Equation (2.7) is commonly referred to as the value equation.

2.2 Dynamic Programming Algorithm

Recursively solving equations (2.7) and (2.8) constitutes the dynamic programming algorithm. There are a several ways to solve this recursion. The method that you choose depends somewhat on preference but mostly on the problem specifics. We present some common methods in this section. Regardless of the method the basic strategy is the same. We consider the possible values for $V_T(x_T)$ and solve $V_{T-1}(x_{T-1})$ for each of the possibilities. We then take another step back to $V_{T-2}(x_{T-2})$ and solve it using the values we found from solving $V_{T-1}(x_{T-1})$. We continue sweeping back in time until we have solved $V_0(x_0)$ where x_0 is the initial state, which is given.

2.2.1 Deterministic Case: Shortest Path Problem. Consider the deterministic problem where the state equation is given by $x_k = f(x_{k-1}, d_{k-1})$. We also suppose that the state space X_k is finite as is the space of admissible decisions $\Gamma(x_k)$. The shortest path problem is one where the initial and terminal conditions x_0 and x_T are given. With each decision there is a cost $R_k(x_k, d_k)$ incurred. The problem is to find a policy $\pi^\star = \{d_0^\star, \ldots, d_{T-1}^\star\}$ such that

$$V_{\pi^\star}(x_0) = \min_{\pi \in \Pi} \left\{ R(x_T) + \sum_{k=0}^{T-1} R(x_k, d_k) \right\}.$$

If $R_k(x_k, d_k)$ is viewed as a distance then we are essentially finding the shortest path from x_0 to x_T.

Example. Consider the problem

$$\min \sum_{k=0}^{3} x_k^2 + u_k^2$$

subject to

$$x_{k+1} = x_k + d_k, \quad k = 0, 1, 2, 3 \tag{2.9}$$

$$x_0 = 0, \ x_4 = 8, \ d_k \in \{0, 1, 2, 3\} \tag{2.10}$$

To put this in the context of dynamic programming we will identify the dynamic system and the value function. We can see that the dynamic system is given by $f(x_k, d_k) = x_k + d_k$ and the cost $R_k(x_k, d_k) = x_k^2 + d_k^2$. Since it is not specified otherwise we let the terminal cost

$R_4(x_4) = 0$. Given (2.9) and (2.10) we can deduce that our state spaces are

$$\begin{aligned} X_0 &= \{0\} \\ X_1 &= \{0,1,2,3\} \\ X_2 &= \{2,3,4,5,6\} \\ X_3 &= \{5,6,7,8\} \\ X_4 &= \{8\} \end{aligned}$$

The feasible decisions are $\Gamma(x_k) = \{d_k \in \{0,1,2,3\} | f(x_k, d_k) \in X_{k+1}\}$.

We will utilize the principle of optimality by solving the truncated subproblem given by the value function

$$V_k(x_k) = \min_{d_k \in \Gamma(x_k)} \{R_k(x_k, d_k) + V_{k+1}(f(x_k, d_k))\}.$$

We start the dynamic programming algorithm by looking at the second to last step $k = 3$. The value equation is

$$V_3(x_3) = \min_{d_3 \in \Gamma(x_3)} \left\{x_3^2 + d_3^2\right\}$$

since $V_4(x_3 + d_3) = R_4(x_4) = 0$. We must find the best decision for each element in X_3. Since $x_4 = 8$ then if $x_3 = 5$ then $d_3 = 3$ because $x_3 + d_3 = 8$. Likewise if $x_3 = 6$ then $d_3 = 2$ and so on. In this case each $x_3 \in X_3$ has only one possible control d_3. We can see that in each case the cost of moving from x_3 to x_4 will be

$$\begin{aligned} V_3(5) &= 5^2 + 3^2 = 34 \Rightarrow u_3 = 3 \\ V_3(6) &= 6^2 + 2^2 = 40 \Rightarrow u_3 = 2 \\ V_3(7) &= 7^2 + 1^2 = 50 \Rightarrow u_3 = 1 \\ V_3(8) &= 8^2 + 0^2 = 64 \Rightarrow u_3 = 0. \end{aligned}$$

We now go back another time step to x_2. We must find the minimum cost and the best control $d_2 \in \Gamma(x_2)$ for each $x_2 \in X_2$.

$$\begin{aligned}
V_2(2) &= \min_{d_3 \in \{3\}} \left\{2^2 + d_3^2 + V_3(2+d_3)\right\} \\
&= 2^2 + 3^2 + 34 = 47 \Rightarrow u_2 = 3 \\
V_2(3) &= \min_{d_3 \in \{2,3\}} \left\{3^2 + d_3^2 + V_3(3+d_3)\right\} \\
&= \min\{3^2 + 2^2 + 34, 3^2 + 3^2 + 40\} = 47 \Rightarrow u_2 = 2 \\
V_2(4) &= \min\{4^2 + 1^2 + 34, 4^2 + 2^2 + 40, 4^2 + 3^2 + 50\} = 51 \Rightarrow u_2 = 1 \\
V_2(5) &= \min\{5^2 + 0^2 + 34, 5^2 + 1^2 + 40, 5^2 + 2^2 + 50, 5^2 + 3^2 + 64\} = 59 \Rightarrow u_2 = 0 \\
V_2(6) &= \min\{6^2 + 0^2 + 40, 6^2 + 1^2 + 50, 6^2 + 2^2 + 64\} = 76 \Rightarrow u_2 = 0
\end{aligned}$$

Again we go back another time step and find the best $d_1 \in \Gamma(x_1)$ for each $x_1 \in X_1$.

$$\begin{aligned}
V_1(0) &= \min\{0^2 + 2^2 + 47, 0^2 + 3^2 + 47\} = 51 \Rightarrow u_1 = 2 \\
V_1(1) &= \min\{1^2 + 1^2 + 47, 1^2 + 2^2 + 47, 1^2 + 3^2 + 51\} = 49 \Rightarrow u_1 = 1 \\
V_1(2) &= \min\{2^2 + 0^2 + 47, 2^2 + 1^2 + 47, 2^2 + 2^2 + 51, 2^2 + 3^2 + 59\} = 51 \Rightarrow u_1 = 0 \\
V_1(3) &= \min\{3^2 + 0^2 + 47, 3^2 + 1^2 + 51, 3^2 + 2^2 + 59, 3^2 + 3^2 + 76\} = 56 \Rightarrow u_1 = 0
\end{aligned}$$

And now we find the minimum cost and best control from $x_0 = 0$.

$$V_0(0) = \min\{0^2 + 0^2 + 51, 0^2 + 1^2 + 49, 0^2 + 2^2 + 51, 0^2 + 3^2 + 56\} = 50 \Rightarrow u_0 = 1$$

We can now see that the minimum cost is $V_{\pi^\star} = 50$ where $\pi^\star = (1, 1, 3, 3)$.

2.2.2 Stochastic Case: Inventory Control. There are many different types of stochastic dynamic programming problems. They vary in which parts of the problem the stochastic variable w_k affects. It can affect the feasible decisions $\Gamma(x_k, w_k)$, or the dynamic system $f_k(x_k, d_k, w_k)$, or the cost function $R(x_k, d_k, w_k)$, or any combination of these. There are also

problems where either the decision is made after or before the random outcome is realized.

In the inventory control problem we suppose a retailer wants to maximize his profit over a finite horizon. We let x_k be the inventory state, d_k be the decision of how much to order, and w_k be the demand during period k. We assume that the distribution of w_k is given. Orders placed in time k are available in time to meet the demand in time $k+1$. The demand w_k is realized at the beginning of the period and the decision d_k is made after the realization. We do not allow backlogging so the inventory state is given by

$$x_{k+1} = x_k + d_k - \min\{x_k, w_k\}.$$

We assume there is a cost of C per item ordered and a holding cost H for every item in stock at the beginning of the period. R is the amount earned per item sold and S is the salvage value of any remaining inventory at the ending time T. When the demand for the period occurs the retailer can only sell as much is on hand so the revenue earned in a period is $R\min\{x_k, w_k\}$. The profit for period k is

$$R(x_k, d_k, w_k) = -Cd_k - Hx_k + R\min\{x_k, w_k\}.$$

Since w_k is realized before we make the decision the value function with the boundary condition is given by

$$\begin{aligned} V(x_k, w_k) &= \max_{d_k \in \Gamma_k(x_k, w_k)} \{R(x_k, d_k, w_k) + \mathbb{E}[V(x_{k+1}, w_{k+1})]\} \qquad (2.11) \\ V(x_T) &= Sx_T \end{aligned}$$

we could use the notation $V(x_k, w_k = w)$ to make it clear that the decision d_k is made after the realization of w_k. We will use this notation in the paper where convenient.

To demonstrate how to solve this problem we choose our parameters to be $C = 2$, $R = 4$,

$H = 1$, $S = 1$, and $T = 2$. Our permissible spaces are $X_i = \{0, 1, 2\}$ and $W_i = \{0, 1, 2\}$. The set $\Gamma(x_k, w_k) = \{0, 1\}$ if $x_k - w_k + 1 \leq 2$ otherwise it is $\{0\}$. We need a distribution for w_i so we let $\mathbb{P}(w_i = j) = 2/5$ for $j \in \{1, 2\}$ and $\mathbb{P}(w_i = 0) = 1/5$ meaning that demand for each period cannot exceed 2.

In our last example we solved the dynamic program by looking at the second to last time period and finding the best decision of each possible state. In this example we present a method that is useful when solving the problem with a computer. We will first organize all the possible decisions into a tree. A tree is a graphical way to represent the possible ways to get from any state to any other. In terms of programming it is a directed data structure in which each node (except the root) has exactly one parent and can have any number of children. The tree of all possible decisions is represented by Figure 2.1.

Now that we have a way to quickly reference the possible decisions we can move backwards through tree eliminating the suboptimal decisions according to equation (2.11). Just as before, we start with the second to last time and move backwards in time from there. The solution to this problem will appear different then the solution to the last problem because we must consider what the best decision is for any of possible outcomes for w_0 and w_1. Throughout this paper I will refer to this step in the algorithm as "trimming the tree" since the solution is derived from the tree of all possible decisions. Since $S = 1$ then we have $V(x_2) = x_2$. Taking another step back in time it can be shown that

$$\begin{aligned}
\mathbb{E}[V(w_1, x_1 = 2)] &= 18/5 \\
\mathbb{E}[V(w_1, x_1 = 1)] &= 12/5 \\
\mathbb{E}[V(w_1, x_1 = 0)] &= 0.
\end{aligned}$$

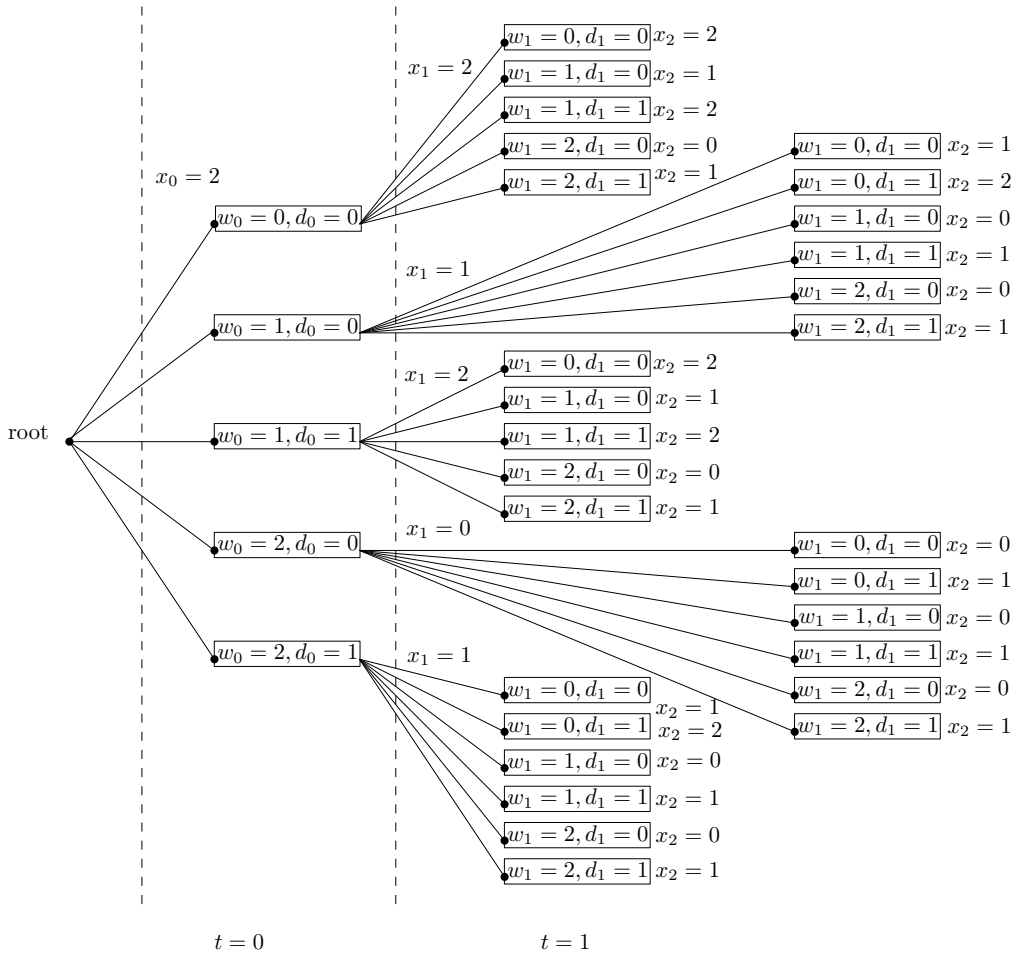

Figure 2.1: This is a tree graph of all possible decisions for in the inventory control problem. The parameters that generate this graph are $R = 4$, $C = 2$, $H = 1$, $S = 1$ and $T = 2$. The terms $x_i = j$ that are not in boxes represent what happens to the state after the decision and random event from the previous node.

Another step back in time gives us

$$\begin{aligned} V(w_0 = 2, x_0 = 2) &= \max\{6 + 0, 4 + 12/5\} \\ V(w_0 = 1, x_0 = 2) &= \max\{2 + 12/5, 0 + 18/5\} = 22/5 \\ V(w_0 = 0, x_0 = 2) &= -2 + 18/5 = 8/5. \end{aligned}$$

As we identify the optimal decisions in the above calculations we eliminate the suboptimal nodes in the tree. Figure 2.2 shows what the tree looks like after the trim. Notice that the

tree keeps track of which decision to make under any possible random outcome.

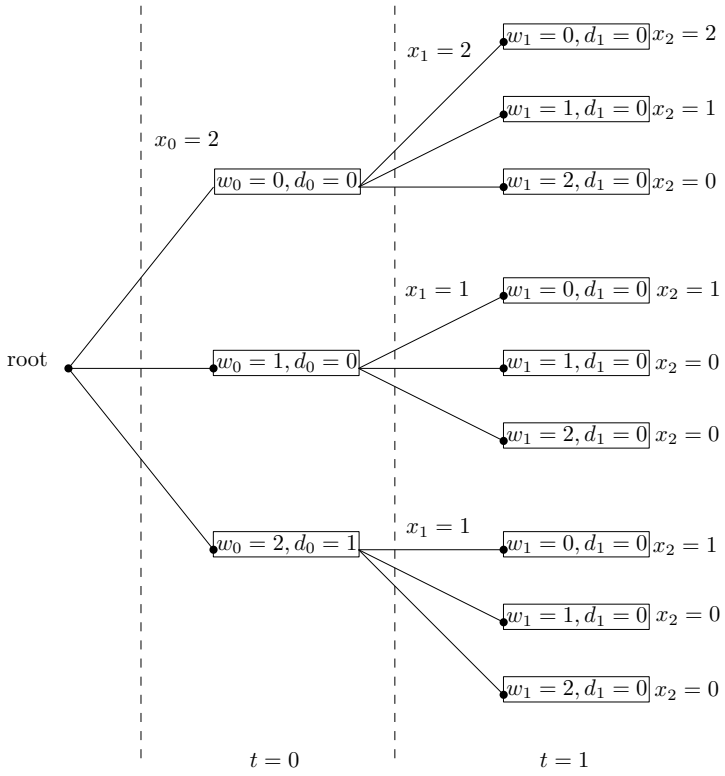

Figure 2.2: This is a tree graph of the optimal decisions in the inventory control problem after the tree in Figure 2.1 has been trimmed.

Chapter 3. The Scheduling Problem

In this chapter we present the appointment scheduling problem. We want to answer the following question: When do I schedule an appointment upon request so that I can maximize my revenue? We state the assumptions and problem formulation. We then present 3 different algorithms for solving the problem with some simplifying assumptions.

3.1 Problem Formulation

In this section we state the problem of interest in terms of a dynamic program. Our object is to determine the profit maximizing appointment time for a patient upon request. We first solve the problem restricting our attention to a single day.

We consider the dynamic scheduling problem where there are T appointment slots in a day and M channels of demand. Let w_t be a random variable representing the demand at time t and be defined by

$$w_t = \begin{cases} 0 & \text{w/ prob } p_0 \\ 1 & \text{w/ prob } p_1 \\ & \vdots \\ M & \text{w/ prob } p_M \end{cases} \tag{3.1}$$

where $\sum p_i = 1$. If no demand occurs then $w_t = 0$ and if demand from channel 1 occurs then $w_t = 1$ and so on. We seek to formulate the problem in terms of a dynamic program. To do this we must define the state variable x_t, decision variable d_t, and revenue function $R(\cdot)$. Our state x_t represents the schedule at time t which we will view as a vector of length T where $x_{t,i}$ is the ith appointment slot. The vector entry $x_{i,t} \in \mathbb{N}$ is the number of people scheduled for appointment slot i. The initial schedule x_0 will represent the empty schedule. We let d_t be the start time of the appointment. The revenue function $R(x_t, w_t, d_t)$ is the revenue of the decision d_t. The ordering of events is as follows: the state x_t is observed, the random event w_t is realized, and the decision d_t is made.

Let $d_t \in \Gamma(x_t, w_t)$, which is to say that d_t must lie in the feasible set of decisions. The set Γ can be governed by a number of heuristics such as a service capacity or willingness to wait. In this problem we will assume a service capacity C so that $x_{t,i} \leq C$ for all i. We let $x_{t+1} = f(x_t, d_t)$ where f tells us how the decision affects the schedule. This leads to the

classic stochastic dynamic program formulation where the value function is given by

$$V(x_t, w_t) = \max_{d_t \in \Gamma(x_t, w_t)} \{\mathbb{E}[R(x_t, w_t, d_t) + V(x_{t+1}, w_{t+1})]\}. \tag{3.2}$$

3.2 Simplified Scheduling problem

In our first solution to this problem we make some simplifying assumptions. We assume that every appointment is m blocks long. We consider the case where there are only two channels of demand; walk-ins and call-ins. When a walk-in or call-in occurs then $w_t = 1$ and $w_t = 2$ respectively. We determine the feasible set for walk-ins by making the assumption that if a walk-in must wait more than ω appointment blocks then he will leave without service. Likewise, a call-in must wait at least κ appointment slots before he can be scheduled. This yields the following expressions for Γ,

$$\begin{aligned}
\Gamma(x_t, 0) &= \{\eta\} && (3.3)\\
\Gamma(x_t, 1) &= \{d_t | t+1 \le d_t \le t+\omega, d_t \le T-m+1 \\
&\qquad \text{and } x_{t,j} < C \text{ for all } d_t \le j \le d_t + m - 1\} \cup \{\eta\} && (3.4)\\
\Gamma(x_t, 2) &= \{d_t | t+\kappa+1 \le d_t, d_t \le T-m+1 \\
&\qquad \text{and } x_{t,j} < C \text{ for all } d_t \le j \le d_t + m - 1\} \cup \{\eta\}. && (3.5)
\end{aligned}$$

where η represents the null decision, or the decision to not schedule an appointment. There are a few things to notice about Γ. A walk-in cannot be scheduled during the same slot they arrive since $t + 1 \le d_t$. The condition $d_t \le T - m + 1$ is there because that is the latest an appointment can begin.

We will assume that the revenue for every appointment is r and that all scheduled

appointments show up at the scheduled time. So

$$R(d_t) = \begin{cases} 0 & \text{if } d_t = \eta \\ r & \text{otherwise} \end{cases} \tag{3.6}$$

and

$$V(x_t, w_t) = \max_{d_t \in \Gamma(x_t, w_t)} \{R(d_t) + \mathbb{E}[V(x_{t+1}, w_{t+1})]\}. \tag{3.7}$$

In this chapter we present three algorithms that solve (3.7). Each of the successive algorithms is a slight modification of the previous one with the intent to reduce the computation time. We present all three to build intuition for the structure of tree.

3.3 Algorithm 1

We solved this problem by building a tree data structure similar to that in the inventory control example from section 2.2.2. Every node in the tree represents a different possible decision corresponding to a different random outcome.

There are two steps to this algorithm; building the tree and trimming the tree. We build the tree by starting with the root node representing the empty schedule x_0 and generating all of the children of this node which are the decisions made at $t = 0$. The children are the decisions in the feasible sets given by equations (3.3), (3.4), and (3.5). We must start at time $t = 0$ otherwise the first appointment slot at $t = 1$ will never get filled. We then create all the nodes at $t = 1$ by generating all of the children of the nodes at $t = 0$. We continue to build the tree until the last time a decision is made, which is $t = T - m$. This tree we have built represents all possible decisions that can be made at any time and any state. Figure 3.1 is a diagram showing the structure of a basic tree before the trim.

After the tree has been built we then trim the tree keeping only the nodes that are revenue maximizing according to (3.7). We start by finding all nodes at $t = T - m - 1$, which is

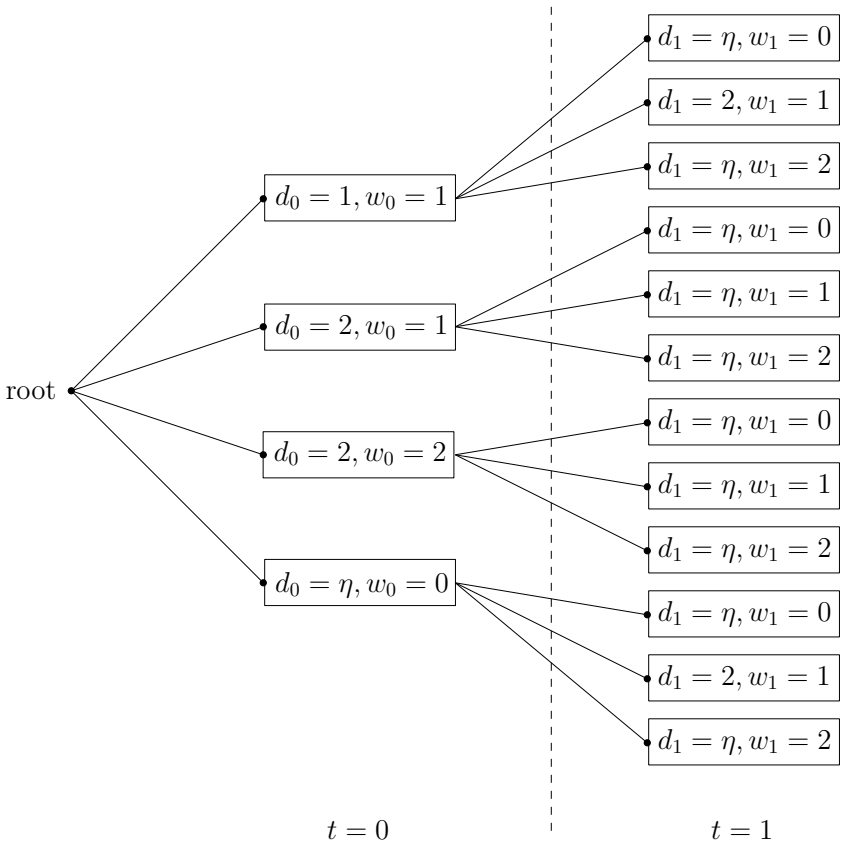

Figure 3.1: This Figure shows the structure of the tree before the trim under Algorithm 1. The parameters for this tree are $T = 2$, $m = 1$, $M = 2$, $C = 1$, $\omega = 2$, and $\kappa = 1$.

the second to last time a decision is made in the schedule. Looping through each node we consider each of the 3 random outcomes and pick the child decision in time $t = T - m$ that gives us the most revenue. We take another step back to $t = T - m - 2$ and look at all of the nodes at that time. Again for each node and for each random outcome we pick the child node that maximizes expected future revenue under each random outcome. In other words we keep the child node that has the decision d_t that maximizes (3.7). We delete the remaining nodes. We repeat this process sweeping back all the way until the root node. This leaves each node in the tree with exactly 3 children as can be seen in Figure 3.2. These three children correspond to the best decision for each random outcome.

This approach to solving (3.7) is exhaustive and inefficient. Even after the tree has been

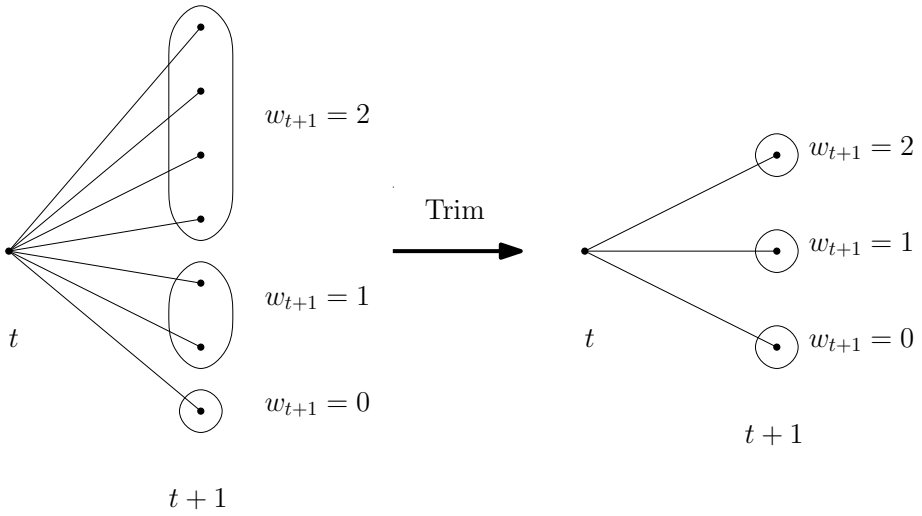

Figure 3.2: Algorithm 1: This Figure looks at the changes that occur in one node and its children when the tree is trimmed by maximizing expected profit.

trimmed it has

$$\sum_{i=0}^{T-m+1} 3^i = \frac{3^{T-m+2} - 1}{2} \tag{3.8}$$

nodes. The number of nodes in the tree after the trim is exponential. The next 2 algorithms provide approaches that are more numerically friendly.

3.4 Algorithm 2

Our second algorithm cuts down on the size of the tree by taking advantage of a particular redundancy. We will explain how this algorithm works. In the first algorithm, depending on the values of ω and κ there may be some overlap in the sets $\Gamma(x_t, 1)$ and $\Gamma(x_t, 2)$. This means that a decision that is valid for a walk-in may also be valid for a call-in. Likewise, in each instance of w_t we include the null decision. This creates some redundancy in our tree. In the second algorithm we take advantage of this structure and merge two nodes that correspond to the same decision into the same node. This idea is illustrated by a diagram in Figure 3.3. Figure 3.4 shows the same tree from Figure 3.1 looks under the model of Algorithm 2.

This reduces the size of the tree dramatically. After the trim in the second algorithm

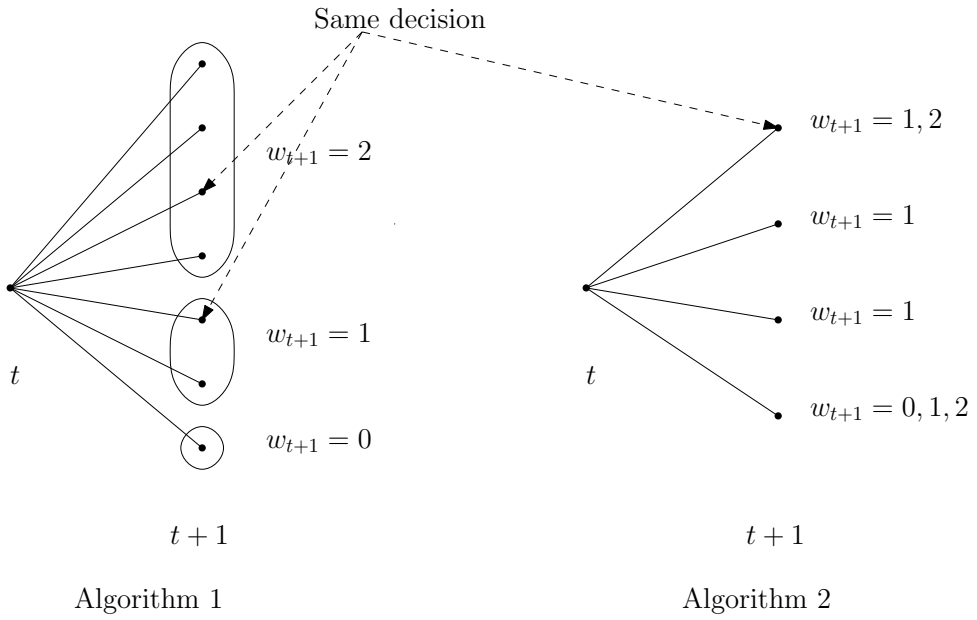

Figure 3.3: This Figure shows how the first algorithm compares to the second. The last node in the second algorithm corresponds to the null decision since it is the only decision that is valid for every random event. This Figure describes the tree before the trim in both algorithms.

the number given by (3.8) is now a very loose upper bound. The code for this algorithm is included in Appendix A.

3.5 Algorithm 3

Before the trim in Algorithm 2 each node has exactly one child that is a null node. The idea behind Algorithm 3 is that we do not include the null nodes but just assume that each node has one as a possible decision. Figure 3.5 is an example of a tree with no null decisions.

This algorithm complicates the tree structure but reduces the number of nodes. In the past two algorithms, every child of a node was at the same time. Notice under this structure that a node can have children at several different times. This complicates the computations necessary to find the expected future profit at a node. In the first two algorithms we summed over profits of all the children of a node multiplied by its probability to determine the expected future profit. We can no longer do this since the children of a node can be at a

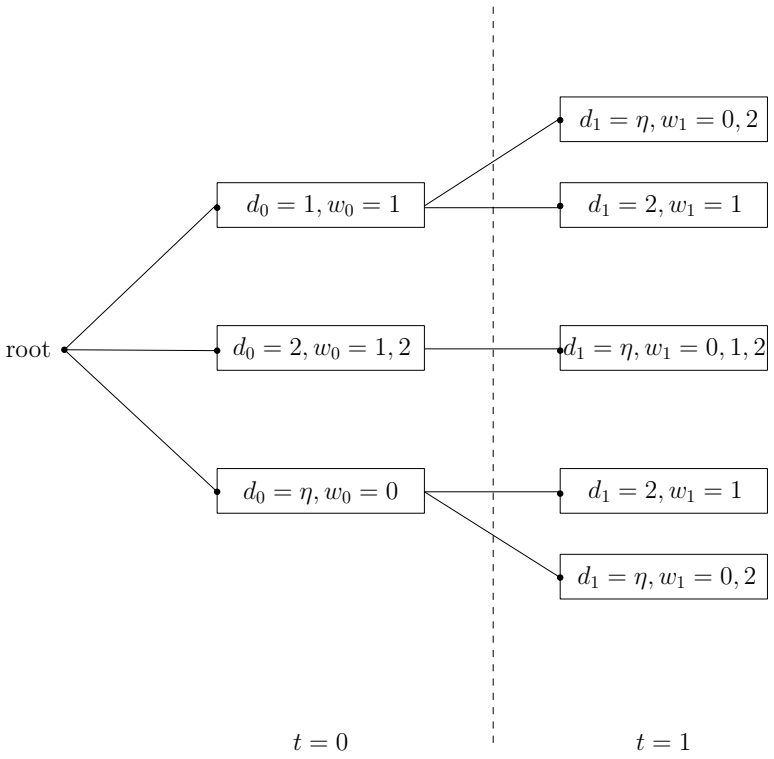

Figure 3.4: This Figure shows the structure of the tree before the trim under Algorithm 2. The parameters for this tree are $T = 2$, $m = 1$, $M = 2$, $C = 1$, $\omega = 2$, and $\kappa = 1$. Compare this tree with Figure 3.1

number of different times.

To deal with this complicated tree structure we employ some new notation. Suppose that through a series of decisions we arrive at a node n with state x_t. We will denote this state as $x^{(n)}$. We seek an expression for $\mathbb{E}[V(x^{(n)}, w_{t^{(n)}+1})]$ in terms of the expected future values

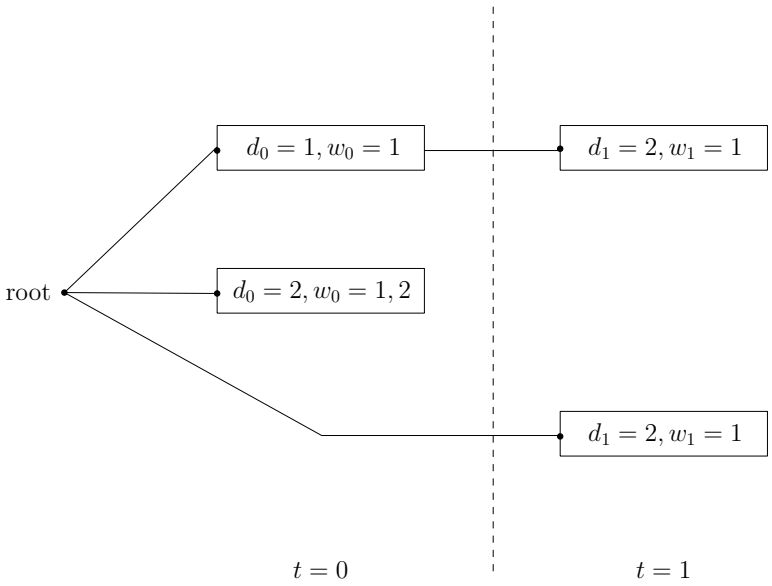

Figure 3.5: This Figure show the structure of the tree before the trim under Algorithm 3. The parameters for this tree are the same as Figures 3.1 and 3.4 to make them easy to compare.

of its children nodes under this tree structure. We define the following.

$$
\begin{aligned}
C_n &= \text{children of node } n \\
C_{n,t} &= \text{children of node } n \text{ at time } t \\
&= \{m \in C_n : t^{(m)} = t\} \\
C_{n,t,w} &= \text{children of node } n \text{ at time } t \text{ valid for random outcome } w \\
&= \{m \in C_{n,t} : w \in w^{(m)}\} \\
t^{(n)} &= \text{time of node } n \\
w^{(n)} &= \text{random events corresponding to node } n \\
d^{(n)} &= \text{decision for node } n \text{ which is the starting time}
\end{aligned}
$$

To help with this onslaught of notation we use superscripts $\cdot^{(n)}$ to denote nodal attributes of n and subscripts $\cdot_n$ do refer to other nodes in the tree with a particular relationship to n.

We also let

$$\mathbb{P}(n \mid t) = \text{probability that we arrive at node } n \text{ given that we are at time } t.$$

For instance, if $w^{(n)} = \{1, 2\}$ then $\mathbb{P}(n \mid t^{(n)} - 1) = p_1 + p_2$ where p_i are defined by (3.1). In general,

$$\mathbb{P}(n \mid t^{(n)} - 1) = \sum_{i \in w^{(n)}} p_i. \tag{3.9}$$

In Algorithm 3 we will be concerned with computing the probabilities of getting from one node to a child node that is not at the next time. In such a case there are null decisions that occur between the two nodes. For a node m that is a child of node n there is $t^{(m)} - t^{(n)} - 1$ null decisions made between nodes n and m. In a tree without null nodes, we need a way to compute the probability of having a null decision. We do this by finding the probability of its compliment. Assuming the independence of successive events, the probabilty of getting to node m from $t^{(n)}$ where n is the parent of m is

$$\mathbb{P}(m \mid t^{(n)}) = \sum_{i \in w^{(n)}} p_i \prod_{s=1}^{t^{(m)} - t^{(n)} - 1} (1 - \sum_{\substack{j \in w^{(k)} \ s.t. \\ k \in C_{n, t^{(n)} + s}}} p_j). \tag{3.10}$$

Equation (3.10) only works if you have trimmed all future nodes.

We can rewrite the value function given by (3.2) with this new notation. We do not want to confuse the notation $w^{(n)}$ and w_t. The first is a set a numbers for which if $i \in w^{(n)}$ and $w_t = i$ then $d^{(n)}$ is a valid decision. Our new value function is

$$V(x^{(n)}, w_{t^{(n)}+1} = w) = \max_{m \in C_{n, t^{(n)}+1, w} \cup \{\eta\}} \left\{ R(d^m) + \mathbb{E}[V(x^{(m)}, w_{t^{(m)}+1})] \right\}$$

where

$$\mathbb{E}[V(x^{(m)}, w_{t^{(m)}})] = \begin{cases} \sum_{l \in C_m} \mathbb{P}(l \mid t^{(m)})(R(d^{(l)}) + \mathbb{E}[V(x^{(l)}, w_{t^{(l)}+1})]) & \text{if } m \in C_n \\ \sum_{\substack{l \in C_n \\ t^{(l)} < t^{(n)}+1}} \mathbb{P}(l \mid t^{(n)} + 1)(R(d^{(l)}) + \mathbb{E}[V(x^{(l)}, w_{t^{(l)}+1})]) & \text{if } m = \eta \end{cases} \tag{3.11}$$

Notice that in the case that $m = \eta$ that we consider the children of node n. This is because η is not a real node so we can't look at it's children. (3.10) can be used to solve (3.11). The code for this algorithm is included in appendix B.

3.5.1 Tree Size. In an effort to understand the efficiency of Algorithm 3 we look at the size of the Tree before the trim. We let N_T be the number of nodes in a tree for a schedule of T appointment slots. We let $n_{T,i}$ be the number of nodes corresponding to decisions made at time i for a schedule with T appointment slots. We have the relationship

$$N_T = 1 + \sum_{t=0}^{T-1} n_{T,t} \tag{3.12}$$

where the 1 in the above sum comes from the root node.

According to our model, the parameters that affect N_T are κ, ω, C, m, and T. Obtaining a closed form expression for N_T in terms of each of these parameters would not only be difficult but also not very useful. We instead seek an upper bound for the case where $C = 1$. We need parameters for κ, ω, and m that will maximize N_T. Letting $m = 1$ makes N_T the largest it can be with regards to m. Also if $\kappa \leq \omega$ then there is overlap in the feasible appointments for walk-ins and call-ins. This condition maximizes N_T with respect to κ and ω since the cardinality of the union of (3.4) and (3.5) is largest.

Under these conditions, we have that the number of nodes in the tree for varying values of T is given in Table 3.1. By observation, we see that there is a recursive relationship with

$n_{T,t}$ given by

$$n_{T,t} = (T - t)\left(1 + \sum_{i=0}^{t-1} n_{T-1,i}\right) \tag{3.13}$$

Programming this recursion further yields numbers to the sequence $\{N_T\}$. The sequence continues as given in Table 3.2.

T	N_T	$n_{T,0}$	$n_{T,1}$	$n_{T,2}$	$n_{T,3}$	$n_{T,4}$	$n_{T,5}$	$n_{T,6}$	$n_{T,7}$
1	2	1							
2	5	2	2						
3	15	3	6	5					
4	52	4	12	20	15				
5	203	5	20	51	74	52			
6	877	6	30	104	231	302	203		
7	4140	7	42	185	564	1116	1348	877	
8	21147	8	56	300	1175	3196	5745	6526	4140

Table 3.1: This Table shows the number of nodes in a tree N_T for varying values of T.

1	2	11	4213597
2	5	12	27644437
3	15	13	190899322
4	52	14	1382958545
5	203	15	10480142147
6	877	16	82864869804
7	4140	17	682076806159
8	21147	18	5832742205057
9	115975	19	51724158235372
10	678570	20	474869816156751

Table 3.2: Gives the number of nodes in each tree according the recursion given by 3.13. The parameters are $m = 1$, $C = 1$, $M = 2$, and $\kappa \leq \omega$.

Coincidentally, this sequence is a the same as the sequence of Bell's numbers [10], which is sequence $A000110$ in the OEIS [11]. We will explore this connection in more detail in section 3.5.2.

3.5.2 Bell's Numbers and an Upper Bound. The Bell numbers give the number of ways to partition a set of T elements, which we will denote B_T. The connection we have is that $B_T = N_{T-1}$.

Let us consider how to count the number of partitions in a set of T elements so that we might understand the connection to our problem. Table 3.3 shows all of the partitions of a sets of $T = 2, 3$ and 4 elements.

T/k	1	2	3	4	No singletons
2	$\{(1),(2)\}$				$\{(1,2)\}$
3	$\{(1),(2,3)\}$ $\{(1),(2),(3)\}$	$\{(2),(1,3)\}$	$\{(3),(1,2)\}$		$\{(1,2,3)\}$
4	$\{(1),(2,3,4)\}$ $\{(1),(2,3),(4)\}$ $\{(1),(3,4),(2)\}$ $\{(1),(2,4),(3)\}$ $\{(1),(2),(3),(4)\}$	$\{(2),(1,3,4)\}$ $\{(2),(1,3),(4)\}$ $\{2),(1,4),(3)\}$	$\{(3),(1,2,4)\}$ $\{(3),(1,2),(4)\}$	$\{(4),(1,2,3)\}$	$\{(1,2,3,4)\}$ $\{(1,2),(3,4)\}$ $\{(1,3),(2,4)\}$ $\{(1,4),(2,3)\}$

Table 3.3: This Table shows all the partitions of $T = 2, 3$ and 4 elements. We can see that the number of partitions equals the number of nodes given in Table 3.1 for $T = 1, 2$ and 3. The first 4 columns represent the partitions with singleton sets in them. The first column has all of the singletons (1), the second (2) and so on without double counting. The 5th column has the partitions without any singletons. We can also compare the number of partitions against the Figures 3.5 and 3.6.

Lets focus our attention on the tree in Figure 3.6. At $t = 0$ the root has 3 children. Each of these children form 3 branches. The number of nodes in each branch including the nodes at $t = 0$ are 5,3, and 2 starting from the top and moving down. Now refer to Tables 3.3 and 3.4 and you will see that the number of singleton sets for B_4 for 1,2, and 3 are also 5,3, and 2. The same thing can be seen by comparing the singleton sets of B_3 and the tree in Figure 3.4. The remainder of the nodes including the root node and the nodes that come from making a null decision at the beginning account for the last column of singleton column and the "No Singletons" column.

Why does this pattern arise? When the node corresponding to $d_0 = 1$ is created we are essentially doing the same thing as fixing the singleton (1) and finding all sets that go with

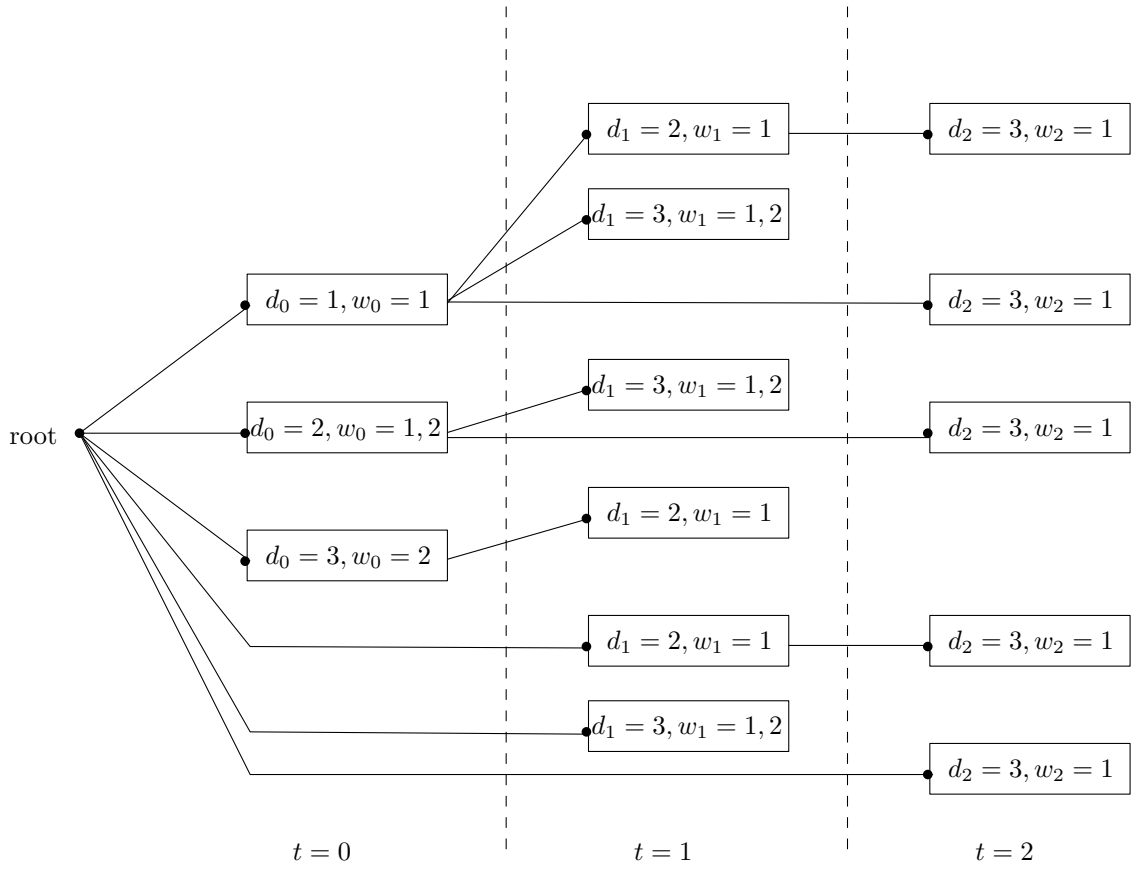

Figure 3.6: This tree from Algorithm 3 has the parameters $T = 3$, $m = 1$, $M = 2$, $C = 1$, $\omega = 2$, and $\kappa = 1$.

it. The same goes with $d_0 = 2$, $d_0 = 3$ and so on. In Table 3.4 there are a few interesting properties we should note. Let $PS_{T,k}$ denote the singleton counts and PN_T the no singleton counts in Table 3.4. Notice that $PS_{T,1} = B_{T-1}$. In terms of decision trees such as 3.5 and 3.6 this happens because when the decision $d_0 = 1$ the problem is then reduced to a schedule with one less appointment slot. In other words, this makes the branch of $d_0 = 1$ from 3.6 the same as the entire tree in 3.5. Also notice that $PS_{T,T} + PN_T = B_{T-1}$. Referring to Figures 3.5 and 3.6, when a null decision is made at $t = 0$ the problem is reduced to a schedule of one less appointment slot.

These things suggest there is a bijection between the partitions of a set of T elements and the number of nodes in a tree from a schedule of length $T - 1$ (with the parameters we have specified). This is useful in finding an upper bound for the number of nodes before the

B_T	T/k	1	2	3	4	5	6	No Singletons
2	2	1						1
5	3	2	1	1				1
15	4	5	3	2	1			4
52	5	15	10	7	5	4		11
203	6	52	37	27	20	15	11	41

Table 3.4: The first column of this Table shows the Bell numbers. The remaining columns show the number of partitions with singletons of 1,2, and so on. The last column show the number of partitions without any singletons. This Table is to be compared with Table 3.3.

trim. Berend and Tassa [12] proved that an upper bound for this sequence is given by

$$N_{T-1} < \left(\frac{0.792T}{\ln(T+1)} \right)^T .$$

This also suggests that a new way to compute Bell's numbers is given by 3.12 and 3.13. Comparing this recursion to those in the OEIS [11] this is a new way to find Bell's numbers.

3.5.3 Computation Time. It should be obvious that the tree in Algorithm 1 is not computationally efficient. When comparing Algorithms 2 and 3 we can see that Algorithm 2 has more nodes than Algorithm 3 but the trimming the tree in 3 will require more computations at each node. The question remains; which algorithm is better? The previous result regarding Bell's numbers implies that Algorithm 3 is NP-hard. So Algorithm 2 is also NP-hard. Regardless of this, the computation times of the respective algorithms yield and interesting result.

Figure 3.7 shows the computation times of Algorithms 2 and 3. Notice that Algorithm 2 performs better until T gets large enough. Then Algorithm 3 performs better. The value of the tradeoff of tree size for computation time depends on how big the tree is. This suggests that for larger more realistic problems Algorithm 3 may provide a better approach.

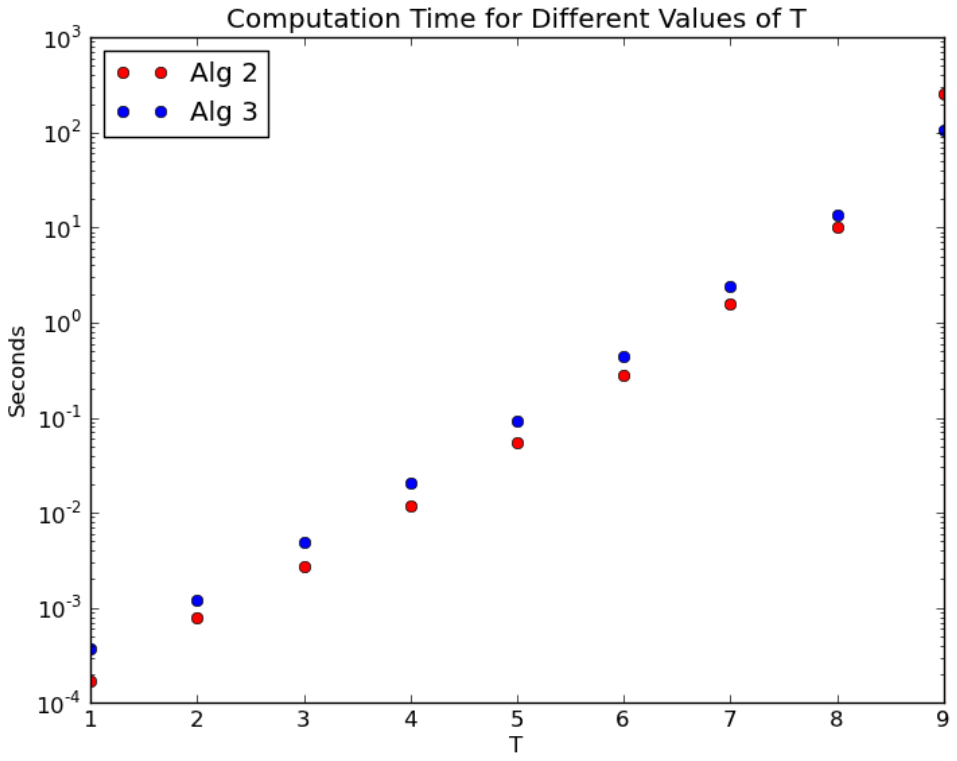

Figure 3.7: The computation times for of Algorithms 2 and 3 for varying values of T. The parameters for the tree are $m = 1$, $M = 2$, $C = 1$, $\omega = 2$, and $\kappa = 1$. The y-axis is a log scale. The Algorithms were programmed in Python and ran on a computer with a 2.4 GHz processor.

Chapter 4. Stochastic Appointment Lengths

In our next problem formulation we assume that appointment lengths are stochastic and that there is a positive probability of a scheduled appointment not showing up. In the deterministic case we assumed that every appointment took up a fixed number of appointment slots m. In this case we define the number of appointment slots $l^{(n)}$ of node n by

$$l^{(n)} = \begin{cases} 0 & \text{w/ prob } q_0 \\ 1 & \text{w/ prob } q_1 \\ \quad \vdots & \\ L & \text{w/ prob } q_L \end{cases} \tag{4.1}$$

where L is the longest appointment possible and $\sum_{i=0}^{L} q_i = 1$. When $l^{(n)}$=0 we have a no-show. In forming the tree of all possible decisions before the trim we include a node n if $x_{t,j} < C$ for $j = d^{(n)}$. In other words the untrimmed tree will be the same as the one from Algorithm 3 for $m = 1$. This means that the Bell numbers will also provide an upper bound to the number of nodes in this tree for varying values of T, κ, and ω.

The challenge in this problem formulation is in the trim. Let S be the event that a scheduled appointment shows up. Then

$$\mathbb{P}(S) = \sum_{i=1}^{L} q_i. \tag{4.2}$$

As in our last problem formulation, we assume that the office has a service capacity of C. Since the length of each appointment is unknown there is a chance that there may not be room for an appointment when its scheduled time comes. Let $T^{(n)}$ be the event that there is room available for the appointment given by node n. We assume that if there is no capacity for a scheduled appointment then the patient does not reschedule and leaves. We account for

this by a cost constant c. By assuming the independence of S and $T^{(n)}$ we have the revenue equation

$$R(n) = \begin{cases} r & \text{w/ prob } \mathbb{P}(S)\mathbb{P}(T^{(n)}) \\ 0 & \text{w/ prob } 1 - \mathbb{P}(S) \\ -c & \text{w/ prob } \mathbb{P}(S)(1 - \mathbb{P}(T^{(n)})). \end{cases}$$

Our value equation is similar to the previous formulation but now $R(m)$ is unknown so we have

$$V(x^{(n)}, w_{t^{(n)}+1} = w) = \max_{m \in C_{n,t^{(n)}+1,w} \cup \mathcal{N}} \{\mathbb{E}[R(m)] + \mathbb{E}[V(m, w_t + 1)]\}$$

where, by equation (4.2)

$$\begin{aligned} \mathbb{E}[R(m)] &= r\mathbb{P}(S)\mathbb{P}(T^{(m)}) + 0(1 - \mathbb{P}(S)) - c\mathbb{P}(S)(1 - \mathbb{P}(T^{(m)})) \\ &= [(r + c)\mathbb{P}(T^{(m)}) - c]\mathbb{P}(S) \\ &= [(r + c)\mathbb{P}(T^{(m)}) - c] \sum_{i=1}^{L} q_i. \end{aligned}$$

We need to find an expression for $\mathbb{P}(T^{(m)})$. Let P_m be the set of nodes that are the posterity of node m and let A_m be the ancestors of node m. Note that P_m is different from C_m in that P_m includes multiple generations. We are interested in the length of the appointments that are within L slots of node m so we let

$$F_m = \{a \in P_m \cup A_m | d^{(m)} - L < d^{(a)} \leq d^{(m)}\}. \tag{4.3}$$

Let $\mathbf{1}_B$ denote the indicator function of a set B where $\mathbf{1}_B(x) = 1$ if $x \in B$ otherwise $\mathbf{1}_B(x) = 0$. Let $X_{a,m} = \mathbf{1}_{l^{(a)} \geq d^{(m)} - d^{(a)} + 1}$ which is to be interpreted as the bernoulli random variable that is 1 if appointment a overlaps appointment m and 0 otherwise. Then the

probability that there is room for node m is

$$\begin{aligned} \mathbb{P}(T^{(m)}) &= \mathbb{P}(\sum_{a \in F_m} X_{a,m} \leq C) && (4.4) \\ &= \mathbb{P}(Y \leq C) && (4.5) \end{aligned}$$

where $Y = \sum_{a \in F_m} X_{a,m}$. If the $X_{a,m}$ were independent and identically distributed then we would have a binomial distribution. However, we can see that Y is a sum of bernoulli trials that are not identically distributed. If we assume they are independent then Y is a poisson-binomial random variable and we can compute the probability we seek by finding the cumulative distribution function of Y. We explore this distribution further in section 4.1.

The cdf of Y is determined by the individual probabilities $\mathbb{P}(X_{a,m} = 1)$ for each $a \in F_m$. These probabilities will be different depending on whether $a \in A_m$ or $a \in P_m$. If $a \in A_m$ then the length of node a, $l^{(a)}$, is conditional on the event $T^{(a)}$. So for $a \in A_m$, by the law of total probability we have

$$\begin{aligned} \mathbb{P}(X_{a,m} = 1) &= \mathbb{P}(l^{(a)} \geq d^{(m)} - d^{(a)} + 1 | T^{(a)}) \mathbb{P}(T^{(a)}) \\ &\quad + \mathbb{P}(l^{(a)} \geq d^{(m)} - d^{(a)} + 1 | (T^{(a)})') \mathbb{P}((T^{(a)})') \\ &= \mathbb{P}(T^{(a)}) \sum_{i=d^{(m)}-d^{(a)}+1}^{L} q_i && (4.6) \end{aligned}$$

since the second conditional probability is 0 and the first conditional probability is given from equation (4.1). We denote the compliment of the event $T^{(a)}$ by $(T^{(a)})'$. If $a \in P_m$ then depending on demand we may or may not reach not a. To account for this uncertainty we multiply (4.6) by the probability that we get to node a given that we are at time $t^{(m)}$ which

is given by (3.10). This gives us

$$\mathbb{P}(X_{a,m}=1)=\begin{cases}\mathbb{P}(T^{(a)})\sum_{i=d^{(m)}-d^{(a)}+1}^{L}q_i & \text{if } a\in A_m\\ \mathbb{P}(a|t^{(m)})\mathbb{P}(T^{(a)})\sum_{i=d^{(m)}-d^{(a)}+1}^{L}q_i & \text{if } a\in P_m\end{cases} \tag{4.7}$$

The recursive formula given by (4.4) and (4.7) allows you to find the probabilities needed for the distribution of Y.

4.1 Poisson-binomial distribution

We let $N=|F_m|$, which is the cardinality of the set F_m which is given by (4.3). We index the elements in F_m and let $\mathbb{P}(X_{a_i,m}=1)=\alpha_i$. Let B_k denote the set of all possible ways to pick k integers from the set $\{1,2,\ldots,N\}$ when order doesn't matter. The elements of B_k are sets b of k elements. We can write the pdf of $Y=\sum_{i=1}^{N}X_{a_i,m}$ as

$$f_Y(k)=\sum_{b\in B_k}\left(\prod_{i\in b}\alpha_i\right)\left(\prod_{j\in b'}(1-\alpha_j)\right)$$

One way to compute $\mathbb{P}(Y\leq\tau)$ is to use the the pdf by summing from $k=0$ to $k=\tau$. An alternative expression for the cdf is given by Fernandez and Williams [13]. In their derivation they used Fourier transforms to get that

$$\begin{aligned}\mathbb{P}(Y\leq\tau) &= \frac{\tau+1}{N+1}+\frac{1}{N+1}\sum_{n=1}^{N}\Bigg\{(1-e^{-i2\pi n(\tau+1)/(N+1)})/(1-e^{-i2\pi n/(N+1)})\\ &\qquad \prod_{k=1}^{N}\{\alpha_k e^{i2\imath n/(N+1)}+1-\alpha_k\}\Bigg\}\end{aligned} \tag{4.8}$$

Chapter 5. The Feedback Loop

The effectiveness of this system of dynamic appointment scheduling will hinge on the the quality and types of data that are collected. Some useful types of data include:

(i) Patient service time - total time elapsed from when the nurse begins examining vitals until s/he leaves the clinic.

(ii) Nurse-patient time - total time elapsed when being seen by the nurse for vitals

(iii) Doctor-patient time - time the doctor spends with the patient.

Radio Frequency Indentification Technology (RFID) are revolutionizing the way companies obtain data about their operations. The most common use of RFIDs is in inventory management. Retailers have found it be a useful and an efficient alternative to barcodes as a way to track inventory. This is particularly useful in distribution centers where RFIDs are placed on pallets and the locations and quantities of inventories are continually tracked. Among several other applications, RFIDs have been used in toll roads, luggage tracking in airports, animal identification, casino chip tracking, and hospitals.

In hospitals, RFIDs offer an attractive solution to many hospital inefficiencies. They have been used to track equipment, patients, and staff. For example, it is common for nurses to lay claim to supplies that they frequently use by placing them somewhere they can access easily. This leads to a shortage of items when needed by other staff. By installing an RFID system, the location of supplies can be tracked and their location can be known at any time. Other hospitals have given each staff member an RFID to collect data on procedure lengths. This data is then analyzed to detect any inefficiencies. This can also be used to find doctors in the hospital if s/he is late for an appointment, is needed for a consult, or in the case of an emergency.

An RFID has two primary components: tags and readers (or interrogators). A tag is placed on the item or person that you wish to track. When the tag is near a reader, it emits a radio signal and the reader registers its location. Readers are strategically placed, such as at the entrance of a door, so that the needed information is gathered. The data is then sent to a computer via an application server where the locations of the tags are displayed.

Example: Suppose that each nurse and physician has an RFID tag on their ID badge. A patient schedules an appointment with the clinic complaining of a sore throat and congestion in the back of his throat. Upon his arrival he is given an RFID tag in the form of a bracelet or card. Then the nurse takes the patient's vitals marking the beginning of the nurse-patient time. When the nurse leaves the reader marks the end of the nurse- patient time. The patient then waits for the doctor. In this room there is another reader that indicates he is waiting for the doctor to come in. The doctor arrives the reader notes the beginning of the doctor-patient time. After some thought the doctor decides he needs to run a strep throat test. The doctor leaves the room, marking the end of the doctor-patient time, as the test is being processed and the patient experiences more waiting time. The test comes back positive, he receives his prescription and leaves. When the patient exits the clinic the reader by the front door notes the end of the service time.

The RFID system has recorded several bits of information about the patient. Lots of similar data could give us great estimates for equation 4.1. The patient service time could be used to set up the queues for the nurses and doctors. Alternatively, the information could be used to create two separate scheduling systems; one for the doctor and one for the nurse. With this sort of information on many different patients we can determine the likelihood that a new patient will receive a certain diagnosis based on his complaints. This would allow even greater accuracy of the probabilities given in equation 4.1 when each decision is made. The diagnosis will have an associated profitability which will be helpful to the appointment system as it seeks to maximize profit. We could use the information to get a better estimation for $\mathbb{E}[d_t]$. Table 5.1 shows how this information could be used.

	Complaints	Diagnosis	Service Time	Revenue
	⋮			
✓	post-nasal drip	35% strep throat	20 min	$250
	swollen glands	40% cold	10 min	$100
✓	sore throat	20% flu	15 min	$150
	fever	5% mono	30 min	$300
	⋮			

Table 5.1: The initial complaints of the patient can indicate the likelihood of certain diagnoses. Data collected overtime could be helpful in determining these probabilities. Data gathered from the RFIDs could be used to determine service times based off of diagnosis. The expected service time in this case is 15.5 minutes and the expected profit is $172.50. Note that these numbers are not based off data.

Chapter 6. Conclusion

In this work we have described 3 different algorithms designed to solve the scheduling problem for deterministic appointment lengths and stochastic demand. We found that the third algorithm solved the problem faster than the second algorithm for large schedules. We also found that Bell's numbers provided an upper bound for the number of nodes in an untrimmed tree under Algorithm 3 [12], demonstrating that the problem is NP-hard. Algorithm 3 provided a useful framework for the case where we include no-shows and stochastic appointment lengths. We formulated the stochastic appointment length case noting that we would need a nice way to compute probabilities in the poisson-binomial distribution [13].

It remains to be shown that there is indeed a bijection between the number of nodes in the untrimmed tree (where all appointments are length 1, the service capacity is 1, and there is overlap in the hueristic parameters κ and ω) and the partitions of a set of elements. We also did not consider the infinite horizon case. A useful appointment system often must be able to see weeks in advance. It might be useful to solve the problem with a moving window approach [14]. That is to make a decision that is optimal when you restrict your attention to some finite horizon. When a time period passes you then move your finite horizon up

to include an additional time period. One of the main results in appointment scheduling is the Baily-Welch rule, which says it is better to double book early in the schedule so that if you have a no-show you don't lose any business. The queue of patients continues to funnel patients to the doctor to make sure the doctor has little idle time. It would be interesting to take the problem formulation under the stochastic case and take away the condition that if someone is not seen on time then they will leave. This would create a queue of patients that are waiting for service. It is likely that under this model the Baily- Welch rule will hold.

While this work was done with health care in mind, there is no reason why it cannot be applied to other situations. We could instead schedule cars for an oil change or repairs for a planes in a hanger. Regardless, of the situation there remains much to be done to derive and understand optimal scheduling systems. Much of the literature is focused on comparing heuristics rather than solving an optimization problem. The reason is that optimization problems grow dramatically in complexity. However, with an infinite horizon approach such as a moving window or policy iteration it may be reasonable to get close to an optimal solution.

Appendix A. Algorithm 2 code

The following is my code for Algorithm 2 in Python. You will notice I preassigned several commonly used objects to save computation time that comes from defining a new object of the same type. Also be aware that Python uses zero indexing. This means the first element in a list A is $A[0]$.

```
class TreeClass:


    def __init__(self,W,T,m,omega,kappa,p,R,capacity):
        rootNode = nodeClass.NodeClass([],-1)
```

```
    rootNode.parent = rootNode
    self.startt = -1  #the time of the root node
    self.endt = (T-m)   #endt is the first time where the nodes of that
     #time have no children
    self.root = rootNode  #root node corresponds to an empty schedule
    self.W = W          #indexes the three possible random outcomes
    self.T = T          #number of blocks in appointment schedule
    self.m = m          #every appointment is m blocks long
    self.omega = omega  #willingness of walk-ins to wait
    self.kappa = kappa  #earliest a call-in can be scheduled
    self.p = p          #p[0]=prob of nothing, p[1]=prob of walk-in,
    #p[2]=prob of call-in
    self.R = R          #every appointment assumes a revenue of R
    self.capacity = capacity   #service capacity constraint
    self.null = 'null'  #a pointer to a null decision
    self.Ran = ([tuple([0]),tuple([0,1]),tuple([0,2]),tuple([0,1,2]),
    ...tuple([1]),tuple([2]),tuple([1,2])])
    #a set indicating all possible combinations of random outcomes
    #node can be valid for.

def addChild(self, parent, t):
    #adds a node to a tree with parent at time t
    newNode = nodeClass.NodeClass(parent, t)
    parent.children.append(newNode)

def findNodesTime(self,nodes,t):
    #returns all the nodes in a tree
```

```
    if nodes[0].t == t:
        return nodes
    else:
        N = []
        for node in nodes:
            N = N + node.children
        return self.findNodesTime(N,t)

def buildTree(self,node):
    #recursively builds tree of all possible decisions
    D,R = node.feasibleSet(node.t,self.T,self.m,self.omega,self.kappa,
    ...self.capacity,self.Ran,self.W,self.null)
    for d in D:
        r = R[D.index(d)]
        self.addChild(node,node.t+1)
        child = node.children[-1]
        child.update(d,r,self.R)
        if child.t != self.endt:
            self.buildTree(child)

def trimTree(self,Parents):
    #only keep optimal decisions through the value function
    S = set()
    for parent in Parents:
        #first we identify the optimal nodes under each random outcome
        c0 = parent.valueFunction(0,self.endt,self.p,self.W)
        c1 = parent.valueFunction(1,self.endt,self.p,self.W)
```

```
            c2 = parent.valueFunction(2,self.endt,self.p,self.W)
            #now we make sure that node parent has only the optimal children
            parent.children = list(set([c0,c1,c2]))
            #now we make sure the optimal nodes are only valid for the
            #random outcome it was optimized over
            c0.random = []; c1.random = []; c2.random = [];
            c0.random.append(0); c1.random.append(1); c2.random.append(2);
            c0.random = self.Ran[self.Ran.index(tuple(c0.random))]
            c1.random = self.Ran[self.Ran.index(tuple(c1.random))]
            c2.random = self.Ran[self.Ran.index(tuple(c2.random))]
            S.add(parent.parent)
        if Parents[0].t != self.startt:
            self.trimTree(list(S))

#this function is used later in method valueFunction
argmax = lambda array: max(izip(array,xrange(len(array))))[1]

class NodeClass:
    #This node class will be used as members of a tree data structure
    #t = period the node lies in, or the period the decision is made
    #d = decision
    #w = randomness that occurs each period, w = [walk-in,call-in]

        def __init__(self, parent, t):
            #Initialize a node by creating the pointer to the parent
            #and creating an empty list for the children
            self.parent = parent
```

```
        self.t = t
        self.children = []

    def update(self,d,r,R):
        self.decision = d   #decision corresponding to node
        self.random = r     #random events that this node is valid for
        if type(d) == int:
            self.profit = R #revenue of node
        else:
            self.profit = 0

    def feasibleSet(self,t,T,m,omega,kappa,capacity,Ran,W,null):
        #This method returns the feasible decisions for self.
        #D is the set of possible decisions
        #R indicates for which random events each element in D
        #is valid for
        D = set(); R = []
        sch = self.assembleSchedule(T,m)
        #cond1 and cond2 make sure that the heuristic parameters
        #kappa and omega are being satisfied
        cond1 = lambda s: s in range(t+1,t+omega+1)
        cond2 = lambda s: s in range(t+kappa+1,T-m+1)
        #keep only those decisions that satisfy the capacity constraint
        for d in range(t+1,T-m+1):
            b = []
            if cond1(d) and sum(sch[d:d+m]+ones([m,1])<=capacity) == m:
                D.add(d)
```

```
            b.append(1)
        if cond2(d) and sum(sch[d:d+m]+ones([m,1])<=capacity) == m:
            D.add(d)
            b.append(2)
        if len(b) != 0:
            R.append(Ran[Ran.index(tuple(b))])
    D = list(D)
    D.append(null)
    R.append(Ran[Ran.index((0,1,2))])
    return D,R

def findChildrenRand(self,w):
    #returns children of self with random event w
    cw = []
    for child in self.children:
        if w in child.random:
            cw.append(child)
    return cw

def prob(self,p):
    #Returns probability of a node given that we are the preceding time.
    P = 0
    for r in self.random:
        a = p[r]
        P = P + a
    return P
```

```
def expectedValue(self,endt,p):
    #returns the expected future value of a node not including
    #the value of self
    E = 0
    if self.t != endt:
        for child in self.children:
            E = E + child.prob(p)*(child.profit+child.expFutVal)
    return E

def valueFunction(self,w,endt,p,W):
    #returns the best node for a child of self for random outcome w
    cw = self.findChildrenRand(w)
    L = []
    for child in cw:
        child.expFutVal = child.expectedValue(endt,p)
        L.append(child.profit+child.expFutVal)
    bestNode = cw[argmax(L)]
    return bestNode

def assembleSchedule(self,T,m):
    sch = zeros([T,1])
    while not(self.parent is self):
        d = self.decision
        if type(d) == int: sch[d:d+m] = sch[d:d+m] + ones([m,1])
        self = self.parent
    return sch
```

```
#the following code utilizes the methods and classes above to solve the problem
Tree = treeClass.TreeClass(startt,W,T,n,m,omega,kappa,p,R,stack,Ran)
Tree.buildTree(Tree.root)
lastNodes = Tree.findNodesTime([Tree.root],Tree.endt-1)
Tree.trimTree(lastNodes)
```

Appendix B. Algorithm 3 code

Here we include the code for Algorithm 3. We will only include those methods that are different or new from those in Algorithm 2.

```
class TreeClass:


    def findNodesTime(self,node,t):
        #returns all decendants of a node with time t
        #each node has children at several different times
        T = list(c for c in node.children if c.t == t)
        if node.t == t: T.append(node)
        L = list(c for c in node.children if c.t < t)
        while len(L) != 0:
            C = []
            for l in L: C = C + l.children
            T = T + list(c for c in C if c.t == t)
            L = list(c for c in C if c.t < t)
        return T


    def buildTree(self,node,t):
```

```
    #builds tree of all possible decisions disregarding optimality
    D,R = node.feasibleSet(t,self.T,self.m,self.omega,self.kappa,
    ...self.capacity,self.Ran,self.W)
    for d in D:
        r = R[D.index(d)]
        self.addChild(node,t+1)
        self.numNodesBeforeTrim += 1
        child = node.children[-1]
        child.update(d,r,self.R)
        if child.t != self.endt:
            self.buildTree(child,child.t)
        else:
            child.expFutVal = 0
    if t != self.endt:
        self.buildTree(node,t+1)

def trimTree(self):
    #deletes nodes on tree that are suboptimal
    T = range(-1,self.endt); T.reverse()
    #we trim the tree at each time t
    for t in T:
        for node in self.findNodesTime(self.root,t):
            #since not all nodes have children we only trim those
            #with children
            if len(node.children) != 0:
                node.paveTheWay(self.endt,self.p,self.W,self.Ran)
                c1 = node.valueFunction(self.W[0],self.endt,self.p,self.W)
```

```
                    c2 = node.valueFunction(self.W[1],self.endt,self.p,self.W)
                    #we don't want to remove all nodes but just the suboptimal ones
                    #at time t+1
                    for c in node.findChildrenTime(node.t+1):
                        if c is c1:
                            c1.random.append(1)
                        if c is c2:
                            c2.random.append(2)
                        if not(c is c1) and not(c is c2):
                            node.children.remove(c)
                            del(c)
                        else:
                            c.random = self.Ran[self.Ran.index(tuple(c.random))]
                    node.expFutVal = node.expectedValue(self.p,self.W)
                else:
                    node.expFutVal = 0


class NodeClass:

        def findChildrenRand(self,w,t):
            #Returns list of children of self at time t with random event w
            cw = list()
            for child in self.children:
                if w in child.random and t == child.t:
                    cw.append(child)
            return cw
```

```
def findChildrenTime(self,t):
    #Returns children of self at time t
    ct = [child for child in self.children if child.t == t]
    return ct

def expectedValue(self,p,W):
    #returns the expected future value of a node self not knowing
    #what the next random event will be.
    E = 0
    for child in self.children:
        if child.t == self.t+1:
            E = E + child.prob(p)*(child.profit+child.expFutVal)
    if len(self.children) != 0:
        a = 1-sum([c.prob(p) for c in self.findChildrenTime
        ...(self.t+1)])
        E = E + a*self.expValofNull(self.t+1,p,W)
    return E

def valueFunction(self,w,endt,p,W):
    argmax = lambda array: max(izip(array,xrange(len(array))))[1]
    #argmax returns index of max in array
    cw = self.findChildrenRand(w,self.t+1)
    L = []
    for child in cw:
        L.append(child.profit+child.expFutVal)
    a = self.expValofNull(self.t+1,p,W)
    if max(L+[a]) != a:
```

```
            best = cw[argmax(L)]
            best.expFutVal = best.expectedValue(p,W)
            best.random = []
        else: best = 'null'
        return best

    def expValofNull(self,t,p,W):
        #null decision is made at time t
        #self is the parent node
        T = list(set(child.t for child in self.children if child.t > t))
        T.sort()
        CT = [self.findChildrenTime(t) for t in T]
        ep = []
        pb = []
        for ct in CT:
            ep.append(sum([c.prob(p)*(c.profit+c.expFutVal) for c in ct]))
            pb.append(1-sum([c.prob(p) for c in ct]))
        if len(pb) != 0: pb.pop()
        pb.insert(0,1)
        a = 0
        for i in range(len(T)):
            a = a + ep[i]*reduce(lambda x,y: x*y,pb[:i+1])
        return a

    def paveTheWay(self,endt,p,W,Ran):
        #prepares self for the use of the valueFunction by eliminating
        #suboptimal nodes that are not first children at time t+1
```

```
argmax = lambda array: max(izip(array,xrange(len(array))))[1]
T = list(set(child.t for child in self.children if child.t
...!= self.t+1))
T.sort(reverse = True)
for t in T:
    bestNodes = set()
    for w in W:
        cw = self.findChildrenRand(w,t)
        L = []
        a = 0
        for child in cw:
            L.append(child.profit+child.expFutVal)
        if t != T[0]:
            a = self.expValofNull(t,p,W)
        L.append(a)
        if max(L) != a:
            best = cw[argmax(L)]
            try:
                best.temp.append(w)
            except AttributeError:
                best.temp = []
                best.temp.append(w)
            bestNodes.add(best)
    for c in self.findChildrenTime(t):
        if not(c in bestNodes): c.delete()
    for b in bestNodes:
        b.random = Ran[Ran.index(tuple(b.temp))]
```

```
            del(b.temp)


def assembleSchedule(self,T,m):
    sch = zeros([T,1])
    while not(self.parent is self):
        d = self.decision
        sch[d:d+m] = sch[d:d+m] + ones([m,1])
        self = self.parent
    return sch


def feasibleSet(self,t,T,m,omega,kappa,capacity,Ran,W):
    D = set(); R = []
    sch = self.assembleSchedule(T,m)
    cond1 = lambda s: s in range(t+1,t+omega+1)
    cond2 = lambda s: (s in range(t+kappa+1,T-m+1))
    for d in range(t+1,T-m+1):
        b = []
        if cond1(d) and sum(sch[d:d+m]+ones([m,1])<=capacity) == m:
            D.add(d)
            b.append(1)
        if cond2(d) and sum(sch[d:d+m]+ones([m,1])<=capacity) == m:
            D.add(d)
            b.append(2)
        if len(b) != 0:
            R.append(Ran[Ran.index(tuple(b))])
    D = list(D)
    return D,R
```

```
#The following lines of code can be run to solve the problem

#You must first specify your parameters

Tree = treeClassS.TreeClassS(W,T,m,omega,kappa,p,R,capacity)

Tree.buildTree(Tree.root,-1)

Tree.trimTree()
```

Bibliography

[1] N.T.J. Bailey. A study of queues and appointment systems in hospital outpatient departments, with special reference to waiting times. *J.R. Stat Soc Ser*, 14(2):185–199, 1952.

[2] J.D. Welch and N.T.J. Bailey. Appointment systems in hospital outpatient departments. *Operational Research Quarterly*, 15(3):224–232, 1964.

[3] B Gendron P. Michelon H. Beaulieu, J. Ferland. A mathematical programming approach for scheduling physicians in the emergency room. *Health Care Management Science*, 3:193–200, 2000.

[4] B. Meenan L. Garg, S. McClean. A non-homogeneous discrete time markov model for admission scheduling and resource planning in a cost or capacity constrained healthcare system. *Health Care Management Science*, 13:155–169, 2009.

[5] Tugba Cayirli and Emre Veral. Outpatient scheduling in health care: A review of literature. *Production and Operations Management*, 12(4), 2003.

[6] Guido C. Kanndorp and Ger Koole. Optimal outpatient appointment scheduling. *Health Care Management Science*, 10:217–229, 2007.

[7] Sabine Sickinger and Rainer Kolisch. The performance of a generalized bailey-welch rule for outpatient appointment scheduling under inpatient and emergency demand. *Health Care Management Science*, 12:408–419, 2009.

[8] Kenneth J. Klassen and Thomas R. Rohleder. Rolling horizon appointment scheduling: A simulation study. *Health Care Management Science*, 5:201–209, 2002.

[9] Kenneth Klassen and Thomas Rohleder. Outpatient appointment scheduling with urgent clients in a dynamic, multi-period environment. *International Journal of Service Industry Management*, 15(2):167–186, 2004.

[10] Eric Temple Bell. Exponential polynomials. *Annals of Mathematics*, 35(2):258–277, 1934.

[11] The on-line encyclopedia of integer sequences. `http://oeis.org`, 2011.

[12] T. Tassa D. Berend. Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205, 2010.

[13] Stuart Williams Manuel Fernandez. Closed-form expression for the poisson-binomial probability density function. *IEEE Transactions on Aerospace and Electronic Systems*, 46(2):803–817, 2010.

[14] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 3 edition, 2007.