MODELS AND ALGORITHMS FOR COMPLEX SYSTEM OPTIMIZATION PROBLEMS: APPLICATIONS TO HOSPITAL LAYOUT AND LED TRAFFIC SIGNAL MAINTENANCE

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science, Industrial Engineering

---

by

Jonathan Hathhorn

Dr. Mustafa Sir, Thesis Supervisor

DECEMBER 2010

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

MODELS AND ALGORITHMS FOR COMPLEX SYSTEM OPTIMIZATION PROBLEMS: APPLICATIONS TO HOSPITAL LAYOUT AND LED TRAFFIC SIGNAL MAINTENANCE

Presented by Jonathan Hathhorn,

A candidate for the degree of Master of Science,

And hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Mustafa Sir

Dr. Esra Sisikoglu

Dr. James Noble

Dr. Carlos Sun

Acknowledgments

I would like to thank my primary advisor, Dr. Mustafa Sir, and my co-advisor, Dr. Esra Sisikoglu. Dr. Sir provided encouragement, guidance, and conceptual and editorial support throughout my graduate research. Dr. Sisikoglu also provided encouragement and guidance, and was instrumental in my understanding of dynamic programming algorithms.

I would also like to thank Dr. Carlos Sun for his insight into the operation of LED traffic signals and providing useful references, and Dr. James Noble for lending his expertise in healthcare systems design. Both provided editorial support as part of my thesis committee.

I am very grateful to Kara Bono for all of her research and work on the hospital layout model, and I would like to thank Osagie Evbuomwan for his input as well. I would also like to thank Chris Hathhorn for his technical support and Python programming language assistance. Finally, I would like to thank Brent Hathhorn for his editorial support.

Contents

List of Figures

List of Tables

Abstract

Due to rising healthcare costs, it is increasingly important to design health care buildings to be efficient and effective. One aspect of a healthcare facility's design is the size and layout of the building and departments. In this paper we review hospital design and the various layout methods that can be applied to hospitals. We formulate a mixed-integer linear programming model to determine the optimal size (i.e. width and length of each floor and number of floors) and department layout of a hospital. The model has multiple objectives; we consider department size requirements to determine a cost-efficient facility size and then place departments to minimize inter-departmental flows. Finally, we use the model to design a multi-floor hospital with seven departments and test the computation time for a variety of scenarios.

The Energy Policy Act of 2005 specifies that all traffic signals manufactured after January 1, 2006 must realize the energy efficiency achieved by LED technology [10]. These new LED traffic signals use less energy and last longer than their predecessors, but they deteriorate gradually and require customized maintenance schedules to optimize their useful life and maximize public safety. In the second half of this paper we review the advantages of LED traffic signals and the current literature on their maintenance. We present

three models and algorithms to compute optimal maintenance schedules. The first model is designed to model routine maintenance and includes routing costs. The second model is an approximation of the first model that can be solved for scenarios which include very large quantities of traffic signals. The final model allows for two actions, inspection and replacement, and introduces stochastic deterioration. We test the computation time of each algorithm and assess the resulting schedules.

1 Introduction

The rising costs of healthcare are in the limelight of public criticism, compelling the healthcare industry to scrutinize the efficiency of its operations. A key area of investigation in the search for greater efficiency is the layout of a healthcare facility. Industrial engineers have long studied the efficient layout of facilities with variations of the Quadratic Assignment Problem (QAP) and other models. A typical objective is determining the placement of departments to optimize interdepartmental flows. In the case of a hospital, optimizing the interdepartmental flows means minimizing the distance that nurses, patients, and other mobile entities have to traverse.

The first half of this paper focuses on developing and solving a large mixed-integer, linear programming model that incorporates the decisions and costs that arise when building a facility in addition to optimizing the department layout of the facility. We investigate methods to balance all of these costs and decrease the solve time of the model.

A new, energy-efficient light source illuminates many traffic signals today: light-emitting diodes (LED). While it is clear that these new LED traffic signals have a longer useful life than their predecessors, they deteriorate slowly over time instead of simply failing. As a result of this progressive deterioration, the decision of when to replace them is no longer certain, but can be aided by on-site testing. This need for interim inspection and testing

creates the problem of balancing the different costs of this process against the steadily increasing need for replacement.

In the second half of this paper, we propose a model to optimize the timing and routing of maintenance services. Next, we examine an approximation model that becomes necessary to handle large scenarios. To optimize the useful life of LEDs, we discuss an inspection strategy and model with the assumption of stochastic decay. Finally, we develop and test algorithms for each of these three models.

The models and algorithms presented in this paper are works in progress. This paper evaluates their capabilities and identifies ways to improve their performance, although all of the algorithms struggle with unreasonably long computation times for realistic-sized scenarios. The evaluation focuses on the computation time required to solve the different models and methods to decrease the potential computation time.

2 Hospital Layout Introduction

The health care industry faces increasing costs from expensive new treatments, technology, and higher customer expectations. Treatments and equipment are becoming more specialized and more difficult to integrate within the facility effectively. Patients, as customers of the hospital, expect to be serviced promptly to minimize their cost of waiting. In addition to patient expectations, the community in which the hospital operates expects the hospital to meet all customer demands at minimal cost. As the cost of operating hospitals continues to increase, the pressure on hospital administrators to manage hospital operations, resources, and equipment more efficiently and more effectively increases. Consequently, attention must be given to the way in which hospitals are designed to meet these demands.

Hospital design is a highly integrated problem since hospital functions and design issues are closely intertwined. A primary hospital design issue is the hospital layout—the layout of departments within the hospital. Often hospitals are designed as a multi-floor, integrated facility. However, little research has been done to determine when a multi-floor facility—as opposed to a single-floor facility—is a more efficient design choice. In the first chapter, we present a review of hospital design issues along with common facility layout design models and multi-floor design models. Chapter 4 presents a formulation to optimize a multiple

floor facility layout that can be used for hospital layout. Chapter 5 demonstrates our multi-floor facility layout optimization formulation with an example instance. We conclude with possible future research and extensions to the facility layout model.

3 Hospital Layout Literature Review

3.1 Hospital Design

A hospital is an institution for health care that provides diagnosis and treatment to a wide range of patients by specialized staff and equipment. Hospital patients are often grouped into three broad categories: outpatients, inpatients, and emergency patients. Outpatients visit the hospital—often by appointment—for diagnosis or treatment and do not stay overnight. Inpatients have been admitted into the hospital, assigned a room, and are expected to stay at least one night. Emergency patients, on the other hand, generate hospital demand—without an appointment—and may be discharged the same day or admitted into the hospital as an inpatient.

Depending on its ability to admit and care for inpatients, a hospital may be categorized as general, specialized, teaching, or clinic. While a clinic generally provides only outpatient service—and some specialized hospitals provide only certain types of treatment, e.g., cancer or orthopedic—most hospitals service all kinds of patients. The hospitals which service a wide variety of patients must consequently provide a wide range of services and functions. From diagnostic and treatment (cardiology, orthopedics, imaging, laboratories, etc.) to

hospitality (food services, cleaning services, etc.), the hospital has the responsibility to provide each function with the utmost care and personalized attention to every patient.

Though the general functions of the hospitals are very much interrelated, the relationships between functional units within and between each general function are not fully understood by hospital management. Tzortzopoulos et al. comment that many hospital processes tend to be organized around functional units. Managers and designers continue to approach each functional unit as though it provided an isolated function [34]. Since functional units can have competing needs and priorities, it is important to balance mandatory requirements, functional requirements (including relationships to other units), and individual unit preferences as well as the needs of patients and hospital staff [12]. Many problems faced in healthcare service delivery today are directly related to poor interactions between services and building design; managers and designers must adopt an integrated and comprehensive approach when making building design decisions [34].

Hospital layouts depend on the operations management of functional units and between functional units. The patient capacity, for example—the number of beds per unit—directly relates to the physical configuration and logistics of a hospital. Congestion within a hospital indicates that operational control and building design are not optimal. Flow congestion increases waiting and throughput times, sometimes resulting in a negative impact both on the quality of service delivered and the capacity utilization of the hospital, and always on the patients' impression of the hospital's efficiency and competence [41]. Reducing delays and unclogging bottlenecks depend on assessing and improving flows between functional units. Patient flow should be improved across the system as a whole rather than within isolated units [16]. Thus, the integrated approach to hospital layout should depend on the patient demand for beds, patient capacity, patient flow, functional unit relationships, and

the hospital logistics system.

3.2 Facility Layout Problem (FLP)

The facility layout problem is concerned with solving the physical organizational puzzle within a production system to minimize the material handling flow costs between departments. Solutions arrange a discrete number of departments within the bounds of a facility. A layout's efficiency can be measured in terms of its material handling cost—the sum of the distances between each department multiplied by the flow or weight between those departments, i.e. the value of the objective function. Other concurrent objectives of the facility layout problem include minimizing lead time, inventory, and required space in addition to maximizing flow, capacity, throughput, quality, and efficiency. With any set of objectives, the layout must also satisfy numerous constraints. Constraints might include departmental area requirements, departmental location restrictions, aisle network requirements, or department overlapping restrictions. The facility layout problem formulation can also be applied in many other applications, e.g. the arrangement of electronic components on a circuit board.

In developing solutions to the facility layout problem, researchers have utilized various modeling approaches, formulations, and solution algorithms. Three of the most popular modeling approaches for the facility layout problem are the quadratic assignment problem, the mixed-integer programming model, and the graph-theoretic approach. Each approach has advantages and disadvantages in modeling and solving.

3.3 Quadratic Assignment Problem (QAP) Approach

The quadratic assignment problem (QAP) is one of the most common facility layout approaches. Its objective is typically to minimize material flow cost, but it might vary between minimizing cost, distance, flow, or time. The solution is typically obtained by assigning a discrete set of equal-sized departments to a discrete number of locations in order to minimize the flow and distance between departments.

These assumptions—that all departments are equal in size and that the available locations are discrete—are a special case of this approach. These assumptions make the model very simple to understand and to optimally solve for a relatively large number of departments compared to other formulations; however, they ignore the highly likely possibility that departments are unequal in size and that locations are continuous instead of discrete. Thus, even a highly optimal solution may not actually represent a good approximation of a real facility.

Finding solutions to the QAP is known to be NP-hard, so as the number of departments to be assigned in the facility layout problem increases, the model complexity and solve time increases enormously.

3.4 Mixed-Integer Programming (MIP) Approach

The mixed-integer programming formulation for the facility layout model, first proposed by Montreuil [19], uses a distance-based objective similar to that of the QAP. That is, the MIP formulation attempts to achieve the same objective as the QAP approach: to minimize the material handling cost within a facility by decreasing the weighted distance between departments. However, unlike the QAP, which assigns a discrete number of equal-

sized departments to a set of discrete locations, the MIP approach attempts to position a discrete number of unequal-sized departments into a continuous solution space (constrained by the overall facility size).

Even though the MIP approach constrains the department size and orientation to a rectangular shape that is known and fixed prior to formulation, one of its advantages is the flexibility it provides to the department sizes. These flexible department sizes—unlike QAPs equal-sized departments—create a more realistic representation of the facility layout problem. However, the complexity associated with allowing unequal-sized departments increases greatly as the number of departments in the problem increases, thus making the MIP approach potentially much more complex than QAP. Newer algorithms, including a new branch and bound algorithm, have increased the level of optimality obtained for larger sized problems, but many extensions to the MIP approach make the optimal solution to the problem even more difficult to achieve efficiently. For example, extensions might utilize the departmental spacing constraints to establish an aisle network or additional constraints to establish locations for pick-up and drop-off severely increase the complexity of the model.

3.5 Graph Theoretic Approach

The graph-theoretic approach varies significantly from the more common facility layout approaches, MIP and QAP. While the objective of MIP and QAP was to minimize the material handling cost within a facility, the objective of the graph-theoretic approach is to maximize the weighted relationships between adjacent departments. The graph-theoretic approach relies mainly on the relationships between the departments. In other words, it primarily assumes that the desirability of locating each pair of departments adjacent to each other is known. Other parameters, such as the size and shape of the departments, which are

necessary for the formulation of either the MIP or the QAP, are initially ignored. The graph-theortic approach uses department relationships to construct a maximally weighted planar graph. Nodes in the graph represent departments and arcs represent adjacency relationships between departments. The number of adjacency relationships is the crucial constraint of this model: a department may be limited, for example, to six adjacent departments to represent a certain manufacturing department. Once the graph is close to optimal it can be converted into a block layout so department shapes and sizes can be considered.

The graph-theoretic approach has many disadvantages. First, it initially ignores the area and shape of departments. Although the problem may be easier to solve without involving sizes (especially unequal sizes), the sizes and shapes of departments can limit the number of department adjacencies. Additionally, the graph-theoretic approach encounters a similar disadvantage of the QAP: the problems that consist of unequally-sized departments are very difficult and cannot guarantee optimality even for a small number of departments. Due to these disadvantages, it is possible that the approach cannot guarantee a realistic representation of a facility with varying department sizes. Since the graph-theoretic approach sacrifices the ability to indicate interdepartmental distances, it is unable to incorporate space and aisle travel considerations—unlike the MIP approach, which may be extended to handle these considerations. Finally, it is difficult to determine if a planar graph representation of the relationship actually exists, let alone what the weights of the department adjacencies might be. As a result, heuristics are often used to construct the maximally weighted adjacency graph.

3.6 Multi-Floor Extension

The multi-floor facility layout problem is an extension to the single-floor facility layout problem in which a discrete number of departments are assigned to a multiple-floor facility. The main objective of the multi-floor facility layout is identical to that of the single floor facility layout: minimize material handling by assigning departments to effective positions in the facility. However, the additional decision variables—such as the number of floors, the number of elevators, and the location of elevators—add complexity to the problem. They also add additional constraints. For example, multi-floor layouts can be infeasible if the total area of departments assigned to any floor is greater than the total area available on the floor [8].

Yet, multi-floor layouts may be necessary when single-floor facilities are either not appropriate or not applicable. Buildings with multiple floors may be necessary for situations when the cost of land is high, the amount of land is limited, or when a compact building allows for more efficient environmental or operational control [14]. Furthermore, a multi-floor facility layout may be more appropriate when renovating or performing a re-layout of an old multi-floor building—especially when the cost of building a new facility exceeds the cost of renovation [8].

Some facility layout problem literature explores extensions to the QAP, MIP, and graph-theoretic approaches with respect to the multi-floor facility layout. Extensions might vary assumptions of department sizes, facility size, and elevators.

Hahn and Krarup review the history of a QAP formulation applied to a multi-floor hospital facility [15]. Their model assumes a discrete number of equal-sized hospital departments in addition to an upper bound on the length, width, and number of floors. Thus,

along with the decision of where to effectively assign the hospital's departments between floors and on each floor, their model must decide the number of floors and the size of each floor.

Later heuristics including simulated annealing, tabu search, and branch and bound have been used to find solutions to layout problems with larger numbers of departments. Hahn and Krarup's formulation does not specify a flow path to the next floor via stairs or an elevator. Thus, the layout formulation does not encompass all the issues within a multi-floor facility layout.

To address the case where department shapes are not equal, Bozer et al. extend two single-floor facility layout algorithms to create a multi-floor facility layout approach named MULTIPLE [8]. Their research combines two well known algorithms: CRAFT and the space-filling curve. CRAFT begins with an initial layout and performs exchanges between departments to reduce the layout cost. Since CRAFT is only capable of exchanging departments adjacent to one another and equal in area, Bozer et al. combine it with the space-filling curve in an attempt to increase the number of department exchanges and allow flexible departmental area requirements.

The space-filling curve method divides the possible location assignment area into a matrix of many small equal-sized locations and assigns them an order. This order assignment should make any subset of sequenced locations sufficiently adjacent to be combined to form the area of a reasonably compact department. The "space-filling curve" is represented by a line that passes through the center of each location in their assigned order. The space-filling curve generates a layout represented by the matrix. Each department is assigned an integer number of locations with respect to its lower and upper area bound. This algorithm can be solved quicker because the department assignments are simply defined by ordering

the departments. The department ordered first occupies the first set of locations (the set size large enough to satisfy the departments size requirements), and the next department is assigned the next set of locations, etc., until all the departments are assigned a sufficient number of locations. Since departments are allocated along the curve and according to their sequence, departments are never split and non-adjacent exchanges can easily be made by altering the curve sequence [8]. The shape of each department is constrained by a perimeter to area ratio in order to avoid non-acceptable department shapes which can result from the non-adjacent exchanges and the flexible area requirements.

MULTIPLE is capable of restricting departments to certain floors and having fixed floors. However, it assumes that the floor dimensions, number of floors, and number and positions of elevators are known *a priori*. Furthermore, the department locations must be considered approximate since they are made up of a set of discrete, equal-sized squares. These assumptions do not encompass many of the integrated issues that are involved in a multi-floor facility layout, yet they do result in a much simpler computation. Even for relatively large problems, the computation time with efficient algorithms is small. If MULTIPLE used a MIP formulation, like the next three approaches, the computation time would be much larger.

Lee et al. present the multi-floor facility layout problem with inner walls and passages as a combination of a mixed-integer programming model and a graph-theoretic model [21]. Such a model can be used for the re-layout process of an old building or the layout of a new building in which inner walls that outline existing rooms and hallways are present and fixed. Given the initial layout skeleton of the facility and several assumptions that simplify the problem, an optimal layout is pursued by allocating departments within the layout skeleton with respect to the multi-objective function. The multi-objective problem attempts

to minimize the material handling cost and maximize the interdepartmental adjacencies. Since multi-objective problems and mixed-integer problems both increase the complexity of the assignment problem, many assumptions are made to simplify it. Some of the primary assumptions include interdepartmental material flow values, interdepartmental relationship values, the number of floors, the number of elevators, the number, positions and sizes of elevators, and the boundary shape for each floor. Consequently, the assumptions result in a model that may not represent the fully integrated multi-floor facility layout. However, even with the assumptions, the multi-objective function leads to a large-scale MIP problem. Lee et al. test the model by applying a genetic algorithm to a multi-deck compartment ship and comparing the layout solution with an existing compartment layout of an actual ship.

Goetschalckx and Irohara also apply a mixed-integer programming model to the multi-floor facility layout problem [14]. In particular, Goetschalckx and Irohara attempt to create efficient formulations for multi-floor facility layout problems with elevators. Similar to MULTIPLE, the position of elevators is a decision variable; yet unlike MULTIPLE and any other model reviewed thus far, the number of elevators is also a decision variable. Goetschalckx and Irohara investigate two formulations of the problem with elevators: one in which full-service elevators stop at all floors and another in which some full-service elevators can stop at all floors in addition to less expensive elevators that do not provide service to all floors. For each formulation, the assumed parameters include the number of floors, the maximum number of elevators, the length and width of each department, and the material flow and cost between departments. The model assumes lower and upper bounds for the length and width of each floor. This allows each floor to be a different size as long as the length and width of each floor is equal to or shorter than the floor directly below it.

With these additional decision variables, the formulation results in a large mixed-integer

model that is very difficult and time consuming to solve optimally. Goetschalckx and Irohara alter the formulations to eliminate symmetry and utilize valid lower bounds on distances in order to reduce computation times. By testing different alterations of the formulations, Goetschalckx and Irohara were able to find the most effective combinations of symmetry-breaking constraints. Finally, by comparing the two elevator models, Goetschalckx and Irohara show that the second model with non-full-service elevators is more economical.

Though elevators are not incorporated into the optimal multi-floor process plant layout introduced by Patsiatzis and Papageorgiou, they alter the cost objective function to improve the representation of all the layout costs that are present in a multi-floor facility [29]. The objective of their model is to determine the number of floors, land area, equipment-floor allocation and detailed layout of each floor so as to minimize the total plant layout cost. The total plant layout cost includes the basic material handling cost function that is used by all facility layout problems (the product of interdepartmental flows, cost of traveling between departments, and distances between each department) and the additional total fixed cost of all floors (fixed cost per floor multiplied by the total number of floors), the total area dependent construction cost (area dependent floor construction cost multiplied by the number of floors and the area of each floor), and the total cost of land for the facility layout (area of facility multiplied by the cost of the land). Since the decision variables include the number of floors, the allocation of departments, and the length and width of the facility (or area), the formulation results in a mixed-integer, non-linear program. Patsiatzis and Papageorgiou make the formulation linear by providing a set of possible facility areas. They then apply the linear formulation to three process plant examples as a test.

The model we propose borrows much from the formulations presented by Patsiatzis and Papageorgiou [29] and Goetschalckx and Irohara [14]. Similar to Goetschalckx and

Irohara, our formulation includes the number and position of elevators as a decision variable. Additionally, and also similar to Goetschalckx and Irohara (and exclusive to them among the methods we reviewed), elevators in our formulation will have zero area. Other key characteristics of our formulation are modeled after Patsiatzis and Papageorgiou's model of the process plant.

We have altered our objective function to incorporate the total layout cost. The total layout cost is the sum of the material handling cost, the land cost of the facility area, the total construction cost per floor, and the total cost of the elevators. Thus, the decisions will be to determine the number of elevators, the position of elevators, the number of floors, the size of floors, the size of departments, and the allocation of departments. This results in a mixed-integer, non-linear program that can be approximated with linearization. Table 3.1 compares the flexibility of our model to the other models we reviewed.

	CRAFT	Hahn and Krarup	MULTIPLE	Lee, Han and Roh	Goetschalckx and Irohara	Patsiatzis and Papageorgiou	**Our Formulation**
Discrete Department Locations	X	X	X				
Continuous Department Locations				X	X	X	X
Fixed Equal Department Shape	X	X					
Fixed Unequal Department Shape					X	X	
Variable Discrete Department Shape			X				
Variable Continuous Department Shape				X			X
Single Floor	X						
Fixed Number of Floors			X	X	X	X	
Variable Number of Floors		X					X
No elevators	X	X				X	
Fixed Number of Elevators			X	X			
Variable Number of Elevators					X		X
No elevators	X	X				X	
Fixed Elevator Location			X	X			
Variable Continuous Elevator Location					X		X
Fixed Facility Size	X	X	X	X	X		
Variable Continuous Facility Size						X	X

Table 3.1: Comparison of FLP models.

4 Multi-Floor Facility Layout Model

4.1 Problem Statement

An optimal facility layout which includes multiple floors is practical when land is scarce, when a compact facility allows for more efficient control, or when renovating an already existing facility. The decision objective for a multi-floor facility layout is to minimize the facility total flow cost and layout cost. Finding an optimal facility layout so as to minimize these costs requires the following decisions: the number of floors, the land area required by the facility, the number of elevators within the facility, the length and width of each department, the department and elevator allocation to each floor and placement within each floor, and the assignment of flows to elevators.

4.2 Assumptions

- Facility floors have a rectangular shape.
- Each facility floor has an identical length, width, and height.
- Vertical travel between floors can only occur through elevators. All elevators are vertically bidirectional. Elevators cannot move horizontally.

- Elevator capacity is not considered.
- Elevators have zero area.
- All departments have a rectangular shape and height equal to the floor height.
- All department area lower bounds are specified as parameters.
- A department cannot be split among multiple floors.
- Elevators cannot be located within a department; however, elevators may be located on the boundary of a department(s).
- Movement between departments (and elevators) is modeled as rectilinear centroid-to-centroid movement. Movement is not explicitly modeled through an aisles network and the movement aisles do not consume any area.
- Material flows between departments are assumed.
- Fixed material flow costs and layout costs are assumed.

4.3 Notation

4.3.1 Indices

i, j	Departments; $\{i = j = 1, 2, 3, \ldots, N\}$.
k	Floors; $\{k = 1, 2, 3, \ldots, M\}$.
e	Elevators; $\{e = 1, 2, 3, \ldots, E^{MAX}\}$.

4.3.2 Parameters

N	Number of departments (integer).
M	Maximum number of floors (integer).
E^{MAX}, E^{MIN}	Maximum, minimum number of elevators (integer).
L, W	Maximum length and width of the facility.
F_{ij}	Material handling flow between departments i and j; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.
C_{ij}^{H}	Horizontal transportation cost per unit distance between departments i and j; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.
C_{ij}^{V}	Vertical transportation cost per unit distance between departments i and j; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.
A_i	Lower bound of the area for department i; $\{i = 1, \ldots, N\}$.
S_i	Lower bound of the length of any side for department i (i must be > 0); $\{i = 1, \ldots, N\}$.
H	Floor height.
C^{L}	Cost placed on making the facility one unit greater in length.
C^{W}	Cost placed on making the facility one unit greater in width.
C^{H}	Cost placed on adding floors to the facility.

C^E Cost placed on adding elevators to the facility.

4.3.3 Binary Variables

v^E_{ije} 1 if traffic between departments i and j travels through elevator e, 0 otherwise; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N; e = 1, \ldots, E^{MAX}\}$.

v_{ik} 1 if department i is assigned to floor k, 0 otherwise; $\{i = 1, \ldots, N; k = 1, \ldots, M\}$.

z_{ij} 1 if departments i and j are assigned to the same floor, 0 otherwise; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.

t^X_{ij} 1 if departments i and j do not overlap along the x-axis and i has x-coordinate closer to the origin; $\{i = 1, \ldots, N; j = 1, \ldots, N; i \neq j\}$.

t^Y_{ij} 1 if departments i and j do not overlap along the y-axis and i has a y-coordinate closer to the origin; $\{i = 1, \ldots, N; j = 1, \ldots, N; i \neq j\}$.

q^{XL}_{ie}, q^{XR}_{ie} 1 if the department i does not overlap elevator e along the x-axis and is either to the left (xL) or to the right (xR) of elevator e; $\{i = 1, \ldots, N; e = 1, \ldots, E^{MAX}\}$.

q^{YB}_{ie}, q^{YA}_{ie} 1 if the department i does not overlap elevator e along the y-axis and is either below (YB) or above (YA) elevator e; $\{i = 1, \ldots, N; e = 1, \ldots, E^{MAX}\}$.

p_e 1 if elevator e is utilized in the solution, 0 otherwise; $\{e = 1, \ldots, E^{MAX}\}$.

4.3.4 Continuous Variables

d_{ij}^H Horizontal rectilinear distance between the centroids of departments i and j $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.

d_{ij}^V Vertical distance between the centroids of departments i and j; $\{i = 1, \ldots, N-1; j = i+1, \ldots, N\}$.

d_{ie}^E Horizontal rectilinear distance between the centroid of departments i and elevator e; $\{i = 1, \ldots, N; e = 1, \ldots, E^{MAX}\}$.

b^L, b^W Length and width of the facility along an arbitrary x and y axis.

l_i, w_i Length and width of department i along an arbitrary x and y axis; $\{i = 1, \ldots, N\}$.

x_i, y_i The x and y coordinates for the centroid of department i; $\{i = 1, \ldots, N\}$.

e_e^X, e_e^Y The x and y coordinates of elevator e; $\{e = 1, \ldots, E^{MAX}\}$.

4.3.5 Integer Variables

u Number of floors used in the solution.

r Number of elevators used in the solution.

4.4 Objective Function

$$\text{Minimize} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} [F_{ij}(C_{ij}^{H} d_{ij}^{H} + C_{ij}^{V} d_{ij}^{V})] + C^{L} b^{L} + C^{W} b^{W} + C^{H} u + C^{E} r \tag{4.1}$$

The overall objective function for the multiple floor facility is shown in (4.1). The first term in the objective function is the sum of the transportation costs between each pair of departments weighted by the flow between each pair of departments. The transportation cost between each set of departments is equal to the sum of the horizontal distance multiplied by the horizontal transportation cost and the vertical distance multiplied by the vertical transportation cost. The second, third, and fourth terms represent the facility costs. The second and third terms represent the costs of making the building greater in length and width respectively. The fourth term represents the cost of adding floors to the facility. The fifth term represents the cost of adding elevators to the facility. Note that the facility costs could readily be converted to a monetary value, whereas it would be much more difficult to estimate a monetary value for the flow costs.

4.5 Floor Constraints

$$\sum_{k=1}^{M} v_{ik} = 1 \qquad \forall i = 1, \ldots, N \tag{4.2}$$

$$z_{ij} \geq v_{ik} + v_{jk} - 1 \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; k = 1, \ldots, M \tag{4.3}$$

$$z_{ij} \leq 1 - v_{ik} + v_{jk} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; k = 1, \ldots, M \tag{4.4}$$

$$z_{ij} \leq 1 + v_{ik} - v_{jk} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; k = 1, \ldots, M \tag{4.5}$$

$$u \geq k v_{ik} \qquad \forall i = 1, \ldots, N; k = 1, \ldots, M \tag{4.6}$$

$$u \leq M \tag{4.7}$$

$$u \geq 1 \tag{4.8}$$

Constraint (4.2) assures that each department is assigned to only one floor. Constraints (4.3), (4.4), and (4.5) constrain the variable z_{ij}. If two departments i and j are allocated to the same floor ($v_{ik} = v_{jk}$), then z_{ij} will be equal to one by constraint (4.3), while constraints (4.4) and (4.5) remain inactive. Conversely, if departments i and j are allocated to different floors, then constraint (4.3) is inactive and constraints (4.4) and (4.5) ensure the value of z_{ij} is zero. Constraints (4.6), (4.7), and (4.8) constrain the number of floors used in the solution, which must be greater than or equal to the floor number that each department is assigned to, less than or equal to the maximum number of floors specified, and at least one.

4.6 Elevator Constraints

$$r = \sum_{e=1}^{E^{MAX}} p_e \tag{4.9}$$

$$r \geq E^{MIN} \tag{4.10}$$

$$r \leq E^{MAX} \tag{4.11}$$

$$\sum_{e=1}^{E^{MAX}} v_{ije}^{E} = 1 - z_{ij} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.12}$$

$$p_e \geq v_{ije}^{E} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.13}$$

Constraint (4.9) defines the total number of elevators used in the solution as the total number of elevators being utilized. Constraints (4.10) and (4.11) require the total elevators used in the solution to be at least as great as the minimum number specified but less than the maximum number specified as a parameter. Each pair of departments that are not assigned to the same floor are required by (4.12) to route their flow through an elevator. All elevators that have flows routed through them must be considered actively used in the solution by (4.13).

4.7 Department Dimension Constraints

$$l_i \geq S_i \qquad \forall i = 1, \ldots, N \tag{4.14}$$

$$w_i \geq S_i \qquad \forall i = 1, \ldots, N \tag{4.15}$$

$$l_i \leq b^L \qquad \forall i = 1, \ldots, N \tag{4.16}$$

$$w_i \leq b^W \qquad \forall i = 1, \ldots, N \tag{4.17}$$

$$l_i w_i \geq A_i \qquad \forall i = 1, \ldots, N \tag{4.18}$$

Constraints (4.14) through (4.17) require each department's length and width to be between the specified minimum side length for that department and the maximum length and width of the facility. Constraint (4.18) requires that each department's area be greater than the minimum department area specified for each department.

4.8 Non-Overlapping Constraints

$$t^X_{i,j} + t^X_{j,i} + t^Y_{i,j} + t^Y_{j,i} = z_{ij} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; i \neq j \tag{4.19}$$

$$x_j - x_i \geq \frac{l_i + l_j}{2} - L(1 - t^X_{i,j}) \qquad \forall i = 1, \ldots, N; j = 1, \ldots, N; i \neq j \tag{4.20}$$

$$y_j - y_i \geq \frac{w_i + w_j}{2} - W(1 - t^Y_{i,j}) \qquad \forall i = 1, \ldots, N; j = 1, \ldots, N; i \neq j \tag{4.21}$$

$$q^{XL}_{ie} + q^{XR}_{ie} + q^{YB}_{ie} + q^{YA}_{ie} = 1 \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.22}$$

$$e^X_e - x_i \geq \frac{l_i}{2} - L(1 - q^{XL}_{ie}) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.23}$$

$$x_i - e^X_e \geq \frac{l_i}{2} - L(1 - q^{XR}_{ie}) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.24}$$

$$e^Y_e - y_i \geq \frac{w_i}{2} - W(1 - q^{YB}_{ie}) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.25}$$

$$y_i - e^Y_e \geq \frac{w_i}{2} - W(1 - q^{YA}_{ie}) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.26}$$

Constraints must be activated to prevent departments from occupying the same physical location. Constraint (4.19) forces one of the two constraints (4.20) or (4.21) to be active if two departments are located on the same floor. Constraint (4.20) prevents departments from overlapping on the x-axis, while (4.22) prevents departments from overlapping on the y-axis. Therefore departments on the same floor may be located next to one another but will not overlap.

Similar to preventing departments from occupying the same physical location, it is necessary to prevent elevators from occupying the same location as a department. Though our model assumes elevators do not occupy any space, it restricts their placement to only department boundaries, not to department interiors. Constraints (4.22) through (4.26) prevent the elevator from overlapping departments in the same manner as departments are prevented from overlapping. Constraint (4.22) forces one of the four constraints (4.23),

(4.24), (4.25), or (4.26) to be active. Constraints (4.23) and (4.24) prevent the elevator from overlapping each department on the x-axis, while (4.25) and (4.26) prevent the elevator from overlapping each department on the y-axis. Thus, the elevator will not overlap any department.

4.9 Distance Constraints

$$d_{ij}^V \geq H \sum_{k=1}^{M} k(v_{ik} - v_{jk}) \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.27}$$

$$d_{ij}^V \geq H \sum_{k=1}^{M} k(v_{jk} - v_{ik}) \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.28}$$

$$\begin{aligned} d_{ij}^H &\geq ((x_i - x_j) + (y_i - y_j)) - (1 - z_{ij})(L + W) \\ &\forall i = 1, \ldots, N-1; j = i+1, \ldots, N \end{aligned} \tag{4.29}$$

$$\begin{aligned} d_{ij}^H &\geq ((x_j - x_i) + (y_j - y_i)) - (1 - z_{ij})(L + W) \\ &\forall i = 1, \ldots, N-1; j = i+1, \ldots, N \end{aligned} \tag{4.30}$$

$$\begin{aligned} d_{ij}^H &\geq ((x_i - x_j) + (y_j - y_i)) - (1 - z_{ij})(L + W) \\ &\forall i = 1, \ldots, N-1; j = i+1, \ldots, N \end{aligned} \tag{4.31}$$

$$\begin{aligned} d_{ij}^H &\geq ((x_j - x_i) + (y_i - y_j)) - (1 - z_{ij})(L + W) \\ &\forall i = 1, \ldots, N-1; j = i+1, \ldots, N \end{aligned} \tag{4.32}$$

$$d_{ie}^E \geq ((x_i - e_e^X) + (y_i - e_e^Y)) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.33}$$

$$d_{ie}^E \geq ((e_e^X - x_i) + (e_e^Y - y_i)) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.34}$$

$$d_{ie}^{E} \geq ((x_i - e_e^X) + (e_e^Y - y_i)) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.35}$$

$$d_{ie}^{E} \geq ((e_e^X - x_i) + (y_i - e_e^Y)) \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.36}$$

$$\begin{gathered} d_{ij}^{H} \geq (d_{ie}^{E} + d_{je}^{E}) - 2(z_{ij})(L + W) - 2(1 - v_{ije}^{E})(L + W) \\ \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; e = 1, \ldots, E^{MAX} \end{gathered} \tag{4.37}$$

Constraints (4.27) and (4.28) define the vertical distances between two departments i and j as the height of each floor multiplied by the number of floors separating the two departments. When the departments are located on the same floor ($V_{ik} = V_{jk}$), the value of the vertical distance is zero.

Since two departments i and j can either occupy the same floor or different floors, and an elevator must only be used if the departments i and j occupy different floors, there are two constraint sets for the horizontal rectilinear distances. For the first case, when the departments i and j are assigned to the same floor ($z_{ij} = 1$), an elevator does not need to be utilized; thus, the horizontal distance between the two departments is the rectilinear distance between the centroids of the two departments i and j. Constraints (4.29) through (4.32), which define the rectilinear distance between two departments, are active in this case but inactive when the departments i and j are assigned to different floors ($z_{ij} = 0$). For the second case, when departments i and j are assigned to different floors, the horizontal distance between the two departments is dependent on the rectilinear distance between the elevator and each department. The distance between department i and each elevator is obtained from constraints (4.33) through (4.36). Then (4.37) sets the vertical distance between the two departments equal to the sum of the rectilinear distance between each

department and the elevator determined by (4.33) through (4.36). This constraint is only active when departments i and j are assigned to different floors ($z_{ij} = 0$) and only when evaluating the elevator used to transport flow between the two departments ($v^E_{ije} = 1$). Accordingly, the horizontal rectilinear distance for any two departments i and j is either (1) the rectilinear distance between the centroids of the two departments, if the departments are assigned to the same floor, or (2) the total rectilinear distance from department i to the elevator and from the elevator to department j when the two departments are assigned to different floors.

4.10 Facility Bounding Constraints

$$x_i \geq \frac{1}{2} l_i \qquad \forall i = 1, \ldots, N \tag{4.38}$$

$$y_i \geq \frac{1}{2} w_i \qquad \forall i = 1, \ldots, N \tag{4.39}$$

$$b^L \geq x_i + \frac{1}{2} l_i \qquad \forall i = 1, \ldots, N \tag{4.40}$$

$$b^W \geq y_i + \frac{1}{2} w_i \qquad \forall i = 1, \ldots, N \tag{4.41}$$

$$e^X_e \geq 0 \qquad \forall e = 1, \ldots, E^{MAX} \tag{4.42}$$

$$e^Y_e \geq 0 \qquad \forall e = 1, \ldots, E^{MAX} \tag{4.43}$$

$$e^X_e \leq b^L \qquad \forall e = 1, \ldots, E^{MAX} \tag{4.44}$$

$$e^Y_e \leq b^W \qquad \forall e = 1, \ldots, E^{MAX} \tag{4.45}$$

$$b^L \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.46}$$

$$b^W \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.47}$$

$$b^L \leq L \qquad \forall i = 1, \ldots, N \tag{4.48}$$

$$b^W \leq W \qquad \forall i = 1, \ldots, N \tag{4.49}$$

Constraints (4.38) and (4.39) require each department to be located within the lower bounds of the facility. Constraints (4.40) and (4.41) require each department to be located within the upper bounds of the facility. The elevators are similarly bounded by constraints (4.42) through (4.45). Constraints (4.46) through (4.49) constrain the facility to the non-negative region less than the upper bounds specified as parameters.

4.11 Additional Constraints

$$d_{ij}^H \geq 0 \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.50}$$

$$d_{ij}^V \geq 0 \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.51}$$

$$d_{ie}^E \geq 0 \qquad \forall i = 1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.52}$$

$$x_i \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.53}$$

$$y_i \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.54}$$

$$l_i \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.55}$$

$$w_i \geq 0 \qquad \forall i = 1, \ldots, N \tag{4.56}$$

$$b^L \geq 0 \tag{4.57}$$

$$b^W \geq 0 \tag{4.58}$$

$$v_{ije}^E \in \{0, 1\} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N; e = 1, \ldots, E^{MAX} \tag{4.59}$$

$$v_{ik} \in \{0, 1\} \qquad \forall i = 1, \ldots, N; k = 1, \ldots, M \tag{4.60}$$

$$z_{ij} \in \{0, 1\} \qquad \forall i = 1, \ldots, N-1; j = i+1, \ldots, N \tag{4.61}$$

$$t_{ij}^{X} \in \{0,1\} \qquad \forall i = 1,\ldots,N; j = 1,\ldots,N \tag{4.62}$$

$$t_{ij}^{Y} \in \{0,1\} \qquad \forall i = 1,\ldots,N; j = 1,\ldots,N \tag{4.63}$$

$$q_{ie}^{XL} \in \{0,1\} \qquad \forall i = 1,\ldots,N; e = 1,\ldots,E^{MAX} \tag{4.64}$$

$$q_{ie}^{XR} \in \{0,1\} \qquad \forall i = 1,\ldots,N; e = 1,\ldots,E^{MAX} \tag{4.65}$$

$$q_{ie}^{YB} \in \{0,1\} \qquad \forall i = 1,\ldots,N; e = 1,\ldots,E^{MAX} \tag{4.66}$$

$$q_{ie}^{YA} \in \{0,1\} \qquad \forall i = 1,\ldots,N; e = 1,\ldots,E^{MAX} \tag{4.67}$$

$$p_e \in \{0,1\} \qquad \forall e = 1,\ldots,E^{MAX} \tag{4.68}$$

$$u \in \mathbb{Z} \tag{4.69}$$

$$r \in \mathbb{Z} \tag{4.70}$$

Constraints (4.50) through (4.52) require the horizontal and vertical rectilinear distances between any two departments, as well as the horizontal distance between a department and an elevator, to be non-negative. Constraints (4.53) through (4.58) require the length and width of the building, the length and width of each department, and the centroid of each department to be non-negative. All binary variables are restricted to the binary values of one or zero. The integer variables—the number of floors and elevators used in the solution—are restricted to integer values.

4.12 Linearization

Constraint (4.18) requires the area of each department to be at least as great as the minimum area parameter specified for that department; however, the area calculation is not linear. Linear models are easier to solve than non-linear models and, due to the size of this model, the ability to solve a meaningfully sized instance is a relevant concern. Therefore, when

solving the model, we will substitute for the non-linear department area constraint a set of linear approximation constraints shown below:

$$l_i + w_i \geq (2\sqrt{A_i}) + (l_i - w_i)\frac{A_i/S_i + S_i - 2\sqrt{A_i}}{A_i/S_i - S_i + 0.001} \qquad \forall i = 1, \ldots, N \tag{4.71}$$

$$l_i + w_i \geq (2\sqrt{A_i}) + (w_i - l_i)\frac{A_i/S_i + S_i - 2\sqrt{A_i}}{A_i/S_i - S_i + 0.001} \qquad \forall i = 1, \ldots, N \tag{4.72}$$

These constraints approximate the curved graph of the area with two straight lines. We have graphed the original non-linear constraint, along with these two new constraints, in a plot of the area in Figure 4.1. The horizontal axis of this graph represents the length of a department and the vertical axis represents the width of a department. The lower bound of feasible length-width combinations is a red curved line. This line represents the nonlinear constraint (4.18). The length-width combinations below this red line are infeasible because the resulting area will be less than the minimum area required. The purple line represents constraint (4.71) and the green line represents constraint (4.72). In place of the red line, departments with length-width combinations below this line have insufficient area and are infeasible. Notice that a department's length or width must be greater than the minimum side length parameter S_i. Also, due to the nature of the optimization problem, it is unlikely that a side will exceed the value of A_i/S_i since this would result in a department area that is larger than the minimum requirements. Within this region $[S_i, A_i/S_i)$ bounding each side length, the linear constraints are reasonably accurate approximations of the nonlinear constraint.

Some department sizes meet the minimum side and area requirements but will still be infeasible due to the error in this approximation. The shaded region on the graph represents these length-width combinations. The solution's department sizes, however, will always be

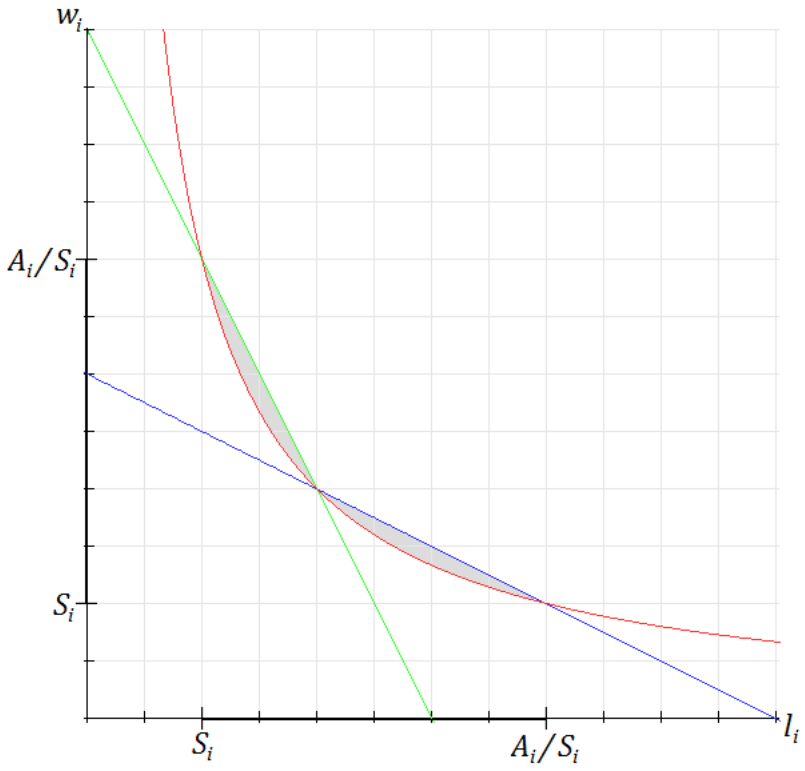

Figure 4.1: Graph of the department area constraints.

equal to or greater than the minimum area parameter.

To avoid dividing by 0 in these constraints, the minimum department side length parameter S_i must be greater than 0, which is realistic. Also, $A_i/S_i - S_i$ must be greater than 0. This could occur if the minimum area assigned to a department is exactly equal to the square of the minimum length assigned to that department. To circumvent this scenario, we add a very small value to the denominator.

4.13 Balancing Multiple Objectives

The terms in the objective function can logically be divided into two different objectives: minimizing the interdepartmental flows (represented by the first group of terms) and minimizing the facility size variables (the last four terms). These two objectives cannot easily

be equated in a single unit. The terms for the facility construction costs could be in a monetary unit, while the interdepartmental flows are primarily an efficiency indicator that would be hard to meaningfully transform into a monetary unit.

This makes balancing the costs placed on these two conflicting objectives difficult. Placing too much weight on the interdepartmental flows results in an efficient layout in terms of interdepartmental flows, but the department areas do not efficiently utilize the facility's space. For example, all departments might be on one floor or—depending on the vertical transportation cost parameters—a floor might be created for each department (with lots of unutilized space). On the other hand, too much weight placed on the facility construction costs results in the departments being arranged to take up the smallest area possible with little regard to the efficiency of interdepartmental flows. While any of these solutions are Pareto optimal, and a balance between the two objectives can be struck with careful parameter calibration, this balance would likely be arbitrary and not meaningful.

We employed the method of lexicographic ordering (LO) to balance multiple objectives in a more meaningful manner. In substance, LO solves a sub-optimization problem for each objective, considering a single objective at a time [17]. Each objective's optimal value is then used as a benchmark to constrain the optimality of the solution with respect to that objective.

In the case of two different objectives, LO first solves a sub-optimization problem that only considers the first objective. The solution is then incorporated into the final model in the form of a constraint. This constraint requires that the solution of the final model be within some range of optimality with respect to the first objective. The first objective is then removed from the objective function of the final model so that it only considers the second objective.

In our model, we have chosen the objective of minimizing the facility size to be optimized first. This is an intuitive choice since we primarily want a facility size just large enough to house all departments. However, during the second optimization the model will be given some slack with regard to the facility size costs, so that the departments can be moved around to optimize interdepartmental flows. For example, the result of the sub-optimization may be that the smallest possible facility that can feasibly contain all of the departments costs 10 million units. This value can be used to limit the final model to determine a facility layout that does not exceed, for example, 11 million units. In other words, the solution must optimize the facility size to within 10% of optimal for this objective.

A few changes must be made to incorporate the LO method into our model. For the sub-optimization problem, the terms that represent the objective of minimizing interdepartmental flows must be removed from the objective function. The rest of the model remains the same. This results in the new objective below:

$$\text{Minimize } C^L b^L + C^W b^W + C^H u + C^E r \tag{4.73}$$

While this is the only change that must be made, we are only interested in determining the smallest facility size possible, so many variables and constraints are no longer necessary. Therefore, we remove them to reduce the model complexity and the time required to solve. The number of elevators will never exceed one—more elevators are not strictly necessary and would increase costs—so the variable r can be considered binary. The number of elevators will equal 1 if there are multiple floors and 0 otherwise. This can be modeled by adding the

constraint below to the sub-optimization model:

$$999r - u \geq -1 \tag{4.74}$$

The parameters for the minimum and maximum number of elevators are no longer needed and the location where the one elevator is assigned does not matter. The flow between the departments and elevators no longer needs to be considered so F_{ij} can be removed. Also, the parameters for the horizontal and vertical transportation costs between each set of departments can be removed. The variables v_{ije}^{E}, q_{ie}^{XL}, q_{ie}^{XR}, q_{ie}^{YB}, q_{ie}^{YA}, x_i, y_i, and p_e represent the use and assignment of elevators and the assignment of flow to elevators, and prevent departments and elevators from overlapping. They are used in the Elevator Constraints (Section 4.6), Non-Overlapping Constraints (Section 4.8) and the Facility Bounding Constraints (Section 4.10, (4.44) and (4.45)). These aspects of the model are not considerations and the variables and constraints can be removed from the sub-optimization model. Similarly, we use the variables d_{ij}^{H}, d_{ij}^{V}, and d_{ie}^{E} to represent the distances between departments and elevators and to calculate flow costs. These variables are no longer necessary and can be removed along with all of the Distance Constraints (Section 4.9), which bound these variables.

The second optimization model is the same as the original model, but the objective function only contains the terms that represent the flow costs:

$$\text{Minimize} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} F_{ij}(C_{ij}^{H} d_{ij}^{H} + C_{ij}^{V} d_{ij}^{V}) \tag{4.75}$$

However, the results of the sub-optimization problem are incorporated into the second

optimization model by adding the following constraint:

$$C^L b^L + C^W b^W + C^H u + C^E r \leq (1+\gamma) g_{sub} \tag{4.76}$$

Where g_{sub} represents the objective function value of the solution to the sub-optimization problem and γ represents a slack parameter that determines how close to optimal the solution must be for the first objective. This assures that the solution to the second optimization problem will be as close to optimal for the first objective as we require. The parameter γ can be increased for a larger building with more efficient interdepartmental flows, or decreased for a more compact building with less efficient interdepartmental flows.

5 Hospital Layout Results

This model was created using Python 2.6.5 to interact with the Gurobi Optimizer version 3.0. The Gurobi Optimizer is designed to utilize multiple processing cores. All computations were executed on a personal desktop computer running the Microsoft Vista 32-bit operating system on an Intel Core 2 Quad Q6600 CPU, with 4 physical cores at 2.40 GHz, and 4 GB of RAM.

5.1 Scenario Parameters

We developed a sample scenario to explore computation times using the Gurobi Optimizer software and demonstrate the model. The parameters accommodate a scenario of up to seven departments. (In experiments not documented here, scenarios with eight or more departments required days or weeks of computation time.) For scenarios of six departments, the parameters for the seventh department will be removed. The units of length are meters, the units of facility cost are in ten thousands of dollars, and the units of flow are not defined.

The minimum area of each department was generated randomly using the Microsoft Excel function RANDBETWEEN(700, 5000) which returns a random integer between 700 and 5000 inclusive. The minimum side length parameter for each department was generated using the same function with the upper bounds set to the square root of the area for that

department and the lower bounds set to half the value of the upper bounds. This assures that each department has a feasible minimum side length and that the aspect ratio of each department is no greater than 4:1. These values are shown in Table 5.1.

	Dept. A	Dept. B	Dept. C	Dept. D	Dept. E	Dept. F	Dept. G
Minimum side length:	24	63	42	52	33	30	36
Minimum area:	797	4180	3001	4194	3669	1015	4185

Table 5.1: Scenario department size parameters.

The maximum length and width of the facility parameter was determined by the sum of each department's minimum area divided by that department's minimum side length. This is an upper bound for the possible length of the facility: it is the length of the facility if all the departments were rectangular and lined up lengthwise, and therefore is not a constraining value. The floor height was set to four. The cost of a meter of facility length or width was set to six. The cost of floors and elevators was set to 500 and 30 respectively.

The horizontal cost per unit distance parameter matrix was populated using the Excel function RANDBETWEEN(0, 100) to create a relatively large variance between the flows between different departments. The vertical and horizontal transportation cost per unit distance matrices were populated using the Excel function RANDBETWEEN(10, 20) to create a smaller variance. These parameters can be found in Appendix F.

5.2 Sample Layout Output

```
  ***  LayoutC7 PART I  ***

Cost:                   2259.0
# Floors:                  1.0
# Elevators:               0.0
Facility Length:         137.0
Facility Width:          156.2

DEPT.  FLOOR       CENTROID         LENGTH    WIDTH
  1      1    (  14.1 ,  14.1 )      28.2      28.2
  2      1    (  33.3 ,  61.5 )      66.7      63.0
  3      1    ( 101.8 ,  21.3 )      70.3      42.5
  4      1    (  33.3 , 124.6 )      66.7      63.2
  5      1    (  83.2 , 100.7 )      33.0     111.0
  6      1    (  45.7 ,  15.0 )      34.9      30.0
  7      1    ( 118.3 ,  99.3 )      37.3     113.6

DEPT.   AREA          MIN_AREA    ERROR
  1      797.0          797          0.0 %
  2     4200.9         4180          0.5 %
  3     2991.0         3001         -0.3 %
  4     4211.1         4194          0.4 %
  5     3663.0         3669         -0.2 %
  6     1046.6         1015          3.1 %
  7     4241.4         4185          1.3 %
```

Figure 5.1: Example output of sub-optimization model.

These parameters were entered into a Python program that interacts with the Gurobi solver. For this example, all seven departments are included, and the maximum number of floors is set to two with the maximum number of elevators at one. The output in Figure 5.1 describes the layout resulting from the sub-optimization model.

The centroid, length, and width of each department are specified. The area of each department and the percent they are above or below the minimum area parameter are also displayed. This is a measure of the accuracy of the linear approximation constraints. The largest error in this scenario was the size of department three which was ten square meters below specification. The Python program includes code to graph the solution layout. The graph produced by the program for this layout is shown in Figure 5.2.

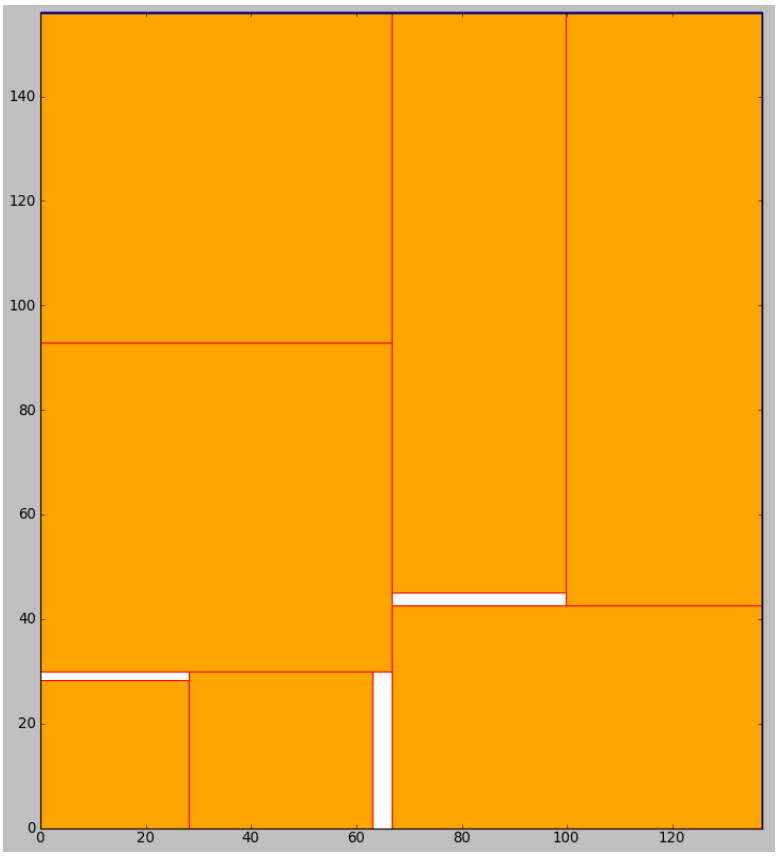

Figure 5.2: Example layout of sub-optimization model.

The sub-optimization model seeks only to minimize the cost of the facility and ignores the cost of the interdepartmental flows. The departments are compacted into the least facility cost formation possible, in this case on a single floor. This resulted in a facility cost of 2,259, the optimal cost when interdepartmental flows are ignored. This is the optimal objective function value for the objective of minimizing the facility cost. The second optimization model, which minimizes the interdepartmental flow cost, was solved with a slack parameter of 1 percent. This requires the layout produced by the second model to have a facility cost that is within 1 percent of the optimal value. The resulting layout is shown in Figure 5.4. The Python code for the sub-optimization model can be found in Appendix L and the code for the second optimization model can be found in Appendix M.

```
   ***   LayoutC7 PART II  ***

Pt. I Optimal Facility Cost:        2259.0
Upper Limit Facility Cost:          2281.6
Slack Used:                          100.0 %
Actual Facility Cost:               2281.6
Flow Cost:                       1093716.5
Flow + Facility Cost:            1095998.1

# Floors:                  1.0
# Elevators:               0.0
Facility Length:         158.0
Facility Width:          138.9

DEPT.  FLOOR       CENTROID         LENGTH    WIDTH
  1      1    (  79.4 ,  55.9 )      32.9      24.1
  2      1    (  31.5 ,  34.0 )      63.0      66.7
  3      1    (  81.8 , 103.4 )      42.0      71.0
  4      1    ( 127.0 ,  34.0 )      62.1      67.9
  5      1    ( 130.4 , 103.1 )      55.2      70.4
  6      1    (  79.4 ,  28.2 )      32.9      31.2
  7      1    (  30.4 , 103.1 )      60.8      71.6

DEPT.   AREA          MIN_AREA    ERROR
  1      792.5          797        -0.6 %
  2     4200.9         4180         0.5 %
  3     2982.0         3001        -0.6 %
  4     4218.8         4194         0.6 %
  5     3885.8         3669         5.9 %
  6     1027.1         1015         1.2 %
  7     4355.6         4185         4.1 %
```

Figure 5.3: Example output, 1% slack.

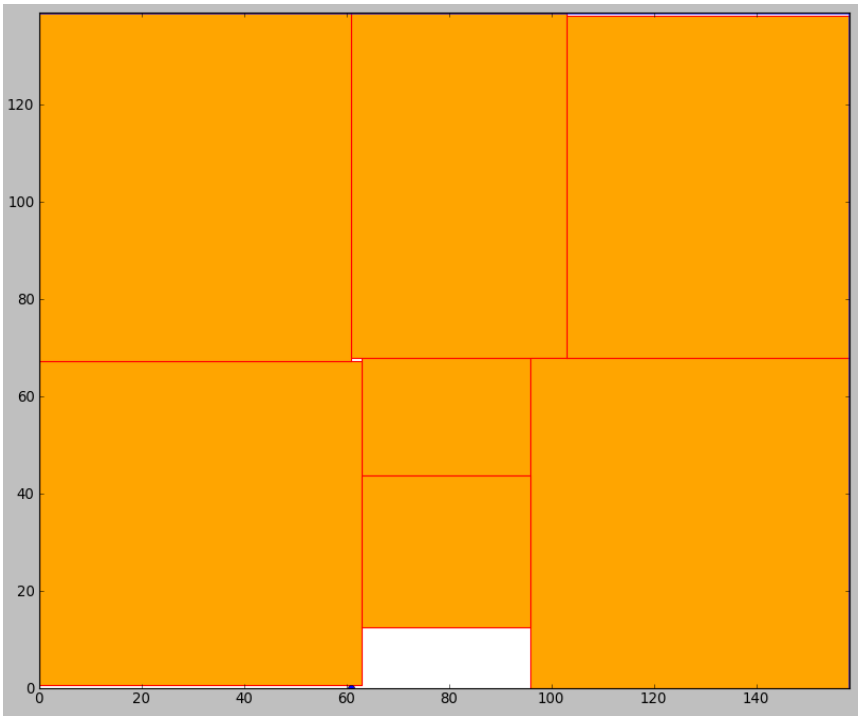

Figure 5.4: Example layout, 1% slack.

```
   ***   LayoutC7 PART II  ***

Pt. I Optimal Facility Cost:        2259.0
Upper Limit Facility Cost:          2315.5
Slack Used:                          100.0 %
Actual Facility Cost:               2315.5
Flow Cost:                         986397.0
Flow + Facility Cost:              988712.5

# Floors:                  2.0
# Elevators:               1.0
Facility Length:          103.2
Facility Width:           111.0

DEPT.  FLOOR       CENTROID         LENGTH    WIDTH
  1      1    (  12.0 ,  81.4 )     24.0      33.0
  2      1    (  32.2 ,  32.5 )     64.5      64.9
  3      2    (  35.1 ,  86.2 )     70.2      42.6
  4      2    (  35.1 ,  34.8 )     70.2      60.2
  5      2    (  86.7 ,  55.5 )     33.0     111.0
  6      1    (  39.9 ,  80.8 )     31.9      31.9
  7      1    (  83.8 ,  55.5 )     38.8     111.0

DEPT.   AREA          MIN_AREA    ERROR
  1      792.0          797        -0.6 %
  2     4182.9         4180         0.1 %
  3     2992.1         3001        -0.3 %
  4     4226.6         4194         0.8 %
  5     3663.0         3669        -0.2 %
  6     1015.0         1015        -0.0 %
  7     4306.3         4185         2.9 %

Elevator   Centroid
  1         (  35.1 ,  64.9 )

Elevator utilized to accommodate flow between departments:
   2  3  4  5  6  7
1 [0, 1, 1, 1, 0, 0]
2    [1, 1, 1, 0, 0]
3       [0, 0, 1, 1]
4          [0, 1, 1]
5             [1, 1]
6                [0]
```

Figure 5.5: Example output, 2.5% slack.

This layout uses a single floor. The facility cost of this layout is 2,281.6, while the flow cost is 1,093,716.5. For comparison, the second optimization model was solved again, except this time the slack parameter is increased to 2.5 percent. The resulting layout is shown in Figure 5.6.

This layout uses both floors. The second floor is graphed above the first floor on the vertical axis. A line separates the first floor form the second floor. The location of the

elevator is represented by a dot. This layout has a greater facility cost (2,315.5) but a lower flow cost (986,397) than the previous layout.

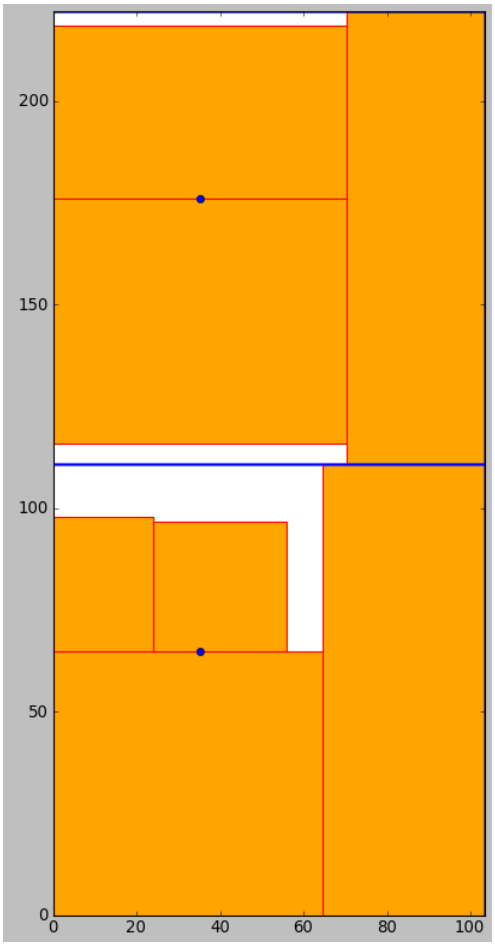

Figure 5.6: Example layout, 2.5% slack.

The parameters determine whether a multi-floor facility is more efficient. The horizontal and vertical transportation costs were varied with the same underlying distribution. Setting the horizontal transportation cost equal to or less than the vertical transportation cost tends to favor multi-floor facilities when optimizing the interdepartmental flows. The cost of additional floors, elevators, and each additional unit of facility length largely determine if a multi-floor facility is cost effective. In this case a multi-floor facility has greater facility cost and lower flow cost.

Perhaps the greatest drawback with the model's current facility cost structure is the use of a parameter for the cost of adding an additional unit of length or width to the facility. This is very unrealistic and strongly favors a square facility. This could be improved by including a linear approximation of the area of the facility and a single parameter for the construction costs per square foot—a much more common metric. Furthermore, since all floors are assumed to be equal size, the cost of an additional floor could be more accurately modeled with a fixed cost and a variable cost proportionate to the square footage of the facility.

5.3 Symmetry Breaking Constraints

We introduced and tests a simple set of symmetry-breaking constraints. Goetschalckx and Irohara demonstrated these constraints in a similar model and found them to effectively reduce solving time for some scenarios. These constraints are referred to as the "position q method" (PQM):

$$x_q \leq 0.5 b_l \tag{5.1}$$

$$y_q \leq 0.5 b_w \tag{5.2}$$

$$v_{qM} = 0 \tag{5.3}$$

These constraints do not change the model, but attempt to reduce computation time by eliminating duplicate feasible solutions. Constraints (5.1) and (5.2) restricts department q's centroid to the left-bottom quarter of the facility, where department q is the department with the largest sum of flows with other departments. This eliminates duplicate symmetric solutions about the x- and y-axis. Constraint (5.3) restricts department q from being located

on the highest floor M. This final constraint will eliminate the symmetry for the floors and will only be included when the maximum number of floors is greater than one.

5.4 Computation Times

We solved models for scenarios with five, six, and seven departments. For each number of departments, the maximum number of floors and the maximum number of elevators were varied between two sets of values. The first restricts the facility to a single floor and no elevators to represent the case of a single floor facility. The second set allows two floors and one elevator, a reasonable maximum number for a facility of only 7 departments. Finally, the value of the slack parameter is set to 3 percent and 6 percent. The resulting computation times are shown in Table 5.2. The quantity of nodes explored, and the resulting facility and flow costs for these scenarios, are documented in Appendix G.

Computation Time

			Without PQM			With PQM		
Departments	Floors	Slack	Part 1	Part 2	Total	Part 1	Part 2	Total
5	Single	3.0%		0.47	0.78		0.25	0.33
		6.0%	0.31	0.38	0.69	0.08	0.27	0.35
	Multiple	3.0%		0.8	1.41		0.23	0.34
		6.0%	0.61	0.78	1.39	0.11	0.67	0.78
6	Single	3.0%		2.34	4.75		0.75	2.08
		6.0%	2.41	1.88	4.29	1.33	0.78	2.11
	Multiple	3.0%		20.01	25.07		2.09	4.67
		6.0%	5.06	11.2	16.26	2.58	3.83	6.41
7	Single	3.0%		223.22	628.08		54.5	117.95
		6.0%	404.86	82.98	487.84	63.45	15	78.45
	Multiple	3.0%		77.39	176.03		17.39	47.06
		6.0%	98.64	951.22	1049.9	29.67	39.09	68.76

Table 5.2: Computation time in seconds for various scenario sizes.

In every scenario, the position q method (PQM) significantly reduced the computa-

tion time. Increasing the number of departments unsurprisingly increases the computation time. Allowing multiple floors and an elevator generally increased the computation time, but unexpectedly decreased it in some scenarios. The slack percentage appears to have a significant effect on computation time, but no correlation was apparent.

We solved new parameter ranges for the scenarios with seven departments to further investigate the effect of the position q method, multiple floors, and the slack parameter on computation time. All other parameters were kept the same as the previous scenarios. Data tables for the following experiments can be found in Appendix H.

First, we solved the scenarios with and without the PQM constraints over a range of slack parameter values. The computation times are shown in Figure 5.7.

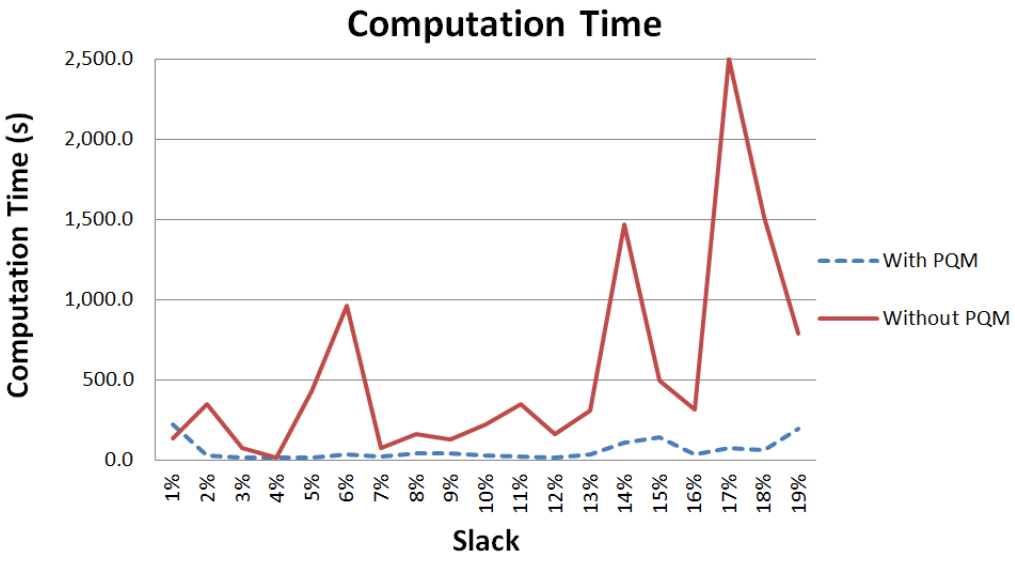

Figure 5.7: Computation time with and without PQM for a range of slack parameter values.

The PQM reduced computation time in every instance except for those where the slack parameter was set to 1 percent. The computation times without using the constraint were erratic and sometimes very large. We look at the fluctuation of computation time when using the PQM constraints in more detail in the next test.

We recorded the computation time for a range of slack values, including the cases of a single floor and multiple floors. The position q method constraints were instated for all of these computations. The results are shown in Figure 5.8.

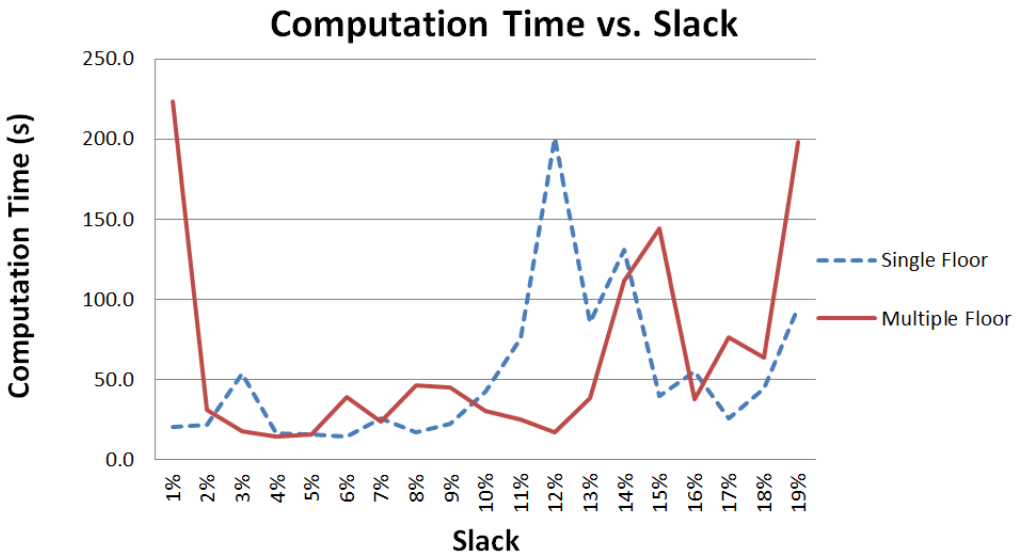

Figure 5.8: Computation time for single and multiple floors for a range of slack parameter values.

The maximum number of floors and the slack parameter appear to have a significant effect on the computation time, but no correlation could be determined from these experiments. These results are likely due to the solving strategies used by Gurobi Optimization's path-dependent algorithm.

5.5 Flow Cost

This set of scenarios also demonstrates the effects of increasing the slack and allowing multiple floors on the resulting layout's flow cost. The flow cost is graphed below for the previous set of scenarios.

Recall that the slack parameter represents a percentage of the optimal facility cost when

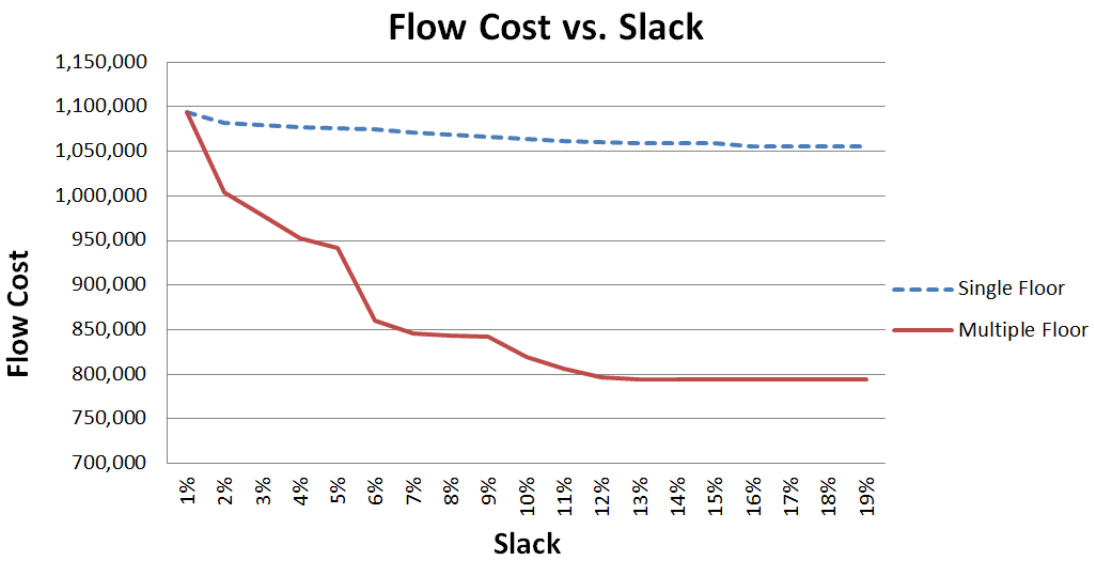

Figure 5.9: The flow cost for single and multiple floors as slack is increased.

the objective of minimizing the flows is removed from the objective function. As the slack is increased, the second optimization model is allowed to "spend" more on the facility to decrease the cost of the interdepartmental flows. When the number of floors is limited to one, the flows can only be marginally improved. Allowing a second floor quickly reduces flow costs when the slack is increased. In this case, increasing the slack beyond 12 percent no longer reduces flow costs for the multiple floor facility and 16 percent for the single floor facility.

6 Hospital Layout Conclusions

Our facility layout model accounts for many design factors—more than any other model reviewed. This unfortunately results in unreasonably long computation times for scenarios of practical size. Adding PQM constraints seems to consistently reduce computation times but not enough to make the model practical. Perhaps other strategies similar to PQM could be used to further reduce computation times.

The parameters are the most important factor in determining whether a multi-floor facility is more efficient than a single floor facility. Before the size and number of floors in the facility can be effectively modeled as decision variables, the model's facility cost structure needs to be improved to better represent realistic costs of construction. Accurate construction cost data must be input in the model for a truly optimal facility layout to be generated.

The objectives of minimizing the interdepartmental flow costs and the facility construction costs can be balanced by the goal programming method. The data describing the additional efficiency gained from additional investment in the facility's structure could greatly benefit the architect or other decision makers involved in the building's construction.

The traditional facility layout problem formulations are well suited for determining the placement of manufacturing departments. While this model includes many design factors, it

is far from complete. A hospital layout requires the integration of many more objectives and constraints specific to hospital design. A hospital layout model must include these objectives and constraints, in addition to reasonable solve times for realistic scenarios before it will be useful to facility designers.

7 Hospital Layout Extensions

There are many possible extensions to this research project. Further research may include linear approximations of the size of the overall facility in cubic feet to supplement the linear approximations of the department sizes we employ. Our model represents the construction costs linearly with the length, width, and height of the facility. This is not realistic and favors square facilities. A linear approximation of the total cubic feet of the facility could be used to model the construction costs more realistically. Additionally, more precise linear methods of approximating the areas could be employed to improve accuracy (when computational resources permit). Alternatively, a program with the capability of solving a mixed-integer, non-linear formulation could be developed or implemented when available.

A sensitivity analysis of the multi-floor facility by varying the number of elevators and the number of floors could be performed. Moreover, the cost parameters should be varied to test the sensitivity of the formulation. This would permit further observation of whether the formulation truly represents a multi-floor layout decision.

The formulation should be solved with real or approximate hospital data and compared to existing hospital layouts. The basic department relationships should first be observed and modeled to understand the basic design of the hospital. If layouts and flow relationship

data from existing hospitals can be found and applied to the model, this may reveal even more significant conclusions about the way hospitals are designed.

The basic relationships between hospital functions should be designed to incorporate the relationships between units within and between hospital functions. For example, the waiting areas and the beds should be incorporated into the flow data as well as other unit data. This will increase the problem complexity and decrease the chance of finding the optimal layout while allowing the model to more accurately represent actual hospitals. Another extension would be to completely separate the beds and waiting rooms from the departments and determine the amount of waiting rooms needed to service the flow. The beds may also be determined this way, although typically simulations or capacity analysis is used to determine the number of beds in a hospital.

Hospital policies—such as beds needing to be next to windows to improve patient recovery time—may be incorporated into the multi-floor facility layout. Also, perhaps by adding a graph-theoretic approach, one may more realistically capture the relationships between outpatient, inpatient, and emergency patient departments.

The multi-floor layout model can also be extended into the re-layout process of a hospital. Often, hospitals incur a large cost from adding extra rooms or extra buildings without taking the time to perform a re-layout optimization. The re-layout of a hospital without adding any new buildings should be considered as an option in addition to a re-layout including new buildings. Likewise, the hospital layout model could be extended to incorporate a dynamic layout or a more flexible layout. Hospitals must be flexible in order to quickly incorporate new technology and adapt to changing amounts of flow and demand. A hospital might become more flexible, for example, by structuring departments, waiting rooms, and beds in such a way that these departments could be exchanged without significantly impacting

service delivery.

Finally, the multi-floor hospital layout with respect to separate hospital buildings should be investigated. A common trend is for outpatient clinics to be completely separate from the hospital. Future research might investigate when this practice would be economical for a hospital network—especially in the case of clinics that share the same requirements as the inpatient facilities for expensive equipment. Costs of redundant equipment must be accounted for if the problem scope spans a network of hospitals.

8 LED Traffic Signal Maintenance Scheduling Introduction

The use of Light-Emitting Diodes (LEDs) to light traffic signals is relatively new. LEDs were first widely used in traffic signals in the mid-1990s [5]. Early versions of LEDs were very low powered and dim, but new LEDs can achieve very high brightness. LEDs are very small, measuring less than $1\,\mathrm{mm}^2$ [27]. They utilize very low wattage compared to other light sources such as incandescent bulbs, which were previously used to light traffic signals.

The prevalence of LED traffic signals is increasing. The California Energy Commission defined a LED traffic signal module as "the individual 8-inch (200 mm) or 12-inch (300 mm) round traffic signal or the pedestrian signal [and a] module consists of the light source, lens, and all parts necessary for operation." The commission surveyed cities and counties in California regarding their use of LED Traffic signals and found that between 1999 and 2004, the percent of respondents reporting that they had installed LED traffic signals increase by 16 percent to 73 percent [9]. The Energy Policy Act of 2005 requires all traffic signals manufactured after January 1, 2006 to meet the performance requirements of the EPA's Energy Star program for traffic signals [13]. This requires the traffic signal design to realize the energy efficiency achieved by LED technology [10]. There are over 250,000 traffic signals in the U.S. today [28], resulting in the rapid adoption of a new application

of a light source technology that is still in its infancy. This consequently presents a great opportunity for research and methodology improvement. Specifically, while optimal maintenance schedules for LED traffic signals are in high demand, relatively little research has been done in modeling this function.

Chapter 9 reviews the benefits of LED traffic signals and the new challenges they present in regards to maintenance. In Chapter 10 we present three models and algorithms. The first model is for optimal maintenance scheduling. The second model is an approximation of the first model which is designed to be quickly solved for very large scenarios. The third model is for scheduling replacement and inspection activities, and introduces stochastic deterioration. In Chapter 11 we test the algorithms' computation times and compare the cost savings of their maintenance schedules.

9 LED Traffic Signal Maintenance Scheduling Literature Review

9.1 Benefits of LEDs

Although LEDs cost more than their incandescent predecessors, there are many potential advantages of utilizing LEDs in traffic signals. The government has mandated their use because of their energy efficiency, reducing the burden on the nation's energy infrastructure and the environment. They also have a longer lifespan than the incandescent bulbs they would replace. Finally, LED traffic signals potentially require less maintenance than incandescent bulbs, which must be frequently replaced.

9.1.1 Energy Efficiency

Perhaps the most persuasive and straightforward benefit of LEDs is the reduced energy costs. LED traffic signals use 7 to 29 watts, while incandescent traffic signals use 135 to 165 watts [2] [3]. On average, LED Traffic Signals use a compelling 80% less energy than incandescent lamps [10]. This can facilitate a payback period as short as 1.8 years for the least expensive red LEDs [2]. The government entities that are primarily responsible for maintaining traffic signals are very interested in the cost savings from the operation of this

new light technology [9]. A Missouri city anticipates reducing power bills by over $1,000 annually per intersection with LED installations [35].

Increased safety might be another advantage of the energy efficiency of the LED traffic signal. The California Energy Commission concluded that the low power requirements of LED traffic signal modules make fully functional battery backup systems for power outages viable [9]. These systems improve safety by allowing the signals to operate from a local battery power supply during a power outage, decreasing the chance of accidents or congestion due to traffic signal light outages [9]. A battery backup system costing $3,000 can power the traffic signals of an intersection for over two hours [10].

9.1.2 Lifespan

LEDs have been marketed as having a longer life span than traditional incandescent traffic signals [5]. Traffic signals illuminated by incandescent lamps fail completely, requiring immediate replacement. New LED traffic signals have a longer lifespan, but the luminous intensity deteriorates with age [20]. The luminance of a signal is roughly a measure of its brightness [10]. A LED traffic signal is deemed to have failed when its luminous intensity falls below a minimum standard. Initially there were many reports of the LEDs dimming significantly, resulting in shorter-than-promised lifespans. As the technology matured, the lifespans have increased and the deterioration of brightness has been significantly reduced [5]. The functional lifespan of modern LEDs can vary, however.

The lifespan of LEDs has been reported over a large range. A 2000 article published by the Institute of Transportation Engineers estimated the lifespan of LED traffic signals to be between 5 and 10 years on average [2]. Responders of a 2004 California Energy Commission survey anticipated that they would need to replace the LED traffic signal modules after

4.5 years on average due to decreasing light intensity [9]. Widely accepted specifications released 2004–2007 for LED traffic signals by the Joint Industry and Traffic Engineering Council Committee of the Institute of Transportation Engineers (ITE) defines minimum luminous intensity and a host of other performance criteria of traffic signal modules for 60 months [37] [39] [40] [36] [38]. For example, section 3.3.4 of an ITE purchase specification states "The module lens shall not crack, craze or yellow due to solar irradiation typical for a south-facing Arizona Desert installation after a minimum of 60 months in service." [37]. The purchase specification recommends this 60 month term for a warranty period, but suggests that the useful life may be longer [39]. A 2006 survey coordinated by the ITE received responses from 75 public agencies and 6 from vendors, with 73% of responders indicating that they use a 5-year warranty period. Furthermore, 52% of responders indicated that they intend to replace the signals after a period greater than 6 years. In a case study by the Road Commission of Oakland County, LED traffic signals were lab tested in 2006, with results concluding that 10 years is a reasonable lifespan for the LED signals [20]. It was noted that LEDs tend to last longer in the cooler climates where this study took place. In a 2009 report by the National Cooperative Highway Research Program, the expected life ranged from 7 to 10 years. By contrast, the lifespan of incandescent bulbs is usually not much more than a year [35].

Cost savings from reduced maintenance are frequently touted by marketers, but practice has shown the extent of the savings is uncertain [5]. In a 1999 survey, the California Energy Commission found that the reduced maintenance cost was one of the main reasons that government entities were switching to LED traffic signals, but in a more recent 2004 survey, only 56 percent of responding entities who implemented LED traffic signals reported achieving savings from maintenance [9].

9.2 Maintenance

The short life of incandescent bulbs resulted in frequent replacements and emergency calls when the bulbs burnt out prematurely. It was believed initially that LEDs would be installed and there would be no maintenance for the duration of the signal's life. But confirming that LED traffic signals are still bright enough to meet standards for public safety is an important maintenance obligation. Furthermore, by determining accurately when an LED traffic signal needs to be replaced, the signal can be used until the true end of its lifespan. Preventative maintenance and lens cleaning are also needed to maximize the safety and utility of the traffic signals.

Many agencies use the warranty period as the lifespan of a traffic signal and replace the signals at the end of this duration. Manufacturers, however, carefully calculate the duration of the warranty they provide for their traffic signals. They limit the warranty length to ensure that few of the signals will fail within this time period to minimize the potential replacement costs they will have to cover. Traffic signals with a 5 year warranty will often maintain conforming luminosity levels for an additional 2 or more years [5]. This provides an additional opportunity to improve on a basic schedule that calls for replacing traffic signals after their warranty expires. In some cases, a bid request specifies a warranty period, and then the manufacturer will simply increase the price to reflect replacing the number of units expected to fail. In either case, the warranty period may be a poor indicator of the actual lifetime of the product. An inspection schedule allows for the useful lifespan of each traffic signal to be maximized while maintaining safety standards.

9.2.1 Inspection Schedules

An inspection generally consists of a field test of the light output of a traffic signal using a hand-held light meter or a device that fits over the circular module of a traffic receiver to block outside light [10]. Inspections may also be performed in a lab by measuring the luminance at different angles and distances. While lab tests are more accurate, a field inspection of an LED traffic signal provides the accurate condition of the signal if performed properly [5].

While LEDs usually outlast their warranty, some will still fail early. The gradual deterioration of the LEDs can result in traffic signals remaining in service long after their signal luminosity is no longer sufficient, since it is not immediately clear the signal needs to be replaced [20]. Failed LED traffic signals are often not reported if they still emit light. New maintenance schedules are necessary that include inspections to identify failed signals. Inspection allows for the early replacement of failing traffic signals while maximizing the utilization of signals with longer lifespans. For example, signals with a longer duty cycle—defined by the ITE as "the fraction of time during a specified time period that the module is energized, expressed as a percent of the specified time period" [39]—will likely be the first signals in an intersection to become too dim. A simple replacement schedule would replace all the signals at once, truncating the useful life of some of the signals. A maintenance schedule that includes inspections more frequently than the alternative simple replacement schedule could ensure higher levels of public safety by identifying and replacing signals that fail early, while still lowering costs by reducing the total average replacement frequency. Unfortunately, it is likely that many of the agencies responding to a 2006 survey by ITE do not monitor the condition of the traffic signals, as 78 percent indicated that they have

inadequate funding for monitoring and replacement programs. Furthermore, there were considerably fewer survey responses for questions concerning monitoring procedures [20]. Establishing a replacement schedule and determining the cost of the schedule will allow for the organizations responsible for the upkeep of LED traffic signals to accurately budget the costs of keeping the signals in compliant condition.

Frequent inspections can be costly and diminish the benefits of a longer lifespan. Therefore, an efficient maintenance schedule is imperative. An efficient schedule would greatly benefit from maintaining a database that contains information about each traffic signal that is installed [5]. This database may include a variety of attributes, such as the model information, the manufacturer, the date and location of installation, the duty cycle, the dates of past inspection or servicing, and the last known luminosity. This information can be used in inspection sampling to track and predict the deterioration of the traffic signals. This in turn will allow for efficient inspection and maintenance scheduling. In one study, LED traffic signal failure rates over time were found to follow the "bathtub" curve, with a higher number of failures at the beginning and end of the signal's life [10]. This implies that new traffic signals may need to be inspected once in the first few months, but older signals at the end of their estimated life will also need to be watched closely. Further, traffic signals with longer duty cycles may need to be inspected sooner and a greater rate of deterioration would be expected.

9.2.2 Preventative Maintenance

The condition of the lens and housing also affects the visibility of a traffic signal. It is recommended that the lens be cleaned and the housing inspected about once a year, depending upon environmental variables. This should be done routinely, not just in response

to complaints or accidents. Preventative maintenance on traffic signals has been shown to effectively identify and fix problems before they become emergency calls, while improving safety and lessening workload variability [1].

10 Maintenance Scheduling Models and Algorithms

10.1 A Dynamic Programming Model with Routing

We present a basic model for the maintenance of LED traffic signals. This model is a starting point for determining optimal maintenance schedules of LED traffic signals. This model is primarily intended to be used for the routine maintenance of the signals to improve and affirm visibility, such as removing debris, cleaning the lens, and visually inspecting the housing and visibility of the LED traffic signal. The cost of performing these maintenance procedures is modeled as a variable cost for each traffic signal maintained in a period. These maintenance procedures may be performed once a year or multiple times a year depending upon the environment—a much more frequent pattern than the replacement cycle of the LED. This results in the basic time units of this model (periods) representing a shorter period of time. A period may represent a unit of time as short as a week or as long as three months depending upon the number of signals being considered and their required maintenance frequency. Due to the relatively short time periods, a capacity limit is included for the quantity of signals that can be feasibly maintained in a single period.

This limitation results in a staggered maintenance schedule—a small subset of signals

are chosen for maintenance each period instead of performing maintenance on a large quantity of signals in one period. This staggering is an important aspect of municipal project management and maintenance schedules. Staggered maintenance schedules facilitate an even balance of demand on municipal resources, such as manpower or cash flow. Clearly, it would not be practical to attempt maintenance on every traffic signal in a large city on the same week regardless as to whether the city performs the maintenance in house or outsources the maintenance to the private sector.

Conversely, too much staggering is inefficient. Instead of the case where maintenance is performed on one single traffic signal in one time period, and one single traffic signal in the next, it may be more efficient to perform maintenance on both signals in the same period. Traveling expenses aside, there are fixed costs associated with performing any maintenance in a time period (or an economy of scale from performing maintenance on multiple signals). This could result from the learning curve of the required maintenance work, the time spent by the maintenance crew to organize the required tools and equipment, or the time spent consulting maps, procedures, or other documentation. This cost is represented in the model by a fixed cost parameter for performing any maintenance in a time period. This fixed cost parameter can be set appropriately; it may be relatively small or negligible in some cases but significant in others. It is not intended to represent traveling costs. This parameter adds to the model's versatility for modeling different scenarios.

The costs of travel and the locations of the traffic signals should be considered when determining an optimal maintenance schedule. This results in a schedule that is similar to a group maintenance policy, which considers replacing a group of machines once a single failure occurs [42]. Traffic signals located near each other (or in the same intersection) should optimally be maintained in the same period to reduce travel expenses that would otherwise

be incurred from making the trip to the signals and the return trip twice. This saves lost wages in travel time, vehicle wear, and fuel. We model the distance between traffic signals with a matrix of distance parameters and a travel cost parameter. The distance parameters represent the distance between any set of two traffic signals and the distance between any traffic signal and the office. The office represents the starting location for the maintenance crew, and may be the location where supplies or equipment is stored. Although the unit of distance used is arbitrary, a reasonably precise unit may be miles or kilometers. The model uses these distances to calculate the total distance traveled to perform maintenance on the traffic signals. Because this calculation can be computationally intensive, approximations for the total distance traveled are used in the solution algorithms. The travel cost parameter is the cost incurred from traveling one unit of distance. This parameter corresponds to the units chosen for distance and places a cost on travel.

This model assumes that the condition of a traffic signal is deterministic and deteriorates linearly with time. The condition of a traffic signal is therefore equivalent to the number of time periods since maintenance was last performed on the signal. This is referred to as the age of the signal. The state in the dynamic program model consists of the age of each traffic signal. After each period, the age of each signal increases by one unit unless maintenance is performed on the signal. More complex methods could be used that may model the deterioration of a signal more accurately, but this relatively simple method is a reasonable starting point that uses minimal computations.

The frequency that a signal requires maintenance is primarily influenced by a negative utility parameter. This parameter is a vector that contains a negative utility value for each possible age of a signal. It represents the cost resulting from the deterioration of a traffic signal. This cost represents the costs associated with the increased risk of accidents

resulting from a signal failure. Note that this cost can be very direct, as the parties involved in the accident may be able to sue the city for inadequate signal maintenance in states where the legislature has permitted such lawsuits ("sovereign immunity"). This negative utility must be balanced by the cost of more frequent maintenance. A traffic signal located far from the office and far from other signals may be maintained less frequently than traffic signals located near the office and near other signals due to the greater replacement cost overcoming the negative utility.

The value of the negative utility parameter will generally be set to zero for a signal of age zero, and increase as the age increases. This models the deteriorating functionality of the traffic signal. Once the signal has aged past the maximum allowable age of a signal—when the longest period of time has passed without maintenance that is acceptable and the signal absolutely must be maintained—the negative utility will be an arbitrarily large number.

10.1.1 Parameters

L	Total number of signals.
m	Maximum allowable age of a traffic signal.
d_{ij}	Distance from one signal to another; $i = 0, 1, 2, \ldots, L; j = i+1, i+2, \ldots, L$, where index 0 represents the office (the starting point for the maintenance crew).
c_v	Variable cost of performing maintenance on a signal.
c_f	Fixed cost of dispatching a maintenance crew in a time period.
c_d	Cost to drive one unit distance.

α Discount rate.

κ Maximum number of replacements in a period (i.e., capacity limit).

10.1.2 Dynamic Program

1. State: $s_i \in \{0, 1, 2, \ldots, m\}$ is the condition of signal i. In vector form,

$$\mathbf{s} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_L \end{bmatrix}.$$

Let S be the set of states. Note that $|S| = (m+1)^L$.

2. Actions:

$$x_i = \begin{cases} 1 & \text{if maintenance is performed on signal i,} \\ 0 & \text{otherwise.} \end{cases}$$

In vector form,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{bmatrix}.$$

Let $X(\mathbf{s})$ be the set of feasible actions given state $\mathbf{s}$. Since the maximum allowable

age of a tragic signal is m, $X(\mathbf{s})$ is given by

$$X(\mathbf{s}) = \left\{ \mathbf{x} \in \{0,1\}^L : \; x_i = 1 \text{ if } s_i = m \text{ and } \sum_{i=1}^{L} x_i \leq \kappa \right\}.$$

3. System Dynamics: Let s_i^t denote the age/condition of signal i in period t. The system dynamics can be defined as:

$$s_i^{t+1} = \begin{cases} s_i^t + 1 & \text{if } x_i^t = 0 \\ 0 & \text{if } x_i^t = 1 \end{cases}$$

We will use the short-hand notation $\mathbf{s}^{t+1} = \Gamma(\mathbf{s}^t, \mathbf{x}^t)$ to represent the state transition function. Note that change in the age/condition of a signal is deterministic (i.e., the age/condition of a signal changes by one unit in each period).

4. Dynamic Program Recursion: The Bellman equation can be written as

$$J(\mathbf{s}) = \min_{\mathbf{x} \in X(\mathbf{s})} \left\{ c(\mathbf{s}, \mathbf{x}) + \alpha J\big(\Gamma(\mathbf{s}, \mathbf{x})\big) \right\}$$

where $c(\mathbf{s}, \mathbf{x})$ is a complex function representing the immediate cost of action $\mathbf{x}$ while in state $\mathbf{s}$:

$$c(\mathbf{s}, \mathbf{x}) = \left(\sum_{i=1}^{L} x_i \right) c_v + I_{[(\sum_{i=1}^{L} x_i) \geq 1]} c_f + g(\mathbf{x}) c_d + \sum_{i=1}^{L} h_i(s_i)$$

Where:

- $I_{[.]}$ is the identity function.
- $g(\mathbf{x})$ is the shortest distance tour starting from the office, visiting each signal

that is being serviced (determined by $\mathbf{x}$), and back to the office. Note that $g(\mathbf{x})$ can be obtained by solving an optimization model for a given $\mathbf{x}$.

- $h_i(s_i)$ is the negative utility of allowing signal i reach state s_i.

10.2 Value Iteration Algorithm for Routing Model

A value iteration type of algorithm was developed to determine an exact solution to the model. The value iteration algorithm functions by initializing an approximate value for every possible system state. Then each approximate value is in turn refined using the preceding approximate values of the other states. The value of each feasible system state is approximated once during each iteration. The algorithm runs many iterations until the series of successive approximate values converge to optimal values.

Let $\bar{J}^t(s_i)$ be the estimate of the value of each state $\mathbf{s} \in S$, $J(\mathbf{s})$, in iteration t and consider the problem in an infinite horizon setting since we do not have a planning horizon. A value iteration algorithm is given by

1. Initialization:

   (a) Initialize $\bar{J}^0(\mathbf{s})$ for each state $\mathbf{s} \in S$.

   (b) Set $t \leftarrow 1$.

2. For each state $\mathbf{s} \in S$, solve:

$$\bar{J}^t(\mathbf{s}) = \min_{\mathbf{x} \in X(\mathbf{s})} \left\{ c(\mathbf{s}, \mathbf{x}) + \alpha \bar{J}^{t-1}\big(\Gamma(\mathbf{s}, \mathbf{x})\big) \right\}$$

   by calculating $c(\mathbf{s}, \mathbf{x}) + \alpha \bar{J}^{t-1}\big(\Gamma(\mathbf{s}, \mathbf{x})\big)$ for each value of $\mathbf{x} \in X(\mathbf{s})$ and then determining the minimum value.

3. Set $t \leftarrow t + 1$.

4. Stop if a termination criterion is met, go to Step 2 otherwise.

Due to the complexity of calculating the optimal shortest-path tour (which is a variation of the Traveling Salesman Problem known to be NP-hard), an approximation is substituted for the function $g(\mathbf{x})$ (which is utilized by the function $c(\mathbf{s}, \mathbf{x})$ in this algorithm). Each traffic signal is assigned an index parameter based on the location of the traffic signal. The traffic signals that require maintenance are visited in order of their index number. The index numbers are thus assigned in a manner which attempts to maximize the efficiency of this routing method. While there are many algorithms that could be used to assign index numbers to traffic signals, one simple and effective method is presented below:

1. Choose the traffic signal that is nearest the office to assign the first index.

2. Assign the next index to the non-indexed traffic signal that is nearest the traffic signal previously chosen.

3. Repeat step 2 until all traffic signals are assigned indices.

We used the Python programming language to implement this value iteration algorithm. In an effort to reduce computation times and expand the size of problems that can be feasibly solved using this method, we exerted considerable effort to improve the efficiency of the code to eliminate redundant computations and utilize multiprocessors cores. While there are more steps and stored values, the resulting algorithm is conceptually the same. The documented program is included in the appendix.

10.3 Model Size and Clustering

The computational complexity and resulting solve times for various sized models are a relevant concern due to the potentially enormous problem instances. This places importance on making reasonable trade-offs between modeling accuracy and the computational complexity of solving the model for an optimal schedule. A good model and algorithm must produce a good solution in a reasonable amount of time.

The unit size and representation of the parameters of this model can be scaled to control the size, scope, and detail of the problem instance and resulting solution. In a micro scenario, only a few intersections may be considered. Each traffic signal in the model would represent and correspond to a specific traffic signal. If one of the intersections has eight traffic signals, each of those traffic signals would be modeled with the same distance parameter value to the office and a distance parameter value of zero between the eight traffic signals. An optimal solution would then contain the detail to include performing maintenance on some subset of traffic signals within the intersection.

However, this level of modeling precision is often impractical. Large amounts of detailed data are needed to create accurate parameters, and the resulting schedule may be too complex to implement. Furthermore, large problems would be too complex to solve in a reasonable amount of time. To scale the model to handle larger problems, modeling simplifications can be introduced to trade off detail for reduced computational complexity. This can easily be done by modeling an entire intersection by what previously represented a single traffic signal. Since a traffic signal in the model may no longer represent an actual traffic signal, we'll refer to it as a node. One node can represent all signals in an intersection, or all the signals in multiple nearby intersections that are clustered together due to their

proximity. The distance parameters now represent the approximate distance between a node and the office or between two nodes. To accommodate various-sized clusters, the variable cost in the model is changed to a vector that contains a separate cost value for the costs of performing maintenance on all the signals in a specific cluster.

10.4 An Approximate Dynamic Programming Model Using an Approximate Cost Structure

A realistic scenario may involve modeling several hundred nodes. This would result in a state space too large to be solved in a reasonable amount of time by the value iteration algorithm. A simplified model, however, can greatly reduce computation time to enable large scenarios while still producing effective solutions. We present an Approximate Dynamic Programming (ADP) [30] [6] model below that approximates the cost structure of the previous model, allowing the costs of each node to be computed independent of the state of the other nodes.

10.4.1 Simplifying Assumptions

We can significantly simplify the cost structure by neglecting the fixed cost of sending a maintenance crew out (i.e., $I_{[\sum_{i=1}^{L} x_i \geq 1]} c_f$) and replacing the cost associated with the shortest distance tour starting from the office, visiting each signal to be replaced (determined by $\mathbf{x}$), and back to the office (i.e., $g(\mathbf{x})c_d$) with $c_d \sum_{i=1}^{L} d_{0i} x_i$. This essentially replaces the routing costs with a fixed cost for traveling to each signal that is independent of the route or the other signals that are replaced in the same time period.

Based on this simplification, the immediate cost reduces to

$$c(\mathbf{s}, \mathbf{x}) = \left(\sum_{i=1}^{L} x_i \right) c_v + c_d \sum_{i=1}^{L} d_{0i} x_i + \sum_{i=1}^{L} h_i(s_i)$$

$$= \sum_{i=1}^{L} \big((c_v + c_d \cdot d_{0i})x_i + h_i(s_i)\big).$$

This simplification allows us to assume that the immediate cost and the cost-to-go can be approximated as additive functions across signals:

$$\tilde{c}(\mathbf{s}, \mathbf{x}) = \sum_{i=1}^{L} \tilde{c}_i(s_i, x_i),$$
$$\tilde{J}(\mathbf{s}) = \sum_{i=1}^{L} \tilde{J}_i(s_i).$$

Using this approximate cost structure, the Bellman equation becomes:

$$\begin{aligned}\tilde{J}(\mathbf{s}) &= \min_{\mathbf{x}\in X(\mathbf{s})} \left\{ \sum_{i=1}^{L} \tilde{c}_i(s_i, x_i) + \alpha \sum_{i=1}^{L} \tilde{J}_i\big(\Gamma_i(s_i, x_i)\big) \right\} \\ &= \min_{\mathbf{x}\in X(\mathbf{s})} \left\{ \sum_{i=1}^{L} \Big[\tilde{c}_i(s_i, x_i) + \alpha \tilde{J}_i\big(\Gamma_i(s_i, x_i)\big)\Big] \right\}\end{aligned}$$

Instead of using $X(\mathbf{s})$ as the feasible set of actions and restricting the states of a signal to the set $\{0, 1, 2, \ldots, m\}$, we find it more convenient to let the feasible set of actions

$$X = \left\{ \mathbf{x} \in \{0,1\}^L : \sum_{i=1}^{L} x_i \leq \kappa \right\},$$

And let the set of states for a signal $\{0, 1, 2, \ldots, m, m+1, m+2, \ldots\}$ with a cost of $\tilde{J}_i(s_i) = \infty$ for all $s_i > m$.

With these conventions and letting

$$a_i \equiv \Big[\tilde{c}_i(s_i, 0) + \alpha \tilde{J}_i\big(\Gamma_i(s_i, 0)\big)\Big] - \Big[\tilde{c}_i(s_i, 1) + \alpha \tilde{J}_i\big(\Gamma_i(s_i, 1)\big)\Big] \qquad i = 1, 2, \ldots, L,$$

It can be readily shown that the right-hand side of the Bellman equation above is

equivalent to the following $0-1$ knapsack problem:

$$\max_x \left\{ \sum_{i=1}^{L} a_i x_i \right\} \qquad \text{subject to } \sum_{i=1}^{L} x_i \leq \kappa, \qquad x_i \in \{0,1\}, i = 1,2,\ldots,L.$$

The solution of this problem can be obtained as follows:

1. Sort the indices so that:

$$a_{i_1} \geq a_{i_2} \geq \cdots \geq a_{i_n} \geq 0 \geq a_{i_{n+1}} \geq \cdots \geq a_{i_L}.$$

2. Find $r = \min\{\kappa, n\}$.

3. The solution is given by

$$x_{i_j} = 1 \qquad \text{for } j = 1,2,\ldots,r,$$

$$x_{i_j} = 0 \qquad \text{for } j = r+1, r+2,\ldots,L,$$

10.5 Value Function Approximation Algorithm

Let $\bar{J}_i^t(\mathbf{s})$ be the estimate of the (approximate) value of state s_i of signal i, $\tilde{J}_i(s_i)$, in iteration t and consider the problem in an infinite horizon setting since we do not have a planning horizon. An ADP algorithm is given by:

1. Initialization

   (a) Initialize $\bar{J}_i^0(s_i)$ for all $i = 1,2,\ldots,L$ and

   $s_i \in \{0,1,2,\ldots,m,m+1,m+2,\ldots\}$ (Note that $\bar{J}_i^0(s_i) = \infty$ if $s_i > m$.)

   (b) Initialize $\mathbf{s}^0$.

(c) Set $t \leftarrow 0$.

2. Solve:

$$\dot{\mathbf{x}}^t \leftarrow \underset{\mathbf{x}\in X}{\operatorname{argmin}}\left\{ \sum_{i=1}^{L} \tilde{c}_i(s_i, x_i) + \alpha \sum_{i=1}^{L} \bar{J}_i^t\big(\Gamma_i(s_i, x_i)\big) \right\}$$

Using the procedure described in Section 10.5.

3. Update:

$$\bar{J}_i^{t+1}(s_i^t) \leftarrow \tilde{c}_i(s_i^t, \dot{x}_i^t) + \alpha \bar{J}_i^t\big(\Gamma_i(s_i^t, \dot{x}_i^t)\big) \qquad i = 1, 2, \ldots, L$$

$$\mathbf{s}^{t+1} \leftarrow \Gamma(\mathbf{s}^t, \mathbf{x}^t)$$

4. Set $t \leftarrow t + 1$.

5. Stop if a termination criterion is met, go to Step 2 otherwise. The termination criterion may be when a certain number of iterations have been executed or when the sum of squares between the old and new cost vector is below some specified threshold value.

It is more difficult to determine convergence for this algorithm because one iteration considers only one system state (the current condition of each node in the scenario), whereas in the previous value iteration algorithm all system states were considered in each iteration. The cost vector is only updated for the states that are considered instead of all states. If the cost vector does not change between two iterations, it may not necessarily mean that the algorithm has converged. The stop criteria used in the program is therefore to run iterations until m consecutive iterations all result in a new cost vector that has a magnitude of difference from the previous cost vector that is less than a beta parameter, where m is the number of conditions being considered in the scenario.

10.6 Penalty Cost Modification

The initialized state values resulted in an early iteration where the number of traffic signals that required maintenance exceeded the capacity limit in some problem instances. Furthermore, these traffic signals were at their most deteriorated condition anticipated by the model, where the next state has a negative utility value of infinity. Due to the capacity limit, the signals that could not be replaced entered the next state in the next period. While these signals are eventually replaced in future periods, a very large cost is carried back to the value of the most deteriorated condition (that is still anticipated by the model and feasible). This causes the algorithm to never again visit these states. While in some cases this may still result in the optimal schedule, this is usually not the case. The model should ideally revisit these states and give them a second chance to make sure they are visited in an optimal schedule. If the model revisited these states and this time the number of traffic signals requiring maintenance does not exceed the capacity limit, then the deteriorated signals would all be replaced. The values of these states may then be very low and the resulting optimal schedule may include visiting these states.

To prevent infinite negative utility values from unduly raising the value of the previous states, the cost assigned to a state resulting from each traffic signal is capped at the cost resulting from the action of replacing the traffic signal, plus a penalty factor. This is reflected in a new function $\bar{J}_i^{t+1}$ shown below.

$$\bar{J}_i^{t+1} = \min_{\mathbf{x}\in\{0,1\}^L} \left\{ \tilde{c}_i(s_i^t, \dot{x}_i^t) + \alpha \bar{J}_i^t\big(\Gamma_i(s_i^t, \dot{x}_i^t)\big) \right\} + I_{[a_{i_{\kappa+1}}>0]} P_i(n^t, \kappa, \mathbf{s}^t),$$

$$i = 1, 2, \ldots, L.$$

Where $P_i(n^t, \kappa, \mathbf{s}^t)$ represents the penalty cost function:

$$P_i(n^t, \kappa, \mathbf{s}^t) = (n^t - \kappa)\varphi\left(\frac{s_i^t}{\sum_{j=1}^{L} s_j^t}\right)$$

And where:

- κ is the parameter specifying the maximum number of replacements in a period.
- $a_{i_{\kappa+1}}$ is the cost benefit of replacing the $\kappa + 1$ traffic signal when they are ordered by the value of a from greatest to least.
- n^t is the number of traffic signals in period t that have a positive a value and thus are to be replaced.
- φ is a penalty cost parameter.

This function represents a penalty to a state because the optimal set of actions specifies maintenance to more traffic signals than the capacity limit. This modification to the algorithm only changes the calculation for the value of the state, and the optimal set of actions, $\dot{\mathbf{x}}^t$, remains the same. The penalty cost function is formulated to scale the penalty based on the number of signals exceeding the capacity limit that should be maintained $n^t - \kappa$ and the condition of the current traffic signal being considered. The penalty is therefore greater if the number of lights exceeding the capacity limit is greater, since this is a particularly undesirable state. The penalty is even greater if the current condition of the signal is greater (i.e. more deteriorated) since this is an indicator of the degree to which they are contributing to the problem. The current states of all signals are thus affected by the capacity limit violation, and not just the signals requiring maintenance in the current period.

Although the concept of a penalty cost may be realistic (resulting from overtime wages,

outsourcing, or hiring temporary help), the cost is not intended to capture or model a real cost but instead to guide the algorithm to converge on a feasible solution. Therefore the approximate model is not intended for scenarios where the optimal schedule includes violating the replacement capacity limit. It is not always clear from the parameters that this is the case, since it may be efficient to replace signals long before they reach the condition threshold where the negative utility approaches infinity. Such scenarios may result in non-converging patterns as the algorithm attempts to balance the schedule and penalties are shifted between signals, or result in state values that perpetually increase due to the unavoidable, recurring penalty.

There are some scenarios for which the model cannot reach an optimal solution. First, the model determines a staggering schedule and the condition at which each signal should be replaced. However, it is possible that the optimal scenario involves a pattern where a signal is not always replaced after the same number of periods. While the value iteration algorithm for the first model could solve and represent this schedule, the approximate algorithm cannot. Instead, the number of periods between replacements, which optimally may not be a constant number, will be rounded to an earlier period such that it can always be replaced after the same number of periods and in the same condition.

Furthermore, if this situation occurs, it is because the capacity limit is constraining the number of traffic signals being replaced, causing the penalty factor to discourage these states. The penalty factor will eventually cause the signals to be maintained at an earlier period until there is no conflict with the capacity limit constraint. However, there is no guarantee that the right combination of signals in the right order will be maintained in earlier states to result in even the optimal solution that can be represented by the value function approximation algorithm.

Considering these weaknesses, the approximation model is most effective for instances where the sum of the optimal rate of maintenance for all the signals is significantly smaller than the capacity limit. The value function approximation algorithm also represents a schedule that is simpler than the optimal schedule of the value iteration algorithm, making it more practical to implement. The greatest advantage to the approximation model is its ability to solve very large problem scenarios, which is discussed in more detail in the results section.

10.7 A Dynamic Programming Model for Stochastic Deterioration

The previous models have assumed that the condition of a traffic signal deteriorated deterministically with time. This may be a reasonable assumption in many cases, such as performing frequent, routine maintenance. A stochastic representation of a traffic signal deterioration rate may also be plausible since the failure of circuitry is often modeled stochastically and failures of LED traffic signals are frequently caused by the circuitry [10]. This method may be useful to model LED traffic signals that utilize an array of LEDs (frequently dozens) that can individually fail, with each failure marginally diminishing the effectiveness of the traffic signal. The final model presented is a stochastic model that assumes that traffic signals decay by one unit each period with a known probability.

If the failure of a traffic signal was always catastrophic with no intermediate states, then the first model would be sufficient for stochastic inspection and replacement. The probability of signal failure and resulting expectation of negative utility would increase with the passing of each period of time. Replacement would not have to be modeled as a decision: once inspected, the replacement rule would always be to replace if the signal had failed and never replace if it has not failed. Gradually diminishing utility with multiple

adequate intermediate states lends to a two-decision model.

The previous models have also assumed that the current condition of each traffic signal was known with reasonable certainty. If the decay rate of a traffic signal is not deterministic then this may no longer be a realistic assumption. Traffic signals must be inspected to accurately determine their condition, since the condition is not continuously monitored. This results in a model that can be classified as a partially observable Markov decision process [26]. The stochastic model assumes two actions are available each period. First, the traffic signal can be inspected. Once inspected, the accurate condition of the traffic signal is revealed. Second, if a traffic signal is inspected, it can then be replaced. The decision to replace a traffic signal can be made using the accurate condition information gained by the first action.

Maintenance scheduling has long been modeled for stochastically-deteriorating equipment where the condition is not continuously observed. The maintenance policy can be viewed as a function. The action space is the range of the maintenance policy function [22]. An optimal schedule in the case of LED traffic signals would minimize the long run expected negative utility and cost per unit of time.

The state in the model represents two attributes that are known about each traffic signal. The first attribute is the number of periods since each traffic signal has been inspected. The second attribute is the condition of the traffic signal when it was last inspected. (Of course, the last known condition is new if a traffic signal was replaced after being inspected.) These two attributes along with the stochastic decay rate are then used to estimate the condition of the traffic signals and develop an optimal inspection and replacement schedule.

Due to the stochastic, unknown condition of the traffic signals captured by this model, the inspection and replacement decisions are made in the same period but not simultaneously—

the condition of a traffic signal is not known until after the decision is made to inspect it. Once the condition is known, the decision is made whether to replace or not replace the signal. If the condition was always known, the model would represent a situation where the conditions are stochastic but continuously monitored, resulting in an optimal schedule that would never choose to inspect a signal without also choosing to replace the signal. Therefore, the action determination sequence is a key aspect of this model.

Note that this model does not include the cost of traveling or the effects of a capacity limit on the number of inspections or replacements in a time period. While these features may be added to this model, they have already been discussed in the previous models and therefore been neglected in order to shift the focus to the new attributes introduced in this model. This also allows for an algorithm to consider each signal independently in scenarios with multiple signals.

10.7.1 Parameters

L	Total number of signals.
m	Worst (maximum) allowable condition of a traffic signal.
p	Probability that a traffic signal deteriorates by one unit in one period.
c_v	Variable cost of inspecting a signal.
c_r	Variable cost of replacing a signal.
α	Discount rate.

10.7.2 Dynamic Program

1. State: The state is represented by the vector $[\mathbf{s}, \mathbf{r}]$ where

$s_i \in \{0, 1, 2, \ldots, m\}$ is the last known condition of traffic signal i (i.e., the condition when the signal was last inspected) and $r_i \in \{1, 2, \ldots, m\}$ is the number of time periods since signal i was last inspected. In vector form,

$$[\mathbf{s}, \mathbf{r}] = [\begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_L \end{bmatrix}, \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_L \end{bmatrix}].$$

Let S be the set of traffic signal conditions, R be the set of quantities of periods, and $[S, R]$ be the set of states. Note that,

$$|S| = (m+1)^L$$

$$|R| = m^L$$

$$|[S, R]| = (m+1)^L m^L$$

2. First Action: The model must first determine whether to inspect a traffic signal (or not inspect) and then use the information obtained from the first action to determine whether to replace the traffic signal (or not to replace). The actions and resulting dynamics are presented in sequence.

$$x_i = \begin{cases} 1 & \text{if signal } i \text{ is inspected,} \\ 0 & \text{otherwise.} \end{cases}$$

In vector form,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{bmatrix}.$$

Let $X(\mathbf{s}, \mathbf{r})$ be the set of feasible actions for $\mathbf{x}$ given state $[\mathbf{s}, \mathbf{r}]$. Since the worst (maximum) allowable condition of a tragic signal is m, $X(\mathbf{s}, \mathbf{r})$ is given by

$$X(\mathbf{s}, \mathbf{r}) = \{\mathbf{x} \in \{0, 1\}^L : x_i = 1 \text{ if } s_i + r_i = m\}.$$

3. System Dynamics Following First Action: Let s_i^t denote the last known condition of signal i in period t and r_i^t denote the number of periods since signal i in period t was inspected. The system dynamics can be defined as:

$$s_i^{'t} = \begin{cases} s_i^t & \text{if } x_i^t = 0 \\ s_i^t + \xi_i^t & \text{if } x_i^t = 1 \end{cases}$$
$$r_i^{t+1} = \begin{cases} r_i^t + 1 & \text{if } x_i^t = 0 \\ 1 & \text{if } x_i^t = 1 \end{cases}$$

Where $s_i^{'t}$ represents the actual condition of the traffic signal i in period t if the signal is inspected, or the last known condition if it is not inspected and ξ_i^t represents the stochastic deterioration of traffic signal i in period t during the past r_i^t time periods;

$$\xi_i^t \in \{0, 1, 2, \ldots, r_i^t\} \text{ and } \xi_i^t = n \text{ with probability } \Psi(n, r_i^t) = \frac{p^n p^{(r_i^t - n)} r_i^t!}{n!(r_i^t - n)!}$$

Note that ξ_i^t is only utilized if traffic signal i in period t is inspected ($x_i^t = 1$), because

otherwise the value of ξ_i^t is unknown.

4. Second Action:

$$y_i = \begin{cases} 1 & \text{if signal } i \text{ is replaced,} \\ 0 & \text{otherwise.} \end{cases}$$

In vector form,

$$\mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{bmatrix}.$$

Let $Y(\mathbf{s}', \mathbf{x})$ be the set of feasible actions for $\mathbf{y}$ given the actual or last known condition $\mathbf{s}'$, and the action $\mathbf{x}$. A signal must first be inspected before it can be replaced, and m is the worst (maximum) condition allowed. Therefore $Y(\mathbf{s}', \mathbf{x})$ is given by

$$Y(\mathbf{s}', \mathbf{x}) = \{\mathbf{y} \in \{0,1\}^L : y_i = 0 \text{ if } x_i = 0 \text{ and } y_i = 1 \text{ if } s_i' = m\}$$

5. System Dynamics Following Second Action:

$$s_i^{t+1} = \begin{cases} s_i'^t & \text{if } y_i^t = 0 \\ 0 & \text{if } y_i^t = 1 \end{cases}$$

We will use the following short-hand notation $[\mathbf{s}^{t+1}, \mathbf{r}^{t+1}] = \Gamma(\mathbf{s}^t, \mathbf{r}^t, \mathbf{x}^t, \mathbf{y}^t, \xi^t)$ to represent the state transition functions. Recall that the term ξ_i^t is only utilized by this

function if traffic signal i in period t is inspected ($\xi_i^t = 1$), because otherwise the value of ξ_i^t is unknown. Therefore, if $x_i^t = 0$, the function $\Gamma(\mathbf{s}^t, \mathbf{r}^t, \mathbf{x}^t, \mathbf{y}^t, \xi^t)$ does not utilize the term ξ_i^t; thereby preserving the unknown nature of the stochastically deteriorating condition of a traffic signal.

6. Dynamic Program Recursion: The Bellman equation using expectation can be written as

$$J(\mathbf{s}, \mathbf{r}) = \min_{\mathbf{x} \in X(\mathbf{s},\mathbf{r})} \left\{ \sum_{i=1}^{L} \sum_{\xi_i=0}^{r_i} \min_{\mathbf{y} \in Y(\mathbf{s}',\mathbf{x})} \left\{ \Psi(\xi_i, r_i) \Big(c(s_i, r_i, x_i, y_i) + \alpha J\big(\Gamma(s_i, r_i, x_i, y_i, \xi_i)\big) \Big) \right\} \right\}$$

where $c(s_i, r_i, x_i, y_i)$ represents the immediate cost of actions x_i, y_i while in state $[s_i, r_i]$:

$$c(s_i, r_i, x_i, y_i) = x_i c_v + y_i c_r + \sum_{n_i=0}^{r_i} \Psi(n_i, r_i) h_i(s_i + n_i)$$

where $h_i(s_i + n_i)$ is the negative utility of allowing signal i to reach the (known) state $(s_i + n_i)$, and can be defined as a parameter for $\{0, 1, 2, \ldots, m\}$ for each signal i. The negative utility component within the immediate cost function, $c(s_i, r_i, x_i, y_i)$, is formulated as the expectation of the negative utility of the state $[s_i, r_i]$ and is not a function of the actions x_i and y_i (nor the current increment of the value of ξ_i in $J(\mathbf{s}, \mathbf{r})$).

10.8 Value Iteration Algorithm for Stochastic Model

We present a value iteration algorithm for the stochastic model. This algorithm uses the same technique as the value iteration algorithm for the first model.

Let $\bar{J}^t(\mathbf{s}, \mathbf{r})$ be the estimate of the value of each state $[\mathbf{s}, \mathbf{r}] \in [S, R]$, $J(\mathbf{s})$ in iteration t and consider the problem in an infinite horizon setting since we do not have a planning horizon. The values for the states of each traffic signal can be determined independently because this model does not include routing costs or capacity limits. The algorithm presented below therefore determines the values for the states of a single traffic signal and can be duplicated for each traffic signal in a problem instance. A value iteration algorithm for the stochastic, unknown model is given by:

1. Initialization

   (a) Initialize $\bar{J}^0(s, r)$ for each state $[s, r] \in [S, R]$

   (In this algorithm, $|[S, R]| = (m + 1)m$ since $L = 1$.)

   (b) Set $t \leftarrow 1$.

2. For each state $[s, r] \in [S, R]$, solve:

$$\bar{J}^t(s, r) = \min_{\mathbf{x} \in X(\mathbf{s}, \mathbf{r})} \left\{ \sum_{\xi=0}^{r} \min_{\mathbf{y} \in Y(\mathbf{s}', \mathbf{x})} \left\{ \Psi(\xi, r) \Big(c(s, r, x, y) + \alpha \bar{J}^{t-1}\big(\Gamma(s, r, x, y, \xi)\big) \Big) \right\} \right\}$$

   (a) By first setting $x = 0$, $y = 0$ and calculating

$$j_{x=0} = c(s, r, x, y) + \alpha \bar{J}^{t-1}\big(\Gamma(s, r, x, y)\big).$$

   (Recall that ξ is not utilized by the function Γ when $x = 0$, and therefore this term is not included in the terms of Γ in this equation.)

   (b) Next set $x = 1$ and solve:

$$j_{x=1} = \sum_{\xi=0}^{r} \min_{\mathbf{y} \in Y(\mathbf{s}', \mathbf{x})} \left\{ \Psi(\xi, r) \Big(c(s, r, x, y) + \alpha \bar{J}^{t-1}\big(\Gamma(s, r, x, y, \xi)\big) \Big) \right\}$$

by first calculating $\Psi(\xi, r)\Big(c(s, r, x, y) + \alpha \bar{J}^{t-1}\big(\Gamma(s, r, x, y, \xi)\big)\Big)$ for each value of $y \in Y(s', x)$ and each value of $\xi \in \{0, 1, 2, \ldots, r\}$. Then determine the minimum value for each ξ, and sum these minimum values. Finally:

$$\bar{J}^t(s, r) = \min\{j_{x=0}, j_{x=1}\}$$

3. Set $t \leftarrow t + 1$.

4. Stop if a termination criterion is met, go to Step 2 otherwise.

The Python code for this algorithm can be found in Appendix K.

11 LED Traffic Signal Maintenance Scheduling Results

All computations for all traffic signal algorithms were executed on a desktop computer running the Microsoft Vista 32-bit operating system on an Intel Core2 Quad Q6600 CPU with 4 physical cores at 2.40 GHz, and 4 GB of RAM. We used Python 2.6.5 to calculate the results.

11.1 Value Iteration Algorithm Computation Time

Number of Nodes	Number of Conditions: 4	5	6	7	8	9
4	0.8	1.0	1.7	2.5	4.0	6.7
5	1.6	3.6	8.2	17.8	36.8	68.7
6	7.3	27.4	92.8	258.3	610.7	1,360.9
7	31.3	181.5	788.1	2,947.0		
8	263.1	2,065.7				
9	1,514.9					

Table 11.1: Computation time in seconds for various scenario sizes.

Table 11.1 displays the computation time in seconds for the value iteration algorithm to converge on a solution. Each cell in the table represents the solve time for a different-sized

scenario. The rows correspond to the number of nodes (traffic signals or groups of traffic signals) evaluated in the scenario and the columns correspond to the number of conditions (states). The parameters that define these scenarios are randomly generated or are arbitrary estimations of realistic values. The full set of parameters and a brief explanation for the scenarios that we evaluated when determining the computation times in Table 11.1 can be found in Appendix A. The algorithm assumes convergence and stops when an iteration results in a new cost vector that has a magnitude of difference from the previous cost vector that is less than a beta parameter.

As the data indicates, the computation time increases as the scenario includes more traffic signals and more conditions. During each iteration, the algorithm considers each possible set of actions for each possible set of states. This is equal to $m^L * 2^L$ where m is the number of conditions and L is the number of traffic signals in the scenario. This value increases exponentially with an increase in L and geometrically with an increase in m.

	Number of Processors Utilized			
	1	**2**	**3**	**4**
Preprocessing Time:	760	1154	1353	1534
Iteration Time:	1855	1006	687	532
Total Computation Time:	2615	2160	2040	2066
Average Time per Iteration:	10.5	5.7	3.9	3.0

Scenario: 8 Lights, 5 Conditions; 177 Iterations

Table 11.2: Computation time using multiple processors.

The table above demonstrates the algorithms' ability to split up the computations between multiple processors. During the preprocessing, a single processor is used to compute values (such as the cost-to-go) for each set of states and each set of actions. These values are used in the computations of each iteration. These values are also copied and stored to a sep-

arate location for each processor to access independently during the simultaneous iteration computations. The average time per iteration is reduced by a factor approximately equal to the number of processors utilized. However, while the computation time during iterations was reduced, each additional processor added computation time in the preprocessing. This additional time is a result of the machine internally making a copy of stored data for each processor. In this instance, using three processors is slightly more efficient than using four. The advantage of using more processors increases as the number of iterations executed increases. The number of iterations increases with a smaller beta stop-gap parameter or a larger discount rate parameter.

11.2 Value Function Approximation Algorithm Computation Time

The algorithm presented above takes advantage of the approximate cost structure to decrease the computation required to determine an optimal schedule. The value function approximation algorithm operates similar to the value iteration algorithm, using a series of successive approximations that build on each other, but differs by recording state values for each node's state independent of the state of the other nodes. This results in reduced computation time due to two properties of the algorithm. First, there are far fewer states to compute and record values for: $|S| \cong mL$ (in contrast to $|S| = m^L$, the number of states in the value iteration algorithm). This is the greatest advantage realized from the simplified cost structure. Second, the approximation algorithm does not compute the value of every feasible state during each iteration (which was the method used by the value iteration algorithm). Instead, the approximation algorithm determines the values and optimal action for an initialized system state in the first iteration. In each consecutive iteration the algorithm calculates the approximate values for only the system state that results from the optimal

action computed in the previous iteration. This effectively simulates the process being modeled thereby allowing the algorithm to compute new approximate values for only states that are visited by the process. This is an advantage, for example, when there are very costly states that are feasible but still impractical, and clearly would not be included in an optimal schedule. While the value iteration algorithm spends equal computation resources on all states, this approximation algorithm may only visit these impractical states a few times. Their large cost will deter the algorithm from returning, and no further computation time will be allocated to determining the exact values for these impractical states. As the iteration converges on an optimal schedule, it will only refine the approximations of the values of the states visited by that schedule.

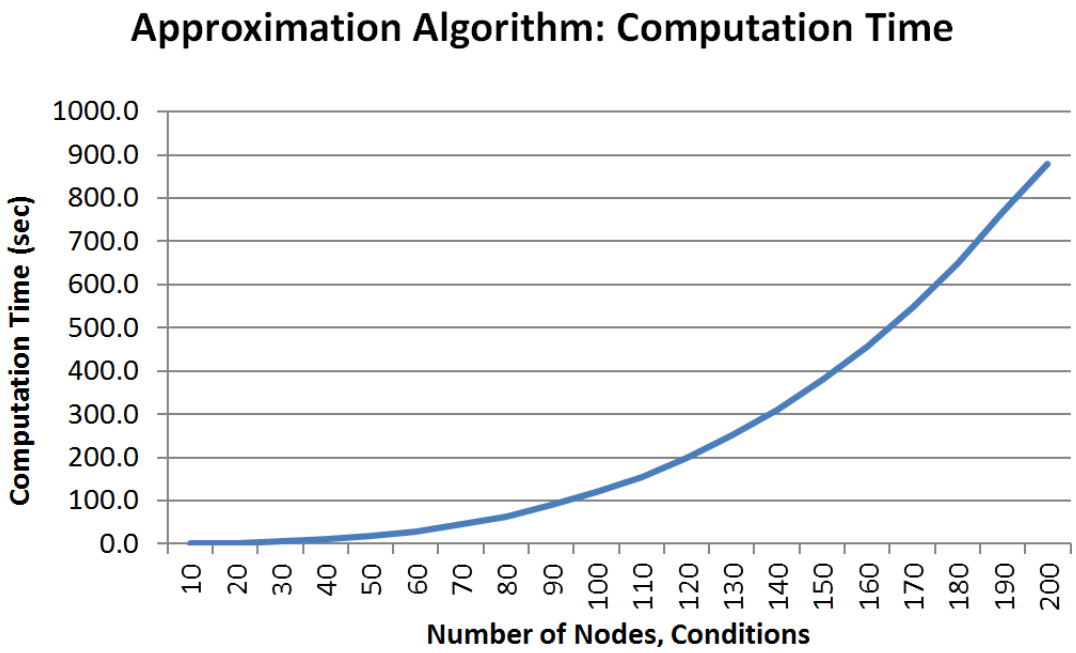

Figure 11.1: Computation time of the value function approximation algorithm.

Figure 11.1 shows the computation time as the scenario size increases. The horizontal axis represents the number of nodes and conditions in the scenario. (The number of nodes and the number of conditions are equal for each scenario.) It is clear that this value function

approximation algorithm can solve much larger scenarios in a more reasonable amount of time than the first algorithm: state values for a scenario considering 170 nodes and 170 conditions each converge in less than 10 minutes. By contrast, the first algorithm required about 50 minutes to solve a scenario considering 7 nodes and 7 conditions each.

Table 11.3 compares the increase in computation time when the number of nodes is increased while the number of conditions is held constant and vice versa.

	Nodes=150				Conditions=150		
Conditions	Computation Time (sec)	# of Iterations	Comp. Time per Iteration (msec)	Nodes	Computation Time (sec)	# of Iterations	Comp. Time per Iteration (msec)
25	26.4	4977	5.3	25	52.0	17046	3.1
50	82.0	9993	8.2	50	108.6	17839	6.1
75	137.3	12511	11.0	75	171.3	18338	9.3
100	207.0	14971	13.8	100	238.5	18951	12.6
125	293.4	17632	16.6	125	312.2	19502	16.0
150	379.2	20119	18.8	150	379.2	20119	18.8
175	509.3	22859	22.3	175	479.2	20703	23.1
200	613.6	25406	24.2	200	578.4	21324	27.1
225	758.1	28157	26.9	225	676.4	21921	30.9
250	907.8	30743	29.5	250	790.6	22613	35.0
275	1121.4	33503	33.5	275	893.4	23121	38.6
300	1311.4	36311	36.1	300	1028.1	23777	43.2

Table 11.3: Comparison between computation times as the number of nodes or conditions increase.

The number of iterations required for convergence increases at a greater rate when the number of conditions is increased than when the number of nodes is increased. The computation time per iteration increases at a greater rate for an increase in the number of nodes than for an increase in the number of conditions. The overall computation time increases faster with an increasing number of conditions than with an increasing number of nodes. The algorithm might best be employed to solve a scenario with a very large number of nodes and a limited number of conditions. Using more conditions will likely improve modeling precision, but too many conditions provides unnecessary precision. We discuss

the parameters used to determine the computation times in these scenarios in Appendix B along with the sensitivity of computation time and the number of iterations.

11.3 Value Iteration Algorithm and Value Function Approximation Algorithm Comparison

To compare the results of these algorithms we developed a fictional scenario. Since the scenario must be solved by the value iteration algorithm, we chose a limited size of four nodes with fifteen conditions each, which can be solved in approximately fifty seconds. Each period could realistically represent 1–2 months. Each node represents all the traffic signals in a single intersection. The location of the intersections and the office from which the maintenance trips are launched were chosen arbitrarily. Figure 11.2 illustrates their approximate locations. The office is located at A and the traffic signals are located at B, C, D, and E.

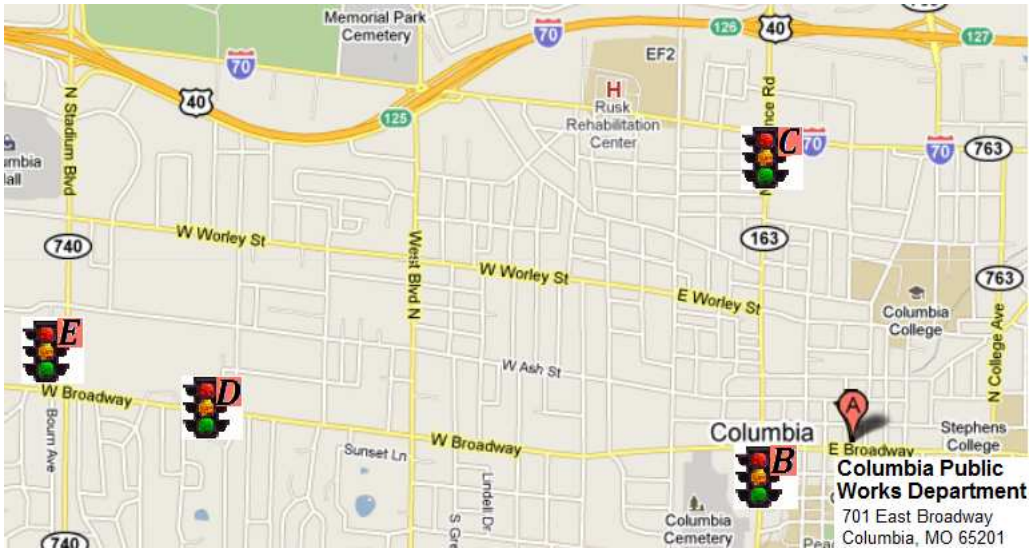

Figure 11.2: Traffic signal location map.

These locations must be translated into the distance matrix parameters. For the value iteration algorithm, this includes the distance between the office and each node, and between

each set of nodes. The value function approximation algorithm only requires the distance between each node and the office. These distances are estimated in meters from the map in Figure 11.2 and displayed in the from–to matrix in Table 11.4. The parameters used by the value function approximation algorithm are shaded.

		To			
		B	C	D	E	A (Office)
From	A (Office)	500	1250	3750	4250	0
	B		1150	3650	4150	500
	C			4400	4900	1250
	D				500	3750
	E					4250

Table 11.4: Distance parameters from–to Matrix (in meters).

Note that the ordering of the traffic signals corresponds to the shortest path tour algorithm proposed in Section 10.2: the traffic signal closest to the office is chosen first, signal B. The traffic signal nearest B is signal C, and so forth with traffic signals D next and E last. The cost of travel is estimated at $1 per kilometer.

Assume that the cost of a maintenance technician is $18 per hour. Two workers are sent out to perform maintenance, and it takes them one hour on average to service all of the traffic signals in an intersection. The intersection D at Stadium and Broadway has a particularly large quantity of traffic signals however, so we allot an extra 15 minutes for this intersection. For the value iteration algorithm, the variable costs to perform maintenance are thus set at $36 for intersections B, C, and D, while intersection E costs $45. When these variable cost parameters are entered into the value function approximation algorithm, the fixed cost is split evenly between them, since this is the lower bound of the fixed cost incurred by performing maintenance in a period. We made the assumption that it takes

an average of 45 minutes (for one technician) to gather the equipment and prepare for a maintenance task, so the fixed cost parameter is set at \$12. This results in the variable cost vector parameter $[39, 39, 39, 48]$ for the value function approximation algorithm.

The technicians are very busy and only have time to perform maintenance on two intersections in a single period (to employ the capacity limit constraint). The discount rate is set at 0.99 for both algorithms since the criterion later used for evaluating the efficiency of the schedules is by long-run average cost per period, which does not include a discount rate. Finally, we assign the estimated cost of allowing a traffic signal to reach a certain level of deterioration per time period arbitrarily. We chose a curve that starts at \$0 for the newest condition and approaches infinity for the 16th condition (since this scenario only allows for 15 conditions). These values are plotted for each increasingly deteriorated condition in Figure 11.3 and can be found in Appendix D.

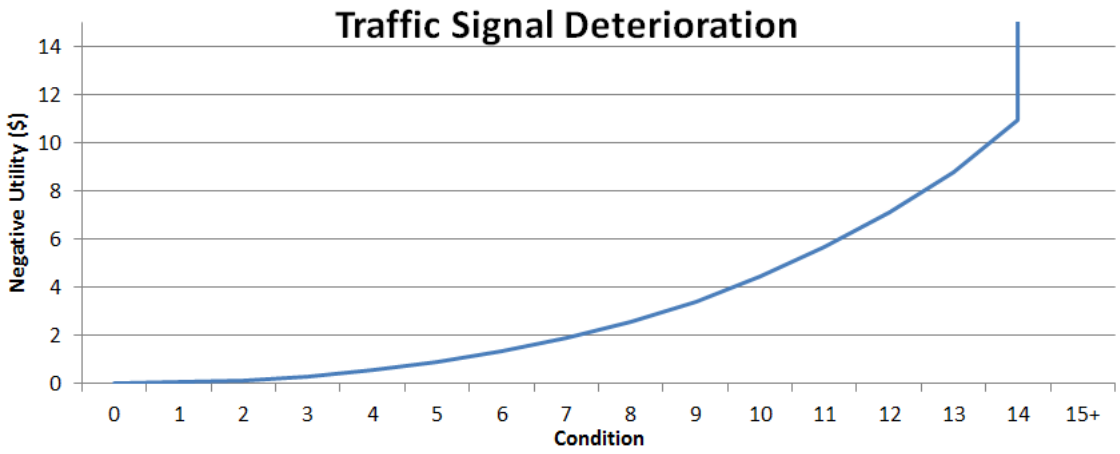

Figure 11.3: Traffic signal deterioration parameters.

With these parameters, the value function approximation algorithm converged on an optimal schedule of $[10, 10, 11, 11]$, representing that the nodes at location B and C should be maintained whenever they reach the condition 10 and nodes D and E should be maintained when they reach the condition 11. This is a very simple and straightforward schedule.

With the same parameters, the value iteration algorithm also converged on a schedule. However, the schedule resulting from the value iteration algorithm specifies an action for every possible system state. A portion of the schedule-output from the program is shown in Figure 11.4.

```
 2377.41     [6, 14, 13, 14]     [0, 1, 0, 1]
 2314.53     [6, 14, 14, 0]      [0, 1, 1, 0]
 2321.12     [6, 14, 14, 1]      [0, 1, 1, 0]
 2328.34     [6, 14, 14, 2]      [0, 1, 1, 0]
 2336.41     [6, 14, 14, 3]      [0, 1, 1, 0]
  2344.5     [6, 14, 14, 4]      [0, 1, 1, 0]
 2351.46     [6, 14, 14, 5]      [0, 1, 1, 0]
 2354.21     [6, 14, 14, 6]      [0, 1, 1, 0]
 2357.26     [6, 14, 14, 7]      [0, 1, 1, 0]
 2360.25     [6, 14, 14, 8]      [0, 1, 1, 0]
 2363.36     [6, 14, 14, 9]      [0, 1, 1, 0]
 2367.07     [6, 14, 14, 10]     [0, 1, 1, 0]
 2370.45     [6, 14, 14, 11]     [0, 1, 1, 0]
 2373.53     [6, 14, 14, 12]     [0, 1, 1, 0]
 2377.31     [6, 14, 14, 13]     [0, 1, 1, 0]
999999.0     [6, 14, 14, 14]     No Feasible Actions
 2214.02     [7, 0, 0, 0]        [0, 0, 0, 0]
```

Figure 11.4: Output representing a portion of the schedule.

The first column is the converged cost of each state. The second column contains the system state vector. There is a row for every possible system state vector—there are 50,625 states in this scenario. Each value in the vector corresponds to the state of each node. The third column represents the optimal action for this state. Each value in the action vector corresponds to the action for each node, where "1" represents performing maintenance on this node and "0" represents not performing maintenance.

The first row of this schedule, for example, indicates that if the first node is in condition six, the second node is in condition fourteen, the third node is in condition thirteen, and the fourth node is in condition fourteen, then the optimal action is to perform maintenance on the second and fourth nodes. For the second-to-last row in the output, there is no feasible action. This is because this scenario only considers fifteen states (zero through fourteen)

and has a capacity limit of two. The state vector indicates three of the four nodes will enter into state fifteen if they are not maintained, yet maintaining all of these nodes will violate the capacity limit. This state has a very large cost so it will not be entered by an optimal schedule.

We created five additional baseline schedules for comparison with the two schedules generated by the algorithms. These baseline schedules call for maintaining each node once it has reached a specific condition. Recalling that the schedule generated by the value function approximation algorithm was $[10, 10, 11, 11]$, the baseline schedules we chose are:

$$Baseline(8) = [8, 8, 8, 8]$$

$$Baseline(9) = [9, 9, 9, 9]$$

$$Baseline(10) = [10, 10, 10, 10]$$

$$Baseline(11) = [11, 11, 11, 11]$$

$$Baseline(12) = [12, 12, 12, 12]$$

$Baseline(8)$, for example, always performs maintenance on nodes when they reach state or condition 8 and may represent a schedule that places a higher priority on safety. $Baseline(12)$ represents a schedule that performs maintenance on nodes when they reach condition 12 and will have high costs from negative utility but lower costs from maintenance.

To evaluate these seven schedules, we developed a program that simulates using each schedule over a set number of periods. The program records the costs of each schedule for each period, and then computes an average cost per period for each schedule. This program runs at the end of the value iteration algorithm and can be found in Appendix I. A small portion of the output from this program is shown in Figure 11.5.

```
************** PERIOD: 25  ***********
<Value Iteration Schedule:>
STATE:      [3, 3, 9, 9]
ACTION:     [0, 0, 0, 0]
NEW STATE: [4, 4, 10, 10]
Cost: 7.4 Running Total Cost: 534.7

<Value Approximation Schedule:>
STATE:      [4, 3, 9, 8]
ACTION:     [0, 0, 0, 0]
NEW STATE: [5, 4, 10, 9]
Cost: 6.8 Running Total Cost: 596.05

<Baseline(8) Schedule:>
STATE:      [8, 7, 5, 5]
ACTION:     [1, 0, 0, 0]
NEW STATE: [0, 8, 6, 6]
Cost: 55.25 Running Total Cost: 637.25
```

Figure 11.5: Sample output from the comparison simulation.

This output shows the actions during period 25 of the simulation. The current state for each schedule, the action dictated by that schedule, and the resulting states are shown. The cost of this period and the running total cost for each schedule are also displayed. The initial state is the same for each schedule and is determined by the final state visited by the value function approximation algorithm so that the nodes will be staggered and the schedules will never result in a capacity limit violation. The simulation ran for 100,000 periods and the average cost per period for each schedule is displayed. These results are shown in Table 11.5.

Schedule	Average cost per period	Percent above optimal
Value Iteration	$22.80	0.0%
Value Approximation	$25.13	10.2%
Baseline [11]	$25.48	11.8%
Baseline [10]	$25.72	12.8%
Baseline [12]	$25.72	12.8%
Baseline [8]	$25.78	13.1%
Baseline [9]	$26.51	16.3%

Table 11.5: Results for the simulation after 100,000 periods.

The schedules are ordered by the least to greatest average cost per period. (We assumed the "optimal" cost to be that of the least-cost schedule—the value iteration schedule.) In this scenario, the value function approximation schedule resulted in 10.2 percent greater costs per period on average than the value iteration schedule. The value function approximation schedule was only marginally better than the best three baseline schedules.

In additional scenarios not documented here, the results varied. While we found no schedule to achieve a lower cost than the schedule produced by the value iteration algorithm, there are many instances where the value iteration, value approximation, and best baseline schedule are all effectively the same and have equal cost. One example where this occurs is when the fixed cost is very small, all nodes have equal variable costs, and all nodes are evenly spaced out or the travel cost is very small. In some cases, where the value function approximation algorithm poorly balances the fixed cost and the staggering of maintenance due to the capacity limit constraint, the best baseline schedule outperforms the value approximation schedule. The results we present in this comparison only represent the effectiveness of the algorithms for a single scenario. Investigating the effectiveness in scenarios with a wide range of parameters would help determine the true performance of these algorithms.

11.4 Stochastic Model Value Iteration Computation Time

The graph in Figure 11.6 shows the computation time for the stochastic model for a single traffic signal. The horizontal axis represents the number of conditions (or states) that the scenario considers. Similar to the first value iteration algorithm, this algorithm assumes convergence and stops when an iteration results in a new cost vector that has a magnitude of difference from the previous cost vector that is less than a beta parameter. This model

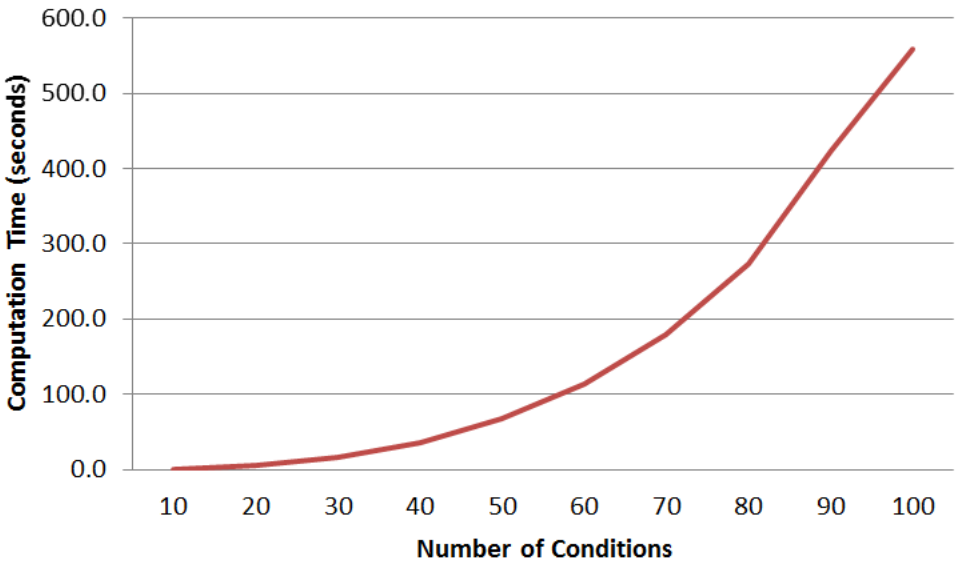

Figure 11.6: Graph of computation time vs. scenario size.

treats each node independently, so each additional node will require additional solve time approximately equal to the first. This algorithm converges faster than the deterministic value iteration model because each traffic signal is considered independently, but not as fast as the value function approximation algorithm. The complete set of parameters for the scenarios used to determine the solve times in the graph are listed in Appendix C.

Instead of an entire intersection, we assumed that in the last scenario discussed in the comparison section, each node represented a single traffic signal. The parameters from that scenario were entered into the stochastic value iteration algorithm, with the cost to replace a traffic signal set to the previous scenario's cost to perform maintenance. We also assumed that the traffic signals deteriorate each period with a probability of 0.65 and the cost to inspect a signal is \$8. A sample of the resulting schedule is shown in Figure 11.7.

This part of the schedule shows only the actions for the first node when it is in state $[6, 7]$. This state represents that the node was inspected seven periods ago and was in

```
STATE:   [6, 7]              COST  /  DANGER  /  TOTAL
 WAIT                         324.9    114.4      439.3
  inspect ->  6 -> wait       313.8    119.9      433.7
  inspect ->  6 -> replace    328.2    110.6      438.8
                                       optimal action:  wait
  inspect ->  7 -> wait       320.1    118.2      438.3
  inspect ->  7 -> replace    328.2    110.6      438.8
                                       optimal action:  wait
  inspect ->  8 -> wait       323.3    118.7      442.0
  inspect ->  8 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
  inspect ->  9 -> wait       329.7    115.2      444.9
  inspect ->  9 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
  inspect -> 10 -> wait       332.9    113.6      446.5
  inspect -> 10 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
  inspect -> 11 -> wait       332.9    114.7      447.7
  inspect -> 11 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
  inspect -> 12 -> wait       332.9    116.1      449.1
  inspect -> 12 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
  inspect -> 13 -> wait       332.9    117.7      450.6
  inspect -> 13 -> replace    328.2    110.6      438.8
                                       optimal action:  replace
 INSPECT                      328.1    110.7      438.8

OPTIMAL ACTION: INSPECT
```

Figure 11.7: A sample schedule for one state.

condition six at that time (and has not been replaced since). The optimal (first) action is indicated in the very last line: inspect. Once inspected, it will be revealed that the traffic signal's actual condition will be six if it never deteriorated in the past seven periods, thirteen if it deteriorated in every one of the past seven periods, or somewhere in between if it deteriorated in some but not all of the periods. Every possibility is displayed, along with the optimal action for each case: if the actual condition is seven or less, do not replace (wait), but if the condition is greater than 7, then replace the node. Finally, the cost and the negative utility (danger) and the total of the two are displayed for each possibility. The decision cost to inspect was computed by a weighted sum of the optimal action costs for each possible outcome resulting from inspecting. These values are weighted by their probability of occurrence, which is not shown on this output. Output displaying the complete set of

probabilities for this scenario is shown in Appendix E.

12 LED Traffic Signal Maintenance Scheduling Conclusions

The value iteration algorithm for the deterministic model was effective for small problem instances. While the algorithm can take advantage of multi-core computing technology to improve the computation times, large scenarios still cannot be solved in a reasonable amount of time. It can solve only scenarios of very limited size. This does not make it a feasible means for producing a schedule for the very large scenarios that traffic signal maintenance scheduling may entail. The schedule produced by this model is also very complex, and may be difficult to implement.

We designed a value function approximation algorithm to compute an approximate solution of the deterministic model for large scenarios. This algorithm can effectively solve much larger scenarios. The value function approximation algorithm can solve scenarios with hundreds of nodes and conditions in reasonable time. The schedule is also much simpler, making it a more feasible option to implement.

In some cases the value iteration algorithm produces a schedule that performs significantly better than the baseline schedules, and always appears to produce schedules that are at least as good. The value function approximation algorithm sometimes produces schedules that are superior to baseline schedules. A range of parameters must be investigated

before the effectiveness of these algorithms can be determined.

The stochastic model can solve scenarios of reasonable size but the computation time is still greater than that of the value function approximation algorithm. The stochastic model does not take into account the travel costs or the effects of a capacity limit on the number of replacements that can be performed per period. While the model includes inspection and stochastic decay, it has not been confirmed from research whether this is a realistic deterioration distribution. The schedule is also complex and may be difficult to implement.

None of these scheduling models consider all of the pertinent aspects of LED traffic signal maintenance scheduling, and the process of schedule implementation has not been examined. A good model should integrate the important elements of maintenance scheduling, be solvable for large, realistic scenarios, and produce a schedule that can be implemented. More research is needed to formulate an efficient model for scheduling the maintenance of traffic signals that meets all of these criteria.

13 LED Traffic Signal Maintenance Modeling Extensions

While much research modeling maintenance activities has been done on single-unit systems [42], perhaps the most unique and important aspect of scheduling the maintenance of LED traffic signals is their huge quantity. A group maintenance policy should therefore be further investigated. The stochastic model considers each traffic signal independently, and the simplified model only considers capacity limits in regard to group maintenance policies. Although the first model and the value iteration algorithm can produce a group maintenance policy that may dictate replacement of a traffic signal or signals when a nearby traffic signal fails, this is only due to the cost savings of reduced travel. A group maintenance policy for LEDs could take into account the type of LED and manufacturer, using statistical sampling and inference to predict the behavior of a population to determine replacement decisions in addition to just their proximity.

There are a variety of factors that could be incorporated into a maintenance model to improve the deterioration prediction of each traffic signal. The duty cycle, color, or manufacturer may all be indicators of signal lifespan [20]. Signals may require inspection or maintenance more frequently depending upon the weather, season, or other environmental factors. Maintenance activities could further be broken down into different components that

could be performed independently, such as snow removal after a snow storm. Each activity could have varying time and costs requirements, and an optimal inspection schedule would determine which subsets of activities to perform during scheduled maintenance.

A more precise modeling of the negative utility of utilizing deteriorated traffic signals may include other methods of estimating the risk besides expectation. More weight, for example, may be placed on higher-risk states that would otherwise have little weight due to being statistically uncommon. The models presented allow for different negative utility values for individual traffic signals. Future investigation could include methods to meaningfully assign parameters, such as a greater negative utility placed on individual signals or intersections that have more traffic or a higher accident rate.

While the first model explored frequent, routine, and deterministic maintenance where travel costs are likely to be significant, this is just one component of a maintenance schedule. Another component explored by the stochastic model was the inspection and replacement activities. These two components must be combined into a single schedule. The effort spent by a technician to access the signal housing may represent the greatest cost component of both the cleaning and inspection procedures [5]. Combining these two tasks may therefore be imperative to maintenance efficiency and so an optimal maintenance schedule would integrate both of these actions.

It can be very challenging to solve complex models in reasonable time for the very large problem instances of LED traffic signal maintenance. The computation time required by algorithms should therefore be considered when developing future models. Modeling accuracy can be sacrificed for reduced algorithm computation times, ideally when the trade-off only marginally diminishes optimality while greatly reducing computation.

The complexity of a schedule must also be gauged by its ability to be implemented.

It must be possible to translate the results of a model into a practical schedule for traffic signal replacement. The additional optimality of a full model may be impractical; but after simplifying for implementation purposes, an approximate model may provide an equally efficient schedule. Future models will likely be designed to be solved for an optimal schedule that is no more complex than the most complex schedule which can be implemented.

A Maintenance Scheduling: Example Parameters for the Value Iteration Algorithm

These scenarios assume the travel cost is approximately $1 per mile and the office is located in the center of a small district that contains the traffic signals to be maintained. The distances were generated randomly using the function RANDBETWEEN(1, 5) in Microsoft Excel, which returns integers between 1 and 5 inclusive with uniform probability. N1 represents node 1, N2 represents node 2, etc., with the distance between nodes in miles. Each box represents the distance from the node labeled on the row on the left side of the table and the node labeled on the column at the top of the table.

We estimated the variable cost to perform maintenance on each node, which includes the cost of labor and equipment, to be between $35 and $45 per node. The function RANDBETWEEN(35, 45) generated this value for each node separately. The variable cost

Fixed Cost:	29
Distance Cost:	1
Discount Rate:	0.9
Number of Processors:	4
Convergence Gap:	0.001

Table A.1: Parameters for the value iteration algorithm.

Node #:	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15
Variable Cost:	42	45	44	43	37	35	41	39	42	36	36	44	36	40	38
Condition:	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
Negative Utility:	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28

Table A.2: Parameters for the value iteration algorithm: variable cost and negative utility.

	TO															
	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12	N13	N14	N15	Office
Office	3	2	4	1	3	2	1	1	2	2	5	4	4	3	2	0
N1	3	4	5	2	3	4	2	1	4	2	5	1	1	1	3	
N2	1	5	4	5	4	2	3	5	3	5	2	2	1	3		
N3	1	4	4	5	5	1	1	4	3	5	3	3	2			
N4	2	4	4	3	2	3	5	5	1	3	3	4				
N5	2	3	4	1	4	2	5	3	5	3	5					
N6	3	4	3	1	3	3	2	5	5	5						
FROM N7	2	1	2	3	1	2	2	4	1							
N8	1	5	3	1	1	5	3	3								
N9	4	5	4	2	4	2	1									
N10	2	2	4	3	1	5										
N11	3	1	5	5	3											
N12	4	1	4	3												
N13	3	1	5													
N14	3	4														
N15	2															

Table A.3: Parameters for the value iteration algorithm: distance.

of each traffic signal may be constant in practice, but may be varied if certain signals or models are known to be more costly to maintain. In the case of nodes, each node may represent an intersection with a different number of traffic signals. The negative utility for each state is the same for each traffic signal in this scenario. The fixed cost is estimated at \$29. The negative utility starts at zero for a traffic signal that was maintained last period and increases by \$2 with each increasingly deteriorated condition. The maintenance capacity limit was set at half the number of nodes in each instance, rounded down (an unrestricting value to ensure a feasible scenario).

Problems that have fewer than 15 nodes or states will not utilize all of these parameters.

Number of Nodes		Number of Conditions								
		4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
4	Comp. Time:	0.8	1.0	1.7	2.5	4.0	6.7	9.4	14.1	19.9
	# Iterations:	*136*	*140*	*143*	*146*	*148*	*150*	*151*	*153*	*155*
5	Comp. Time:	1.6	3.6	8.2	17.8	36.8	68.7	121.5	202.4	318.8
	# Iterations:	*144*	*149*	*153*	*156*	*159*	*162*	*164*	*167*	*168*
6	Comp. Time:	7.3	27.4	92.8	258.3	610.7	1360.9			
	# Iterations:	*152*	*159*	*162*	*167*	*170*	*174*			
7	Comp. Time:	31.3	181.5	788.1	2947.0					
	# Iterations:	*160*	*167*	*172*	*177*					
8	Comp. Time:	263.1	2065.7							
	# Iterations:	*168*	*177*							
9	Comp. Time:	1514.9								
	# Iterations:	*174*								

Table A.4: The computation time and the number of iterations for problems of various sizes.

For instance, a problem with only 5 nodes will use the variable cost of the first 5 values in the variable cost array. Table A.4 displays the computation time and the number of iterations for problems of various sizes.

B Maintenance Scheduling: Example Parameters for the Value Function Approximation Algorithm

For all of the scenarios, the negative utility parameters were set to 0 for the newest condition and increased by one for each deteriorated condition. Additionally, the most deteriorated three conditions were set to a very large number to discourage the algorithm from entering these conditions. The capacity limit was set to half the number of nodes considered by the problem, rounded down to the nearest integer. The penalty cost was set to equal twice the number of conditions considered by the scenario, and the variable cost was set to four times the number of conditions. One unit of distance was set to equal one unit of cost. The convergence gap was set to 0.0000001, in addition to the criterion described after the algorithm is presented. For the first half of the nodes, the distance parameter was set to 1 for the first node and increased by one for each node thereafter. This was repeated, starting from 1, for the second half of the nodes. Finally, the discount rate was set to 0.90. We compiled the computation times in Table B.1 for the scenarios with an equal number of nodes and conditions.

The number of iterations that must be completed to solve a scenario is more sensitive to the parameters of the scenario in the approximation algorithm than the value iteration

Nodes	Conditions	Computation Time (sec)	Iterations	Comp. Time per Iteration (msec)
10	10	0.3	1391	0.2
20	20	1.8	3480	0.5
30	30	4.8	5084	1.0
40	40	9.8	6262	1.6
50	50	18.0	7533	2.4
60	60	29.0	8830	3.3
70	70	44.4	10103	4.4
80	80	64.0	11350	5.6
90	90	89.0	12673	7.0
100	100	119.0	13831	8.6
110	110	154.9	15056	10.3
120	120	199.1	16342	12.2
130	130	251.9	17679	14.2
140	140	310.8	18871	16.5
150	150	379.2	20119	18.8
160	160	454.9	21408	21.3
170	170	547.3	22808	24.0
180	180	648.5	24143	26.9
190	190	766.9	25432	30.2
200	200	879.4	26650	33.0

Table B.1: The computation times for the scenarios with an equal number of nodes and conditions.

algorithm. Comparing the time to run a thousand iterations may therefore be a better indicator of the proportion of time it will take to solve any scenario of one size compared to another size. These values are therefore presented along with the total computation times. This sensitivity is due to the approximation algorithms technique of only visiting practical states. If the scenario parameters make a large portion of the states very costly and impractical, the algorithm will quickly rule them out and concentrate on the remaining states, reaching convergence sooner. The number of iterations is correlated to the sum of the iterations required for each state to converge. During the execution of the value

iteration algorithm many of the states may converge early on, but it will continue to run until the last state converges. The number of iterations that the value iteration algorithm completes is proportional to the number of iterations required by single state requiring the most iterations. The convergence of one state affects the convergence of connected states, so the correlation is not perfect.

C Maintenance Scheduling: Example Parameters for the Stochastic Deterioration Algorithm

	Number of Conditions									
	10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Solve Time:	0.9	4.9	15.9	36.4	67.1	113.9	180.0	273.4	424.3	558.6
Iterations:	*121*	*127*	*142*	*149*	*148*	*150*	*153*	*157*	*157*	*158*

Table C.1: Computation time and number of iterations for the stochastic deterioration algorithm.

The negative utility values are the same as in Scenario A.

	Number of Conditions	
	<= 40	40<
Probability of deterioration:	0.5	0.5
Inspect Cost:	25	250
Replace Cost:	250	2500
Discount Rate:	0.9	0.9
Convergence Gap Beta:	0.001	0.001

Table C.2: Parameters for the computation time analysis of the stochastic deterioration algorithm.

D Maintenance Scheduling: Negative Utility Parameters for Comparison Scenario

Condition:	Negative Utility:
0	0.00
1	0.05
2	0.15
3	0.30
4	0.55
5	0.90
6	1.35
7	1.90
8	2.55
9	3.40
10	4.45
11	5.70
12	7.15
13	8.80
14	10.95
15+	9999.99

Table D.1: Negative utility parameters for comparison scenario.

E Maintenance Scheduling: Probability Matrix

Table E.1 is the probability matrix utilized by the stochastic value iteration algorithm when solving an instance with 15 states and the stochastic decay parameter is set to 0.65. The first row is used when 1 period has elapsed since a traffic signal was inspected: the first column is the probability that the traffic signal has not deteriorated, and the second column is the probability that it has deteriorated by one unit. Each row thereafter represents the situation where one more period has elapsed since the traffic signal was last inspected.

```
 0.35  0.65
0.122 0.455 0.423
0.043 0.239 0.444 0.275
0.015 0.111 0.311 0.384 0.179
0.005 0.049 0.181 0.336 0.312 0.116
0.002  0.02 0.095 0.235 0.328 0.244 0.075
0.001 0.008 0.047 0.144 0.268 0.298 0.185 0.049
  0.0 0.003 0.022 0.081 0.188 0.279 0.259 0.137 0.032
  0.0 0.001  0.01 0.042 0.118 0.219 0.272 0.216   0.1 0.021
  0.0 0.001 0.004 0.021 0.069 0.154 0.238 0.252 0.176 0.072 0.013
  0.0   0.0 0.002  0.01 0.038 0.099 0.183 0.243 0.225  0.14 0.052 0.009
  0.0   0.0 0.001 0.005  0.02 0.059 0.128 0.204 0.237 0.195 0.109 0.037 0.006
  0.0   0.0   0.0 0.002  0.01 0.034 0.083 0.155 0.215 0.222 0.165 0.084 0.026 0.004
  0.0   0.0   0.0 0.001 0.005 0.018 0.051 0.108 0.176 0.218 0.202 0.137 0.063 0.018 0.002
  0.0   0.0   0.0   0.0 0.002  0.01  0.03 0.071 0.132 0.191 0.212 0.179 0.111 0.048 0.013 0.002
  0.0   0.0   0.0   0.0 0.001 0.005 0.017 0.044 0.092 0.152 0.198 0.201 0.155 0.089 0.035 0.009 0.001
************States= 15 ******************
```

Table E.1: The probability matrix utilized by the stochastic value iteration algorithm.

F Hospital Layout: Department MH Flow and Transportation Cost Parameters

Material handling flow matrix

	Dept. B	Dept. C	Dept. D	Dept. E	Dept. F	Dept. G
Dept. A	49	62	17	6	34	0
Dept. B		27	90	48	88	82
Dept. C			7	4	27	92
Dept. D				65	78	37
Dept. E					15	83
Dept. F						45

Table F.1: MH flow cost parameters.

Horizontal transport cost per unit distance

	Dept. B	Dept. C	Dept. D	Dept. E	Dept. F	Dept. G
Dept. A	11	12	16	16	12	17
Dept. B		13	19	11	13	17
Dept. C			14	10	13	10
Dept. D				20	20	11
Dept. E					17	12
Dept. F						11

Table F.2: Horizontal transport cost parameters.

Vertical transport cost per unit distance

	Dept. B	Dept. C	Dept. D	Dept. E	Dept. F	Dept. G
Dept. A	10	10	11	18	19	10
Dept. B		11	10	14	15	12
Dept. C			19	16	12	18
Dept. D				18	18	16
Dept. E					19	16
Dept. F						13

Table F.3: Vertical transport cost parameters.

G Hospital Layout: Facility Cost and Flow Cost of Example Scenarios

COST

			Without PQM		With PQM	
Departments	Floors	Slack	Part 1	Part 2	Part 1	Part 2
5	Single	3.0%		390360		390360
		6.0%	2045.6	373054	2045.6	373054
	Multiple	3.0%		390360		390360
		6.0%	2045.6	373054	2045.6	373054
6	Single	3.0%		685663		685663
		6.0%	2069.2	665158	2069.2	665158
	Multiple	3.0%		685663		685663
		6.0%	2069.2	662632	2069.2	662632
7	Single	3.0%		1E+06		1E+06
		6.0%	2259	1E+06	2259	1E+06
	Multiple	3.0%		978804		978804
		6.0%	2259	859930	2259	859930

Table G.1: Facility cost and flow cost for various test scenarios.

The costs under "Part 1" of Table G.1 represent the facility cost and the costs under "Part 2" represent the flow cost.

NODES EXPLORED

			Without PQM			With PQM		
Departments	Floors	Slack	Part 1	Part 2	Total	Part 1	Part 2	Total
5	Single	3.0%		7,177	14,676		2,068	2,708
		6.0%	7,499	6,538	14,037	640	3,066	3,706
	Multiple	3.0%		5,978	19,674		675	1,697
		6.0%	13,696	6,162	19,858	1,022	7,082	8,104
6	Single	3.0%		36,013	114,861		8,868	46,356
		6.0%	78,848	28,766	107,614	37,488	8,762	46,250
	Multiple	3.0%		226,130	322,583		18,745	58,182
		6.0%	96,453	143,175	239,628	39,437	43,630	83,067
7	Single	3.0%		3,362,001	13,731,213		769,787	1,721,094
		6.0%	10,369,212	1,248,022	11,617,234	951,307	195,064	1,146,371
	Multiple	3.0%		777,725	2,437,934		149,080	538,403
		6.0%	1,660,209	10,746,342	12,406,551	389,323	381,989	771,312

Table G.2: Nodes explored while solving various test scenarios.

H Hospital Layout: Computation Time and Flow Cost of Example Scenarios

	Multiple Floors without PQM		Multiple Floor with PQM		Single Floor, with PQM	
Slack	Time	Flow Cost	Time	Flow Cost	Time	Flow Cost
1%	136.0	1,093,717	223.2	1,093,717	20.2	1,093,717
2%	347.1	1,003,933	30.9	1,003,933	22.0	1,081,591
3%	77.9	978,804	17.4	978,804	53.9	1,078,870
4%	20.4	952,449	14.4	952,449	16.6	1,077,363
5%	427.4	941,602	15.7	941,602	15.7	1,075,539
6%	961.6	859,930	39.3	859,930	14.7	1,074,538
7%	79.5	846,079	23.6	846,079	25.5	1,071,288
8%	161.0	843,792	46.1	843,792	17.3	1,068,979
9%	132.4	842,613	45.1	842,613	22.3	1,066,669
10%	223.4	819,903	30.3	819,903	42.6	1,064,360
11%	352.1	805,711	25.1	805,711	75.4	1,062,051
12%	164.7	796,605	16.9	796,605	200.6	1,060,160
13%	307.1	794,220	38.7	794,220	85.9	1,059,642
14%	1,470.4	794,220	111.8	794,220	131.0	1,059,642
15%	495.8	794,220	144.1	794,220	39.9	1,059,023
16%	314.8	794,220	38.0	794,220	55.1	1,055,488
17%	2,502.1	794,220	76.4	794,220	25.8	1,055,488
18%	1,515.2	794,220	63.9	794,220	44.3	1,055,488
19%	790.9	794,220	197.9	794,220	94.9	1,055,488

Table H.1: Computation time and flow cost for a range of slack parameters, with and without PQM.

I Python Code: Value Iteration and Evaluation Simulation

```
# !/usr/bin/env python
import math
import time
from multiprocessing import Pool, Array
# ======================================================================
# ================ Output Options ======================================
# ======================================================================
NumProcesses = 4
# Number of new processes to start. Set to the number of physical cores.
ShowIteration = 1000
# While running, the program will print out the current GAP for every iteration
# of the above value.
ShowInitialization = False
# Prints out cost-to-go of each action.
PrintSolution = False
# Controls whether the solution is displayed.
RunComparison = True
# Determines if the program runs the comparison seciton at the end. If true,
# there are additional parameters in this section.
# ======================================================================
# ================ Parameters ==========================================
# ======================================================================
L = 4
# Number of nodes
B = 15
# Number of conditions/states for each node;
# NOTE: The youngest a node will be is 1, which is the age of a node when it
# was replaced the previous period.
TIME = 2
# Max number of nodes that can be replaced in a single period.
D1 = [500,1250,3750,4250,0]
D2 = [1150,3650,4150,500]
D3 = [4400,4900,1250]
D4 = [500,3750]
D5 = [4250]
# Distance between each set of nodes. The first and last entries represent the
# office. Number of rows should equal L+1.
```

```
# ***Last entry of first row must be 0 since this is the distance from the
# office to the office.
NegUtility = []
for i in range(L):
    DANG = []
    DANG += [0,.05,.15,.3,.55,.9,1.35,1.9,2.55,3.4,4.45,5.7,7.15,8.8,10.95]
    NegUtility += [DANG]
# Negative utility level of each age of a node.
#      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
CV = [36,36,36,45]
# Variable cost to maintain each node.
CF = 12
# Fixed cost to do any maintenance in a time period.
CD = 0.001
# Cost to drive one unit of distance.
ALPHA = 0.99
# Discount rate for one time period.
BETA = 0.0001
# Stop-gap between the cost vector values between iterations.
# ===============Derived constants=====================================
D1[L] = 0
D = [D1, D2, D3, D4, D5]
# Completes the distance matrix.
S = B ** L
# Number of states.
A = 2 ** L
# Number of actions.
# ======================================================================
# ======== State Vector to Index Translation ==========================
# ======================================================================
def StateToIndex(nextstate):
    # Determine next states index number:
    nextindex = 0
    for k in range(L):
        nextindex += (nextstate[k] - 1) * B ** (L - k - 1)
    nextindex = int(nextindex)
    return nextindex
# ======================================================================
# ======== Route Matrix To DistanceTraveled ===========================
# ======================================================================
def Distance(route):
    # Determines the distance traveled from a route.
    distance = 0
    for k in range(L + 1):
        for m in range(L + 1 - k):
            distance += route[k][m] * D[k][m]
    return distance
# ======================================================================
# ====== Action Vector To NumberNodesInspected =========================
# ======================================================================
def NodesInspected(action):
    # Given action vector, determines the cost from inspecting nodes.
    CostInspecting = 0
    for k in range(1, L + 1):
```

```
        CostInspecting += action[k] * CV[k - 1]
    if CostInspecting > 0:
        CostInspecting += CF
    return CostInspecting
# ========================================================================
# ==== Action Vector To Route Matrix Translation ======================
# ========================================================================
def ActionToRoute(action):
    # Returns the route matrix of an action vector. This is a 2D matrix that
    # corresponds to the distance parameter matrix. 1 represents that this path
    # between two nodes is taken, 0 otherwise.
    route =  []
    for k in range (L + 1):
        TheEnd = 0
        rtd = []
        for m in range(k + 1, L + 2):
            if (((action[k] == 1) and (action[m] == 1)) and (TheEnd == 0)):
                rtd += [1]
                TheEnd = 1
            else:
                rtd += [0]
        route += [rtd]
    return route
# ========================================================================
# ========== Action Index To Vector Translation ========================
# ========================================================================
def IndexToAction(index):
    # Determine action vector from the action index number.
    action = []
    action += [1]
    top = A
    bottom = 0
    for k in range(L):
        middle = bottom + ((top-bottom)/2)
        if (index + 1) <= middle:
            action += [0]
            top = top - ((top-bottom)/2)
        else:
            action += [1]
            bottom += (top-bottom)/2
    action += [1]
    return action
# ========================================================================
# ======================== Format Action Vector ==========================
# ========================================================================
def FormatAction(Action):
    # Removes the placeholder value at the begining & end of action vector.
    FNAction = []
    FNAction[:] = Action[:]
    delete = FNAction.pop()
    delete = FNAction.pop(0)
    return FNAction
# ========================================================================
# ======================== Format State Vector ===========================
```

```
# ========================================================================
def FormatState(State):
    # Subtracts one from each state value. This is a cosmetic change so that
    # this algorithm will match the paper model.
    NewState = []
    for i in range(len(State)):
        NewState += [State[i]-1]
    return NewState
# ========================================================================
# ========== State Index To Vector Translation ===========================
# ========================================================================
def IndexToState(index):
# Returns a state vector given the index.
    state = []
    top = S
    for k in range(L):
        marker = top - B ** (L - k - 1)
        q = B
        found = 0
        while (found == 0):
            if (index + 1) >= marker + 1:
                found = 1
                state += [q]
                top = marker + B ** (L - k - 1)
            else:
                marker = marker - B ** (L - k - 1)
                q = q - 1
    return state
# ========================================================================
# ========= DIFFERENCE FUNCTION DEFINITION ===============================
# ========================================================================
def TakeDifference(old, new):
    # Computes the GAP between cost vectors.
    z = 0
    for i in range(S):
        z += (abs(old[i] - new[i])) ** 2
    z = math.sqrt(z)
    return z
# ========================================================================
# ====== Compute Cost Of This Schedule ===================================
# ========================================================================
def ScheduleCost(tcost, CurrentState):
    expense = 0
    i = CurrentState
    cost = []
    expensecost=[]
    thisaction = []
    ResultingState=[]
    for j in range(len(next_state_index[i])):
        cost += [innate_cost[i][j] +\
                 ALPHA * tcost[next_state_index[i][j]]]
        expensecost += [innate_cost[i][j]]
        ResultingState += [next_state_index[i][j]]
        thisaction += [short_action[i][j]]
```

```
    if len(cost) >0:
        # Prints out the optimal action.
        optIndex = cost.index(min(cost))
        optaction = thisaction[optIndex]
        expense = expensecost[optIndex]
        mynextstateindex = ResultingState[optIndex]
    else:
        print 'ERROR!!! NO FEASIBLE ACTIONS FOR THIS STATE!!'
    return expense, mynextstateindex, optaction
# ======================================================================
# ============ WORKER FUNCTION DEFINITIONS ============================
# ======================================================================
my_innate_cost = None
my_oldcost = None
my_next_state_index = None
def worker_init(innate_cost, oldcost, next_state_index):
    global my_innate_cost, my_oldcost, my_next_state_index
    my_innate_cost = innate_cost
    my_oldcost = oldcost
    my_next_state_index = next_state_index
def worker(given_state):
    arr = [InnateCost + ALPHA * my_oldcost[NextState]
        for (InnateCost, NextState) in\
        zip(my_innate_cost[given_state],\
            my_next_state_index[given_state])]
    arr += [999999]
    return min(arr)
# ======================================================================
# ====== MAIN FUNCTION DEFINITION WITH PRINT ==========================
# ======================================================================
def RunPrintIteration(tcost):
    newtcost = []
    for i in range(S):
        cost = []
        thisaction = []
        for j in range(len(next_state_index[i])):
            cost += [innate_cost[i][j] +\
                     ALPHA * tcost[next_state_index[i][j]]]
            thisaction += [IndexToAction(short_action[i][j])]
        if len(cost) > 0:
            # Assigns lowest cost as the cost for this action.
            newtcost += [min(cost)]
            # Prints out the optimal action.
            optIndex = cost.index(min(cost))
            optaction = thisaction[optIndex]
            print str(round(newtcost[i], 2)).rjust(8), '   ',
            print FormatState(age[i]), '   ', FormatAction(optaction)
        else:
            newtcost += [0]
            print str(round(tcost[i], 2)).rjust(8), '   ',
            print FormatState(age[i]), '    No Feasible Actions'
    return newtcost
# ======================================================================
# ================== Preprocessing =====================================
```

```
# ====================================================================
if __name__ == '__main__':
    start = time.time()
    print'**********TRAFFIC SIGNAL PROGRAM V7*****************'
    print ' Number Nodes=', L, 'Actions=', A
    print ' Max Age=', B, 'States=', S, 'Capacity Limit=', TIME
    print'*****************************************************'
    print
    print 'Preprocessing..'
    print '    Stage 1 of 4..',
    if ShowInitialization:
        print
        print 'Cost-to-go of each action:'
    thiscost = []
    x = []
    for i in range(A):
        # Determine action vector from the action index number:
        x += [IndexToAction(i)]
        # Create rout matrix from the action vector:
        v = ActionToRoute(x[i])
        # Determine the costs from inspection of this action:
        inspectCosts = NodesInspected(x[i])
        # Compute distance traveled by this route:
        dist = Distance(v)
        # Determine total cost-to-go of this action, no neg util.
        thiscost += [inspectCosts + CD * dist]
        if ShowInitialization:
            print FormatAction(IndexToAction(i)), '   ', thiscost[i]
    print ' Complete.'
    print '    Stage 2 of 4..',
    age = []
    for i in range(S):
        # Determine each state's age vector from index number:
        age += [IndexToState(i)]
    print ' Complete.'
    print '    Stage 3 of 4..',
    # Determine feasibility of each action/state combination. And if feasible,
    # determine parameters.
    next_state_index = []
    # "next_state_index" contains an array for each state. Each array contains
    # a value for each feasible action for that state.  The value is the index
    # of the next state resulting from performing that action in that state.
    innate_cost = []
    # "innate_cost" contains an array for each state.  Each array contains a
    # value for each feasible action for that state. The value is equal to the
    # cost-to-go of that action. The cost-to-go includes the variable costs of
    # maintenance, the distance travel costs, the fixed cost, and the risk
    # incurred from the resulting state.
    short_action = []
    # "short_action" contains an array for each state. Each array contains a
    # value for each feasible action for that state.  The value is the index of
    # that action.
    fullnextindex = []
    for i in range(S):
```

```
        IndexA = []
        thiscostA = []
        shortaction = []
        for j in range(A):
            # Determine feasibility.
            ReplaceCount = 0
            for k in range(L):
                if ((x[j][k + 1] == 0) and (age[i][k] == B)):
                    break
                if (x[j][k + 1] == 1):
                    ReplaceCount += 1
                if (ReplaceCount > TIME):
                    break
            else: # Only runs when the loops completes without breaking,
                  # meaning the action is feasible.
                shortaction += [j]
                nextstate = []
                thisDang = 0
                for k in range(L):
                    thisDang += NegUtility[k][age[i][k]-1]
                    if (x[j][k + 1] == 1):
                        nextstate += [1]
                    else:
                        nextstate += [age[i][k] + 1]
                IndexA += [StateToIndex(nextstate)]
                thiscostA += [thiscost[j]+thisDang]
        # This section initializes the values of the arrays below. This is so
        # that they will not have to be re-computed each iteration.
        innate_cost += [thiscostA]
        next_state_index += [IndexA]
        short_action += [shortaction]
# ========================================================================
# ================ MAIN ==================================================
# ========================================================================
    # Initialize states:
    print ' Complete.'
    print '    Stage 4 of 4..',
    newcost = [1 for i in range(S)]
    oldcost = Array('d', S, lock = False)
    diff = BETA + 1
    it = 0
    cutnum = 1
    # Initialize processors:
    mapfunc = None
    if (NumProcesses > 1):
        pool = Pool(processes = NumProcesses, initializer = worker_init,\
                    initargs = (innate_cost, oldcost, next_state_index))
        mapfunc = pool.map
    else:
        worker_init(innate_cost, oldcost, next_state_index)
        mapfunc = map
    # Record times:
    OldTime = time.time()
    IterationStart = time.time()
```

```
    print ' Complete.'
    print 'Preprocessing Complete.  Elapsed Time:',
    print round((time.time()-start),2)
    print
    print 'Beginning iterations..(', NumProcesses,'Processes )'
    # Main loop:
    while diff > BETA:
        oldcost[:] = newcost[:]
        newcost = mapfunc(worker, range(S))
        it += 1
        diff = TakeDifference(oldcost, newcost)
        if ShowIteration > 0:
            if (int(it / ShowIteration) == cutnum):
                cutnum += 1
                print 'Iteration:', str(it).rjust(3),
                print '   GAP=', str(round(diff, 4)).rjust(10),
                print '    Processing time:',
                print round((time.time() - OldTime), 2)
                OldTime = time.time()
    elapsed = round((time.time() - start), 2)
    if PrintSolution:
        print
        print '--- Cost  /   State   /   Action ---'
        newcost = RunPrintIteration(oldcost)
    print '-----------------------------------------'
    print 'Elapsed time:', str(elapsed).rjust(7),
    print '   Avg Iteration time:',
    print round((time.time()-IterationStart)/it, 2)
    print 'Total iterations:', str(it).rjust(3),
    print '   Final GAP=', str(round(diff, 4)).rjust(8)
# ======================================================================
# ====== Main Body of Schedule Cost Computation ========================
# ======================================================================
    if RunComparison:
        # ****************Parameters*************
        InitialState = [2,1,9,8]
        schedule = [10,10,11,11]
        base = [8,8,8,8]
        # NOTE: the "youngest state" is 1, so one is added to each of these.
        Turns = 25
        LongRun = 100000
        # ***************Body********************
        for i in range(L):
            base[i] += 1
            schedule[i] += 1
            InitialState[i] += 1
        NewStateIndex = StateToIndex(InitialState)
        NewExpense = 0
        TotalExpense = 0
        OldStateIndex = 0
# Approx.
        NewActionIndex = 0
        SNewStateIndex = StateToIndex(InitialState)
        SNewExpense = 0
```

```
        STotalExpense = 0
        SNewActionIndex = 0
        OverLimit = 0
# Baseline.
        PNewStateIndex = StateToIndex(InitialState)
        PNewExpense = 0
        PTotalExpense = 0
        PNewActionIndex = 0
        POverLimit = 0
        for h in range(LongRun):
            if (h < Turns):
                print '************** PERIOD:',h+1,' **************'
                print '<Value Iteration Schedule:>'
                print 'STATE:    ', FormatState(age[NewStateIndex])
            OldStateIndex = NewStateIndex
            NewExpense, NewStateIndex, NewActionIndex = \
                        ScheduleCost(newcost, OldStateIndex)
            TotalExpense += NewExpense
            if (h < Turns):
                print 'ACTION:   ', FormatAction(IndexToAction(NewActionIndex))
                print 'NEW STATE:', FormatState(age[NewStateIndex])
                print 'Cost:', NewExpense, 'Running Total Cost:', TotalExpense
                print
                print '<Value Approximation Schedule:>'
                print 'STATE:    ', FormatState(age[SNewStateIndex])
            SOldState = IndexToState(SNewStateIndex)
            SNewAction = [1]
            SNewState = []
            SNegUt = 0
            replacements = 0
            for i in range(L):
                if SOldState[i] >= schedule[i] and replacements == TIME:
                    OverLimit += 1
                if SOldState[i] >= schedule[i] and replacements<TIME:
                    SNewAction += [1]
                    SNewState += [1]
                    replacements += 1
                else:
                    SNewAction += [0]
                    SNewState += [SOldState[i]+1]
                    if SNewState[i]>B:
                        print 'ERROR! State out-of-range!',
                        print 'Infeasible Schedule!'
            SNewAction += [1]
            for i in range(L):
                SNegUt += NegUtility[i][SOldState[i]-1]
            SNewStateIndex = StateToIndex(SNewState)
            SNewExpense = Distance(ActionToRoute(SNewAction))\
                         *CD+SNegUt+NodesInspected(SNewAction)
            STotalExpense += SNewExpense
            if (h < Turns):
                print 'ACTION:   ', FormatAction(SNewAction)
                print 'NEW STATE:', FormatState(SNewState)
                print 'Cost:', SNewExpense, 'Running Total Cost:',
```

```
        print STotalExpense
        print
        print '<Baseline(8) Schedule:>'
        print 'STATE:    ', FormatState(age[PNewStateIndex])
    POldState = IndexToState(PNewStateIndex)
    PNewAction = [1]
    PNewState = []
    PNegUt = 0
    Preplacements = 0
    for i in range(L):
        if POldState[i] >= base[i] and Preplacements == TIME:
            POverLimit += 1
        if POldState[i] >= base[i] and Preplacements<TIME:
            PNewAction += [1]
            PNewState += [1]
            Preplacements += 1
        else:
            PNewAction += [0]
            PNewState += [POldState[i]+1]
            if PNewState[i]>B:
                print 'ERROR! State out-of-range!',
                print 'Infeasible Schedule!'
    PNewAction += [1]
    for i in range(L):
        PNegUt += NegUtility[i][POldState[i]-1]
    PNewStateIndex = StateToIndex(PNewState)
    PNewExpense = Distance(ActionToRoute(PNewAction))\
                 *CD+PNegUt+NodesInspected(PNewAction)
    PTotalExpense += PNewExpense
    if (h < Turns):
        print 'ACTION:   ', FormatAction(PNewAction)
        print 'NEW STATE:', FormatState(PNewState)
        print 'Cost:', PNewExpense,
        print 'Running Total Cost:', PTotalExpense
print
if OverLimit+POverLimit > 0:
    print '# Capacity Limit Restrictions, Approximate:', OverLimit
    print '# Capacity Limit Restrictions, Baseline:', POverLimit
    print
print'********* Long Run Costs **********'
print 'Exact:         ',round(TotalExpense/LongRun,2),'per period'
print 'Approximate:   ',round(STotalExpense/LongRun,2),'per period'
print 'Baseline:      ',round(PTotalExpense/LongRun,2),'per period'
print 'Exact/Approx Difference: ',
print round((STotalExpense-TotalExpense)/LongRun,3),'per period; ',
print round(((STotalExpense-TotalExpense)/LongRun)\
           /(STotalExpense/LongRun)*100,2),
print '%'
print 'Approx/Baseline Difference: ',
print round((PTotalExpense-STotalExpense)/LongRun,3),'per period; ',
print round(((PTotalExpense-STotalExpense)/LongRun)\
           /(PTotalExpense/LongRun)*100,2),
print '%'
print 'Exact/Baseline Difference: ',
```

```
print round((PTotalExpense-TotalExpense)/LongRun,3),'per period; ',
print round(((PTotalExpense-TotalExpense)/LongRun)\
            /(PTotalExpense/LongRun)*100,2),
print '%'
```

J Python Code: Value Function Approximation

```
import sys
import math
import time
import csv
# ======================================================================
# ================ Output Options ======================================
# ======================================================================
ShowIteration = 1000
# While running, the program will print out the current GAP for every iteration
# of the above value.
PrintCost = False
# Prints the cost of each states.
PrintOutput = True
# Prints results to an output file for analysis.
rintRestarts = 0
# Prints a note every time that the algorithm reaches N iterations within the
# convergence gap but then restarts. 0 prints nothing.
PrintSolution = False
# Prints out the state that each node should be maintained.
PrintThisState = False
# Prints the state for the range below.
PrintUtilCost = False
# Prints the cost and the negative utility for each state of each node Also for
# the range below:
FirstIts = 0
# Set equal to an integer >= 0. Prints this data for the first "FirstIts"
# iterations.
LastIts = 0
# Prints this data for the last "LastIts" iterations. MUST be less than B.
# Also, may print other iterations if they meet enough consecutive convergence
# criteria that they appear to be the final iterations.
# ======================================================================
# ================ Parameters ==========================================
# ======================================================================
L = 10
# Number of nodes.
B = 6
```

```
# Oldest allowable age of a node; NOTE: The youngest a node will be is 1, which
# is the age of a node when it was replaced the previous period.
TIME = int(L/2)
# Number of nodes that can be replaced in one period.
PENALTY = 2*B
# Penalty for exceeding the number of nodes that can be replaced in one period.
DIST = []
OneDist = 1
for x in range(L):
    DIST += [OneDist]
    if OneDist == int(L/2):
        OneDist = 1
    else:
        OneDist += 1
# Distance between each set of nodes.
NegUtility = []
for i in range(L):
    DANG = []
    DangCount = 0
    for x in range(B-int((L+TIME-1)/TIME)):
        DANG += [DangCount]
        DangCount += 1
    for x in range(int((L+TIME-1)/TIME)):
        DANG += [99999]
    NegUtility += [DANG]
# Negative utility of each age of a node. Must be defined for B+1 and must be
# set to 999+ for at least the last (L/TIME) elements within B.
CV = []
for i in range(L):
    CV += [4*B]
#        1    2    3    4    5    6    7    8    9   10
# CV = [18., 18., 18., 18., 18., 18., 18., 18., 18., 18.,
#       18., 18., 18., 18., 18.]
# Variable cost to perform maintenace on each node.
DISTSCALE = 1
# Cost to travel 1 unit distance.
ALPHA = 0.90
# Discount rate for one time period.
BETA = 0.0000001
# Stop-gap between the cost vector values between iterations.
MAXITERATION = 500000
# Stop after this number of iterations regardless of convergence.
JUMP = 1
# Rate that the algorithm changes between costs.
# ==============Derived constants=======================================
for i in range(min(len(DIST),len(CV))):
    DIST[i] = (2*(DIST[i]*DISTSCALE))+CV[i]
# This makes the DIST array contain all the costs-to-go for each node.
StartTime = time.time()
# Record start time.
# ======================================================================
# ================= Main Loop ==========================================
# ======================================================================
def GetAction(oldcost, oldstate):
```

```
WaitCost = []
ReplaceCost = []
CostDiff = []
for i in range(L):
    # Compute the difference in cost of replacing this node.
    WaitCost += [ALPHA*oldcost[i][oldstate[i]+1]+\
                 NegUtility[i][oldstate[i]]]
    ReplaceCost +=  [ALPHA*oldcost[i][0]+DIST[i]\
                    +NegUtility[i][oldstate[i]]]
    CostDiff += [(WaitCost[i])-(ReplaceCost[i])]
numleft = TIME
newstate = []
newcost = []
for i in range(L):
    newstate += [0]
    blank = []
    for j in range(B):
        blank += [oldcost[i][j]]
    newcost += [blank]
    # Initializes new array.
replacements = 0
totalAge = 0
for i in range(L):
    totalAge += oldstate[i]
    if CostDiff[i] > 0:
        replacements += 1
# Determines the number of nodes to be maintained by this action.
if totalAge == 0:
    totalAge = 1
pen = 0
if replacements > TIME:
    NumOverLimit = replacements-TIME
    pen = PENALTY*(replacements-TIME)
else:
    NumOverLimit = 0
for i in range(L):
    first = max(CostDiff)
    index = CostDiff.index(first)
    if (first > 0)and(numleft>0):
    # Computes cost for each node that is maintained. Also computes the
    # resulting state of this action.
        newstate[index] = 0
        newcost[index][oldstate[index]] = JUMP*(ReplaceCost[index]\
            +(oldstate[index]/totalAge)*pen)\
            +(1-JUMP)*oldcost[index][oldstate[index]]
    if (first > 0)and(numleft <= 0):
    # Computes cost for maintained nodes past the capacity limit. Also
    # computes the resulting state of this action.
        newstate[index] = (oldstate[index]+1)
        newcost[index][oldstate[index]] = JUMP*(ReplaceCost[index]\
            +(oldstate[index]/totalAge)*pen)\
            +(1-JUMP)*oldcost[index][oldstate[index]]
    if (first <= 0):
    # Computes new costs and states for nodes that are not maintained.
```

```
            newstate[index] = (oldstate[index]+1)
            newcost[index][oldstate[index]] = JUMP*(WaitCost[index]\
                +(oldstate[index]/totalAge)*pen)\
                +(1-JUMP)*oldcost[index][oldstate[index]]
        CostDiff[index] = -99999
        numleft -= 1
    return newstate, newcost, NumOverLimit
# ======================================================================
# ========= DIFFERENCE FUNCTION DEFINITION ============================
# ======================================================================
def TakeDifference(old, new):
    # Computes the GAP between cost vectors.
    z = 0
    for i in range(L):
        for j in range(B):
            z += (abs((old[i][j])-(new[i][j])))**2
    z = math.sqrt(z)
    return z
# ======================================================================
# ================ Program =============================================
# ======================================================================
# Initialize states.
TotalOverLimit = 0
ThisState = []
MaxState = []
LastCost = []
for i in range(L):
    MaxState += [0]
    ThisState += [0]
    tempcost = []
    for j in range(B):
        tempcost += [NegUtility[i][j]]
    LastCost += [tempcost]
ThisCost = LastCost
diff = BETA+1
it = 0
cutnum = 1
Convergence = 0
NumOverLimit = 0
OldTime = time.time()
OutputFile = csv.writer(open('output.csv','w'))
RunNum = 0
if PrintCost:
    print 'Initialized Costs'
    print 'NODE:','STATE:','COST:'
    for i in range (L):
        for j in range(B):
            print str(i).rjust(4),
            print str(j).rjust(6),
            print str(round(ThisCost[i][j], 1)).rjust(8)
    print
# Main Loop.
print 'Beginning iterations..   '
while ((Convergence<B)and(it<MAXITERATION)):
```

```
    LastCost = ThisCost
    ThisState, ThisCost, NumOverLimit = GetAction(LastCost, ThisState)
    RunNum += NumOverLimit
    it += 1
    diff = TakeDifference(LastCost, ThisCost)
    if diff < BETA:
        for i in range(L):
            if MaxState[i] < ThisState[i]:
                MaxState[i] = int(ThisState[i])
        Convergence += 1
        TotalOverLimit += NumOverLimit
    if diff>BETA and Convergence>0:
        if PrintRestarts>0 and (Convergence>PrintRestarts):
            print '**************Restart:', Convergence
        for i in range(L):
            MaxState[i] = 0
        Convergence = 0
        TotalOverLimit = 0
    if PrintOutput:
        OutputFile.writerow(ThisState)
    if ShowIteration>0:
        if (int(it/ShowIteration) == cutnum)\
           or((it <= FirstIts or Convergence > (B-LastIts))\
              or it>(MAXITERATION-LastIts)):
            cutnum += 1
            print 'Iteration:', str(it).rjust(6),
            print ' GAP=', str(round(diff, 6)).rjust(10),
            print ' Proc Time:',
            print str(round((time.time() - OldTime),2)).rjust(7),
            print ' Penalties:', RunNum
            RunNum = 0
    if ((it <= FirstIts or Convergence > (B-LastIts))\
        or it > (MAXITERATION-LastIts)):
        if PrintThisState:
            print '   New System State:', ThisState
        if PrintUtilCost:
            print 'NODE:   STATE:   NEG_UTIL:     COST:'
            for i in range (L):
                for j in range(B):
                    print str(i).rjust(4), str(j).rjust(6),
                    print str(round(NegUtility[i][j], 1)).rjust(15),
                    print str(round(ThisCost[i][j], 1)).rjust(9)
elapsed = round((time.time() - StartTime), 2)
if PrintSolution:
    print 'Solution:',MaxState
print 'Elapsed time:', str(elapsed).rjust(7),
print '   Total Iterations:', it,
if (TotalOverLimit > 0):
    print '   Infeasibility:', round((TotalOverLimit/B),2)
```

K Python Code: Stochastic Deterioration Algorithm

```
import sys
import math
import time
# ====================================================================
# ================== PRE-SOLVE OPTIONS ===============================
# ====================================================================
ShowPrMatrix = False
# True = Show; False = Don't Show.
# This displays the probability equation and term values for each possible
# number of periods since inspection.
ShowRawPrMatrix = False
# True = Show; False = Don't Show.
# This displays the unformatted probability matrix.
ShowStateDanger = False
# True = Show; False = Don't Show.
# This displays the danger of each state.
ShowRawDangerMatrix = False
# True = Show; False = Don't Show.
# This displays the unformatted danger matrix.
# ====================================================================
# ================ SOLVE OPTIONS =====================================
# ====================================================================
SolveProgram = True
# True = Show; False = Don't Show.
# This solves the dynamic program for the parameters given,
# This must be set to 'True' for the next two display options to execute.
ShowIteration = 10
# Must be an integer >= 0. Displays the iteration and the GAP while solving.
# Shows only every "nth" value. The value of "0" results in none being
# displayed.
ShowSolution = False
# True = Show; False = Don't Show.
# For each state and action, this displays the total cost, negative utility,
# and the sum of the two, once convergence is reached. Also displays the
# optimal decisions, and the total iteration and final gap.
# ====================================================================
# ================ PARAMETERS ========================================
```

```
# ========================================================================
B = 70
# Number of conditions to be considered. The newest a node will be is 0, but
# the earliest it will have been inspected is 1, representing the node was
# inspected last period.
HEADS = 0.5
# Probability that a node will deteriorate by one unit during a period.
DANGER = [0]
for i in range(B+10):
    DANGER += [2*i]
MAXDANGER = 999999999
# This is the set negative utility for any actual decay level greater than B,
# and overrides the DANGER vector.
MAXCOST = 999999999
# This is the total_cost of any state greater than [B, B].
CDANGER = 1
# This is a weight for the value of one unit of neg utility for one period.
CINSPECT = 250
# Cost to inspect a node.
CREPLACE = 2500
# Cost to replace a node, after inspecting it.
COST_ALPHA = 0.90
# Cost discount rate for one time period.
RISK_ALPHA = COST_ALPHA
# Risk discount rate for one time period.
BETA = 0.001
# Stop-gap between the cost/risk vector values between iterations.
# ========================================================================
# ========== DERIVED CONSTANTS ===========================================
# ========================================================================
S = (B+1)*(B)
# Number of states. Last known condition can be 0-B, and the time since last
# inspection can be 1-B.
DANGER = DANGER*CDANGER
# Weights the danger so that it is the same units as the cost.
oldCost = []
oldDang = []
oldTotal = []
for v in range(S):
    oldCost += [0]
    oldDang += [0]
    oldTotal += [0]
# Declaration of the long-run danger and cost vectors.
start = time.time()
# Starts timer.
# ========================================================================
# ========== Create Probability Matrix ===================================
# ========================================================================
# To save computation time, the probability matrix is computed once and saved.
# This matrix contains the probability of the actual condition of the node.
PROB = []
nFact = 1
if ShowPrMatrix:
    print 'Equation:  [p^k] * [p^(n-k)] * [n! / (k!*(n-k)!) ]'
```

```
    print 'Where:'
    print '   p = ', HEADS,', the probability of a node decaying each period'
    print '   k = the number of periods that the node decayed'
    print '   n = the number of periods that have passed'
    print
    print '   n    k  (n-k)       n!          k!       (n-k)!    Probability'
    print
for n in range(B+1):
    PROBrow = []
    kFact = 1
    for k in range(n+2):
        nmkFact = 1
        for j in range((n+1)-k):
            nmkFact = nmkFact*(j+1)
        probabilityA = (HEADS**(k))*((1-HEADS)**((n+1)-k))
        probabilityB = (nFact)/(kFact*nmkFact)
        probability = probabilityA*probabilityB
        PROBrow += [probability]
        if ShowPrMatrix:
            print str(n+1).rjust(4), str(k).rjust(4),
            print str((n+1)-k).rjust(4),str(nFact).rjust(11),
            print str(kFact).rjust(11), str(nmkFact).rjust(11),
            print str(round(probability, 6)).rjust(12)
        kFact = kFact*(k+1)
    PROB += [PROBrow]
    nFact = nFact*(n+2)
if ShowRawPrMatrix:
    print 'Probability Matrix:'
    print PROB
# ================================================================
# ========== Returns Long-Term Danger ============================
# ================================================================
def GetSavedDanger(state):
    if ((state[0] <= (B)) and (state[1] <= B)):
        savedD = oldDang[StateToIndex(state)]
    else:
        savedD = ReturnDanger([0,1])+RISK_ALPHA*oldDang[StateToIndex([0,1])]
    return savedD
# ================================================================
# ============= Returns Long-Term Cost ===========================
# ================================================================
def GetSavedCost(state):
    if ((state[0] <= (B)) and (state[1] <= B)):
        savedC = oldCost[StateToIndex(state)]
    else:
        savedC = CREPLACE+COST_ALPHA*oldCost[0]
    return savedC
# ================================================================
# ======== State Vector to Index Translation =====================
# ================================================================
def StateToIndex(state):
    # Determine next state's index number:
    index = 0
    if state[0] <= B and state[1] <= B and state[0] >= 0 and state[1] >= 1:
```

```
        index = B*state[0]+state[1]-1
        index = int(index)
    else:
        print'***** ERROR: StateToIndex() out of range! *****'
        print'*****',state,'*****'
    return index
# ======================================================================
# ========== State Index To Vector Translation ========================
# ======================================================================
def IndexToState(index):
# Returns a state vector given the index. State vectors are the form:
# [Last_Known_State, Periods_Since_Last_Inspection].
    state = []
    indexA = int(index/B)
    indexB = index-(indexA*B)+1
    state += [indexA, indexB]
    if index > S:
        print'***** ERROR: IndexToState() out of range! *****'
        print'***** Index:',index,'*****'
    return state
# ======================================================================
# ================ MAIN FUNCTION DEFINITION ============================
# ======================================================================
def RunIteration(prntB):
    newcostsdangers = []
    newcosts = []
    newdangers = []
    for i in range(S):
        # For each state:
        costs = []
        dangers = []
        costsdangers = []
        action = []
        # Determine this state's [last_condition, time_since_inspection] vector
        # from index number:
        wear = IndexToState(i)
        for j in range(1, 3):
            # Determine next states' vectors and the danger:
            tempstate = []
            thisDanger = 0
            thiscost = 0
            if (j == 1):
                if prntB == 1:
                    print 'STATE:',str(wear).rjust(8),
                    print'             COST  /  DANGER  /  TOTAL'
                tempstate = [wear[0],(wear[1]+1)]
                thisDanger = ReturnDanger(tempstate)\
                            +RISK_ALPHA*GetSavedDanger(tempstate)
                thiscost += COST_ALPHA*GetSavedCost(tempstate)
            if (j == 2):
                # Determine the cost/danger of replaceing:
                tempstate = [0,1]
                ReplaceDanger = ReturnDanger(tempstate)\
                              +RISK_ALPHA*GetSavedDanger(tempstate)
```

```
            ReplaceCost = CINSPECT+CREPLACE\
                          +COST_ALPHA*GetSavedCost(tempstate)
            ReplaceDangerCost = ReplaceDanger+ReplaceCost
            for v in range(wear[1]+1):
            # For each possible state of decay:
                # Determine the cost/danger of inspecting & not replacing:
                tempstate = [wear[0]+v, 1]
                InspectDanger = ReturnDanger(tempstate)\
                                +RISK_ALPHA*GetSavedDanger(tempstate)
                InspectCost = CINSPECT+COST_ALPHA*GetSavedCost(tempstate)
                InspectDangerCost = InspectDanger+InspectCost
                # Determine if it's cheaper to replace or not
                if ReplaceDangerCost<InspectDangerCost:
                    DecisionCost = ReplaceCost
                    DecisionDanger = ReplaceDanger
                else:
                    DecisionCost = InspectCost
                    DecisionDanger = InspectDanger
                # Determine the expected cost/danger of inspecting (vs. not
                # inspecting):
                thisDanger += DecisionDanger*PROB[wear[1]-1][v]
                thiscost += DecisionCost*PROB[wear[1]-1][v]
                if prntB == 1:
                    print str('inspect ->').rjust(12),
                    print str(wear[0]+v).rjust(2),
                    print str('-> wait   ').rjust(10),
                    print str(round(InspectCost, 1)).rjust(8),
                    print str(round(InspectDanger, 1)).rjust(8),
                    print str(round((InspectDangerCost), 1)).rjust(10)
                    print str('inspect ->').rjust(12),
                    print str(wear[0]+v).rjust(2),
                    print str('-> replace').rjust(10),
                    print str(round(ReplaceCost, 1)).rjust(8),
                    print str(round(ReplaceDanger, 1)).rjust(8),
                    print str(round((ReplaceDangerCost), 1)).rjust(10)
                    print'                                        ',
                    print'optimal action: ',
                    if ReplaceDangerCost<InspectDangerCost:
                        print 'replace'
                    else:
                        print 'wait'
        costs += [thiscost]
        dangers += [thisDanger]
        action += [j]
        costsdangers += [thiscost+thisDanger]
        if (prntB == 1):
            if j == 1:
                print str('WAIT   ').rjust(8),
            else:
                print str('INSPECT').rjust(8),
            print str(round(thiscost, 1)).rjust(26),
            print str(round(thisDanger, 1)).rjust(8),
            print str(round((thiscost+thisDanger), 1)).rjust(10)
    optimalIndex = costsdangers.index(min(costsdangers))
```

```
        newcostsdangers += [costsdangers[optimalIndex]]
        newcosts += [costs[optimalIndex]]
        newdangers += [dangers[optimalIndex]]
        if (prntB == 1):
            print '                                              '
            print 'OPTIMAL ACTION:',
            if optimalIndex == 0:
                print 'WAIT'
            else:
                print 'INSPECT'
            print
    return [newcosts, newdangers, newcostsdangers]
# ========================================================================
# ======== DIFFERENCE FUNCTION DEFINITION ================================
# ========================================================================
def TakeDifference(oldCost, oldDang, newCost, newDang):
    z = 0
    for i in range(S):
        z += (abs(oldCost[i]-newCost[i]))**2
    z = math.sqrt(z)
    y = 0
    for i in range(S):
        y += (abs(oldDang[i]-newDang[i]))**2
    y = math.sqrt(y)
    return z+y
# ========================================================================
# =========== Compute Danger of Each State ===============================
# ========================================================================
# The danger of the main states are computed once and saved.
INDEXDANGER = []
for v in range(S):
    theState = IndexToState(v)
    theDanger = 0
    for w in range(theState[1]+1):
        if ((theState[0]+w) <= B):
            theDanger += DANGER[theState[0]+w]*PROB[theState[1]-1][w]
        else:
            theDanger += MAXDANGER*PROB[theState[1]-1][w]
    INDEXDANGER += [theDanger]
if ShowRawDangerMatrix:
    print INDEXDANGER
def ReturnDanger(state):
    danger = 0
    if ((state[0] <= (B)) and (state[1] <= B)):
        danger = INDEXDANGER[StateToIndex(state)]
    if ((state[0] <= (B)) and (state[1] == B+1)):
        for w in range(state[1]+1):
            if ((state[0]+w) <= B):
                danger += DANGER[state[0]+w]*PROB[state[1]-1][w]
            else:
                danger += MAXDANGER*PROB[state[1]-1][w]
    if ((state[0] > (B)) or (state[1] > B+1)):
        danger = MAXDANGER
    return danger
```

```
# ======================================================================
# ================ PROGRAM =============================================
# ======================================================================
prntB = 0
diff = BETA+1
it = 0
newCost = oldCost
newDang = oldDang
newTotal = oldTotal
cutnum = 1
print '************States=',B,'******************'
while ((diff > BETA) and (SolveProgram)):
    oldCost = newCost
    oldDang = newDang
    oldTotal = newTotal
    newCost, newDang, newTotal = RunIteration(prntB)
    it += 1
    diff = TakeDifference(oldCost, oldDang, newCost, newDang)
    if ShowIteration >= 1:
        if (int(it/ShowIteration) == cutnum):
            cutnum += 1
            print 'Iteration:', str(it).rjust(3),
            print'   GAP=', str(round(diff, 5)).rjust(9)
if (ShowSolution and SolveProgram):
    print
    print
    prntB = 1
    newCost, newDang, newTotal = RunIteration(prntB)
if (ShowSolution and SolveProgram) or (ShowIteration >= 1 and SolveProgram):
    print '-------------------------------------------'
    print 'Total Iterations:',
    print str(it).rjust(3),
    print'   Final GAP=', str(round(diff, 5)).rjust(8)
if ShowStateDanger:
    for test in range(B+2):
        for test2 in range(1,B*2):
            Thisisstate = []
            Thisisstate += [test]
            Thisisstate += [test2]
            Thiswasstate = ReturnDanger(Thisisstate)
            print 'State:',str(Thisisstate).rjust(8),
            print '     Danger:',str(round(Thiswasstate,3)).rjust(8)
elapsed = round((time.time() - start), 2)
print 'Elapsed time:', str(elapsed).rjust(7),
```

L Python Code: FLP Model Sub-Optimization

```
# Mixed Integer Layout Program.
# import numpy as np
import math
import sys
from gurobipy import *
# ============================================================
# ==================== PARAMETERS ============================
# ============================================================
BreakC = True
# Include symmetry breaking constraints?
N = 7
# The number of departments (integer).
M = 3
# The maximum number of floors (integer).
L = 513
W = 513
# The maximum Length/Width of the facility.
A = [797,4180,3001,4194,3669,1015,4185]
# Lower bound of the area for each department.
S = [24,63,42,52,33,30,36]
# Lower bound of the side length for each department.
H = 4
# Height of a floor.
CL = 6
# Cost place on making the facility one unit greater in length.
CW = 6
# Cost placed on making the facility greater in width.
CF = 500
# Cost placed on adding floors to the facility.
CE = 30
# Cost placed on adding an elevator to the facility.
# ============================================================
# ==================== BINARY VARIABLES ======================
# ============================================================
m = Model("GOAL LAYOUT Part 1")
# Model.
v = []
```

```
for i in range(N):
    newcolumn = []
    for k in range(M):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'v')
        newcolumn += [newvar]
    v += [newcolumn]
# 1, if department i is assigned to floor k; 0 otherwise.
z = []
for i in range(N - 1):
    newcolumn = []
    for j in range(i+1, N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'vz')
        newcolumn += [newvar]
    z += [newcolumn]
# 1, if departments i and j are assigned to the same floor; 0 otherwise.
tx = []
for i in range(N):
    newcolumn = []
    for j in range(N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'tx')
        newcolumn += [newvar]
    tx += [newcolumn]
# 1, if departments i and j do not overlap along the x-axis and i has
# x-coordinate closer to the origin.
ty = []
for i in range(N):
    newcolumn = []
    for j in range(N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'ty')
        newcolumn += [newvar]
    ty += [newcolumn]
# 1, if departments i and j do not overlap along the y-axis and i has
# y-coordinate closer to the origin.
r = m.addVar(0.0, 1.0, CE, GRB.BINARY, 'r')
# **** The number of elevators used in the solution. ******
# ============================================================
# ==================== CONTINUOUS VARIABLES ==================
# ============================================================
bl = m.addVar(0.0, L, CL, GRB.CONTINUOUS, 'bl')
bw = m.addVar(0.0, W, CW, GRB.CONTINUOUS, 'bw')
# Length and width of the facility along an arbitrary x- and y-axis.
l = []
for i in range(N):
    newvar = m.addVar(S[i], L, 0.0, GRB.CONTINUOUS, 'l')
    l += [newvar]
w = []
for i in range(N):
    newvar = m.addVar(S[i], W, 0.0, GRB.CONTINUOUS, 'w')
    w += [newvar]
# Length and width of department i along an arbitrary x- and y-axis.
x = []
for i in range(N):
    newvar = m.addVar(0.0, L, 0.0, GRB.CONTINUOUS, 'x')
    x += [newvar]
```

```
y = []
for i in range(N):
    newvar = m.addVar(0.0, W, 0.0, GRB.CONTINUOUS, 'y')
    y += [newvar]
# The x and y coordinates for the centroid of department i.
# ============================================================
# ==================== INTEGER VARIABLES =====================
# ============================================================
u = m.addVar(1, M, CF, GRB.INTEGER, 'u')
# The number of floors used in the solution.
# ============================================================
# ===================== OBJECTIVE FUNCTION ===================
# ============================================================
# OBJECTIVE FUNCTION: Minimize cost of weighted flows betweeen departments.
m.ModelSense = 1;
# Update model to integrate new variables.
m.update()
# ============================================================
# ===================== GOAL PART I CONSTRAINTS ==============
# ============================================================
LHS = LinExpr([99, -1],[r, u])
m.addConstr(LHS, GRB.GREATER_EQUAL, -1, "Goal Constraint 1")
# ============================================================
# ===================== FLOOR CONSTRAINTS ====================
# ============================================================
for i in range(N):
    sumv = LinExpr()
    for k in range(M):
        sumv.addTerms(1,v[i][k])
    m.addConstr(sumv,GRB.EQUAL, 1, "FC1")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,-1,-1],[z[i][j-i-1],v[i][k], v[j][k]])
            m.addConstr(LHS, GRB.GREATER_EQUAL, -1, "FC2")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,1,-1],[z[i][j-i-1],v[i][k], v[j][k]])
            m.addConstr(LHS, GRB.LESS_EQUAL, 1, "FC3")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,-1,1],[z[i][j-i-1],v[i][k], v[j][k]])
            m.addConstr(LHS, GRB.LESS_EQUAL, 1, "FC4")
for i in range(N):
    for k in range(M):
        LHS = LinExpr([1,-(k+1)],[u,v[i][k]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FC5")
m.addConstr(u, GRB.LESS_EQUAL, M, "FC6")
m.addConstr(u, GRB.GREATER_EQUAL, 1, "FC7")
# ============================================================
# ============== DEPARTMENT DIMENSION CONSTRAINTS ============
# ============================================================
```

```
for i in range(N):
    m.addConstr(l[i], GRB.GREATER_EQUAL, S[i], "DDC1")
    m.addConstr(w[i], GRB.GREATER_EQUAL, S[i], "DDC2")
    m.addConstr(l[i], GRB.LESS_EQUAL, bl, "DDC3")
    m.addConstr(w[i], GRB.LESS_EQUAL, bw, "DDC4")
for i in range(N):
    CONSTANT = ((A[i]/S[i])+S[i]-2*math.sqrt(A[i]))/((A[i]/S[i])-S[i]+0.001)
    LHS = LinExpr([1,1,-CONSTANT, CONSTANT],[l[i],w[i],l[i],w[i]])
    RHS = 2*math.sqrt(A[i])
    m.addConstr(LHS, GRB.GREATER_EQUAL, RHS, "DDC5")
    LHS = LinExpr([1,1,CONSTANT, -CONSTANT],[l[i],w[i],l[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, RHS, "DDC6")
# ============================================================
# ============== NON-OVERLAPPING CONSTRAINTS =================
# ============================================================
for i in range(N-1):
    for j in range(i+1, N):
        LHS = LinExpr([1,1,1,1,-1],[tx[i][j],tx[j][i],\
                        ty[i][j],ty[j][i],z[i][j-i-1]])
        m.addConstr(LHS, GRB.EQUAL, 0, "NOC1")
for i in range(N):
    for j in range(N):
        LHS = LinExpr([1,-1,-0.5,-0.5,-L],[x[j],x[i],l[i],l[j],tx[i][j]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -L, "NOC2")
        LHS = LinExpr([1,-1,-0.5,-0.5,-W],[y[j],y[i],w[i],w[j],ty[i][j]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -W, "NOC4")
# ============================================================
# ============== FACILITY BOUNDING CONSTRAINTS ===============
# ============================================================
for i in range(N):
    LHS = LinExpr([1, -0.5],[x[i],l[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC1")
    LHS = LinExpr([1, -0.5],[y[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC2")
    LHS = LinExpr([1,-1,-0.5],[bl,x[i],l[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC3")
    LHS = LinExpr([1,-1,-0.5],[bw,y[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC4")
# ============================================================
# ============ SYMMETRY BREAKING CONSTRAINTS ==================
# ============================================================
if BreakC:
    LHS = LinExpr([1, -0.5],[x[1],bl])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "SBC1")
    LHS = LinExpr([1, -0.5],[y[1],bw])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "SBC2")
    if M>1:
        m.addConstr(v[1][M-1], GRB.EQUAL, 0, "SBC3")
# ============================================================
# ============== OTHER REDUNDANT CONSTRAINTS =================
# ============================================================
for i in range(N):
    m.addConstr(tx[i][i], GRB.EQUAL, 0, "ORC1")
    m.addConstr(ty[i][i], GRB.EQUAL, 0, "ORC2")
```

```
# ============================================================
# ============== SOLVE AND PRINT SOLUTION ====================
# ============================================================
def printSolution():
    if m.Status == GRB.OPTIMAL:
        floor = []
        for i in range(N):
            floorsum = 0
            for k in range(M):
                floorsum += int(round((k+1)*v[i][k].X, 1))
            floor += [floorsum]
        Area = []
        Error = []
        for i in range(N):
            Area += [l[i].X*w[i].X]
            Error += [(l[i].X*w[i].X-A[i])/A[i]]
        print '********************************',
        print '************************************'
        print'  ***  LayoutC7 PART I  ***'
        print '\nCost:           ',
        print str(round(m.ObjVal, 1)).rjust(13)
        print   '#  Floors:       ',
        print str(u.X).rjust(13)
        print   '#  Elevators:    ',
        print str(r.X).rjust(13)
        print   'Facility Length:',
        print str(round(bl.X, 1)).rjust(13)
        print   'Facility Width: ',
        print str(round(bw.X, 1)).rjust(13)
        print '\nDEPT.  FLOOR       CENTROID        LENGTH    WIDTH '
        for i in range(N):
            print ' ',i+1,
            print '   ',floor[i],
            print'  (', str(round(x[i].X, 1)).rjust(5),
            print ',', str(round(y[i].X, 1)).rjust(5), ')',
            print str(round(l[i].X, 1)).rjust(8),
            print str(round(w[i].X, 1)).rjust(9)
        print '\nDEPT.   AREA         MIN_AREA    ERROR'
        for i in range(N):
            print ' ',i+1,'  ', str(round(Area[i], 1)).rjust(7),
            print ' ',str(A[i]).rjust(8),'   ',
            print str(round(Error[i]*100, 1)).rjust(8), '%'
        print
        print 'Note: Elevator location is not calculated or shown in graph'
# ================Graphics==================
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches
        floors = int(u.X+0.1)
        plt.axis([0,round(bl.X+.01,3),0,round(bw.X*floors+.01,3)])
        for f in range(floors):
            floorlines = plt.axhline(y = bw.X*(f+1), color = 'blue', linewidth = 2)
        FacilityWidth = plt.axvline(x = bl.X, color = 'blue', linewidth = 2)
        for f in range(floors):
            for i in range(N):
```

```
                if floor[i] == (f+1):
                    xy = (x[i].X-(0.5*l[i].X)),\
                         ((y[i].X-(0.5*w[i].X))+bw.X*(f))
                    width = l[i].X
                    height = w[i].X
                    p = mpatches.Rectangle(xy, width, height,\
                                           facecolor = "orange", edgecolor = "red")
                    plt.gca().add_patch(p)
        plt.show()
# ===============End Graphics===============
    else:
        print 'No solution'
# Solve:
m.optimize()
printSolution()
```

M Python Code: FLP Model Second Optimization

```
# Mixed Integer Layout Program -- GOAL PROGRAM PART II
# import numpy as np
import math
import sys
from gurobipy import *
# ============================================================
# ==================== PARAMETERS ============================
# ============================================================
GFACT = 1.025
GOAL = 2259
# Part two parameters.
BreakC = True
# Include symmetry breaking constraints?
N = 7
# The number of departments (integer).
M = 2
# The maximum number of floors (integer).
EMAX = 1
EMIN = 0
# The max/min number of elevators.
L = 513
W = 513
# The maximum Length/Width of the facility.
DepartmentA = [49,62,17,6,34,0]
DepartmentB = [27,90,48,88,82]
DepartmentC = [7,4,27,92]
DepartmentD = [65,78,37]
DepartmentE = [15,83]
DepartmentF = [45]
F = [DepartmentA, DepartmentB, DepartmentC,\
     DepartmentD, DepartmentE, DepartmentF]
# Material handeling flow matrix.
DepartmentA = [11,12,16,16,12,17]
DepartmentB = [13,19,11,13,17]
DepartmentC = [14,10,13,10]
DepartmentD = [20,20,11]
```

```
DepartmentE = [17,12]
DepartmentF = [11]
CH = [DepartmentA, DepartmentB, DepartmentC, DepartmentD,\
      DepartmentE, DepartmentF]
# Horizontal transportation cost per unit distance matrix.
DepartmentA = [10,10,11,18,19,10]
DepartmentB = [11,10,14,15,12]
DepartmentC = [19,16,12,18]
DepartmentD = [18,18,16]
DepartmentE = [19,16]
DepartmentF = [13]
CV = [DepartmentA, DepartmentB, DepartmentC, DepartmentD,\
      DepartmentE, DepartmentF]
# Vertical transportation cost per unit distance matrix.
A = [797,4180,3001,4194,3669,1015,4185]
# Lower bound of the area for each department.
S = [24,63,42,52,33,30,36]
# Lower bound of the side length for each department.
H = 4
# Height of a floor.
CL = 6
# Cost place on making the facility one unit greater in length.
CW = 6
# Cost placed on making the facility greater in width.
CF = 500
# Cost placed on adding floors to the facility.
CE = 30
# Cost placed on adding an elevator to the facility.
# ============================================================
# ===================== BINARY VARIABLES =====================
# ============================================================
m = Model("LAYOUT")
# Model.
ve = []
for i in range(N - 1):
    newcolumn = []
    for j in range(i+1, N):
        newdimension = []
        for e in range(EMAX):
            newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 've')
            newdimension += [newvar]
        newcolumn += [newdimension]
    ve += [newcolumn]
# 1 if traffic between departments (i, j) travels through elevator e, 0
# otherwise.
v = []
for i in range(N):
    newcolumn = []
    for k in range(M):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'v')
        newcolumn += [newvar]
    v += [newcolumn]
# 1, if department i is assigned to floor k; 0 otherwise.
z = []
```

```
for i in range(N - 1):
    newcolumn = []
    for j in range(i+1, N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'vz')
        newcolumn += [newvar]
    z += [newcolumn]
# 1, if departments i and j are assigned to the same floor; 0 otherwise.
tx = []
for i in range(N):
    newcolumn = []
    for j in range(N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'tx')
        newcolumn += [newvar]
    tx += [newcolumn]
# 1, if departments i and j do not overlap along the x-axis and i has
# x-coordinate closer to the origin.
ty = []
for i in range(N):
    newcolumn = []
    for j in range(N):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'ty')
        newcolumn += [newvar]
    ty += [newcolumn]
# 1, if departments i and j do not overlap along the y-axis and i has
# y-coordinate closer to the origin.
qxl = []
for i in range(N):
    newcolumn = []
    for k in range(EMAX):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'qxl')
        newcolumn += [newvar]
    qxl += [newcolumn]
# 1, if the department i does not overlap elevator e along the x-axis and is
# either to the left (XL) or to the right (XR) of elevator e.
qxr = []
for i in range(N):
    newcolumn = []
    for e in range(EMAX):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'qxr')
        newcolumn += [newvar]
    qxr += [newcolumn]
# 1, if the department i does not overlap elevator e along the x-axis and is
# either to the left (XL) or to the right (XR) of elevator e.
qyb = []
for i in range(N):
    newcolumn = []
    for e in range(EMAX):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'qyb')
        newcolumn += [newvar]
    qyb += [newcolumn]
# 1, if the department i does not overlap elevator e along the y-axis and is
# either below (YB) or above (YA) elevator e.
qya = []
for i in range(N):
```

```
    newcolumn = []
    for e in range(EMAX):
        newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'qya')
        newcolumn += [newvar]
    qya += [newcolumn]
# 1, if the department i does not overlap elevator e along the y-axis and is
# either below (YB) or above (YA) elevator e.
p = []
for e in range(EMAX):
    newvar = m.addVar(0.0, 1.0, 0.0, GRB.BINARY, 'p')
    p += [newvar]
# 1 if elevator e is used in the solution, 0 otherwise.
# ============================================================
# ==================== CONTINUOUS VARIABLES ==================
# ============================================================
dh = []
for i in range(N - 1):
    newcolumn = []
    for j in range(i+1, N):
        newvar = m.addVar(0.0, 2*(L+H), F[i][j-i-1]*CH[i][j-i-1],\
                          GRB.CONTINUOUS, 'dh')
        newcolumn += [newvar]
    dh += [newcolumn]
# Horizontal rectilinear distance between the centroids of departments i and j.
dv = []
for i in range(N - 1):
    newcolumn = []
    for j in range(i+1, N):
        newvar = m.addVar(0.0, H*M, F[i][j-i-1]*CV[i][j-i-1],\
                          GRB.CONTINUOUS, 'dv')
        newcolumn += [newvar]
    dv += [newcolumn]
# Vertical distance between the centroids of departments i and j (z-direction).
de = []
for i in range(N):
    newcolumn = []
    for e in range(EMAX):
        newvar = m.addVar(0.0, (L+H), 0.0, GRB.CONTINUOUS, 'de')
        newcolumn += [newvar]
    de += [newcolumn]
# Horizontal rectilinear distance between the centroid of departments i and
# elevator e.
bl = m.addVar(0.0, L, 0, GRB.CONTINUOUS, 'bl')
bw = m.addVar(0.0, W, 0, GRB.CONTINUOUS, 'bw')
# Length and width of the facility along an arbitrary x- and y-axis.
l = []
for i in range(N):
    newvar = m.addVar(S[i], L, 0.0, GRB.CONTINUOUS, 'l')
    l += [newvar]
w = []
for i in range(N):
    newvar = m.addVar(S[i], W, 0.0, GRB.CONTINUOUS, 'w')
    w += [newvar]
# Length and width of department i along an arbitrary x- and y-axis.
```

```
x = []
for i in range(N):
    newvar = m.addVar(0.0, L, 0.0, GRB.CONTINUOUS, 'x')
    x += [newvar]
y = []
for i in range(N):
    newvar = m.addVar(0.0, W, 0.0, GRB.CONTINUOUS, 'y')
    y += [newvar]
# The x and y coordinates for the centroid of department i.
ex = []
for i in range(EMAX):
    newvar = m.addVar(0.0, L, 0.0, GRB.CONTINUOUS, 'ex')
    ex += [newvar]
ey = []
for i in range(EMAX):
    newvar = m.addVar(0.0, W, 0.0, GRB.CONTINUOUS, 'ey')
    ey += [newvar]
# The x and y coordinates of elevator e.
# ============================================================
# ===================== INTEGER VARIABLES ====================
# ============================================================
u = m.addVar(1, M, 0, GRB.INTEGER, 'u')
# The number of floors used in the solution.
r = m.addVar(EMIN, EMAX, 0, GRB.INTEGER, 'r')
# The number of elevators used in the solution.
# ============================================================
# ===================== OBJECTIVE FUNCTION ===================
# ============================================================
# OBJECTIVE FUNCTION: Minimize cost of weighted flows betweeen departments.
m.ModelSense = 1;
# Update model to integrate new variables.
m.update()
# ============================================================
# ================ GOAL PROGRAM PART II CONSTRAINTS ==========
# ============================================================
LHS = LinExpr([CL, CW, CF, CE],[bl, bw, u, r])
m.addConstr(LHS, GRB.LESS_EQUAL, GFACT*GOAL, "GOAL CONSTRAINT 1")
# ============================================================
# ===================== FLOOR CONSTRAINTS ====================
# ============================================================
for i in range(N):
    sumv = LinExpr()
    for k in range(M):
        sumv.addTerms(1,v[i][k])
    m.addConstr(sumv,GRB.EQUAL, 1, "FC1")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,-1,-1],[z[i][j-i-1],v[i][k], v[j][k]])
            m.addConstr(LHS, GRB.GREATER_EQUAL, -1, "FC2")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,1,-1],[z[i][j-i-1],v[i][k], v[j][k]])
```

```
            m.addConstr(LHS, GRB.LESS_EQUAL, 1, "FC3")
for i in range(N-1):
    for j in range(i+1, N):
        for k in range(M):
            LHS = LinExpr([1,-1,1],[z[i][j-i-1],v[i][k], v[j][k]])
            m.addConstr(LHS, GRB.LESS_EQUAL, 1, "FC4")
for i in range(N):
    for k in range(M):
        LHS = LinExpr([1,-(k+1)],[u,v[i][k]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FC5")
m.addConstr(u, GRB.LESS_EQUAL, M, "FC6")
m.addConstr(u, GRB.GREATER_EQUAL, 1, "FC7")
# ============================================================
# ============== DEPARTMENT DIMENSION CONSTRAINTS ============
# ============================================================
for i in range(N):
    m.addConstr(l[i], GRB.GREATER_EQUAL, S[i], "DDC1")
    m.addConstr(w[i], GRB.GREATER_EQUAL, S[i], "DDC2")
    m.addConstr(l[i], GRB.LESS_EQUAL, bl, "DDC3")
    m.addConstr(w[i], GRB.LESS_EQUAL, bw, "DDC4")
for i in range(N):
    CONSTANT = ((A[i]/S[i])+S[i]-2*math.sqrt(A[i]))/((A[i]/S[i])-S[i]+0.001)
    LHS = LinExpr([1,1,-CONSTANT, CONSTANT],[l[i],w[i],l[i],w[i]])
    RHS = 2*math.sqrt(A[i])
    m.addConstr(LHS, GRB.GREATER_EQUAL, RHS, "DDC5")
    LHS = LinExpr([1,1,CONSTANT, -CONSTANT],[l[i],w[i],l[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, RHS, "DDC6")
# ============================================================
# ============== NON-OVERLAPPING CONSTRAINTS =================
# ============================================================
for i in range(N-1):
    for j in range(i+1, N):
        LHS = LinExpr([1,1,1,1,-1],[tx[i][j],tx[j][i],ty[i][j],\
                                    ty[j][i],z[i][j-i-1]])
        m.addConstr(LHS, GRB.EQUAL, 0, "NOC1")
for i in range(N):
    for j in range(N):
        LHS = LinExpr([1,-1,-0.5,-0.5,-L],[x[j],x[i],l[i],l[j],tx[i][j]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -L, "NOC2")
        LHS = LinExpr([1,-1,-0.5,-0.5,-W],[y[j],y[i],w[i],w[j],ty[i][j]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -W, "NOC4")
for i in range(N):
    for e in range(EMAX):
        LHS = LinExpr([1,1,1,1],[qxl[i][e],qxr[i][e],qyb[i][e],qya[i][e]])
        m.addConstr(LHS, GRB.EQUAL, 1, "NOC6")
for i in range(N):
    for e in range(EMAX):
        LHS = LinExpr([1,-1,-0.5,-L],[ex[e],x[i],l[i],qxl[i][e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -L, "NOC7")
        LHS = LinExpr([-1,1,-0.5,-L],[ex[e],x[i],l[i],qxr[i][e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -L, "NOC8")
        LHS = LinExpr([1,-1,-0.5,-W],[ey[e],y[i],w[i],qyb[i][e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -W, "NOC9")
        LHS = LinExpr([-1,1,-0.5,-W],[ey[e],y[i],w[i],qya[i][e]])
```

```
        m.addConstr(LHS, GRB.GREATER_EQUAL, -W, "NOC10")
# ============================================================
# ============== DISTANCE CONSTRAINTS ========================
# ============================================================
for i in range(N-1):
    for j in range(i+1, N):
        sumk = LinExpr()
        for k in range(M):
            sumk.addTerms(-H*k,v[i][k])
            sumk.addTerms(H*k,v[j][k])
        m.addConstr(dv[i][j-i-1], GRB.GREATER_EQUAL, sumk, "DC1")
for i in range(N-1):
    for j in range(i+1, N):
        sumk = LinExpr()
        for k in range(M):
            sumk.addTerms(H*k,v[i][k])
            sumk.addTerms(-H*k,v[j][k])
        m.addConstr(dv[i][j-i-1], GRB.GREATER_EQUAL, sumk, "DC2")
for i in range(N-1):
    for j in range(i+1, N):
        LHS = LinExpr([1,-1,1,-1,1,-(L+W)],[dh[i][j-i-1],x[i],x[j],\
                                        y[i],y[j],z[i][j-i-1]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -(L+W), "DC3")
        LHS = LinExpr([1,1,-1,1,-1,-(L+W)],[dh[i][j-i-1],x[i],x[j],\
                                        y[i],y[j],z[i][j-i-1]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -(L+W), "DC4")
        LHS = LinExpr([1,-1,1,1,-1,-(L+W)],[dh[i][j-i-1],x[i],x[j],\
                                        y[i],y[j],z[i][j-i-1]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -(L+W), "DC5")
        LHS = LinExpr([1,1,-1,-1,1,-(L+W)],[dh[i][j-i-1],x[i],x[j],\
                                        y[i],y[j],z[i][j-i-1]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, -(L+W), "DC6")
for i in range(N):
    for e in range(EMAX):
        LHS = LinExpr([1,-1,1,-1,1],[de[i][e],x[i],ex[e],y[i],ey[e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0.0, "DC7")
        LHS = LinExpr([1,1,-1,1,-1],[de[i][e],x[i],ex[e],y[i],ey[e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0.0, "DC8")
        LHS = LinExpr([1,-1,1,1,-1],[de[i][e],x[i],ex[e],y[i],ey[e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0.0, "DC9")
        LHS = LinExpr([1,1,-1,-1,1],[de[i][e],x[i],ex[e],y[i],ey[e]])
        m.addConstr(LHS, GRB.GREATER_EQUAL, 0.0, "DC10")
for i in range(N-1):
    for j in range(i+1, N):
        for e in range(EMAX):
            LHS = LinExpr([1,-1,-1,2*(L+W),-2*(L+W)],[dh[i][j-i-1],\
                        de[i][e],de[j][e],z[i][j-i-1],ve[i][j-i-1][e]])
            m.addConstr(LHS, GRB.GREATER_EQUAL, -2*(L+W), "DC11")
# ============================================================
# ============== FACILITY BOUNDING CONSTRAINTS ===============
# ============================================================
for i in range(N):
    LHS = LinExpr([1, -0.5],[x[i],l[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC1")
```

```
    LHS = LinExpr([1, -0.5],[y[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC2")
    LHS = LinExpr([1,-1,-0.5],[bl,x[i],l[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC3")
    LHS = LinExpr([1,-1,-0.5],[bw,y[i],w[i]])
    m.addConstr(LHS, GRB.GREATER_EQUAL, 0, "FBC4")
for e in range(EMAX):
    LHS = LinExpr([1,-1],[ex[e],bl])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "FBC5")
    LHS = LinExpr([1,-1],[ey[e],bw])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "FBC6")
# ============================================================
# ============== ELEVATOR CONSTRAINTS ========================
# ============================================================
sump = LinExpr()
for e in range(EMAX):
    sump.addTerms(1,p[e])
m.addConstr(r, GRB.EQUAL, sump, "EC1")
for i in range(N-1):
    for j in range(i+1, N):
        sumve = LinExpr()
        for e in range(EMAX):
            sumve.addTerms(1,ve[i][j-i-1][e])
        sumve.addTerms(1, z[i][j-i-1])
        m.addConstr(sumve, GRB.EQUAL, 1, "EC2")
for i in range(N-1):
    for j in range(i+1, N):
        for e in range(EMAX):
            m.addConstr(p[e], GRB.GREATER_EQUAL, ve[i][j-i-1][e], "EC3")
# ============================================================
# ============ SYMMETRY BREAKING CONSTRAINTS ==================
# ============================================================
if BreakC:
    LHS = LinExpr([1, -0.5],[x[1],bl])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "SBC1")
    LHS = LinExpr([1, -0.5],[y[1],bw])
    m.addConstr(LHS, GRB.LESS_EQUAL, 0, "SBC2")
    if M>1:
        m.addConstr(v[1][M-1], GRB.EQUAL, 0, "SBC3")
# ============================================================
# ============== OTHER REDUNDANT CONSTRAINTS ==================
# ============================================================
for i in range(N):
    m.addConstr(tx[i][i], GRB.EQUAL, 0, "ORC1")
    m.addConstr(ty[i][i], GRB.EQUAL, 0, "ORC2")
# ============================================================
# ============== SOLVE AND PRINT SOLUTION ====================
# ============================================================
def printSolution():
    if m.Status == GRB.OPTIMAL:
        floor = []
        for i in range(N):
            floorsum = 0
            for k in range(M):
```

```
        floorsum += int(round((k+1)*v[i][k].X, 1))
    floor += [floorsum]
elevator = []
for i in range(N-1):
    elevatorcolumn = []
    for j in range(i+1, N):
        elevatorsum = 0
        for e in range(EMAX):
            elevatorsum += int((e+1)*ve[i][j-i-1][e].X+0.001)
        elevatorcolumn += [elevatorsum]
    elevator += [elevatorcolumn]
Area = []
Error = []
for i in range(N):
    Area += [l[i].X*w[i].X]
    Error += [(l[i].X*w[i].X-A[i])/A[i]]
FacCost = CL*bl.X+CW*bw.X+CF*u.X+CE*r.X
print '*****************************************',
print '*************************'
print '   ***   LayoutC7 PART II  ***'
print
print 'Pt. I Optimal Facility Cost:',
print str(round(GOAL, 1)).rjust(13)
print 'Upper Limit Facility Cost:  ',
print str(round(GOAL*GFACT, 1)).rjust(13)
print 'Slack Used:                 ',
print str(round(((100*FacCost)/(GOAL*GFACT)), 5)).rjust(13), '%'
print 'Actual Facility Cost:       ',
print str(round(FacCost, 1)).rjust(13)
print 'Flow Cost:                  ',
print str(round(m.ObjVal, 1)).rjust(13)
print 'Flow + Facility Cost:       ',
print str(round((FacCost+m.ObjVal), 1)).rjust(13)
print
print   '# Floors:       ', str(u.X).rjust(13)
print   '# Elevators:    ', str(r.X).rjust(13)
print   'Facility Length:', str(round(bl.X, 1)).rjust(13)
print   'Facility Width: ', str(round(bw.X, 1)).rjust(13)
print '\nDEPT.  FLOOR       CENTROID        LENGTH    WIDTH '
for i in range(N):
    print ' ',i+1,
    print '   ',floor[i],
    print'  (', str(round(x[i].X, 1)).rjust(5),
    print ',', str(round(y[i].X, 1)).rjust(5), ')',
    print str(round(l[i].X, 1)).rjust(8),
    print str(round(w[i].X, 1)).rjust(9)
print '\nDEPT.   AREA         MIN_AREA    ERROR'
for i in range(N):
    print ' ',i+1,'  ', str(round(Area[i], 1)).rjust(7),
    print ' ',str(A[i]).rjust(8),'   ',
    print str(round(Error[i]*100, 1)).rjust(8), '%'
print '\nElevator   Centroid'
for e in range(EMAX):
    print ' ',e+1,
```

```
            print’        (’, str(round(ex[e].X, 1)).rjust(5),
            print ’,’, str(round(ey[e].X, 1)).rjust(5), ’)’
        print
        print ’Elevator utilized to accommodate flow between departments:’
        print ’ ’,
        for i in range(N-1):
            print ’’, i+2,
        print
        for i in range(N-1):
            print i+1,
            for j in range(i):
                print’  ’,
            print elevator[i]
# ================Graphics=================
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches
        floors = int(u.X)
        for f in range(floors):
            for e in range(EMAX):
                plt.plot([ex[e].X], [(ey[e].X+bw.X*(f))], ’o’, color = ’blue’)
        plt.axis([0,bl.X,0,bw.X*floors])
        for f in range(floors):
            floorlines = \
                  plt.axhline(y = bw.X*(f+1), color = ’blue’, linewidth = 2)
        FacilityLength = plt.axvline(x = bl.X, color = ’blue’, linewidth = 2)
        for f in range(floors):
            for i in range(N):
                if floor[i] == (f+1):
                    xy = (x[i].X-(0.5*l[i].X)), ((y[i].X-(0.5*w[i].X))\
                                                 +bw.X*(f))
                    width = l[i].X
                    height = w[i].X
                    p = mpatches.Rectangle(xy, width, height,\
                                    facecolor = "orange", edgecolor = "red")
                    plt.gca().add_patch(p)
        plt.show()
# ===============End Graphics===============
    else:
        print ’No solution’
# Solve.
m.optimize()
printSolution()
```

Bibliography

[1] Peggy Vonsherie Allen. Traffic signal preventive maintenance: An ounce of prevention is worth a pound of cure. *ITE Journal*, 79(4):20–25, April 2009.

[2] Anonymous. Frequently asked questions about led traffic signals. *ITE Journal*, 70(2):22–23, January 2000.

[3] Anonymous. Leds signal a bright future for iowa city. *The American City & County*, 117(1):38, January 2002.

[4] Ronald G. Askin and Charles R. Standridge. *Modeling and Analysis of Manufacturing Systems*. Canada: John Wiley & Sons, Inc., 1993.

[5] Nathaniel S. Behura. A survey of maintenance practices of light-emitting diode traffic signals and some recommended guidelines. *ITE Journal*, 77(4):18–22, April 2007.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition, 1996.

[7] Rekha Bhowmik. An approach of the facility layout design optimization. *International Journal of Computer Science and Network Security*, 8(4):212–220, April 2008.

[8] Yavuz A. Bozer, Russell D. Meller, and Steven J. Erlebacher. An improvement-type layout algorithm for single and multiple-floor facilities. *Management Science*, 40(7):918–932, July 1994.

[9] Merry Bronson. Light emitting diode (led) traffic signal survey results. Technical Report CEC-400-2005-003, California Energy Commission, January 2005.

[10] John D. Bullough, Jeremy D. Snyder, Aaron M. Smith, and Terence R. Klein. Replacement processes for light emitting diode (led) traffic signals. Technical Report NCHRP Web-Only Document 146, National Cooperative Highway Research Program, Transportation Research Board, August 2009.

[11] Jonathan Cagan, Kenji Shimada, and Su Yin. A survey of computational approaches to three-dimensional layout problems. *Computer Aided Design*, 34:597–611, 2002.

[12] Robert F. Carr. Hospital. Website, June 2010. `http://www.wbdg.org/design/hospital.php`.

[13] Energy policy act of 2005. Technical Report Public Law 109–58, The United States Congress, August 2005.

[14] Marc Goetschalckx and Takashi Irohara. Efficient formulations for the multi-floor facility layout problem with elevators. Website, February 2007. http://www.optimization-online.org/DB_HTML/2007/02/1598.html.

[15] Peter M. Hahn and Jakob Krarup. A hospital facility layout problem finally solved. *Journal of Intelligent Manufacturing*, 12:487–496, 2001.

[16] Carol Haraden and Roger Resar. Patient flow in hospitals: Understanding and controlling it better. *Frontiers in Health Service Management*, 20:3–15, 2004.

[17] Kyung-Wook Jee, Daniel L. McShan, and Benedick A. Fraass. Lexicographic ordering: intuitive multicriteria optimization for imrt. *Physics in Medicine and Biology*, 52:1845–1861, 2007.

[18] Jane Jerrard. The end of general hospitals. *The Hospitalist*, pages 13–17, December 2005.

[19] Abdullah Konak, Sadan Kulturel-Konak, Bryan A. Norman, and Alice E. Smith. A new mixed integer programming formulation for facility layout design using flexible bays. *Operations Research Letters*, 34(6):660–672, 2006.

[20] Led traffic signal monitoring, maintenance, and replacement issues. Technical Report NCHRP Synthesis 387, National Cooperative Highway Research Program, Transportation Research Board, 2008.

[21] Kyu-Yeul Lee, Seong-Nam Han, and Myung-Il Roh. An improved genetic algorithm for facility layout problems having inner structure walls and passages. *Computers and Operations Research*, 30:117–138, 2003.

[22] John J. McCall. Maintenance policies for stochastically failing equipment: A survey. *Management Science*, 11(5):493–525, March 1965.

[23] Russell D. Meller and Kai-Yin Gau. The facility layout problem: recent and emerging trends and perspectives. *Journal of Manufacturing Systems*, 15(5):351–366, 1996.

[24] Kristine M. Miller. *Planning, Design, and Construction of Health Care Facilities*. Joint Commission Resources, 1st edition, 2006.

[25] Richard L. Miller and Earl S. Swensson. *Hospital and Healthcare Facility Design*. Norton, 2nd edition, 2002.

[26] George E. Monahan. A survey of partially observable markov decision processes: Theory, models, and algorithms. *ITE Journal*, 28(1):1–16, January 1982.

[27] Ivan Moreno and Ching-Cherng Sun. Modeling the radiation pattern of leds. *Optics Express*, 16(3):1808–1819, February 2008.

[28] National traffic signal report card: Technical report. Technical report, National Transportation Operations Coalition, 2007.

[29] Dimitrios I. Patsiatzis and Lazaros G. Papageorgiou. Optimal multi-floor process plant layout. *Computers and Chemical Engineering*, 26:575–583, 2002.

[30] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley, 1st edition, 2007.

[31] John Reiling. Safe design of healthcare facilities. *Quality and Safety in Healthcare*, 15:34–41, 2006.

[32] Request for proposal: Light emitting diode signal module replacement project. Technical report, City of Foster City, California, Public Works Engineering, 2009.

[33] S. P. Singh and R. R. K. Sharma. A review of different approaches to the facility layout problems. *The International Journal of Advanced Manufacturing Technologies*, 30(5):425–433, 2006.

[34] Patricia Tzortzopoulos, Ricardo Codinhoto, Mike Kagioglou, and Lauri Koskela. Design for operational efficiency—linking building and service design in healthcare environments. In *Proceedings: ELAGEC-II Encuentro Latino-Americano de Gestion y Economia de la Construccion, Santiago, Chile*, pages 24–25, January 2008.

[35] Stephen Ursery. Led lights save energy in city's traffic signals. *The American City & County*, 118(8):18–19, July 2003.

[36] Vehicle traffic control signal heads: Light emitting diode (led) circular signal supplement (performance spec). Technical report, Institute of Transportation Engineers, June 2005.

[37] Vehicle traffic control signal heads: Light emitting diode (led) circular signal supplement (purchase spec). Technical report, Institute of Transportation Engineers, June 2005.

[38] Vehicle traffic control signal heads: Light emitting diode (led) vehicle arrow traffic signal supplement (performance spec). Technical report, Institute of Transportation Engineers, April 2006.

[39] Vehicle traffic control signal heads: Light emitting diode (led) vehicle arrow traffic signal supplement (purchase spec). Technical report, Institute of Transportation Engineers, April 2006.

[40] Vehicle traffic control signal heads: Part 3: Light emitting diode (led) vehicle arrow traffic signal modules. Technical report, Institute of Transportation Engineers, September 2003.

[41] Leti Vos, Siebren Groothuis, and Godefridus G. van Merode. Evaluating hospital design from an operations management perspective. *Health Care Management Science*, 10:357–364, 2007.

[42] Zimin (Max) Yang, Dragan Djurdjanovic, and Jun Ni. Maintenance scheduling in manufacturing systems based on predicted machine degradation. *Journal of Intelligent Manufacturing*, 19:87–98, July 2007.