

**AN OPTIMIZED PARKING ALGORITHM TO
CONTROL PASSENGER ELEVATOR GROUPS USING
ARTIFICIAL NEURAL NETWORK**

HARRISON THUKU NGETHA

**MASTER OF SCIENCE
(Telecommunication Engineering)**

**JOMO KENYATTA UNIVERSITY OF
AGRICULTURE AND TECHNOLOGY**

2011

**An Optimized Parking Algorithm to Control Passenger
Elevator Groups Using Artificial Neural Network**

Harrison Thuku Ngetha

**A thesis submitted in partial fulfillment for the degree of
Master of Science in Telecommunication Engineering in the
Jomo Kenyatta University of Agriculture and Technology**

2011

DECLARATION

This thesis is my original work and has not been presented for a degree award in any other university.

Signature..... Date.....

Harrison Thuku Ngetha

This thesis has been submitted for examination with our approval as University Supervisors.

Signature..... Date.....

**Prof. J.N. Nderu,
JKUAT, Kenya.**

Signature..... Date.....

**Dr. L.M. Ngoo,
Multimedia University, Kenya.**

DEDICATION

I dedicate this work to Lucy Wanjiku Ngetha, my mother. Her immense inner strength, and unfailing belief in me, have always greatly inspired and motivated my life.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank God for his hand in having seen me through all the challenges that I have faced in my academic program as well as in formulating and completion of my research and thesis.

I would also like to thank my supervisors Dr. L.M. Ngoo and Prof. J.N. Nderu for their support and guidance throughout my research project. To Dr. E.N. Ndung'u, I say thank you for the many research papers, so generously provided.

To my family, friends and postgraduate colleagues, lots of appreciation for their encouragement and support throughout my studies

TABLE OF CONTENTS

DECLARATION.....	i
DEDICATION.....	ii
ACKNOWLEDGEMENTS.....	iii
TABLE OF CONTENTS.....	iv
LIST OF TABLES.....	viii
LIST OF FIGURES.....	ix
LIST OF APPENDICES.....	xii
LIST OF ABBREVIATIONS.....	xiii
NOMENCLATURE.....	xiv
ABSTRACT.....	xvii
CHAPTER 1.....	1
1. Introduction.....	1
1.1 Background	1
1.2 Statement of the problem.....	4
1.3 Justification of the thesis.....	5
1.4 Objectives	6
1.4.1 General Objective.....	6
1.4.2 Specific Objective.....	6
1.5 Significance of the study.....	7
1.6 Thesis organization.....	7

CHAPTER 2.....	10
2. Literature review	10
2.1 Elevator Control.....	10
2.1.1 Introduction.....	10
2.1.2 Traffic control, Drive control, Travel control.....	10
2.2 Elevator Group Control Systems (EGCSs).....	12
2.2.1 Introduction.....	12
2.2.2 Operation of Elevator group control systems.....	14
2.3 Measurement and forecasting of the passenger traffic.....	18
2.3.1 Measurement of passenger traffic.....	18
2.3.2 Forecasting method	21
2.3.3 Elimination of seasonal variations.....	23
2.3.4 Forecast traffic profile in an office building.....	25
2.4 Parking Algorithm of Passenger Elevators.....	25
2.4.1 Introduction.....	25
2.4.2 Parking Policies and their execution.....	27
2.4.2.1 Parking in Down peak traffic.....	29
2.4.2.2 Parking in Up peak traffic.....	31
2.5 Artificial Neural Networks (ANN).....	33
2.5.1 Introduction.....	33
2.5.2 Biological neural networks.....	35
2.5.3 The Processing unit.....	37
2.5.4 Combination function.....	38

2.5.5 Activation function.....	39
2.5.6 Network topologies.....	42
2.5.7 Network Learning.....	44
2.5.8 Objective function.....	46
2.5.9 Basic Architecture of a Feed-forward network.....	46
2.6 Backpropagation.....	48
2.6.1 Architecture of Backpropagation Net.....	49
2.6.2 Activation function.....	52
CHAPTER 3.....	53
3. Design Methodology.....	53
3.1 Elevator Simulator design.....	53
3.1.1 Introduction.....	53
3.1.2 The model.....	53
3.1.2.1 The description.....	53
3.1.2.2 Division into objects.....	54
3.1.3 The architecture of an elevator supervisory group control system.....	64
3.1.3.1 Data preparation.....	65
3.2 The pattern recognition system.....	67
3.3 The ANN training system trained using Backpropagation.....	68
3.3.1 Introduction.....	68
3.3.4 Simulated System by use of the GUI in ANN tools for MATLAB.....	70
CHAPTER 4.....	74
4. Simulation tests, results and analysis.....	74

4.1 Simulated 12hr passenger traffic pattern.....	74
4.2 Simulated results for the ECGs from the pattern recognition system.....	79
4.3 Simulated Results of the ANN training system trained using Backpropagation.....	84
4.4 Effect of the optimal parking.....	86
CHAPTER 5.....	90
5. Conclusion	90
5.1 Conclusion.....	90
REFERENCES.....	93
APPENDICES.....	96

LIST OF TABLES

Table 1.1	A list of buildings, the number of floors and dates of completion.....	1
Table 3.1	List of eligible objects discovered during analysis.....	55
Table 3.2	List of behaviors discovered during analysis.....	56
Table 3.3	The abstraction of the elevator class.....	57
Table 3.4	The abstraction of the passenger class.....	58
Table 3.5	The abstraction of the observer class.....	59
Table 3.6	The abstraction of the draw elevator simulation class.....	60
Table 3.7	The abstraction of the text menu and prompts class.....	62
Table 3.8	The abstraction of the report class.....	63
Table 3.9	The abstraction of the output class.....	64

LIST OF FIGURES

Figure 2-1:	A floor in a building where an EGCS is installed.....	12
Figure 2-2:	General structure of the EGCS.....	16
Figure 2-3:	Step counting method during an elevator stop.....	19
Figure 2-4:	Three main passenger traffic components.....	25
Figure 2-5a:	Components of a neuron.....	36
Figure 2-5b:	Synapse	36
Figure 2-6:	Processing unit.....	37
Figure 2-7a:	Identity function.....	39
Figure 2-7b:	Binary step function	39
Figure 2-8a:	Sigmoid function	40
Figure 2-8b:	Bipolar sigmoid function.....	41
Figure 2-9a:	Recurrent neural network.....	43
Figure 2-9b:	Feed-forward neural network.....	43
Figure 2-10a:	Supervised learning model.....	45
Figure 2-10b:	Unsupervised learning model.....	45
Figure 2-11:	Single layer Backpropagation Net	50
Figure 3-1:	Elevator life cycle	58
Figure 3-2:	Elevator simulator user interface prototype.....	61
Figure 3-3:	Text menu prototypes	62
Figure 3-4:	A sample report showing the activities of the major objects....	63

Figure 3-5:	The architecture of an elevator supervisory group control system.....	65
Figure 3-6:	A block diagram of the Pattern Recognition System.....	67
Figure 3-7:	The ANN training System.....	69
Figure 3-8:	Network Data Manager in GUI for ANN tools.....	70
Figure 3-9:	Backpropagation system Input Vectors (MatLab 6.5).....	71
Figure 3-10a:	The ELEVNET (creation).....	72
Figure 3-10b:	The ELEVNET (view).....	72
Figure 4-1a:	Passenger Traffic: Morning Up-Peak	74
Figure 4-1b:	Passenger Traffic: Off-Peak hour.....	75
Figure 4-1c:	Passenger Traffic: Lunch time hour.....	75
Figure 4-1d:	Passenger Traffic: Evening Down-Peak.....	76
Figure 4-1e:	Passenger Traffic: Elevator #1 Traffic.....	76
Figure 4-1f:	Passenger Traffic: Elevator #2 Traffic.....	77
Figure 4-1g:	Passenger Traffic: Elevator #3 Traffic.....	77
Figure 4-1h:	Passenger Traffic: Elevator #4 Traffic.....	78
Figure 4-2a:	Passenger elevator traffic patterns: Elevator #1.....	80
Figure 4-2b:	Passenger elevator traffic patterns: Elevator #2.....	80
Figure 4-2c:	Passenger elevator traffic patterns: Elevator #3.....	81
Figure 4-2d:	Passenger elevator traffic patterns: Elevator #4.....	81
Figure 4-3:	Forecast traffic component profile during a day in an office building.....	85
Figure 4-4a:	Average passenger waiting times as a function of the	

	passenger arrival rate.....	86
Figure 4-4b:	Average landing call times as a function of the passenger arrival rate.....	86
Figure 4-5a:	Incoming traffic.....	88
Figure 4-5b:	Outgoing traffic.....	88
Figure 4-5c:	Long Wait Rate (before optimal parking).....	89
Figure 4-5d:	Average Waiting Times (after optimal parking).....	89

LIST OF APPENDICES

Appendix A:	Program Codes.....	96
Appendix B:	Training Algorithm.....	129
Appendix C:	Flowchart for back propagation training of feedforward neural network.....	131

LIST OF ABBREVIATIONS

EGCSs	Elevator Group Control Systems
AWT	Average Waiting Time
LWP	Long Waiting Percent
RNC	RuN count
CC	Car Controller
ESD	Elevator Simulator Design
SCADA	Supervisory Control and Data Acquisition
KEMU	Kenya Methodist University
JKUAT	Jomo Kenyatta University of Agriculture and technology
PC	Personal Computer
ANN	Artificial Neural Networks
BP	Backpropagation
GUI	Graphical User Interface
ARRSES	Adaptive Response Rate Single Exponential Smoothing

NOMENCLATURE

a_t	Smoothed observation value at the time t
b_t	Estimate of a change due to a trend during a time
C	Set of floors of existing car calls
cc_i	Number of new car calls from floor i
$f(k)$	Value of measured load signal k
F_t	Forecast value at the time t
I_t	Traffic intensity value at the time t
K	Set of existing landing calls
L	Number of elevators in group
M	Car size in persons
N	Number of measurements
N	Number of floors in the building
p_{fi}	Number of entering passengers
p_{oi}	Number of exiting passengers
S_t	Moving average at the time t
S_{tt}	Moving average of the moving averages at the time t
t_a	Parking delay time
t_c	Remaining standing time of a stopped car
t_{cc}	Car call time
t_{ct}	Landing call time
t_{eta}	Estimated time of arrival to a landing call
t_{ride}	Passenger ride time inside a car
t_s	Average stop time $t_{service}$ Elevator service time

t_v	A single floor drive time at rated speed v
U	Space definitions for traffic factors
v	Rated elevator speed
x_i	Elevator position in units of floors
X_{av}	Forecast number of passengers during a typical day
X_{day}	Measured number of passengers during the day
Y_i	Observation value i
α, β	Smoothing constants
γ	Weight factor for the cost function
χ^2	Chi-squared test value
σ_i	Weight factor for the landing call i
τ	Average elevator round trip time
i, k	Observation, floor and call indexes
l	Elevator index
m, t	Time indexes
w	Synaptic weight of a neuron
w_b	Weight of the bias unit
b	Bias
y_{in}	Sum of the input weights to a neuron
$f(y_{in})$	Output response from an output neuron
$w_i(new)$	Adjusted weight
α	Learning rate
δ	Weight update factor

ABSTRACT

Elevator group control systems (EGCSs) are the control systems that systematically manage three or more elevators in order to efficiently transport passengers. The optimal parking algorithm of EGCSs is made using the classification of the passenger traffic and system manager's requirements, and the cars are assigned to specific floors by the generated control strategy. Group elevator parking and scheduling is a well-known problem in industrial control and operations research with significant practical implications. The traffic pattern of elevator passengers in buildings, with multiple elevators varies considerably during certain periods of the day. These traffic patterns are known as up-peak and down-peak. Due to the rise in building occupancy, the traffic pattern is also varying. The up-peak and down-peak are occurring at the same time.

Up-peak and down-peak pose extraordinary demands on the parking and scheduling processes for the elevator group, because the passenger arrival rate is high, and the traffic pattern is non-uniform. At the same time, these patterns can have a regular probabilistic structure. Currently two methods are used for parking the lift cars, the basic parking and the dynamic assignment. During the dynamic assignment operation, the group of elevators keeps statistics of the number of calls and the mean waiting time for every floor. The two parameters keep on changing with time, within the day, and also due to significant change in building occupancy. Because of the dynamic behavior, there is need for a system of real time control.

This thesis presents an efficient approach that consists of an experimental design and an Artificial Neural Network (ANN) model to generate models for the simulation of the parking control algorithm in Elevator group control systems (EGCSs).The system

has then been trained to recognize passenger traffic patterns for the four lift cars using a pattern recognition neural network for random patterned inputs. After the training the system has gone through further training using back propagation to enable pre-distribution of lift cars evenly among the ten floors in a building. The system has been designed and trained using C++ and then simulated by MATLAB. The results show that, the parking algorithm adapts to the prevailing traffic pattern. Control actions, such as returning cars automatically to busy traffic floors, or parking cars during light traffic, follow from the forecast traffic pattern as compared to the existing methods.

CHAPTER ONE

INTRODUCTION

1.1 Background

Since the year 1971, skyscrapers have been established within the major towns in Kenya. This is due to the fact that despite the increase in population, the land available is constant or diminishing. There has been an upward trend to build taller and taller buildings as indicated in Table 1.1 below.

Table 1.1 A list of buildings, the number of floors and dates of completion.

Rank	Name	Height	Year
01.	Times Tower	140 m	1997
02.	Kenyatta International Conference Center	105 m	1974
03.	NSSF Building	103 m	1993
04.	I & M Bank Tower	99 m	2001
05.	Government Office Conference Hall	98 m	1972
06.	Provincial Headquarters	84 m	1982
07.	Cooperative Bank House	83 m	1981
08.	National Bank House	82 m	1976
09.	Anniversary Towers	80 m	1992
10.	Reinsurance Plaza	77 m	1982
11.	Uchumi House	71 m	1972
12.	ICEA Building	69 m	1981
13.	International House	66 m	1971
14.	Hilton Hotel	61 m	1969
15.	View Park Towers	60 m	1994
16.	City Hall Annex	60 m	1980
17.	Treasury Building	48 m	1980
18.	Union Towers	48 m	1977

The building's occupancy is also on the rise. For instance, The View Park Towers- an ultra modern building within the Central Business District in Nairobi City which is located in Nairobi CBD, Utalii lane, houses the satellite campuses for both Moi University (19th Floor) and The Kenya Methodist University (KEMU) in the 15th floor. The KEMU Satellite campus was established in September 2005 and the student population has grown from the initial 30 students, to currently in excess of 1000.

The two companies which provide the lift services are Schindler Group Lifts and Otis Lifts. A passenger lift is designed to move people between a building's floors. Passenger elevators capacity is related to the available floor space. Generally passenger elevators are available in capacities from 1,000 to 6,000 lb (455 to 2,727 kg) in 500 lb (230 kg) increments. Generally passenger elevators in buildings with eight floors or less are either hydraulic or electric. The hydraulic lifts have speeds of up to 200 ft/min (1.0 m/s) while electric types attains speeds of up to 500 ft/min (2.5 m/s). In buildings of up to ten floors, electric and gearless elevators are likely to have speeds up to 500 ft/min (2.5 m/s), while those with over ten floors, elevator speeds range between 500 ft/min (2.5 m/s) up to 2000ft/min (10 m/s).

Modern elevators use more complex heuristic Algorithms to decide which request to service next. There are two special operating modes in common use, the Up Peak mode and the Down Peak Mode.

During Up Peak mode, elevator cars in a group are recalled to the lobby to provide expeditious service to passengers arriving at the building, most typically in the

morning as people arrive for work or at the conclusion of a lunch-time period. Elevators are dispatched one-by-one when they reach a pre-determined passenger load, or when they have had their doors opened for a certain period of time. The next elevator to be dispatched usually has its hall lantern or a "this car is leaving next" sign illuminated to encourage passengers to make maximum use of the available elevator system capacity.

The commencement of Up Peak/Down Peak may be triggered by a time clock, by the departure/arrival of a certain number of fully loaded cars from/at the lobby within a given time period, or by a switch manually operated by a building attendant.

During Down Peak mode, elevator cars in a group are sent away from the lobby towards the highest floor served, after which they commence running down the floors in response to hall calls placed by passengers wishing to leave the building. This allows the elevator system to provide maximum passenger handling capacity for people leaving the building.

Most of the lifts operates in a group and thus require a specialized control algorithm known as Elevator Algorithm ^[5]. The elevator algorithm is a simple algorithm, by which a single elevator can decide where to stop, is summarized as follows:

- Continue traveling in the same direction while there are remaining requests in that same direction.
- If there are no further requests in that direction, then stop and become idle, or change direction if there are requests in the opposite direction.

Basic parking is a parking algorithm which utilizes the manual control and is usually implemented by assigning the main floor (usually Ground Floor) as the busiest floor. This floor is given the highest priority and assigned more cars. Other cars are arbitrary assigned other floors depending on the understanding of the operation by the service company^[1].

The other type of control is referred as the Dynamic Assignment. During operation, the group of elevators keeps statistics of the number of calls (landing calls) and the mean waiting time for every floor. These two parameters, for every floor, help the lift system to identify the busiest floors. The group control then assigns more cars to the busiest floors.

However the two parameters keep on changing with time, within the day, and also due to significant change in building occupancy. Because of the dynamic behavior, there is need for a system of real time control.

1.2 Statement of the Problem

In the recent past a number of high rise buildings have come up in the major town in Kenyan, especially the capital city, Nairobi. The existing buildings have in the recent past experienced an up surge in the occupancy of the buildings. For instance the Kenyatta National Hospital, View Park Towers, Union Towers etc. This is because of establishment of University Satellite Campuses (KEMU, Moi University), existing colleges affiliating with public universities like JKUAT, Nairobi, Kenyatta). For the hospitals, the visiting hours are also experiencing heavy traffic

In most of the skyscrapers, the use of lifts is not optimized. This results in congestion and longer service time (waiting time and riding time)

The Up Peak times are morning and late afternoon (2-3 pm). The Down Peak times are early afternoon (12-1.30 pm) and evenings. In the recent past the Up Peak and Down Peak times are occurring at the same time. This results in congestion in the lobby as well as the top floors. A good example is the View Park Towers, which has two satellite campuses and the Kenyatta National Hospital, during visiting times.

1.3 Justification of the thesis

The Group Lift control in most of these buildings is not optimized and thus the waiting time is on the increase. Most of the buildings utilize either the Basic parking algorithm or the Dynamic assignment. Few buildings though use the fuzzy logic control algorithm e.g. the National Bank house along Harambee Avenue, Nairobi. The above mentioned group control algorithms do not offer highly optimized parking control, hence a need for a more optimized parking control algorithm.

The Down Peak and Up peak may occur at the same time. A good example is the Kenyatta National Hospital where the visiting time into the buildings is 12.30 pm to 2 pm. This results in an Up Peak. During this time the building also experiences a Down Peak. This causes traffic congestion. Since the emergency lift is also caught up in the traffic, it cannot handle emergency cases efficiently.

The lifts operating at Kenyatta National Hospital (KNH) are six including one for emergency. These lifts, installed by Otis Lifts, are among the oldest and thus use

basic parking algorithm. The emergency lift is also grouped together with the other lifts. The emergency lift is meant to handle emergency cases only. The buildings housing the satellite campuses and colleges also experience both Up Peak and Down Peak at the same time especially in the evenings due to evening classes. A good example is the Union Towers (three lifts) and View Park Towers (six lifts + a VIP lift).

Due to this phenomenon, these building experience longer waiting times and a lot of delays. The parking and group control mechanisms in place do not handle the control in an optimized manner.

1.4 Objectives

1.4.1 General Objective

The main objective of this study is to develop an intelligent computer program to enable an optimized parking algorithm to control passenger elevator group using Artificial Neural Network (ANN) system. This will be done by recognizing the traffic pattern for each passenger elevator as well as for the floors and then trying to optimally park the elevators so as to minimize the waiting and travel times.

1.4.2 Specific Objectives

- i. To develop an adaptive computer program using Artificial Neural Networks and to train the Net to recognize the pattern of hall call signals at any time and then park the lifts in accordance with the received signals.

- ii. To develop a computer code using C⁺⁺ for providing random input signals conformed to a given pattern for the simulated building and hall calls.
- iii. To combine both the Neural Net and the C⁺⁺ program in order to create an optimized parking algorithm for the group lifts.

1.5 Significant Contributions.

The thesis discusses the impact of the current parking algorithms on the traffic flow in and out of the building and the danger of not parking the group lifts optimally especially in hospitals, where optimal time usage is critical. The study intends to come up with an optimized control algorithm for optimally parking the group lifts. A software system which can successfully predetermine the call patterns from busy floors at different times will be developed. Through pre-determining these patterns, the system will also be able to pre-distribute the group lifts efficiently among different floors. This is done by training the Neural Net for pattern recognition using data generated from the developed program, then, after feeding the output from the Net to the C⁺⁺ program to pre distribute the lifts optimally on the busy floors at each specific time.

1.6 Thesis Organization

This thesis consists of five chapters.

Chapter 1, this chapter includes the objectives and motivations for conducting this research. The expected results and also the benefits of the system once implemented in the existing elevator control systems high rise buildings within the major cities in

the country, towards optimal parking of passenger elevators. This reduces the waiting and landing times and thus eases the congestion in the lobby.

Chapter 2 discusses the overview of the elevator control systems. The areas covered includes, the measurement and forecasting of passenger traffic. It also introduces the elevator simulator design system, the basic concepts of parking algorithm of the passenger elevators and artificial neural networks. The chapter briefly discusses the up peak and down peak passenger traffic, the basics of neural networks, the activation functions used and also different training algorithms. The chapter also discusses the pattern recognition neural network, and the simulation results of the system designed to recognize the specific passenger traffic patterns and the corresponding waiting times for the ten floors being served by four elevators over a 72hr period. The chapter discusses the backpropagation training algorithm which includes the feedforward network.

Chapter 3 gives the simulated results of the four passenger elevators and the ten floors but with the controlled neural network trained by backpropagation the design parameters of the elevator control systems is discussed. The chapter also discusses the disadvantages of long waiting times. Simulation results of a 12hr period call traffic pattern as well as the waiting and landing times are given for the four lifts and ten floors. These patterns are then used to develop a pattern recognition system and also for the design of the neural network controller.

Chapter 4 is about simulation tests at different times of the day and hour in different floors. The results are presented and analyzed.

Chapter 5 includes the conclusion of the work done and suggestions of future work.

CHAPTER TWO

LITERATURE REVIEW.

2.1 Elevator Control

2.1.1 Introduction

On the basis of functionality, elevator control can be divided into three major parts; the Traffic Control, Drive Control and Travel Control. The three parts are fully dependent on one another and have to work together to serve a passenger. In order to better understand the elevator control we will consider a lift serving a passenger from one floor to another, under normal operation. This is termed as serving a call. In normal operation the lift function is initiated by the passenger by pressing the floor call. When the lift arrives at the floor it serves the call by switching off the call light, indicating the direction and maybe ringing a bell. The former two actions confirm a call is served, and if they do not appear when the lift has stopped at a floor and opened the doors it may mean one of three things:

- i. The lift is reserved for special function,*
- ii. The lift is headed in the opposite direction,*
- iii. The lift is full.*

2.1.2 Traffic Control, Drive Control, Travel Control

In this thesis, we consider an elevator group. This is where two or more lifts are interconnected to serve floor calls as a team. In group operation there is at least one Master lift, the other(s) being slave(s). When a passenger presses a call, the master lift assigns this call to the nearest lift. The nearness of a lift to a particular call is

called the call cost of the lift to that call. Call cost is an important parameter; and for a particular lift it is very dynamic with time. It depends on the physical distance the lift is from the call, the direction of the lift, as well as the inside (car) calls and their distribution. The lift with the least value of the call cost is assigned the call. However, the parameter is reconstituted in very short intervals and this is reflected on the re-assignments to the same call until such a time when the differences in the values are definite. When the floor call is served the passenger enters the lift and presses a car (or inside) call. If there is no inside call the lift may serve any available floor call; hence the passenger should give the lift an inside call. If there is an inside call, the lift moves to that floor and then we say the lift has served the car call. After this the lift is ready to serve other floors and car inside calls. The drive controller receives commands from the traffic controller. E.g. move this number of displacement units in this direction. Depending on the displacement units the drive comes up with digital graphs (virtual) of acceleration, maximum speed (nominal speed) and deceleration. It also checks the speed and displacement at all times of travel. In case of any discrepancy an emergency stop is initiated. Serving of calls works when the lift is operating normally. For normal operation the lift has to check travel parameters such as power quantity and quality, safety loop, ambient temperature, actual position, load, direction of travel, speed, acceleration and deceleration. At any time during the travel these parameters have to conform, otherwise an emergency stop is initiated.

2.2 Elevator Group Control Systems (EGCSs)

2.2.1 Introduction

An elevator group requires control systems for the lifts to operate efficiently. The control systems are known as *The Elevator Control Systems* (EGCS's). These are control systems that manage multiple elevators in a building, as shown in Figure 2.1, in order to efficiently transport the passengers. The performance of EGCS's is measured by several criteria such as;

- i. *The average waiting time of passengers,*
- ii. *The percentage of passengers waiting more than 60 s,*
- iii. *Power consumption [1], [2], [3], [4].*

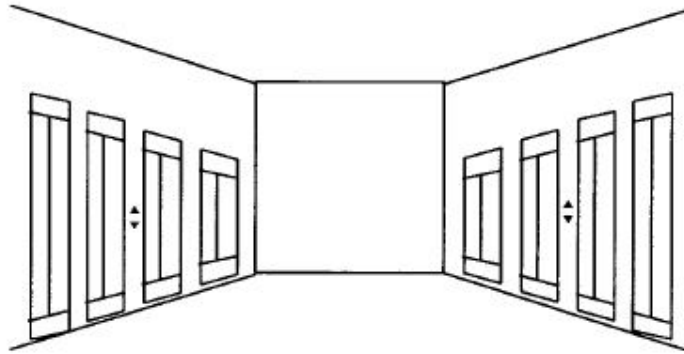


Figure 2.1 A floor in a building where an EGCS is installed.

The EGCS's manage elevators to minimize the evaluation criteria; it is, however, difficult to satisfy all criteria at the same time. Therefore, the EGCS is designed to satisfy each criterion at certain levels while reducing the average waiting time of passengers.

An EGCS consists of hall call buttons, car call buttons, elevators, and a group controller. If a passenger wants to go to another floor, he/she presses a direction (hall

call) button and waits for an elevator to arrive, then enters the elevator and presses a floor (car call) button in the elevator. The group controller selects a suitable elevator when a passenger presses the hall call button. In this case, the group controller considers the current situation of the building to select the most appropriate elevator in the group.

In the EGCS, it is difficult to select a suitable elevator for the following reasons. First, the EGCS is very complex. If a group controller manages elevators and assigns hall calls to the elevators, the controller considers cases. Second, the controller must consider the hall calls which will be generated in the near future. Third, it must consider many uncertain factors, such as the number of passengers at the floors where hall calls and car calls are generated. Fourth, it must be possible for a system manager to change the control strategy. Many studies have been done and significant progress has been made regarding the algorithm to assign hall calls, but none of them are optimal solutions. Some recent studies [5], [6], [7], [8] show more desirable results than previous systems using conventional statistical methods. However, it is still difficult to define and reflect the control strategy and reflect the strategy when the EGCS selects suitable elevators. The method in [5] used fuzzy logic to classify the traffic pattern, but the important features, such as the time and the ratio of in/out passengers, were not used as input variables. The method in [6] classified the traffic patterns and assigned the hall calls by using fuzzy logic, but it was difficult for the system manager to define control strategy. Control strategy defines control variables by vertical traffic type where the major concern is;

- i. The average waiting time in the morning,*
- ii. The energy consumption during business hours,*

iii. The long-wait-percent in the evening.

The method in [7] classified the traffic patterns and calculated waiting times by using a neural network. Hall calls were assigned by the waiting times, so it was difficult to consider the other evaluation criteria such as power consumption. Moreover, the system could not reflect the strategy. The method in [8] considered the control strategy generation, but used the conventional suitability functions at the hall call assignment, so it was difficult to reflect and tune the system for the control strategy. Specifically, we focus on the control strategy generation and the hall call assignment parts of the EGCS. The control strategy generation part prepares for the hall call assignment by using the system manager's request, and the hall call assignment part assigns hall calls to suitable elevators. The parking policy used by the EGCS, also contributes in the optimal use of the elevator group. This thesis explores the optimal parking algorithm that enables the reduction of the waiting times of passengers.

2.2.2 Operation of Elevator group control system (EGCSs)

This section contains the overview of the general structure of EGCSs. There are two hall call (up, down) buttons on a floor, and multiple elevators as shown in Figure 2.1. The EGCS selects an elevator for the passenger who has pressed a hall call button. The selected elevator moves to the floor where the hall call occurred. To understand the EGCS, consider an example of the elevator group control process.

A passenger who is going to the 10th floor from the second floor presses the up hall call button.

- *The hall call signal is transmitted to the EGCS.*
- *The EGCS selects an elevator to service the passenger.*

- *The EGCS sends a message to the selected elevator.*
- *The selected elevator moves to the second floor and the passenger boards.*
- *The passenger presses the car call button for the 10th floor.*
- *The elevator sends a message to the EGCS and moves to the 10th floor.*
- *The elevator arrives at the 10th floor and the passenger leaves.*

The EGCS repeats the process of selecting service elevators for hall calls. We call the selection the hall call assignment. In the EGCS, the hall call assignment is important and the performance depends on the hall call assignment method. Many evaluation criteria are used to estimate the performance of the EGCS [9], [10]. In this thesis, the following three criteria are used.

- i. Average waiting time (AWT) is the time until the service elevator arrives at the floor after a passenger presses a hall call button. AWT is the average of all waiting times in a unit time.*
- ii. Long waiting percent (LWP) is the percentage of the passengers who wait more than 60 s in a unit time.*
- iii. RuN count (RNC) is the number of elevator moves in a unit time and is used to estimate the power consumption of the system since most energy is consumed by starting or stopping the elevator.*

Figure 2.2 shows the general structure of the EGCS. In Fig. 2.2, the EGCS manages four elevators in a building. Each elevator has its own controller represented by the car controller (CC) and communicates with the elevator group controller. The EGCS consists of three main parts and several modules.

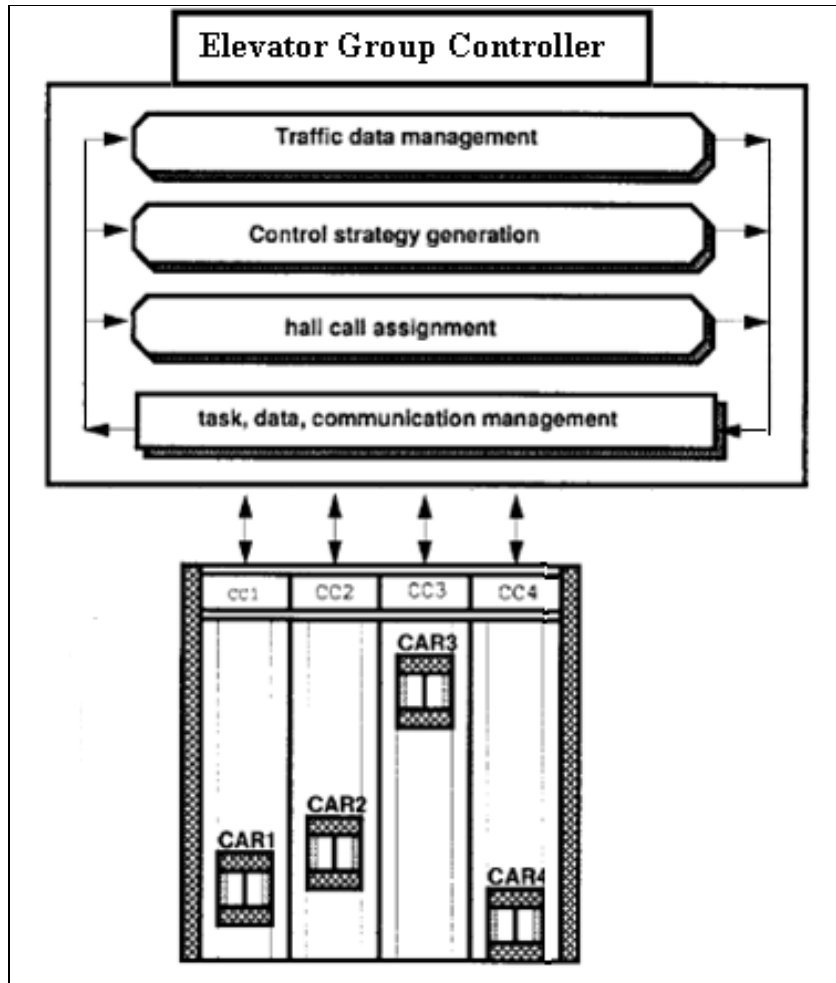


Figure 2.2. General structure of the EGCS.

The three main parts are;

- i. *The traffic data management,*
- ii. *The control strategy generation,*
- iii. *The halls call assignment.*

The **traffic data management part** collects various statistics of traffic data such as the number of hall calls, car calls, and passengers getting on or off the elevators. This part learns the traffic data and predicts the future traffic [11], [12]. The **control strategy generation part** classifies the traffic into one of several modes and

determines which hall call assignment method is suitable for the classified traffic mode. Finally, **the hall call assignment part** selects service elevators for hall calls via the method determined in the control strategy generation part. In addition, there are several small modules to support the main parts such as task management, data management, and communication management parts. The EGCS generally tests the degree of suitability of each elevator for a hall call and selects an elevator with the best suitability. If we consider the above three evaluation criteria, then the suitability can be represented by their combination.

Given that;

$\phi(i)$ the suitability function for the i^{th} elevator;

$T_{LWP}(i), T_{AWT}(i)$ and $T_{RNC}(i)$ the evaluation values of i^{th} elevator for AWT, LWP and RNC , respectively;

$T_{AWT}(i)$ the waiting time of the passenger;

$T_{LWP}(i)$ the probability of the long waiting;

$T_{RNC}(i)$ the number of moves in a time interval when i^{th} elevator assigns for the hall call.

The suitability of i^{th} elevator for a hall call can be represented by the following equation:

$$\phi(i) = v_1 \cdot T_{AWT}(i) + v_2 \cdot T_{LWP}(i) + v_3 \cdot T_{RNC}(i) \quad (2.1)$$

In this equation, the, v_1, v_2 and v_3 are weighing factors of each evaluation criterion.

The elevator with the minimum value of the function is selected.

2.3 Measurement and forecasting of the passenger Traffic

2.3.1 Measurement of passenger traffic

The number of entering and exiting passengers per floor and per direction was chosen as a variable to be forecasted. This data was based on a large number of detailed measurements. To find out the passenger traffic flow in a building, the car load and the photocell signal information are utilized. The benefit of these two detectors is that the passenger is not aware of the measurement at all. Two methods are used since sometimes the old inaccurate load weighing devices are not considered in elevators, or with wide doors the photocell signal information can be inaccurate. If either of the measurement methods fails, only the prevailing method is applicable. If the photocell signals and car load estimation are near to each other, the estimate with the smallest value is chosen. If the passenger arrival rate with either of the methods is 33 per cent less than with the other method, the lower value is rejected. These heuristic rules are based on practical reasoning and in practice they have proven to be accurate enough to distinguish the best estimate.

With an accurate digital car load weighing device the step-wise changes in the load values can be counted during an elevator stop. A step-wise increment in the car load information indicates that a passenger enters the car. A sudden decrease means that passenger exits the car, correspondingly. The threshold load value to recognize an increment of decrement step is scaled to a rated car load. The threshold value is 25 % of the average passenger weight. The threshold value varies between 17-20 kg, depending on the elevator car size. In Figure 2.3 an example of load variation during a stop is shown [12]. With the step counting method three entering and two exiting passengers are recognized. If only arrival load, minimum load and departure load

information were analyzed, one exiting and one entering passenger would be recognized. The information of the car load signal oscillates especially when stopping at a floor.

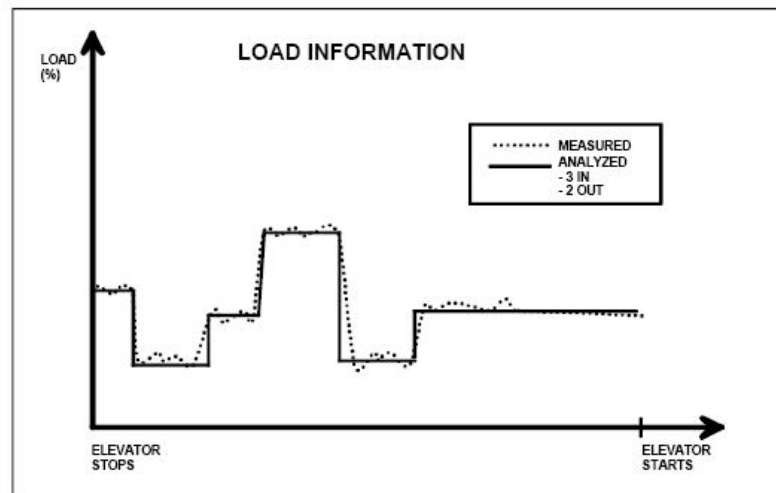


Figure 2.3 Step counting method to count the number of entering and exiting passengers during an elevator stop

The other method to estimate the number of passenger transfers is an electronic safety beam, or a photocell signal device installed in the elevator door opening. An entering or exiting passenger breaks the safety beam or the photocell signal. An estimate to the total number of the transported passengers is obtained by dividing the number of photocell cuts by two since one passenger breaks the light ray twice during one journey. The direction of the transfer, in or out of the car, cannot be deduced without additional traffic data. From the landing call and elevator status data the individual passenger trips can be reconstructed by keeping a record of the number of passengers inside the car during the elevator up and the down trips. Rough rules are used when analyzing the event data. The number of passengers inside the car

during an up or a down trip cannot momentarily exceed the nominal car carrying capacity M . If N is the number of floors, the limits to the entering passengers P_{ii} and exiting passengers P_{oi} at floor i are;

$$0 \leq \sum_{i=1}^k P_{ii} - \sum_{i=1}^k P_{oi} \leq M, \text{ for } k = 1, \dots, N \quad (2.2)$$

At the reversal floors it is assumed that all the entered passengers during the trip have exited the elevator

$$\sum_{i=1}^k P_{ii} - \sum_{i=1}^k P_{oi} = 0 \quad (2.3)$$

It is assumed that the floors of existing up calls form a set K . If there is an up call at the floor 1 where the car stops during the up trip, at least one passenger enters the car

$$1 \leq P_{ii} \leq M, \forall i \in K \quad (2.4)$$

An analogous equation is valid for the down calls. The rules of Eq. (2.2)-(2.4) can be used when the traffic is measured with external analyzers attached to the elevator control board. An accuracy of 10 % for the passenger traffic intensity can be obtained when reconstructing individual passenger trips from elevator traffic events. The accuracy of the photocell signal based method can still be improved.

Within a control system, more information on the elevator status is available than with external traffic analyzers. The passenger trips can be reconstructed more accurately as the car calls given inside each car are known. The number of entering passengers can be adjusted from the number of new car calls. The number of entering

passengers from floor i is at least the number of new car calls cc_i from that floor, but always less than the nominal car size, i.e.;

$$cc_i \leq P_{ii} \leq M, \text{ if } i = 1, \dots, N \quad (2.5)$$

The destinations of entering passengers are obtained from the new or existing car call floors. If the floors of existing car calls form a set C , it can be assumed that at least one passenger exits at the car call floor i , then;

$$1 \leq P_{oi} \leq M, \forall i \in C \quad (2.6)$$

By using the rules of Equations (2.2) - (2.6) an error of 5 % was found for the passenger arrival and destination floors [5]. No exceptional behavior, such as passengers remaining in the car at the reversal floor, or one passenger pressing landing call buttons in both directions, was taken into account. Any exceptional behavior will reduce the accuracy, and sometimes the method fails. That is why the test to find the more accurate method is always used.

2.3.2 Forecasting method

As mentioned in Section 2.3.1, long-term statistical forecasts of the passenger traffic are formed, and the forecasts are utilized in several control operations. The forecasting methods of economical and technical phenomena can roughly be classified either in time series or causal regression [13]. These methods give the best

results in long-term forecasting. In the time series methods the forecast is based on the earlier behavior. The time series methods are suitable for the forecasting of elevator traffic phenomena since the forecast periods are short. According to the traffic measurements of an elevator system, the traffic repeats quite similarly day by day.

Exceptional days are known in advance from the calendar. Within a day 97 per cent of the 12 changes in traffic intensity can be explained by periodical changes [5]. A fifteen minute period was considered to be suitable for elevator traffic phenomena [5]. Half an hour is too long for the peak traffic periods and, on the other hand, five minutes is too short to analyze the elevator round trips. The statistics for the day contains a total of 96 time slots. In the group control, full-day statistical forecasts are formed once a day. In a real time control system there is often shortage of the calculation time. The available memory size sets its limitations to the amount of data to be collected. In most situations a Single Exponential Smoothing method gives the best results

$$F_t = \alpha Y_t + (1 - \alpha) F_{t-1} \quad (2.7)$$

Where the F is a smoothed value and α is a smoothing constant with a value between zero and one. In this method an exponentially decreasing weight is given for the old data. The old data is gradually ignored and the new data is adapted instead. For controls in real buildings the optimal smoothing parameter is not easy to define in advance. A method that continuously optimizes the smoothing parameter, the Adaptive Response Rate Single Exponential Smoothing (ARRSES) method, was found to be the most suitable for the present application [13]

$$F_t = \alpha_t Y_t + (1 - \alpha_t) F_{t-1} \quad (2.8)$$

where

$$\alpha_t = \left| \frac{E_t}{M_t} \right| \quad (2.8a)$$

$$E_t = \beta e_t + (1 - \beta)E_{t-1} \quad (2.8b)$$

$$e_t = Y_t - F_{t-1} \quad (2.8c)$$

Variable F_{t-1} refers to the smoothed value of the previous period, and Y_t to the observed value of the same period. F_t is the new smoothed value, and α_t is the smoothing constant. A constant value between zero and one, e.g. 0.2, is set beforehand for the parameter β . In the group control the smoothing constants in Equation (2.8) are updated per floor and per direction.

2.3.3 Elimination of seasonal variations

In the initialization phase of statistical forecasts single averages for the first five days are gathered. During that time the group control operates without the forecast data, which reduces the control efficiency. Then the landing call times are optimized in the basic call allocation algorithm. Statistics for a full day are collected before being used to modify the statistical forecasts. The seasonal variations in the measured traffic data are eliminated by the calendar, and by making two tests. With these tests the periodical changes, such as Christmas, are not included in the statistical forecasts. Exceptional days, or failures in recording the data, are eliminated. If there is a power failure for more than half an hour during the day, the observed data of that day is rejected.

The traffic intensity profile during the day is checked using a C_2 -test. The hours of the day are divided into six time slices between 7:00 and 19:00. Five degrees of freedom are used in testing. If the observed intensity is Y_t , and the forecast intensity is F_t , the test value is

$$\chi^2 = \sum_{i=1}^6 \frac{(Y_t - F_t)^2}{F_t} \quad (2.9)$$

If the value C_2 is greater than the critical value $C_2 = 0.95$, the observed intensity differs significantly from the forecast value and it is rejected. If the observed intensities are proportional to the intensities of the forecast day, the observed data is accepted.

Another test is made for the passenger arrival rate during the day. The observed arrival rate is compared with the forecast arrival rate of a typical day. The arrival rate is assumed to follow Poisson's distribution and the following test is applied [14]

$$X_{day} < (X_{av} - 3(\sqrt{X_{av}})) \quad (2.10)$$

Where X_{day} is the measured number of passengers during the day and X_{av} is a forecast number of passengers during a typical day. The test is based on the normal distribution approximation for Poisson's distribution. If the number of arriving passengers has been too low, the day will not be accepted in the statistics. If there have been more than ten consecutive days that have been rejected, the control clears the old statistics and starts to collect a new set.

2.3.4 Forecast traffic profile in an office building

The main streams of traffic flows can be divided roughly into three traffic components. The incoming, outgoing and inter-floor passenger traffic components are shown in Figure 2.4. During the incoming traffic, passengers arrive at the building, and during the outgoing traffic they exit the building. In the inter-floor traffic the passengers travel from one populated floor to another inside the building. Real traffic patterns during a day are combinations of these three traffic components.

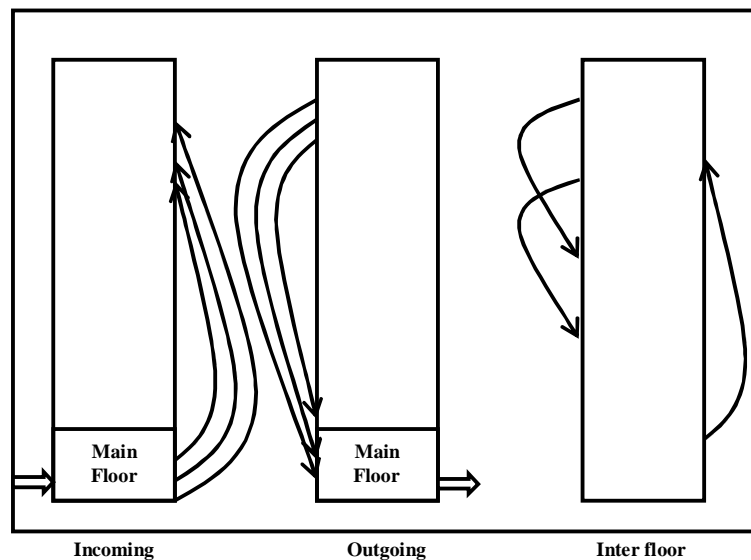


Figure 2.4 Three main passenger traffic components

2.4 Parking Algorithm of Passenger Elevators

2.4.1 Introduction.

Parking of free cars has been identified as an important part of elevator group supervisory control algorithms [15]. In this thesis, we consider the problem of optimally dispatching empty cars to desired parking locations in a manner that

minimizes the usual optimization criterion in group scheduling algorithms: the waiting time of future hall calls [16].

A parking algorithm could actively move empty cars so as to anticipate and intercept future hall calls, long before these calls actually occur. Reasoning about events far into the future and deciding on a suitable course of action can be handled by a planning sub-system that decides upon new parking locations for all free cars, as soon as their number changes, and a corresponding plan-execution sub-system that brings the free cars to these parking locations.

The idea of actively moving empty cars with the explicit purpose of parking them favorably with respect to future hall calls is present in several studies on optimal group elevator scheduling. One possibility is to use the statistical properties of the traffic pattern in order to dispatch cars to the floors where they would be most needed. In the case of pure up-peak traffic, it is clear that a car that has delivered all of its passengers should be dispatched immediately back to the lobby for the next batch of passengers.

This insight has been used in the provably optimal solution for pure up-peak traffic proposed by Pepyne and Cassandras [17]. Although pure up-peak traffic rarely exists in practice and there is almost always additional inter-floor traffic, active parking of cars is an important element of most industrial elevator control systems [15].

Our strategy is to re-park all available cars (i.e., the ones that are not currently serving passengers) as soon as their number changes, due to one of the following two

events: both a car becomes free (empty) and available for service, or a car is assigned to service a new hall call and is no longer empty. We assume that the optimal parking locations for a fixed number of free cars when the arrival rates are fixed remain the same, i.e., we ignore the effects the remaining (busy) cars might have on the optimal parking locations of the free ones. Under this assumption, planning is reduced to the computation of a policy that maps sets of free cars of various cardinality to the corresponding parking locations. In other words, the planning system computes a universal plan [4] covering all possible numbers of free cars, and uses it as long as the traffic flow in the building remains relatively constant; as soon as the stochastic properties of the traffic change, a new universal plan (policy) is computed.

2.4.2 Parking policies and their execution

We are considering a building of F floors equipped with N_c identical elevator cars. At any particular moment, C of these cars are free, i.e., having no hall or car calls assigned to them ($0 \leq C \leq N_c$). When a new hall call is signaled, a scheduling algorithm assigns it to one of the N_c cars; according to the accepted rules, this assignment is never revoked later. As a result, the number C of empty cars will either decrease (if the new hall call is assigned to a free car) or remain the same (if the new hall call is assigned to a busy car). C increases when a car completes servicing all hall and car calls assigned to it, and becomes free. A parking decision is made and executed in this case too, in an identical manner.

We assume that the parking locations always coincide with one of the landings, i.e., a car is never parked between two adjacent landings (a formal proof that such parking positions are optimal in down-peak traffic is presented in [18]).

When a parking decision has to be made, the free cars can be either already parked at a floor, or moving between floors in the process of executing a previous parking decision. We denote by y_i , $i = 1 \dots C$ the floors where each car i is at that moment. If car i is not moving, y_i is simply the floor where the car is located; when car i is moving, y_i is the first floor where it can stop and possibly reverse its direction. (We assume that a car cannot reverse its direction between landings, even though such a possibility is likely to increase the responsiveness and efficiency of the parking algorithm, if it were allowed.) Once the locations of the cars $y = [y_1, y_2, \dots, y_C]$ are known and the desired parking positions x have been determined, a parking plan has to be devised and executed by a parking subsystem. The objective of this plan is to move the cars from their current positions y to the desired parking floors x as quickly as possible. Thus, the parking subsystem has to decide which of the cars should go to each of the parking locations. Since there are $O(C!)$ possible matches between the C parking positions and the C cars, finding the optimal plan (matching) is not a trivial problem.

Currently two parking algorithms are in use; the Basic Parking and Dynamic Assignments algorithms. Basic parking is the manual control and is usually implemented by assigning the main floor (usually Ground Floor) as the busiest floor. This floor is given the highest priority and assigned more cars. Other cars are arbitrary assigned other floors depending on the understanding of the operation by the service company. The other type of control is referred as the Dynamic Assignment. During operation, the group of elevators keeps statistics of the number of calls and the mean waiting time for every floor. These two parameters, for every

floor, help the lift system to identify the busiest floors. The group control then assigns more cars to the busiest floors [15] [16] [17].

When a parking decision has to be made, the free cars can be either already parked at a floor, or moving between floors in the process of executing a previous parking decision.

The objective of this algorithm is to move the cars from their current positions to the desired parking floors as quickly as possible. Thus, the optimal parking algorithm has to decide which of the cars should go to each of the parking locations.

Our general strategy in the two cases described below (down-peak traffic and up-peak traffic) is to first analyze how the passenger flow influences the final positions of the cars when they become free, then identify inefficiencies resulting from uneven distribution of free cars, and finally decide how the cars should be re-parked so that the responsiveness of the system to new hall calls could be improved.

2.4.2.1 Parking in down-peak traffic

Under down-peak traffic, most of the passengers depart from upper floors and are delivered to the lobby. As a result, when a car becomes free, it is usually located at the lobby. If such cars are left where they delivered the last passenger (the lobby), they would be far away from the locations where new calls are likely to originate (the upper floors). In order to amend this mismatch between where the cars are and where they would be needed the most, empty cars can be moved from the lobby to the upper floors as soon as they deliver the last passenger.

In order to make the problem tractable, for the case of down-peak traffic we choose to optimize the waiting time of only the first future hall call.

Furthermore, we make the assumption that the first hall call would be served by one of the free cars, rather than one of the cars that already have passengers assigned to them, and we also assume that the new call would occur only after the desired parking location of the cars has been attained. We can define the expected waiting time of the first future call as a function of the state of free cars \mathbf{x} only:

$$Q(x) = \sum_{f=1}^F pf_{\min_i} T(\chi_i, f); \quad (2.11)$$

Where pf is the probability that the next passenger would arrive at floor f , and $T(X_i, f_2)$ is the time it would take for a car parked at floor f_1 to serve a call originating at floor f_2 .

The number of possible placements of C cars at F landings is F^C , which is exponential in F and exhaustive computation of $Q(\mathbf{x})$ for all possible \mathbf{x} is very expensive. However, intuition suggests that the optimal parking position would be the one that parks the cars as evenly as possible with respect to the arrival distribution of passengers. An even distribution of cars with respect to that probability would be one that positions the C available cars so that their respective probabilities of serving the next call would be equal ($1/C$). One approximate way to achieve this is to split the building into C zones, each served by one of the C cars. Given an array of cumulative arrival probabilities $P_j, j = 1 \dots F$, such that

$$P_i = \sum_{i=1}^j P_i \quad (2.12)$$

This parking policy can be computed by the parking algorithm. This solution would be optimal with respect to the minimization criterion only when the average time to serve a call is the same for each zone. In practice, however, this time would be higher for larger zones, so a correction is necessary; in direction of shrinking relatively larger zones so that they cover arrivals with probability lower than $1/C$.

2.4.2.2 Parking in up-peak traffic

The parking solution based on matching the arrival distribution of passengers to the parking location of the cars, while successful for down-peak traffic, is not sufficient for up-peak traffic. The reason for this is the very uneven distribution of arrival rates, the majority of passengers arrive at the lobby, and most of the waiting time is generated by such passengers. Hence, it is of primary importance to reduce the waiting time at the lobby under this type of traffic. However, parking free cars with respect to only such passengers is not very efficient either, sending each and every car to the lobby immediately after it becomes free leaves the rest of the building uncovered, and the waiting times of passengers arriving at the upper floors start to dominate the overall average waiting time. It is clear that if C free cars are available for service, some proportion of them should be sent to the lobby, while the remaining ones should be parked at the upper floors, again distributed evenly with respect to the arrival rates there. The question then becomes how to determine this proportion.

The simple optimization criterion used for down-peak traffic (the immediate cost (waiting time) $Q(\mathbf{x})$ of only the next arrival) is not adequate for the case of up-peak traffic. If only $Q(\mathbf{x})$ were minimized, the computed optimal number of cars at the lobby would always be only one, since one car is sufficient to answer a potential

single call at the lobby, and the remaining cars would be better used at the upper floors in order to minimize the waiting times of arrivals there. A much more appropriate optimization criterion for this traffic pattern is the average waiting time over a longer time horizon (preferably infinitely long). In this case, it is more convenient to express the optimization criterion, W_N as the average waiting time over a sequence of N future arrivals:

$$W_N = 1/N \left[\sum_{i=1}^N Q(s_i) \right] \quad (2.13)$$

Where

s_i is the state of the elevator bank when call i occurs,

$Q(s_i)$ is the expected waiting time of passenger and

the expectation $[\]$ is taken with respect to the distribution of the next N arrivals.

The true long-term average waiting time of passengers, which is the exact criterion we would like to optimize, is the limit of W_N as N becomes infinitely large (i.e., the horizon becomes infinitely long):

$$W_\infty = \lim_{N \rightarrow \infty} W_N. \quad (2.14)$$

We define the system to be in a particular state represented by a parking position not only when the system has assumed that position, but also when it is in the process of moving towards it.

In order to further reduce the number of states, we will assume that a parking position for the case of up-peak traffic is specified by the pair of numbers (L, U), where L is the number of cars parked at the lobby, while U is the number of cars parked at the upper floors. We further make the assumption that the U cars are spaced evenly along the height of the building.

Thus, once the pair (L, U) is given and the height of the building is known, the corresponding detailed parking position \mathbf{x} can be generated by parking L cars at the lobby and distributing the remaining U cars along the height of the building. As a consequence, we can define the immediate cost $Q(L,U)$ of a state (position) (L,U) as the corresponding immediate cost (waiting time) of the complete position \mathbf{x} :

$$Q(L,U) = Q(\mathbf{x}) \quad (2.15)$$

Under the proposed notation for parking states, the decision that has to be made when C free cars are available is how many of them should be sent to the lobby (L), and how many should be sent to the upper floors ($U = C - L$). For example, if there are three free cars available, the possible decisions are (0, 3), (1, 2), (2, 1), and (3, 0). One very compact representation of such a policy is the N_c -dimensional vector of values L_C , $C = 1 \dots N_c$, whose C^{th} element specifies how many cars should be parked at the lobby when C of all cars are free.

2.5 Artificial Neural Networks (ANN)

2.5.1 Introduction

Artificial Neural Networks (ANN), are computational models that consist of a number of simple processing units that communicate by sending signals to each other

over a large number of weighted connections [5]. They were originally developed from the inspiration of human brains. In human brains, a biological neuron collects signals from other neurons through a host of fine structures called dendrites. The neuron sends out spikes of electrical activity through a long, thin strand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes [19].

Like human brains, neural networks also consist of processing units (artificial neurons) and connections (weights) between them [19]. The processing units transport incoming information on their outgoing connections to other units. The "electrical" information is simulated with specific values stored in those weights that make these networks have the capacity to learn, memorize, and create relationships amongst data [11].

A very important feature of these networks is their adaptive nature where "learning by example" replaces "programming" in solving problems. This feature renders these computational models very appealing in application domains where one has little or incomplete understanding of the problems to be solved, but where training data are available.

Neural networks, sometimes referred to as connectionist models, are parallel-distributed models that have several distinguishing features [20]:

- i) A set of processing units;*
- ii) An activation state for each unit, which is equivalent to the output of the unit;*
- iii) Connections between the units. Generally each connection is defined by a weight w_{jk} that determines the effect that the signal of unit j has on unit k ;*
- iv) A propagation rule, which determines the effective input of the unit from its external inputs;*
- v) An activation function, which determines the new level of activation based on the effective input and the current activation;*
- vi) An external input (bias, offset) for each unit;*
- vii) A method for information gathering (learning rule);*
- viii) An environment within which the system can operate, provide input signals and, if necessary, error signals.*

To understand how an artificial neural network works we have to first of all examine how a biological neuron works in regard to the whole biological nervous system.

2.5.2 Biological neural networks

The neuron has four main regions to its structure. The cell body, or soma, has two offshoots from it, the dendrites, and the axon, which end in presynaptic terminals.

The cell body is the heart of the cell, containing the nucleus and maintaining protein synthesis.

A neuron contains a host of fine structures called dendrites, which branch out in a treelike structure, and receive signals from other neurons. A neuron usually only has one axon which grows out from a part of the cell body called the axon hillock. The axon conducts electric signals generated at the axon hillock down its length. These electric signals are called action potentials. The other end of the axon may split into several branches, which end in a presynaptic terminal. Action potentials are the electric signals that neurons use to convey information to the brain[11]. At the end of each axon branch, a structure called a synapse converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurons [8].

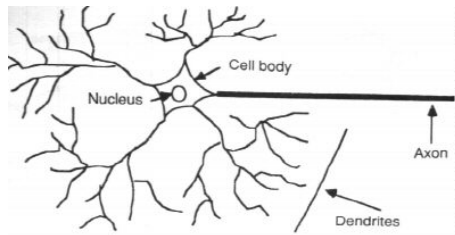


Figure 2.5a Components of a neuron

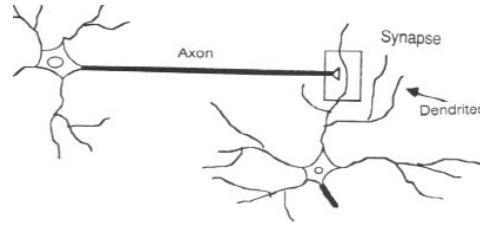


Figure 2.5b Synapse

When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes. Learning repetition of a task, even exposure to a new or continuing

stimulus can cause the brain's connection strengths to change, some synaptic connections becoming reinforced as new ones are being created, others weakened or in some cases disappearing altogether [1].

2.5.3 The Processing Unit

The processing unit (Figure 2.6), also called a neuron or node, performs a relatively simple job; it receives inputs from neighbors or external sources and uses them to compute an output signal that is propagated to other units.

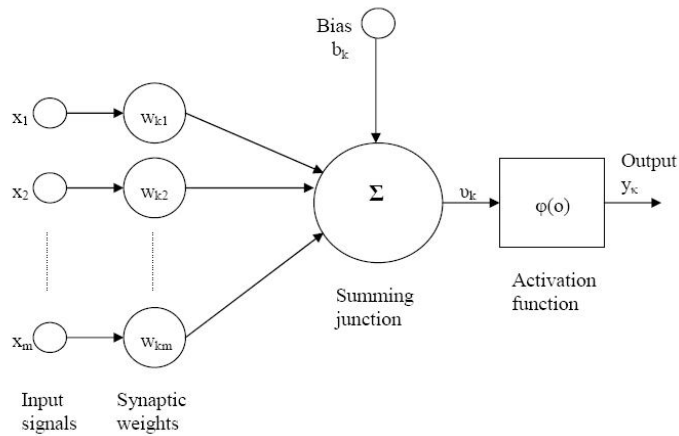


Figure 2.6 Processing unit

Within the neural systems there are three types of units:

- 1) Input units, which receive data from outside of the network;
- 2) Output units, which send data out of the network;
- 3) Hidden units, whose input and output signals remain within the network.

Each unit j can have one or more inputs $x_0, x_1, x_2, \dots, x_n$, but only one output z_j . An input to a unit is either the data from outside of the network, or the output of another unit, or its own output.

2.5.4 Combination Function

Each non-input unit in a neural network combines values that are fed into it via synaptic connections from other units, producing a single value called net input. The function that combines values is called the combination function, which is defined by a certain propagation rule. In most neural networks we assume that each unit provides an additive contribution to the input of the unit with which it is connected. The total input to unit j is simply the weighted sum of the separate outputs from the connected units plus a threshold or bias term, θ_j ;

$$a_j = \sum_{i=1}^n w_{ji} x_i + \theta_j \quad (2.16)$$

The contribution for positive w_{ji} is considered as an excitation and an inhibition for negative w_{ji} . We call units with the above propagation rule sigma units.

In some cases more complex rules for combining inputs are used. One of the propagation rules known as sigma-pi has the following format [20]:

$$a_j = \sum_{i=1}^n w_{ji} \prod_{k=1}^m x_{ik} + \theta_j \quad (2.17)$$

Lots of combination functions usually use a "bias" or "threshold" term in computing the net input to the unit. For a linear output unit, a bias term is equivalent to an

intercept in a regression model. It is needed in much the same way as the constant polynomial '1' is required for approximation by polynomials.

2.5.5 Activation Function

Most units in neural network transform their net inputs by using a scalar-to-scalar function called an activation function, yielding a value called the unit's activation. Except possibly for output units, the activation value is fed to one or more other units. Activation functions with a bounded range are often called squashing functions. Some of the most commonly used activation functions are [21]:

- 1) Identity function (Figure 2.7a)

$$g(x) = x \quad (2.18)$$

It is obvious that the input units use the identity function. Sometimes a constant is multiplied by the net input to form a linear function.

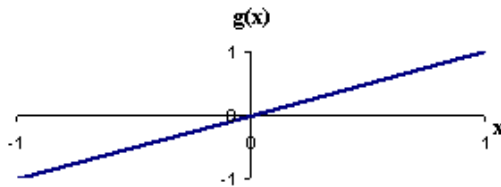


Figure 2.7a Identity function

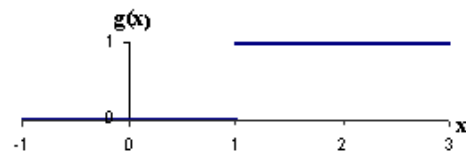


Figure 2.7b Binary step function

- 2) Binary step function (Figure 2.7b)

Also known as threshold function or Heaviside function. The output of this function is limited to one of the two values:

$$g(x) = \begin{cases} 1 & \text{if } (x \geq \theta) \\ 0 & \text{if } (x < \theta) \end{cases} \quad (2.19)$$

This kind of function is often used in single layer networks.

3) Sigmoid function (Figure 2.8)

$$g(x) = \frac{1}{1 + e^{-x}} \quad (2.20)$$

This function is especially advantageous for use in neural networks trained by backpropagation, because it is easy to differentiate, and thus can dramatically reduce the computation burden for training. It applies to applications whose desired output values are between 0 and 1.

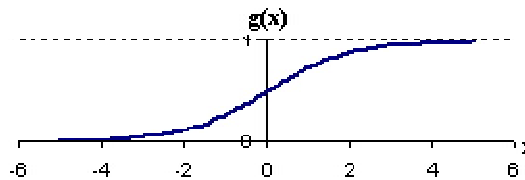


Figure 2.8a Sigmoid function

4) Bipolar sigmoid function

$$g(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \quad (2.21)$$

This function has similar properties with the sigmoid function. It works well for applications that yield output values in the range of $[-1, 1]$.

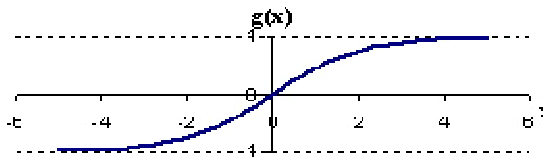


Figure 2.8b Bipolar sigmoid function

Activation functions for the hidden units are needed to introduce non-linearity into the networks. The reason is that a composition of linear functions is again a linear function. However, it is the non-linearity (i.e., the capability to represent nonlinear functions) that makes multi-layer networks so powerful. Almost any nonlinear function can perform the job, although for back-propagation learning it must be differentiable and it helps if the function is bounded. The sigmoid functions are the most common choices [22] [23].

For the output units, activation functions should be chosen to be suited to the distribution of the target values. We have already seen that for binary $[0, 1]$ outputs, the sigmoid function is an excellent choice. For continuous-valued targets with a bounded range, the sigmoid functions are again useful, provided that either the outputs or the targets to be scaled to the range of the output activation function. But

if the target values have no known bounded range, it is better to use an unbounded activation function, most often the identity function (which amounts to no activation function). If the target values are positive but have no known upper bound, an exponential output activation function can be used [22] [23].

2.5.6 Network topologies

The topology of a network is defined by the number of layers, the number of units per layer, and the interconnection patterns between layers. They are generally divided into two categories based on the pattern of connections:

- 1) Recurrent networks (Figure 2.9a), which contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which activation does not change further. In other applications in which the dynamical behavior constitutes the output of the network, the changes of the activation values of the output units are significant.
- 2) Feed-forward networks (Figure 2.9b), where the data flow from input units to output units is strictly feed-forward. The data processing can extend over multiple layers of units, but no feedback connections are present. That is, connections extending from outputs of units to inputs of units in the same layer or previous layers are not permitted. Feed-forward networks are the main focus of this thesis.

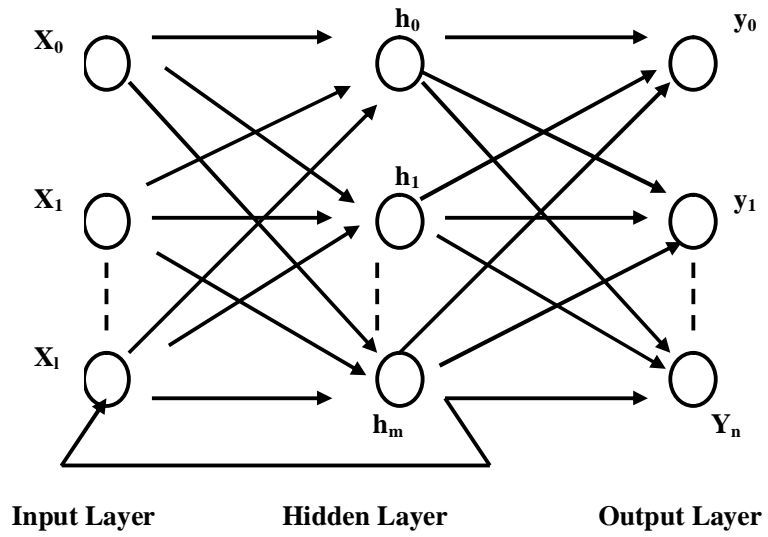


Figure 2.9a Recurrent neural network

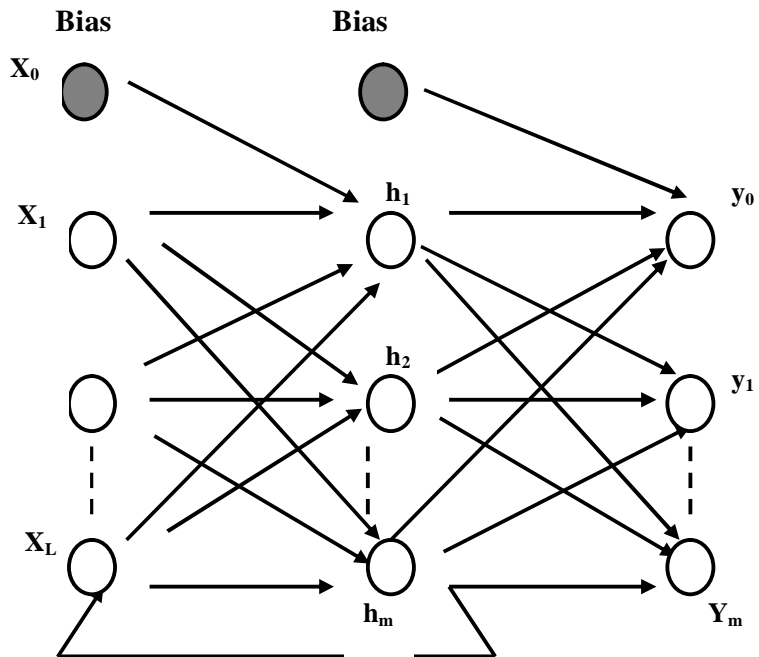


Figure 2.9b Feed-forward neural network

2.5.7 Network Learning

The functionality of a neural network is determined by the combination of the topology (number of layers, number of units per layer, and the interconnection pattern between the layers) and the weights of the connections within the network. The topology is usually held fixed, and the weights are determined by a certain training algorithm. The process of adjusting the weights to make the network learn the relationship between the inputs and targets is called learning, or training. Many learning algorithms have been invented to help find an optimum set of weights that result in the solution of the problems. They can roughly be divided into two main groups:

i) Supervised Learning - The network is trained by providing it with inputs and desired outputs (target values). These input-output pairs are provided by an external teacher, or by the system containing the network. The difference between the real outputs and the desired outputs is used by the algorithm to adapt the weights in the network (Figure 2.10a). It is often posed as a function approximation problem - given training data consisting of pairs of input patterns x , and corresponding target t , the goal is to find a function $f(x)$ that matches the desired response for each training input.

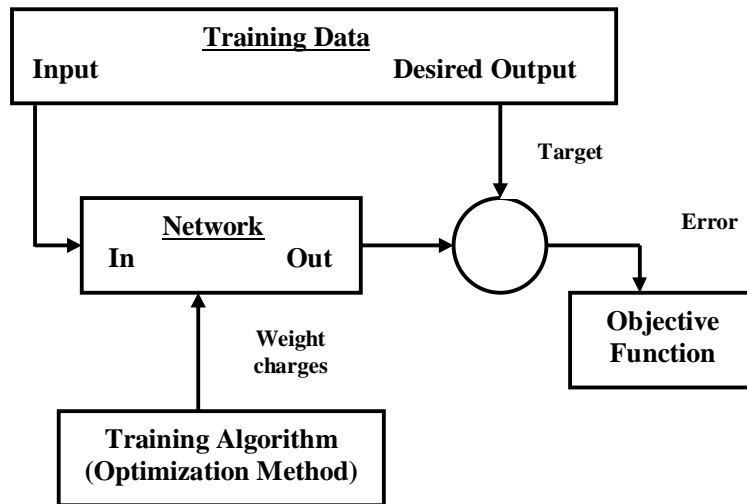


Figure 2.10a Supervised learning model

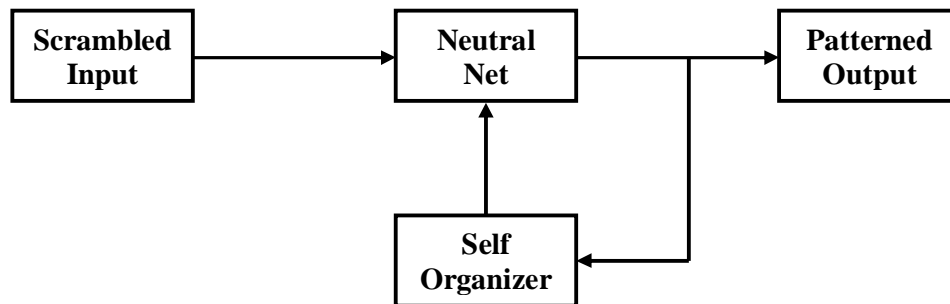


Figure 2.10b Unsupervised learning model

ii) Unsupervised Learning - With unsupervised learning, there is no feedback from the environment to indicate if the outputs of the network are correct (Figure 2.10b). The network must discover features, regulations, correlations, or categories in the input data automatically. In fact, for most varieties of unsupervised learning, the targets are the same as inputs. In other words, unsupervised learning usually performs the same task as an auto-associative network, compressing the information from the inputs.

2.5.8 Objective Function

To train a network and measure how well it performs, an objective function (or cost function) must be defined to provide an unambiguous numerical rating of system performance. Selection of an objective function is very important because the function represents the design goals and decides what training algorithm can be taken. To develop an objective function that measures exactly what we want is not an easy task. A few basic functions are very commonly used. One of them is the sum of squares error function,

$$E = \frac{1}{NP} \sum_{p=1}^P \sum_{i=1}^N (t_{pi} - y_{pi})^2 \quad (2.22)$$

where p indexes the patterns in the training set, i indexes the output nodes, and t_{pi} and y_{pi} are, respectively, the target and actual network output for the i th output unit on the p th pattern. In real world applications, it may be necessary to complicate the function with additional terms to control the complexity of the model.

2.5.9 Basic Architecture of a Feed-forward network.

A layered feed-forward network consists of a certain number of layers, and each layer contains a certain number of units. There is an input layer, an output layer, and one or more hidden layers between the input and the output layer. Each unit receives its inputs directly from the previous layer (except for input units) and sends its output directly to units in the next layer (except for output units). Unlike the Recurrent network, which contains feedback information, there are no connections from any of

the units to the inputs of the previous layers nor to other units in the same layer, nor to units more than one layer ahead. Every unit only acts as an input to the immediate next layer. Obviously, this class of networks is easier to analyze theoretically than other general topologies because their outputs can be represented with explicit functions of the inputs and the weights.

An example of a layered network with one hidden layer is shown in Figure 2.11. In this network there are l inputs, m hidden units, and n output units. The output of the j^{th} hidden unit is obtained by first forming a weighted linear combination of the l input values, then adding a bias,

$$a_j = \sum_{i=1}^l w_{ji}^l x_i + w_{j0}^l \quad (2.23)$$

where $w_{ji}^{(l)}$ is the weight from input i to hidden unit j in the first layer and $w_{j0}^{(l)}$ is the bias for hidden unit j . If we are considering the bias term as being weights from an extra input (2.23) can be rewritten to the form of,

$$a_j = \sum_{i=1}^l w_{ji}^l x_i \quad (2.25)$$

The activation of hidden unit j then can be obtained by transforming the linear sum using an activation function $g(x)$

$$h_j = g(a_j) \quad (2.26)$$

The outputs of the network can be obtained by transforming the activation of the hidden units using a second layer of processing units. For each output unit k , first we get the linear combination of the output of the hidden units,

$$a_j = \sum_{i=1}^m w_{ji}^{(2)} h_i + w_{k0}^{(2)} \quad (2.27)$$

Again we can absorb the bias and rewrite the above equation to,

$$a_k = \sum_{j=0}^m w_{kj}^{(2)} h_j \quad (2.28)$$

Then applying the activation function $g_2(x)$ to (2.26) we can get the k^{th} output

$$y_k = g_2(a_k) \quad (2.29)$$

Combining (2.26), (2.27), (2.28) and (2.29) we get the complete representation of the network as

$$y_k = g_2\left(\sum_{j=0}^m w_{kj}^{(2)} g\left(\sum_{i=0}^i w_{ji}^{(1)} x_i\right)\right) \quad (2.30)$$

The network of Figure 2.12 is a network with one hidden layer. We can extend it to have two or more hidden layers easily as long as we make the above transformation further.

One thing we need to note is that the input units are very special units. They are hypothetical units that produce outputs equal to their supposed inputs. No processing is done by these input units.

2.6 Back-Propagation

The general nature of the back propagation training method means that a backpropagation net (a multilayer, feedforward net trained by backpropagation) can be solved in many areas. As is the case with most neural networks, the aim is to train the net to achieve a balance between the ability to respond correctly to the input

patterns that are used for training (memorization) and the ability to give reasonable (good) response to input that is similar, but not identical, to that used in training (generalization).

The training of a network by backpropagation involves three stages: the feedforward of the input training pattern, the calculation and backpropagation of the associated error and the adjustments of the weights. After training, application of the net involves only the computation of the feedforward phase. Even if training is slow, a trained net can produce its output very rapidly.

2.6.1 Architecture of a Backpropagation Net.

A multilayer neural network with one layer of hidden units (the Z units) is shown in figure 2.12. The output units (as Y units) and the hidden units also may have biases (as shown). The bias on a typical output unit Y_K is denoted by W_{OK} ; the bias on a typical hidden unit Z_{ij} is denoted V_{oj} . These bias terms act like weights on connections from units whose output is always 1. (These units are shown in figure 2.12 but are usually not displayed explicitly). Only the direction of information flow for the feedforward phase of operation is shown. During the backpropagation phase of learning, signals are sent in the reverse direction.

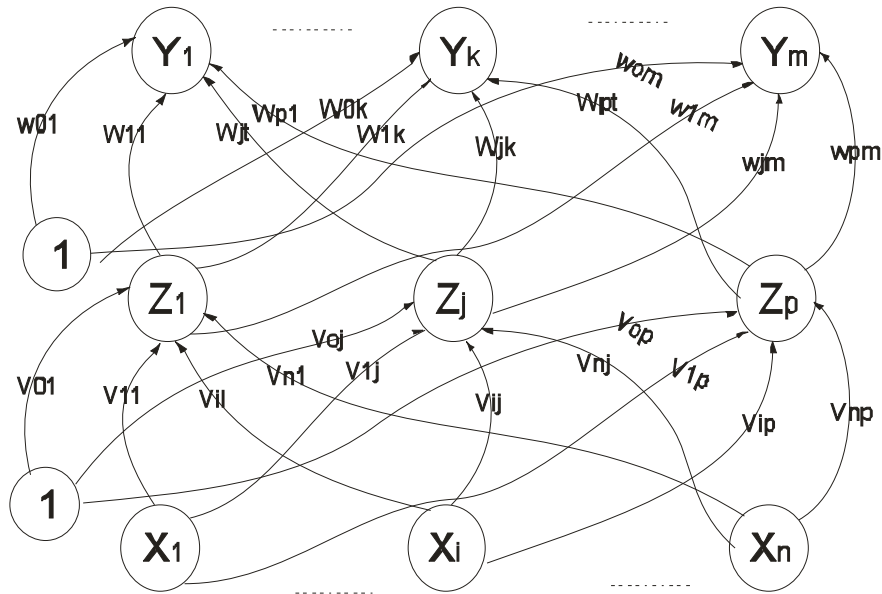


Figure 2.11 Single layer Backpropagation Net

During feedforward, each input unit (X_i) receives an input signal and broad-casts this signal to each of the hidden units Z_1, \dots, Z_p . Each hidden unit then computes its activation and sends its signal (z_j) to each output unit. Each output unit (Y_k) computes its activation (y_k) to form the response of the net for the given input pattern.

During training, each output unit compares its computed activation y_k with its target value t_k to determine the associated error for that pattern with that unit. Based on this error, the factor δ_k ($k = 1, \dots, m$) is computed. δ_k is used to distribute the error at output unit Y_k back to all units in the previous layer (the hidden units that are connected to Y_k). It is also used (later) to update the weights between the output and the hidden layer. In a similar manner, the factor δ_j ($j = 1, \dots, p$) is computed for each hidden unit Z_j . It is not necessary to propagate the error back to the input layer, but δ_j is used to update weights between the hidden layer and the input layer.

After all of the δ factors have been determined, the weights for all layers are adjusted simultaneously. The adjustments to weight w_{jk} (from hidden unit Z_j to output unit Y_k) is based on the factor δ_k and the activation z_j of the hidden unit Z_j . the adjustments to the weights v_{ij} (from input unit X_i to hidden unit Z_j) is based on the factor δ_j and the activation x_i of the input unit.

In the process of developing a training algorithm for back propagation net, the following nomenclature is used:

- X Input training vector: $X = (x_1, \dots, x_i, \dots, x_n)$
- T Output target vector: $t = (t_1, \dots, t_k, \dots, t_m)$
- δ_k Portion of error correction weight adjustments for w_{jk} that is due to an error at output unit Y_k ; also, the information about the error at unit Y_k that is propagated back to the hidden units that feed into unit Y_k .
- X_i Input unit i: For an input unit, the input signal and output signal are the same, namely, x_i .
- v_{oj} Bias on hidden unit j.
- Z_j The net input to Z_j is denoted z_{in_j}

$$Z_{in_j} = v_{oj} + \sum_i x_i v_{ij} \tag{2.31}$$

The output signal (activation) of Z_j is denoted z_j :

$$Z_j = \int(z_{in_j}) \tag{2.32}$$

- w_{ok} Bias on output unit k.
- Y_k Output unit k:
The net input to Y_k is denoted by y_{in_k} :

$$y_in_k = w_{oj} + \sum_i x_i w_{jk} \quad (2.33)$$

The output signal (activation) of Y_k is denoted y_k :

$$y_k = \int(y_in_k) \quad (2.34)$$

2.6.2 Activation function

An activation function for a backpropagation net has several important characteristics: it should be continuous, differentiable and monotonically non decreasing. Furthermore, for computational efficiency, it is desirable that its derivative be easy to computer

The activation function used is the binary sigmoid function, which has range of (0, 1) and is defined as

$$f_1(x) = \frac{1}{1 + \exp(-x)} \quad (2.35)$$

With

$$f_1'(x) = f_1(x)[1 - f_1(x)] \quad (2.36)$$

The function is illustrated in figure 2.8b above, Bipolar Sigmoid, range (0, 1)

CHAPTER THREE

DESIGN METHODOLOGY.

3.1 Elevator Simulator Design

3.1.1 Introduction

This section describes the design and development of a system that simulates the operation of an elevator model. This was necessitated due to the following;

- i. The Lift providers do not keep the passenger traffic data.*
- ii. Getting access to the Elevator machine rooms was difficult.*

The objects required to build the model were identified, and a system developed to simulate the model. Each of the objects acts independently and manages its own data. One or more objects manage the user interface. The model allows the user to specify how many of the principal objects the simulation should include on a given run. The output of the model gives a report showing the activity of the major objects..

3.1.2 The Model

3.1.2.1 The Description

Elevator simulators typically simulate passenger flow, but have also simulated the elevator movement. The passenger flow simulators typically perform one or a combination of three scenarios:

1. Passenger traffic from the lobby to the upper floors.
2. Passenger traffic from upper floors to the lobby.

3. Passenger traffic between floors.

For our model to simulate an elevator and provide meaningful information, the output will determine, for a given number of elevators, how many passengers per hour that it can carry from the lobby to the upper floors of the building.

The user determines the number of elevators in the building. The building contains ten floors. The number of passengers that can fit into the elevator was fixed to sixteen. The passengers were counted as they left the elevator at their destination floor. The destination floor was determined using a "random" Poisson interval. When all of the passengers in the elevator have reached their destination floors, the elevator returns to the lobby to pickup more passengers. The simulation continues until the user cancels it via keystroke. A report was generated showing the activity of the elevators and passengers.

3.1.2.2 Division into objects

The first step of identifying objects and classes began by generating a set of candidate classes and objects using the classical approach and behavior analysis. The eligible classes and objects generated using the classical approaches are shown in Table 3.1. Abstractions that relate to system function points revealed during behavior analysis are shown in Table 3.2.

Table 3.1 List of eligible objects discovered during analysis.

Tangible Things	Roles	Events	Interactions	Organizations	External Systems
Building, Elevator, buttons, cable, motor, pulley, motor controller, hydraulic cylinder, hydraulic pump, hydraulic controller, door edge, closed and locked switch, full open switch, floor indicator, bell/chime, lights, fans	Rider, repair person, inspector, observer	Request to floor, door recycle, door close, door open, go up, go down, stop, ring bell, update floor indication, go to floor, change directions, break down	Elevator & passengers, floor & passengers, elevator & floor	Life system, door system, building floors, elevator passenger requests	Power, building, closed, circuit, television, emergency, phone, security

Table 3.2 List of behaviors discovered during analysis.

Behaviors
<ul style="list-style-type: none">▪ User input for the number of elevators in the building.▪ User input to stop the simulation.▪ A random number of passengers load from lobby onto the elevator and requests a floor.▪ An observer counts the passengers as they load and limit them to the elevator capacity.▪ The elevator closes it's doors after a predetermined amount of time after they are open.▪ The passengers randomly request a floor number for where they want to go.▪ The elevator determines the lowest floor request from all the passengers when traveling up from the lobby and travels to that floor.▪ The passenger unloads at their requested floor.▪ An observer counts the passengers as they unload from the floor.▪ A random floor is selected for each passenger on the elevator.▪ The elevator returns to the lobby for more passengers when all the passengers have departed to their requested floors.▪ The observer reports the results to the user.

After perusing the list of possible objects, the elevator was determined to be a primary object. The roles and responsibilities that this abstraction should encompass was then considered. The elevator is responsible for keeping track of which floor it is on, which direction it is traveling, the passengers on board, and the state of operation. Responsibilities such as direction, floor number, and state may seem like overkill for

a simple lobby-to-floor elevator simulation; however, in anticipation of possible reuse or expansion, they are included.

These responsibilities are turned into services so that the user interface or other objects can acquire this information. These services provide the return of current information. Other services needed for the elevator include loading and unloading of passengers. The abstraction of the elevator class is summarized in Table 3.3.

Table 3.3 The abstraction of the elevator class

Name:	Elevator
Responsibilities:	Keep track of the current floor position, the direction and state of operation, and the number of passengers aboard and their destination.
Operations:	load_passengers, unload_passengers, current_passengers current_floor, current_direction, current_state
Attributes:	passengers, floor, direction, state, elevator_id

Instances of this class have a dynamic life cycle, which can be expressed in the state transition diagram shown in Figure 3.1. Here it can be seen that upon initialization, an instance of this class moves to the idle state, where it begins checking for passengers. The elevator loads the maximum passengers, and begin moving up to the lowest floor request. The elevator then stops to unload the passenger(s). If there are more passengers to unload, the elevator goes to the moving up state. If there are no more passengers in the elevator, it begins to move down until it reaches the lobby. When the elevator reaches the lobby, it goes to the load passengers state.

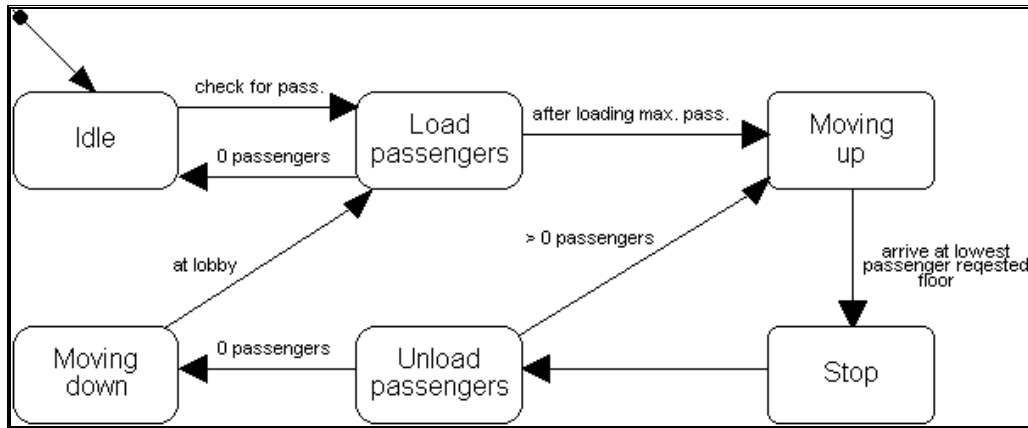


Figure 3.1 Elevator life cycle.

After looking at the elevator object, it was determined that the passenger portion may possess some complexity and would be a good candidate for a subclass. Its primary responsibility would be to keep track of the destination floor of the passenger. The abstraction of the passenger class is summarized in Table 3.4.

Table 3.4 The abstraction of the passenger class

Name:	Passenger
Responsibilities:	Keep track of the destination floor.
Operations:	Current destination
Attributes:	Destination floor

One of the few requirements for the simulator is a user interface. A likely place to handle screen output and input would be from the observer point of view. An observer class could gather statistics from each elevator and display them on the screen, and process any operator requests. It would play the role of a SCADA (Supervisory Control and Data Acquisition) unit. The operator request menus and the elevator simulation action display should be separate from the collector of the

simulation statistics (the observer). The roles and responsibilities of the observer abstraction was then considered. The observer was responsible for keeping track of passengers leaving a floor and entering an elevator, and passengers leaving that elevator and entering a floor. It also kept track of which floor the passengers came from. The observer was tied to one elevator, such that it only knows the statistics of that elevator. The abstraction of the observer class is summarized in Table 3.5.

Table 3.5 The abstraction of the observer class

Name:	Observer
Responsibilities:	Keep track of the passengers that enter and exit the elevator.
Operations:	Total_exit_passengers, Total_entry_passengers Passenger_entry, Passenger_exit
Attributes:	Total_Entry_Passengers, Total_Exit_Passengers

Since the operator request menus and the elevator simulation action display are separate from the observer, separate abstractions were defined for each. Another abstraction that was necessary was an interface between the observer and the displays. By prototyping some displays during analysis before determining the interface needs, it was possible to de-couple the simulation from the particular display package that might be used. Figure 3.2 shows the prototype for the simulation display.

By analyzing the elements in Figure 3.2, an abstraction that can be used to display the simulation was derived. Some of the responsibilities include displaying the passengers per hour information, displaying the static building with elevator shafts and floors, displaying the dynamic elevators traveling up and down the shafts, and

displaying passengers entering and exiting the elevator. The abstraction of the draw elevator simulation class is summarized in Table 3.6

Table 3.6 The abstraction of the draw elevator simulation class

Name:	Display_Elevator_Simulation
Responsibilities:	<p>Display the passengers per hour information.</p> <p>Displaying the static building with elevator shafts and floors.</p> <p>Displaying the dynamic elevators traveling up and down the shafts.</p> <p>Displaying passengers entering and exiting the elevator.</p>
Operations:	<p>Display_passenger_per_hour, Display_static_building</p> <p>Display_dynamic_elevator, Display_passenger_entry</p> <p>Display_passenger_exit</p>

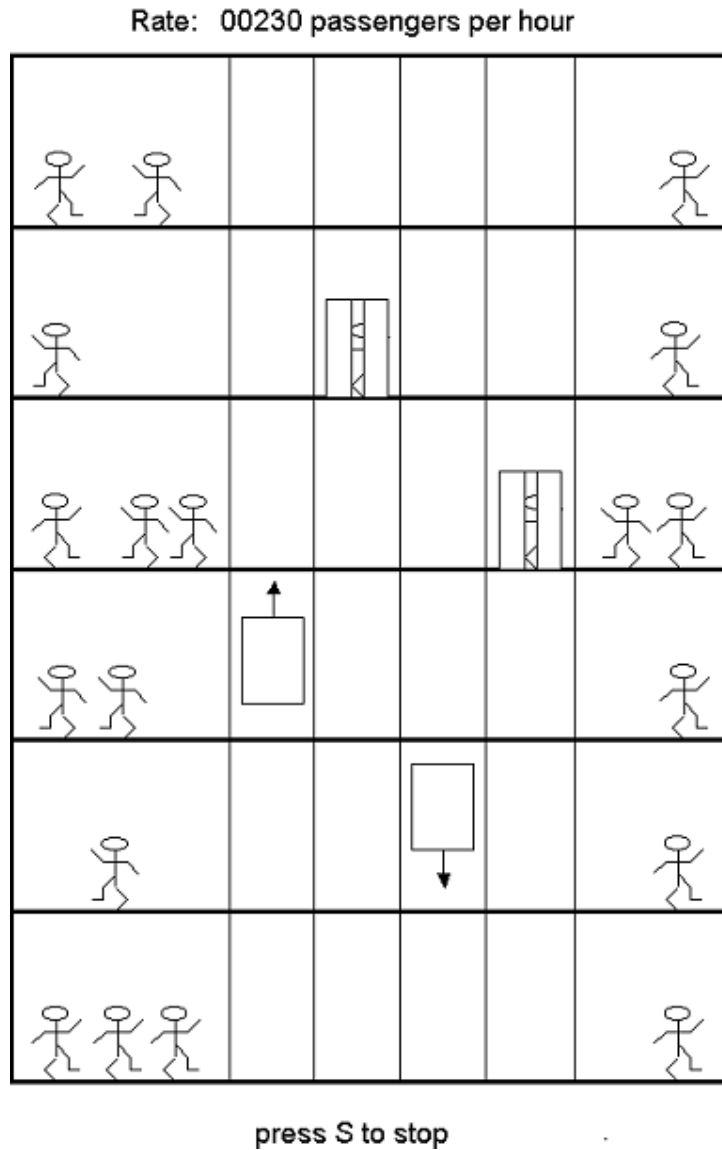


Figure 3.2 Elevator simulator user interface prototype.

The operator request menus are text based or window based. For simplicity, the text based menus were analyzed. Figure 3.3 shows the prototype for the text based menus and prompts display. This abstraction was responsible for displaying the menu and prompting for input. The receipt of input was in the main body of the program. This abstraction of the text menu and prompts class was summarized in Table 3.7.

Table 3.7 The abstraction of the text menu and prompts class

Name:	Menu_and_Prompts
Responsibilities:	Display the menu. Prompt for input.
Operations:	Display_menu Display_number_of_elevators_prompt Display_print_reports_options

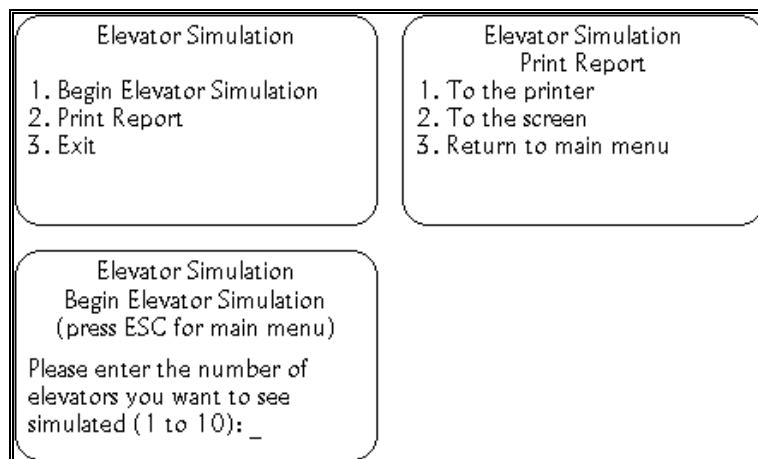


Figure 3.3 Text menu prototypes.

Another requirement for the simulator was the final report on the activities of all the major objects. A reporting class was necessary to print the actual report information during simulation and pass it to the report class, was derived. The report class was responsible for receiving the information and printing the report to the printer or the screen. The report class is summarized in Table 3.8.

Table 3.8 The abstraction of the report class

Name:	Menu_and_Prompts
Responsibilities:	Display the menu. Prompt for input.
Operations:	Display_menu Display_number_of_elevators_prompt Display_print_reports_options

The report contains information about the activities of elevator and passengers. It includes totals from each elevator such as the elevator number, the total passengers, the total time, the flow rate and the total number of passengers who exited at each floor. The report also contains a summary for the building which was the totals of all the elevators. A sample report is shown in Figure 3.4.

<pre> Elevator #1 Total Passengers: 40 Duration: 261 seconds Flow Rate: 551 passengers per hour Floor Passengers 1 2 2 6 3 12 4 10 5 6 6 4 </pre>	<pre> Building Summary Number of Elevators: 1 Total Passengers: 40 Minimum Duration: 261 seconds Maximum Duration: 261 seconds Minimum Flow Rate: 551 pass/hour Maximum Flow Rate: 551 pass/hour Floor Passengers 1 2 2 6 3 12 4 10 5 6 6 4 </pre>
--	--

Figure 3.4 A sample report showing the activities of the major objects.

In order to make the output design flexible, there was a need to de-couple the screen and printer from the report output and from the display output. This required two more classes. These two classes were derived from a super class dedicated to output.

The output class includes such activities as draw text, draw line, draw circle, draw rectangle, set text size, set text style, and set line size. The output class is summarized in Table 3.9.

Table 3.9 The abstraction of the output class

Name:	Output
Responsibilities:	Maintain the commands for output to screen or printer.
Operations (virtual):	Draw_Text, Draw_Line, Draw_Circle, Draw_Rectangle Set_Text_Size, Set_Text_Style, Set_Line_Style, Set_Line_Size
Attributes:	line_style, line_size, text_style, text_size

3.1.3 The architecture of an elevator supervisory group control system

The elevator supervisory group control system was designed for a high rise building with ten populated floors. The output data was sourced from a real existing system and then this data was used to create and simulate the prototype building. The prototype building includes ten floors, served by four shaft elevators, Figure 3.5,

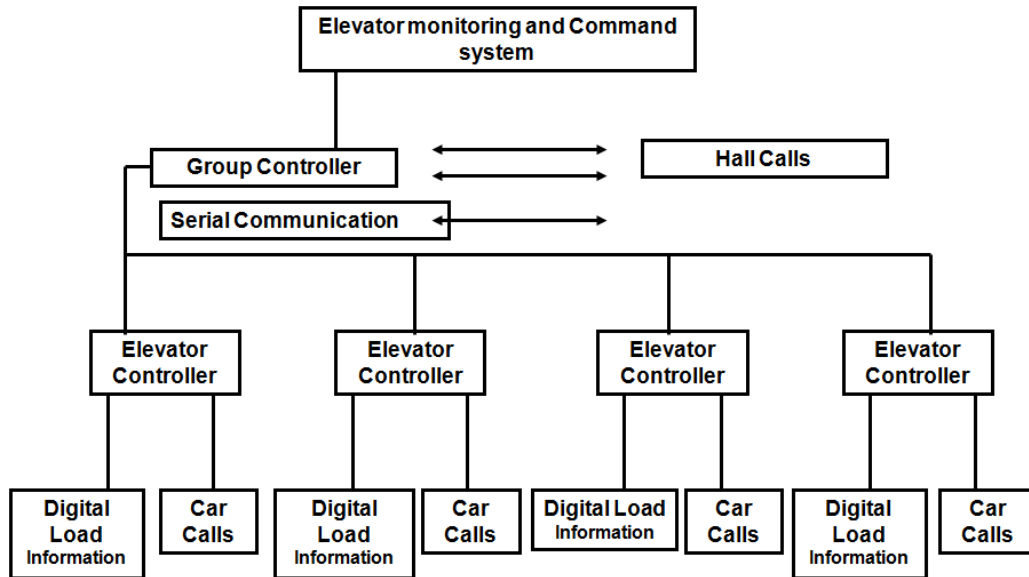


Figure 3.5 The architecture of an elevator supervisory group control system.

The design was done for a high rise building with ten populated floors. The passenger traffic for each elevator at different times and the number of passengers per floor for each elevator was then measured and an average for every fifteen minutes plotted for a 12hr period.

3.1.3.1 Data Preparation

After collecting data from the Elevator Simulator system, the following was done:

1. Data validity checks.

This revealed the unacceptable data that, if retained, would have produced poor results. The average waiting times is 60seconds. A value of 240sec and above was discarded.

2. Partitioning data.

Partitioning is the process of dividing the data into validation sets, training sets, and test sets. By definition, *validation sets* are used to decide the architecture of the network; *training sets* are used to actually update the weights in a network; *test sets* are used to examine the final performance of the network. The primary concerns should be to ensure that: a) the training set contains enough data, and suitable data distribution to adequately demonstrate the properties we wish the network to learn; b) there is no unwarranted similarity between data in different data sets.

3.2 The pattern recognition system

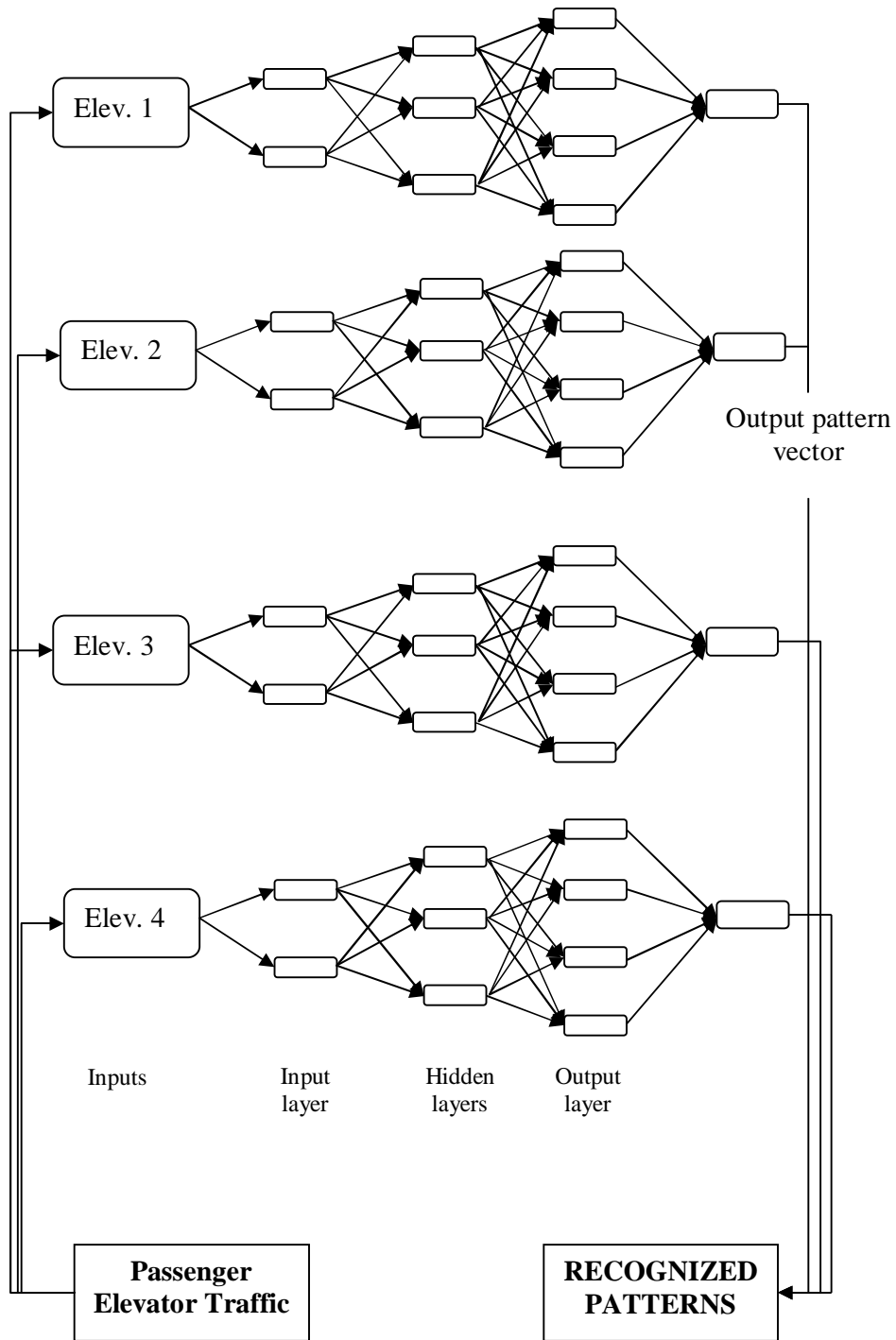


Fig 3.6 A block diagram of the Pattern Recognition System

The signal inputs (waiting times and landing times) from the elevator simulator for the four elevators serving the ten floors into the pattern recognition network (figure 3.6) for a period of three days in real time.

3.3 The ANN training system trained using Backpropagation

3.3.1 Introduction

The proposed system was trained using backpropagation. The ANN system used contains three layers of neurons (the input, the output and the hidden layer). Each layer consists of ten neurons. Ten neurons were chosen as an ideal number after testing the performance of the system. The system was forced to adapt to a set of training vectors (the input and the target vectors). The number of epochs was set to 100, the weights adjusted and the error between the adjusted results in the waiting times per floor and the balanced target output approximated to zero. Figure 3.7 shows the backpropagation (BP) training of the ANN for the system

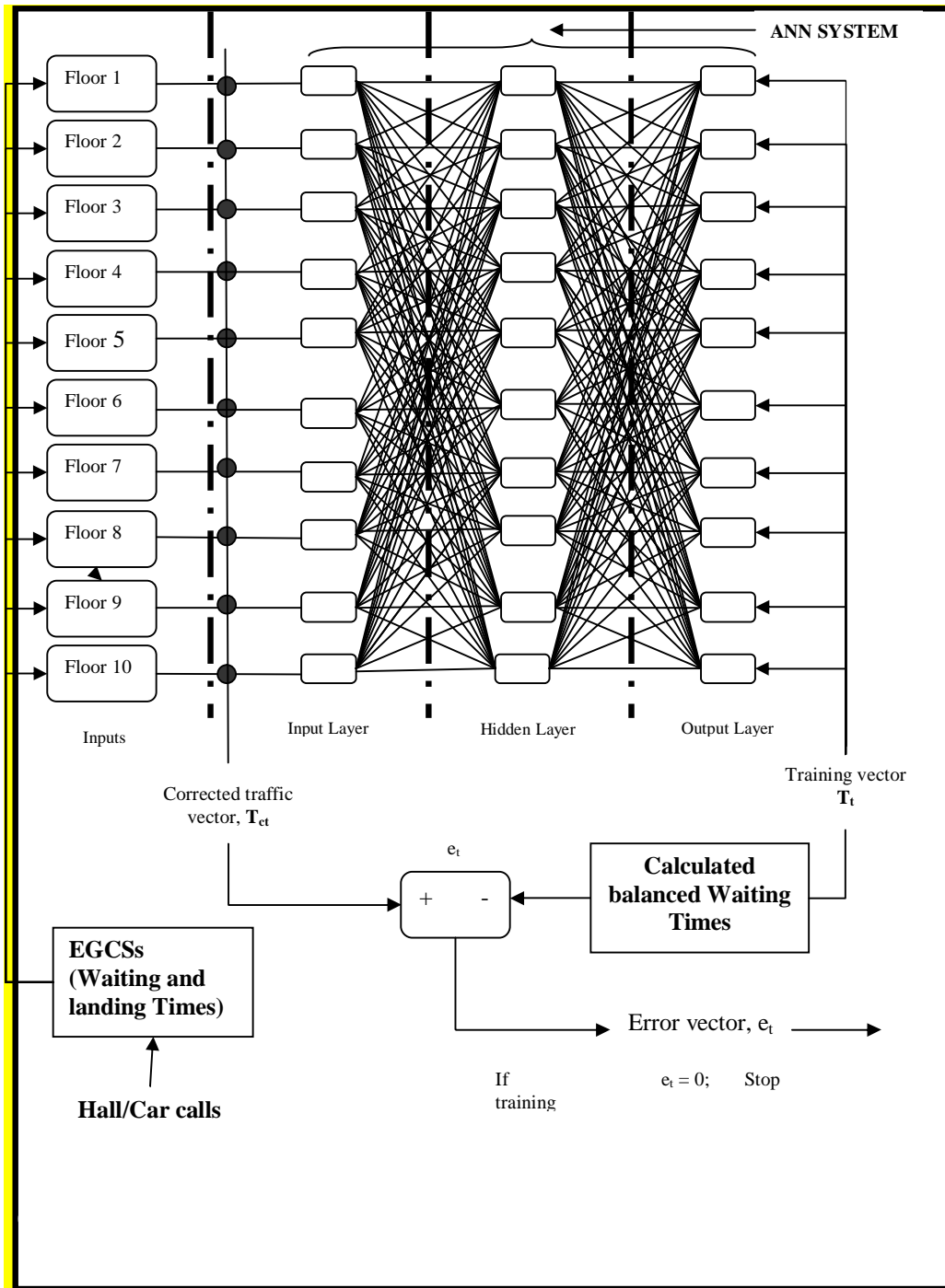


Figure 3.7 The ANN training System

5.4.2 Simulated System by use of the GUI in ANN tools for MATLAB

The waiting times for the passengers on each of the ten floors being served by the four elevators were feed-forwarded through the ten input ten output multilayer backpropagation system by use of the GUI for ANN tools in MATLAB as shown in figure 3.8.

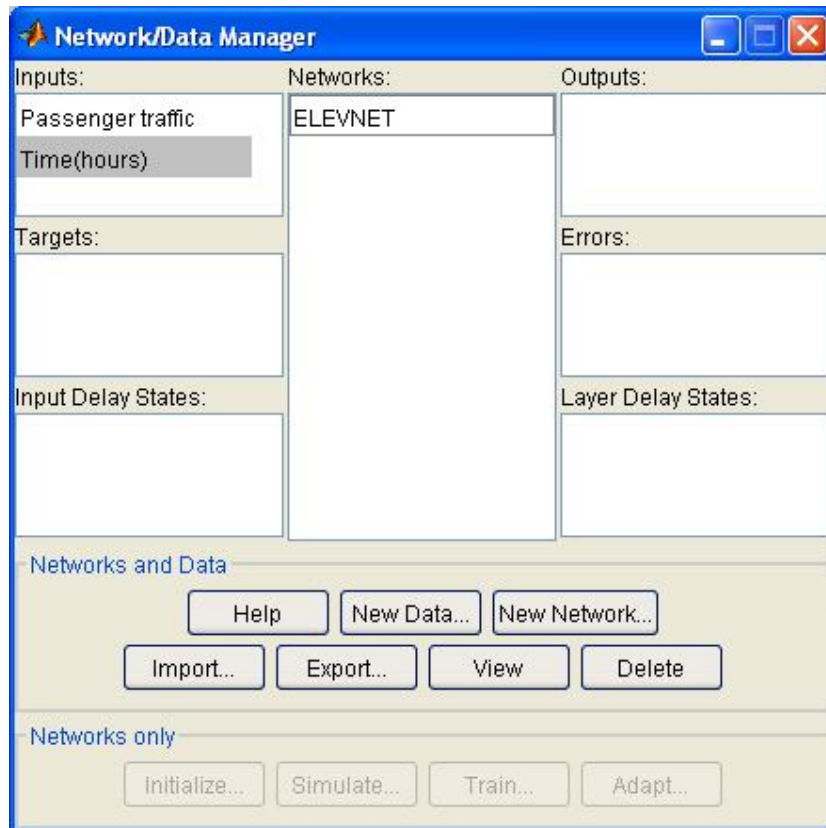


Figure 3.8 Network Data Manager in GUI for ANN tools

Using the Data Manager, the inputs (the waiting times patterns) were specified. They indicated the patterns for all of the ten floors in the building for a 12hr system from the pattern recognition system as shown in figure 3.9. The targets were specified and

the target vector also synchronized with the input vectors for backpropagation training.

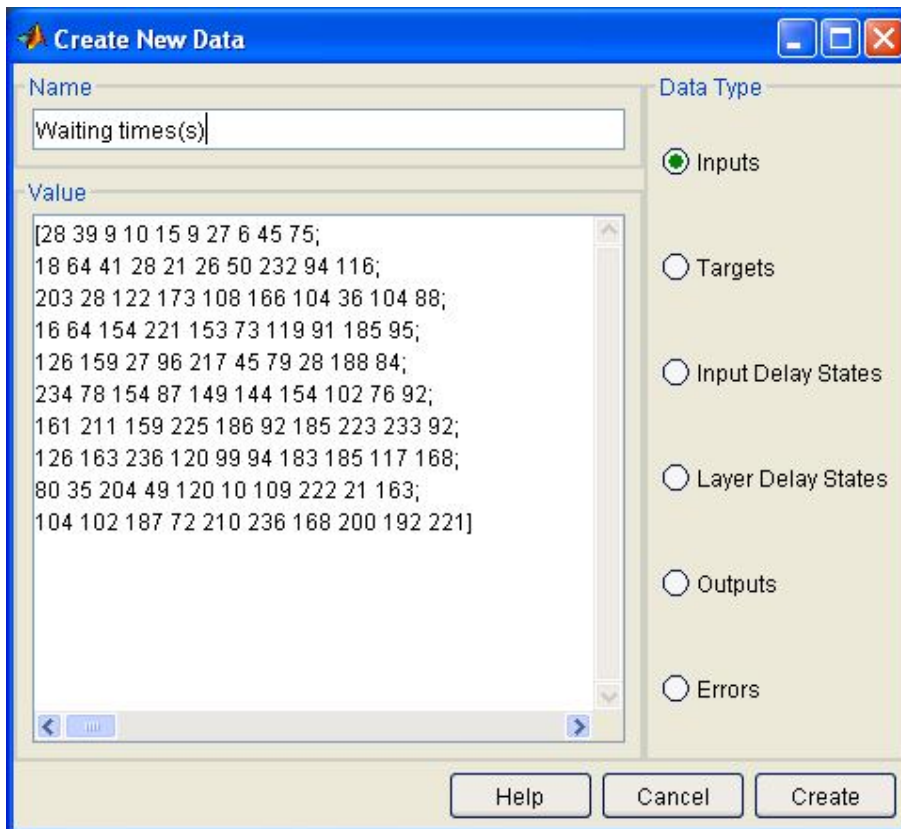


Figure 3.9 Backpropagation system Input Vectors (MatLab 6.5)

A neural network which included both the inputs and the targets was created. The backpropagation training method was specified. The training function to train the net was set to TRAINLN, the adoption learning function was set and three numbers of layers were set, consisting of the input layer, the hidden layer and the output layer as shown in figures 3.10a and 3.10b.

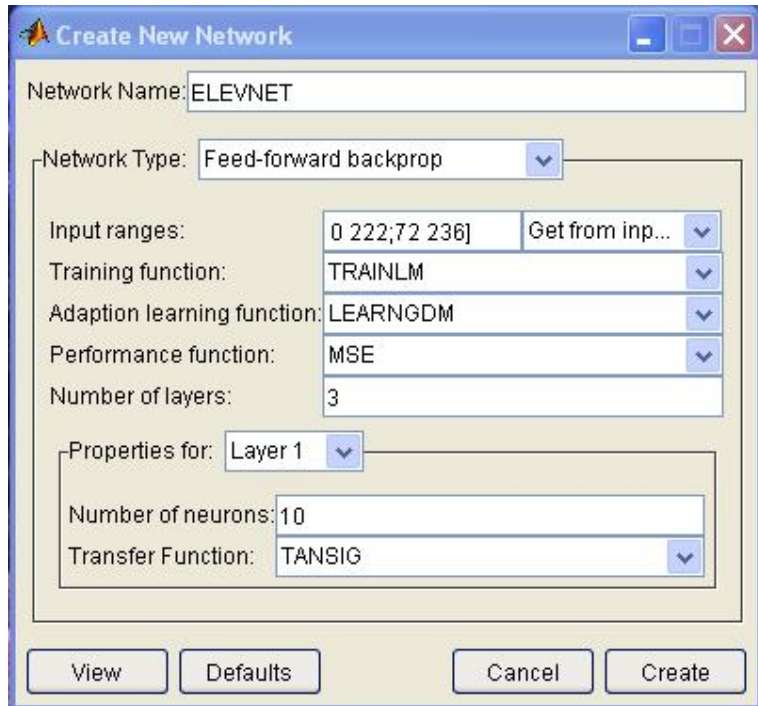


Figure 3.10a The ELEVNET (creation)

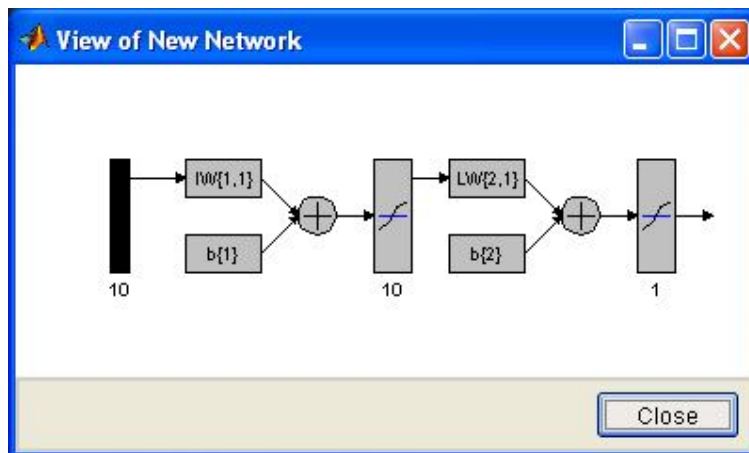


Figure 3.10b The ELEVNET (View)

The training was done for 100 epochs and the system was able to calculate the weights for each neuron interconnection.

CHAPTER FOUR

SIMULATION TESTS, RESULTS AND ANALYSIS.

4.1 Simulated 12hr passenger traffic pattern

The passenger traffic for the four elevators was generated using the elevator simulator and the average passenger traffic was taken after every hour for a 12hr period. The elevator simulator used sourced traffic pattern from an existing building (ten populated floors) to be able to generate passenger traffic patterns for the four elevators for the ten populated floors. The graphs were plotted in four sections of the day to highlight the different passenger traffic patterns in the different times of the day as shown in Figure 4.1 (a...h)

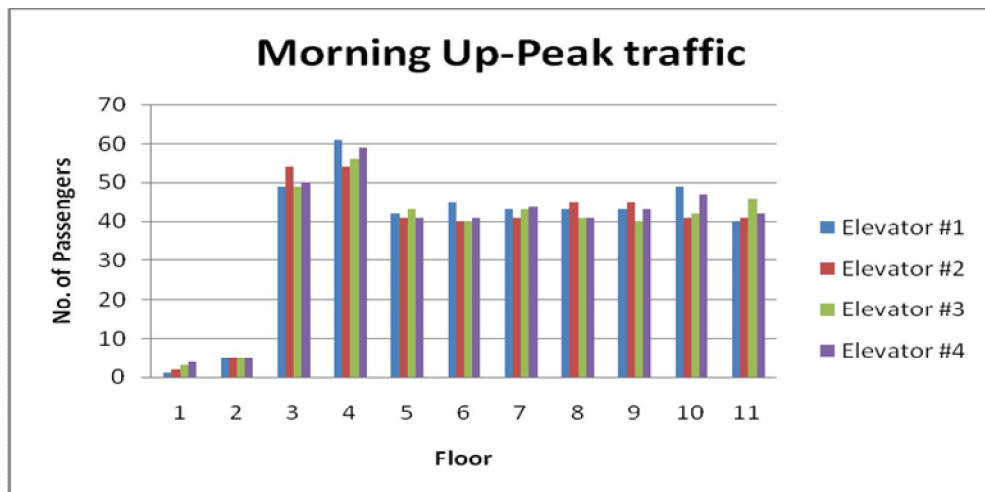


Figure 4.1a. Morning Up-Peak Traffic

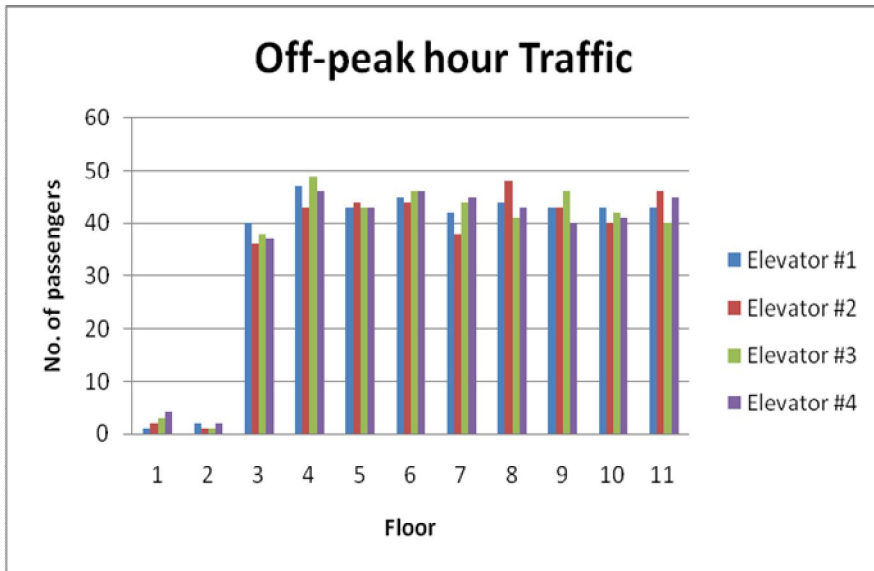


Figure 4.1b. Off Peak Hour Traffic

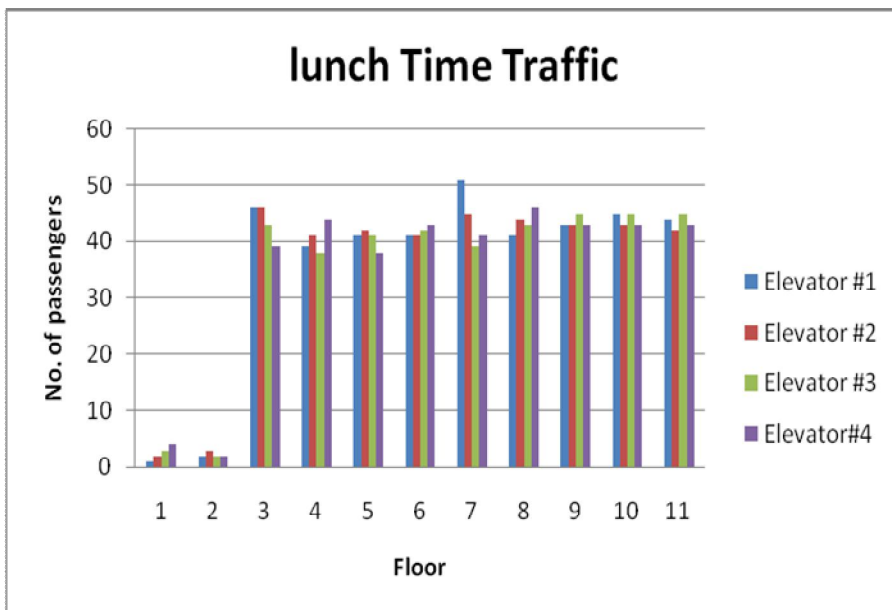


Figure 4.1c. Lunch Time Traffic

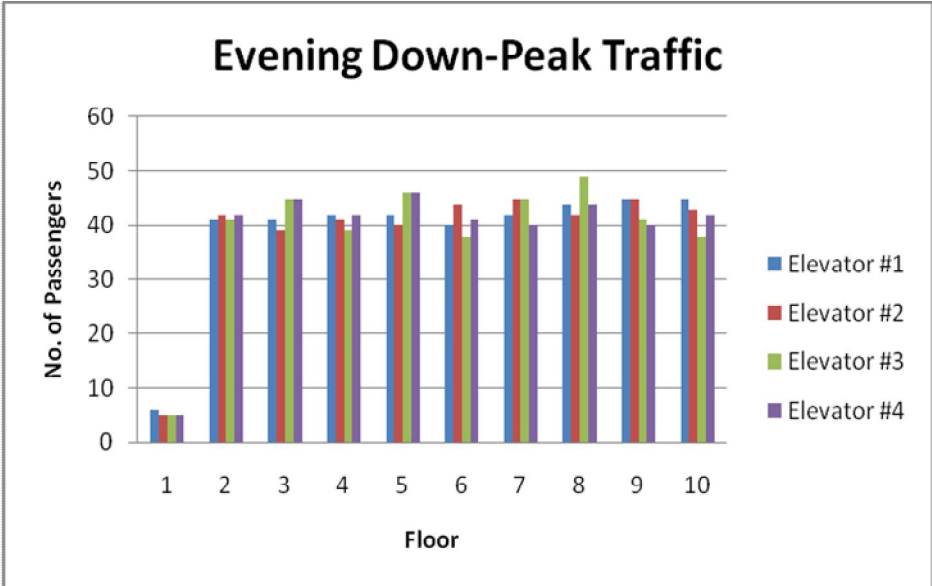


Figure 4.1d. Evening Down Peak Traffic

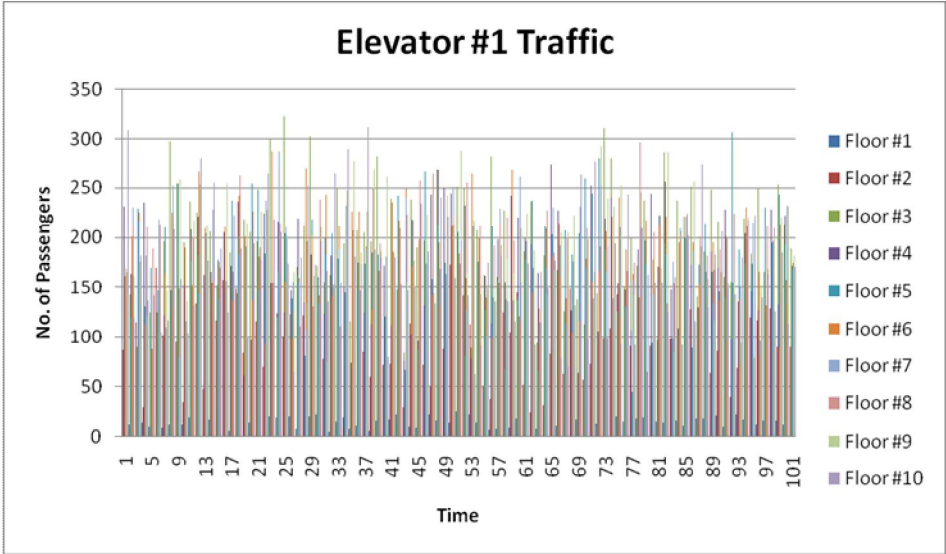


Figure 4.1e. Elevator #1 Traffic

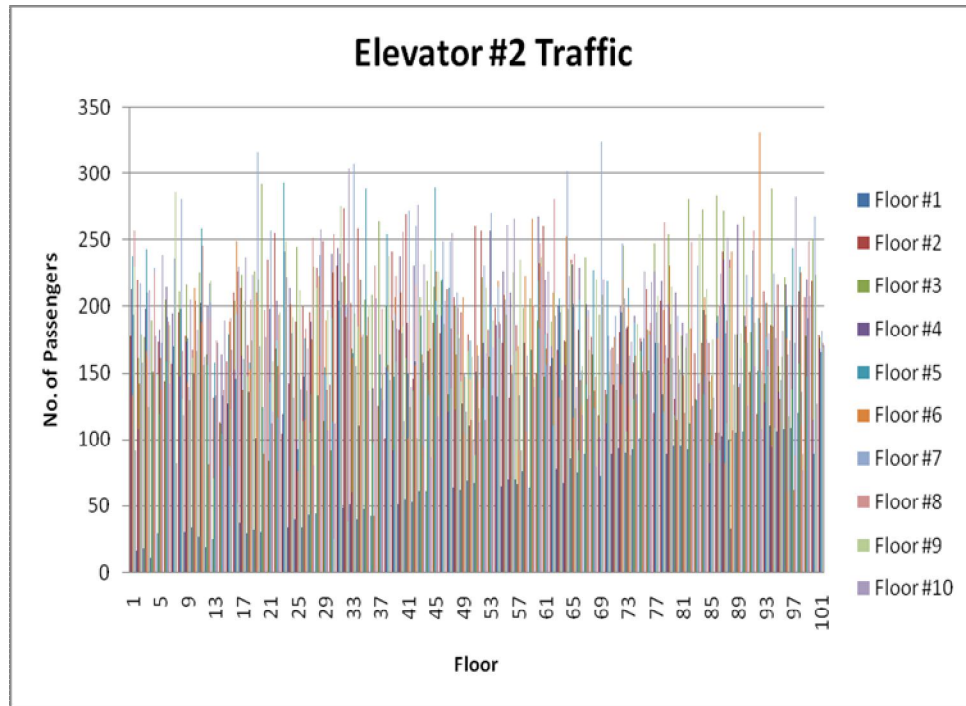


Figure 4.1f. Elevator #2 Traffic

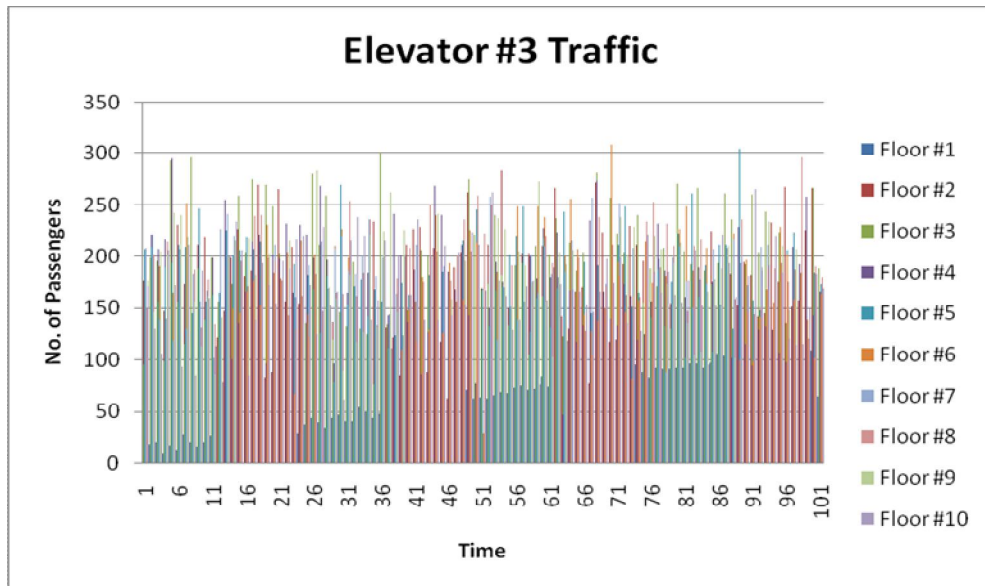


Figure 4.1g. Elevator #3 Traffic

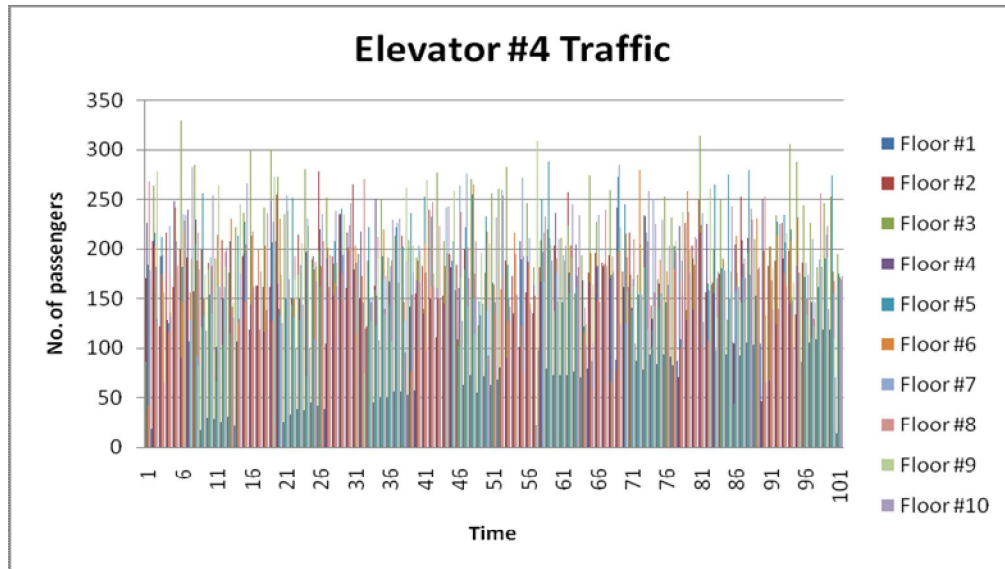


Figure 4.1h. Elevator #4 Traffic

Figure 4.1 Simulated Passenger Elevator traffic patterns

Figures 4.1a.....d, Indicates the traffic patterns for a busy 10 storey building. The traffic patterns do not vary much. During the morning up-peak, people are reporting to the offices and also the Full time students at both Moi University and Kenya Methodist University campuses are reporting for classes. During the off-peak hour, the traffic is also high. The first classes end at 10am or 11am. The cafeterias are located only at ground floor and thus the traffic mainly consists of students.

In addition to the normal lunch time traffic, the second classes are also ending at the same time. Most of the students are leaving the classes for ground floor. At 2pm, the students are reporting to classes and so are the workers, who are returning from lunch.

During evening down peak, the people are leaving the building. At the same time, the evening classes are starting. Thus, there is congestion at the lobby as well as the top floors.

Figures 4.1e.....h

Indicate the individual elevators lifts. The traffic is much more for middle floors (floor 3 to floor 9). Apart from the peak traffic (up and down), there is the inter floor traffic. The campuses occupy more than one floor and thus contribute to inter floor traffic. The tests were done for duration of ten days during daytime only. When the campuses are on recess, the traffic is reduced as indicated in figure 4.3

4.2 Simulated results for the ECGSs from the pattern recognition system

The pattern recognition network came up with simulated results for distinct 12hr passenger elevator traffic patterns for each of the four lifts serving the ten floors as shown in figure 4.2 (a-d). The 12hr system has been divided into four distinct sessions,

7.30am-9 am: The morning Up Peak (from 7.30 to 9.30)

10am-12pm: The off Peak (from 10 to 12)

12.30pm-2.30pm: The Lunch Time (from 12.30 to 2.30)

4.30am-5.30am: The Evening Down Peak (from 4.30 to 5.30)

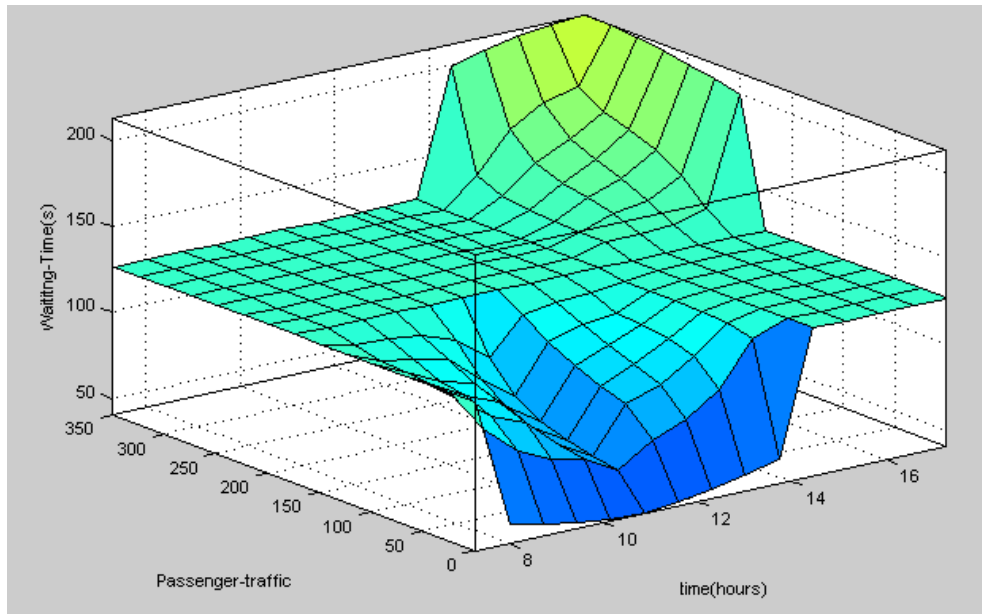


Figure 4.2a Passenger Elevator #1

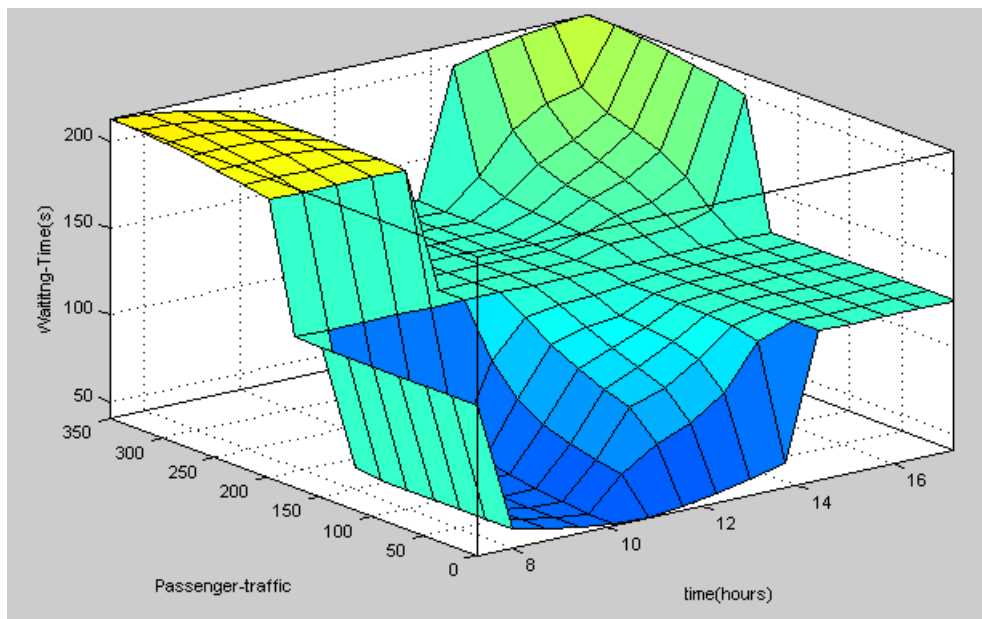


Figure 4.2b Passenger Elevator #2

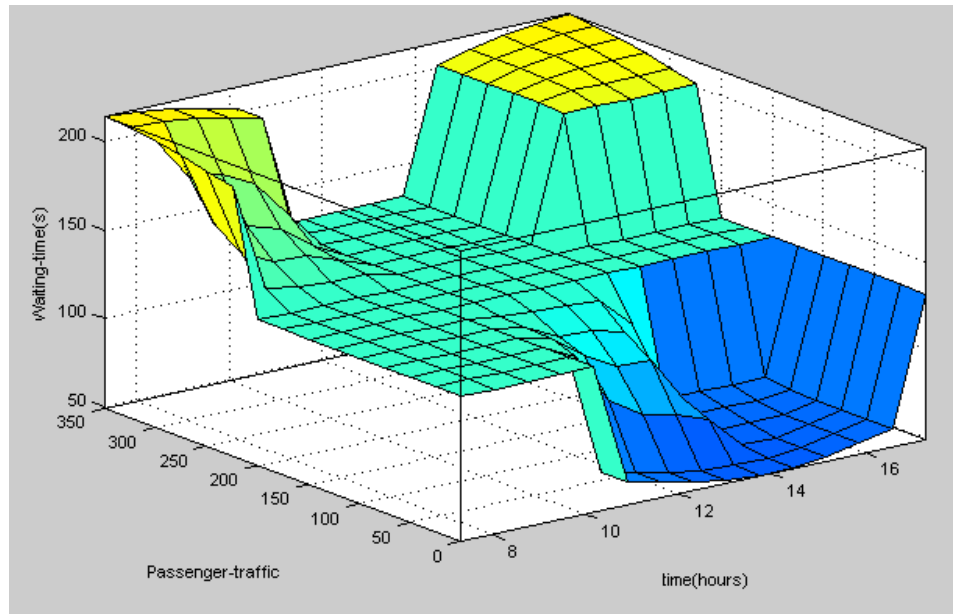


Figure 4.2c Passenger Elevator #3

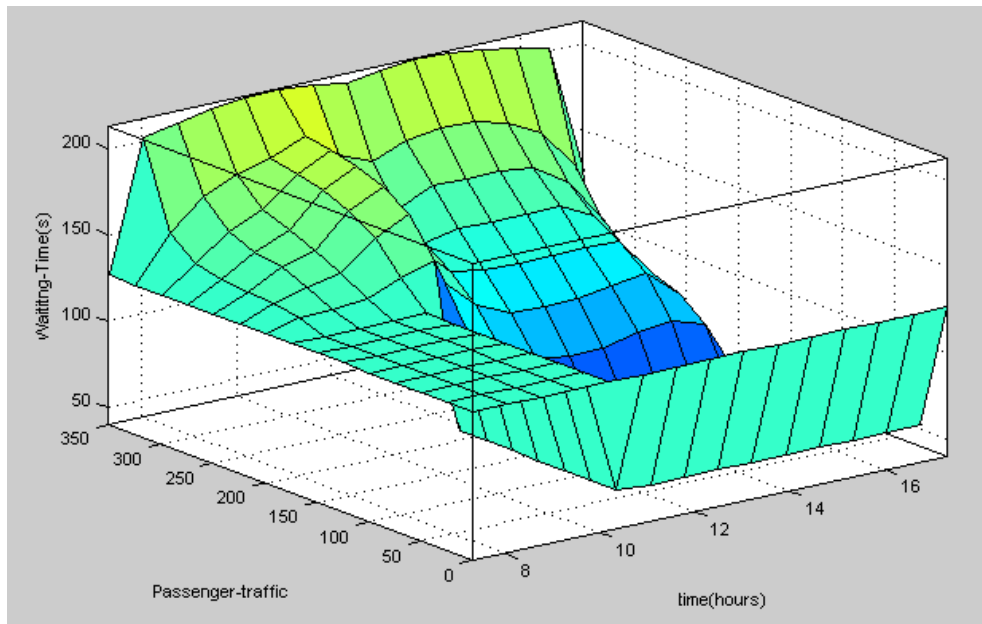


Figure 4.2d Passenger Elevator #4

- As indicated by the figure 4.2a, the waiting times are high during the evening down peak, for high passenger traffic. Only during the off peak time are the

waiting times low. This is attributed to low passenger traffic, coupled with all elevators being functional. Figure 4.2b indicates a typical busy working day where there is a heavy morning up peak and evening down peak. The lunch time traffic is moderate. This results in high waiting times during the peak times. Figure 4.2c indicates almost a similar scenario to figure 4.2b. However the low waiting times are exhibited in the evening for low passenger traffic. Figure 4.2d indicates a less busy day. The waiting times are on the rise as the passenger traffic increases throughout the day.

- Figure 4.2a

During the peak time, the waiting and landing times are high. They are at their highest, when the passenger traffic exceeds 250 passengers/hour.

This floor is configured to serve the floors not occupied by the satellite campuses.

- Figure 4.2b, c

The waiting and landing times are on the increase during the morning up peak and evening down peak times. These two lifts are restricted to serve the floors on which the satellite campuses are located.

- Figure 4.2d

The campuses are on recess and it is a less busy day.

- From passenger elevator traffic patterns from the graphs above it shows that, depending on the time of day and the location of the high rise building, there is slight change in the traffic patterns. The high rise building is located in a densely populated urban area like Nairobi.

- The elevator simulator for the high rise building prototype inputted these patterns into a pattern recognition neural network and three day 12hr iteration was done to get a more definite pattern for the system, and for the neural network to be able to predict the 12hr passenger elevator traffic patterns for each elevator as per floor. Once the pattern recognition system derived the 24hr patterns for the four elevators, the neural network controller was then trained using backpropagation to optimally park and dispatch the four elevators among the ten populated floors in the high rise building.
- A statistical forecast for a typical day in a single tenant office building with common working hours is shown in Figure 4.3. The three passenger traffic components were forecast by the Elevator simulator. According to the figure, traffic intensity is highest in the morning at 8:30 a.m. and during the lunch hour at 12:00 a.m. During the morning up-peak, people arrive at work and it is the most demanding time for the elevator handling capacity. A lot of inter-floor traffic in the morning has also been measured. During the lunch hour there is typically about 40 per cent incoming, 40 per cent outgoing, and 20 per cent inter-floor traffic. The lunch hour traffic is the most demanding for the group control capability since there are a lot of car and landing calls to be served. In the evening, people exit the building, and mostly outgoing traffic is forecast.

4.3 Simulated Results of the ANN training system trained using Backpropagation

The final simulation of both programs was done once the net had been trained, and the code has been written. The random numbers were used to represent the passenger traffic flow, which represented the outgoing traffic. These random inputs were fed initially to the neural network for the network to recognize a unique pattern in each of the ten passenger lifts. The inputs were fed in iterations of 1 hour each for three days so as to get an accurate pattern. These iterations were done in real-time, to mimic the random traffic occurring in any high rise building. Once the patterns were recognized, the inputs to the neural network were forcefully fed by the average of the total output (load) of all the passenger elevator cars to optimally distribute and park the passenger elevator cars. This was done over a number of iterations for the network to be able to optimally park the free passenger elevator cars.

In the following, the effect of optimal parking is compared to the existing parking algorithms. Back propagation was used in the tests. The building has one entrance floor and 10 populated floors. The population distribution is shown in Figure 4.3.

There are six and ten times as many people on the two highest floors as there are on the lower floors. The test is made for an elevator group with five 16-person cars. The speed of the elevators is 2.0 m/s. Outgoing traffic was simulated. The average passenger waiting times with both optimization methods are shown in Figures 4.4a and 4.4b. Figure 4.4a shows the passenger waiting times as a function of the passenger arrival rate. Figure 4.4b shows the landing call times as a function of the passenger arrival rate. The average waiting times are considerably decreased during heavy traffic, but the average landing call times are slightly increased with waiting

time optimization.

The available handling capacity is better utilized as the passenger waiting times are optimized. By optimizing passenger waiting times the average waiting times floor by floor are balanced. Crowded floors with high passenger arrival rates have better service than by optimizing the landing call times. The maximum waiting times are cut at heavily populated floors and the average passenger waiting times become shorter. Waiting time optimization improves passenger waiting times especially in buildings with uneven population distributions.

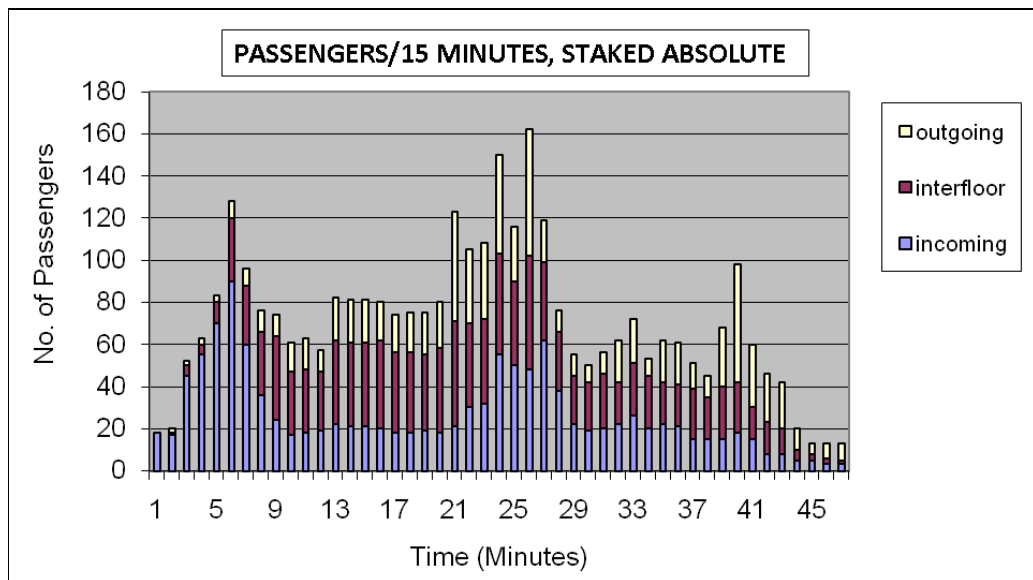


Figure 4.3 Forecast traffic component profile during a day in an office building

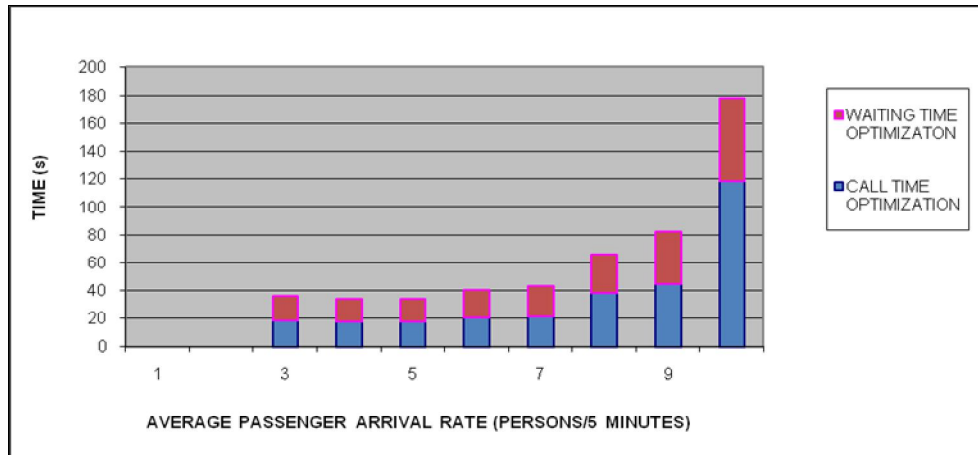


Figure 4.4a Average passenger waiting times as a function of the passenger arrival rate.

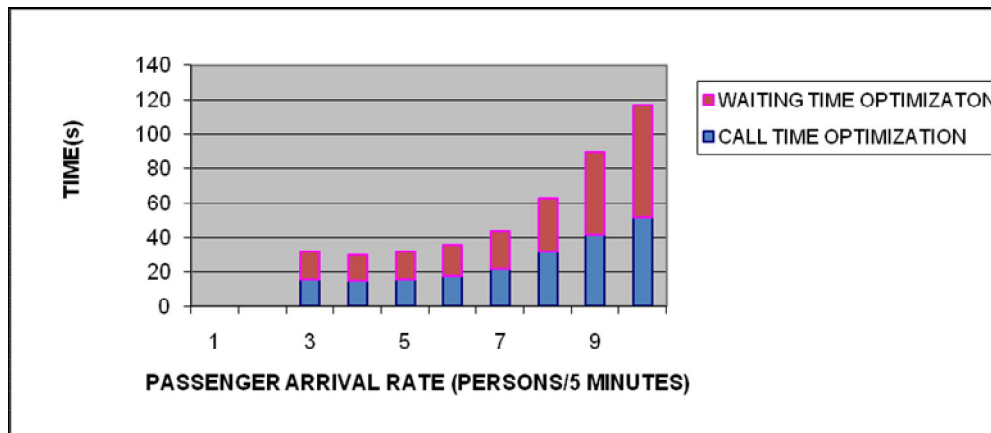


Figure 4.4b Average landing call times as a function of the passenger arrival rate.

4.4 Effect of the optimal parking

The algorithm provides a method and system for optimally parking elevator cars under different patterns of passenger traffic. For the case of down-peak traffic, the cars are distributed equally over the floors of the building so as to minimize the expected waiting time of only the next passenger. This results in immediate savings

in the expected waiting time for low and medium arrival rates. The cars are parked to match the arrival distribution of passengers at the various floors.

Minimizing the expected waiting time of only the first passenger is not sufficient for the case of up-peak traffic, where the main question is how many free cars should be kept at the lobby, given the number of floors and the overall arrival rate of passengers. The proposed solution to the problem of optimal parking for a group of elevators during up-peak traffic is based on the training of the system using Artificial Neural network (ANN) with a small number of states corresponding to candidate parking locations, and a dynamic programming process for minimizing the expected waiting time of future passengers for longer, but still limited time intervals,

Figure 4.5.

According to Figure 4.5a and b, which indicates the simulated service level parameters in a building with one entrance floor during outgoing and incoming traffic intensity, the average waiting time stays below the interval until the passenger arrival rate reaches the handling capacity? In Figure 4.5b, the average interval, average passenger waiting time and average hall call time are shown for the outgoing traffic. The interval for outgoing traffic is shorter than for incoming traffic since the group control can limit the number of stops during outgoing traffic. According to the figures, the average waiting time during up-peak and down peak hours is the same. More passengers are transported during lunch time than during up or down peak times.

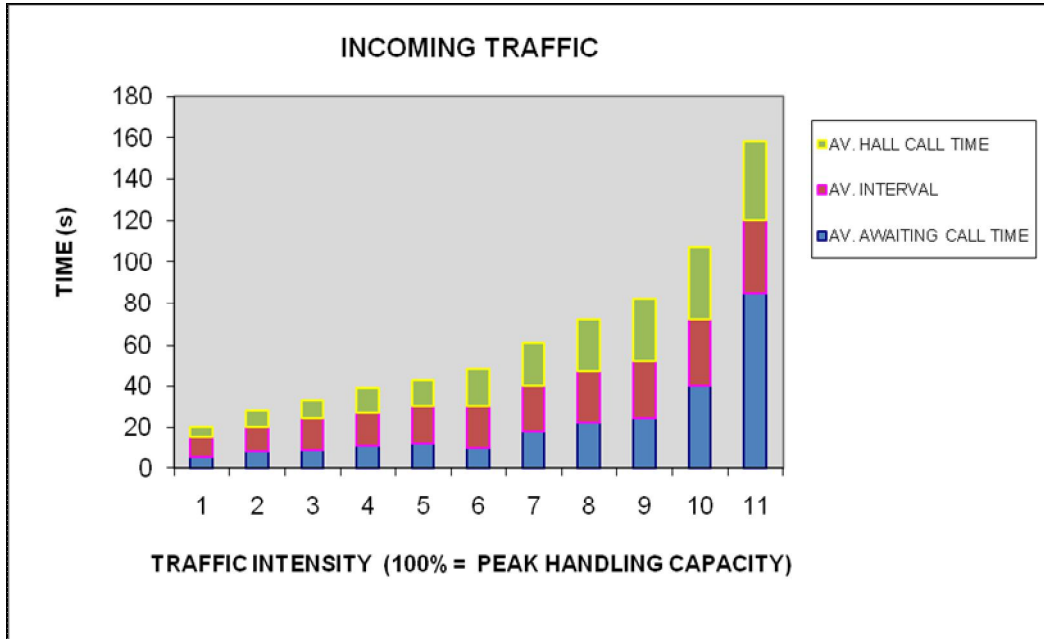


Figure 4.5a, Incoming Traffic

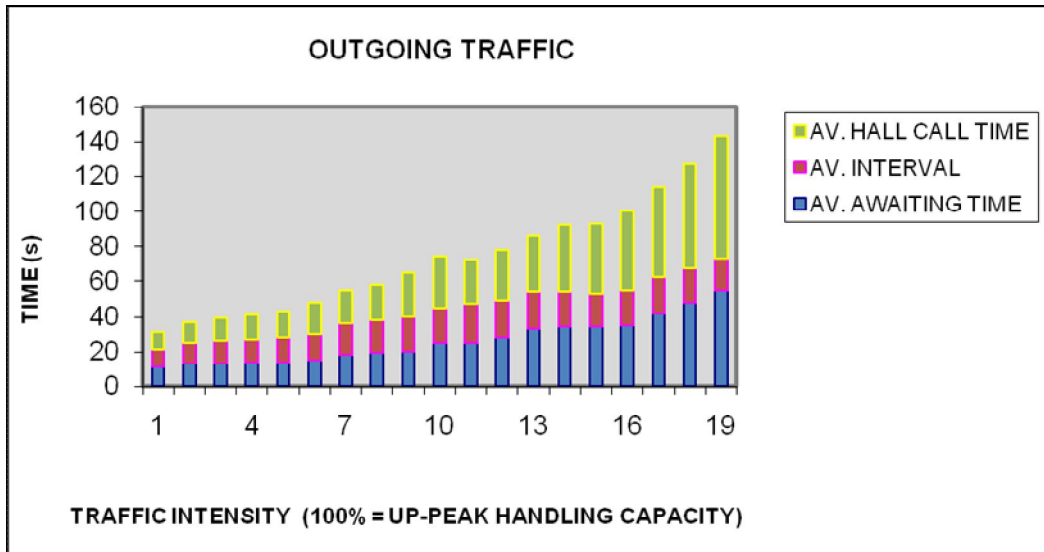


Figure 4.5b Outgoing Traffic

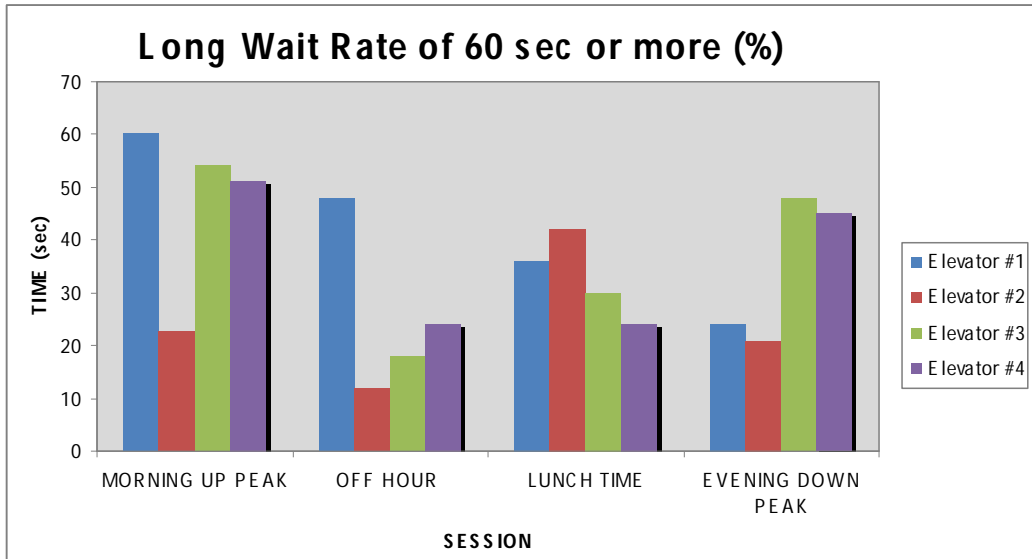


Figure 4.5c, Long Wait Rate (before optimal parking)

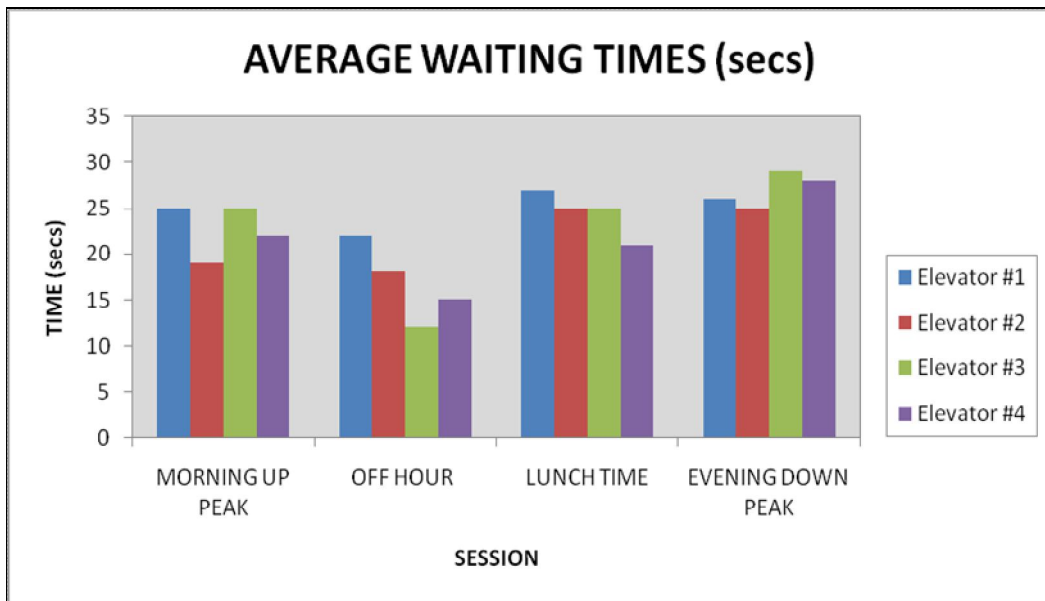


Figure 4.5d Average Waiting Times (after optimal parking)

CHAPTER FIVE

Conclusion

- Passenger waiting times and ride times inside the car are optimized according to the observed passenger arrival rates at each floor and in each direction from the last five minutes.
- The number of waiting passengers behind each call is estimated. By optimizing the parking algorithms, the average waiting times become more balanced floor by floor.
- Crowded floors with high passenger arrival rates are better served using ANN developed method than using basic parking or dynamic assignment.
- The maximum waiting times are cut at heavily populated floors and the average passenger waiting times reduced from about 220s to 60s .
- Optimization of parking of free elevator cars improves the service especially in buildings with unequal passenger arrival rates at different floors and directions.
- The landing call times are slightly decreased but during heavy traffic they can be a little increased.
- The parking algorithm adapts to the prevailing traffic pattern. Control actions, such as returning cars automatically to busy traffic floors, or parking cars during light traffic, follow from the forecast traffic pattern.
- Artificial Neural Network is applied in recognizing the prevailing traffic patterns according to the forecast traffic component and passenger arrival rates. Passenger arrival rates at and exiting rates from each floor and in each

direction are forecast for each time period. Statistical forecasts of the passenger traffic are made in fifteen minute and one hour periods for a typical day, or separately for every weekday.

- Contrary to conventional controls, the peak traffic periods are predicted in advance. Before implementing a forecast traffic pattern, the validity of the forecast is confirmed. If the forecast is in conflict with the short-term statistics, the forecast is not applied in the control and parking of the group elevators. The group control decisions can be further improved by utilizing the statistical forecasts more.
- The reservations of landing calls to cars can be fixed at an earlier stage if the future events are simulated more accurately during the call allocation. Passengers can then be informed earlier about the arriving car, which shortens the psychological waiting time. The number of optimization targets can be increased.
- All the optimization targets cannot be reached simultaneously since they often are in conflict with each other. The optimization targets can be switched according to the forecast traffic pattern. For example, during the up-peak the optimization target could be to increase handling capacity and decrease journey time, and during the down peak to balance car load and to minimize passenger waiting times. The optimization targets should be selected so that they have the greatest positive influence on the defined cost and on the overall performance of the elevator group.
- If the system were to expand and control real elevators, or if the simulator were to grow to encompass more than just a PC, the observer class could be

replaced with a commercial-off-the-shelf SCADA package without affecting the functionality of the simulator or system.

REFERENCES

1. C. B. Kim, K. A. Seong, and H. Lee-Kwang (1993); Fuzzy approach to elevator group control system, in Proc. 5th IFSA Congr. vol. 2, pp. 1218–1221.
2. IEEE Trans. Syst., Man, Cybern (1995); A fuzzy approach to elevator group control system; vol. 25, pp. 985–990.
3. R. D. Peters (1992); The theory and practice of general analysis lift calculations, in Elevcon Proc., pp. 197–206.
4. John McDermott, Ed., Milan, Italy and M. J. Schoppers (1987); Universal plans for reactive robots in unpredictable environments, in Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Morgan Kaufmann publishers Inc. an Mateo, CA, USA; pp. 1039.1046
5. Siikonen and J. Leppala (1991); Elevator traffic pattern recognition, in Proc. 4th IFSA Congr, pp. 195–198; M. L.
6. Mitsubishi, Ltd. Japan (1992); The fuzzy elevator group supervisory system, Patent 2-52875.
7. Mitsubishi, Ltd. Japan (1992); Group supervisory system of elevator cars based on neural network, Patent 4-32472.
9. Dimitri P. Bertsekas (2000); Dynamic Programming and Optimal Control, Athena Scientific, Belmont, Massachusetts, Volumes 1 and 2.
10. G. C. Barney and S. M. dos Santos (1985); Elevator Traffic Analysis, Design, and Control. Stevenage, U.K.: Peregrinus,

11. Daniel Nikovski and Matthew Brand (2003); Decision-theoretic group elevator scheduling, in 13th International Conference on Automated Planning and Scheduling, Trento, Italy, pp. 133.142, AAI
12. G.C. Barney (2003); Elevator Traffic Handbook, Spon Press, London.
13. Gang Bao, Christos G. Cassandras, Theodore E. Djaferis, Asif D. Gandhi, and Douglas P. Looze, Department of Electrical and Computer Engineering, Amherst, Massachusetts (1994); Elevator dispatchers for down- peak traffic, Technical report, University of Massachusetts.
14. David L. Pepyne and Christos G. Cassandras (1997); Optimal dispatching control for elevator systems during up peak traffic, IEEE transactions on control systems technology, vol. 5, no. 6, pp. 629.643.
15. Matthew Brand and Daniel Nikovski (2003); Optimal parking of elevator cars in supervisory group control, in preparation for IEEE Transactions on Systems, Man, and Cybernetics, Part B.
16. Carpenter, G., and S. Grossberg (1988); The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network. IEEE Computer 21: 77-88.
17. Fausett L (1994); Fundamentals of Neural Networks, Prentice Hall.
18. Marko, K., J. Dossdall, and J. Murphy (1990); Automotive control system diagnosis using neural nets for rapid pattern classification of large data sets. In Proceedings of the International Joint Conference on Neural Networks Piscataway, NJ: IEEE Service Center.
19. Russell I. (1991); Self-organization and adaptive resonance theory networks. In Proceedings of the Fourth Annual Neural Networks and Parallel

Processing Systems Conference, Indianapolis, IN: Indiana University-Purdue University,

- 20 Prof Leslie Smith (2003); An Introduction to Neural Networks, Center for Cognitive and Computational Neuroscience, University of Stirling.
21. Ingrid F. Russell (2000); Neural Networks, Department Of Computer Science, University of Hartford.
22. Reading MA (1991); Neural Networks: Addison-Wesley.
23. Freeman, J., and D. Skapura (1997); An Introduction to Neural Networks, UCL Press; Gurney K.
24. AITT (2008); Artificial Neural Networks, Artificial technologies guide.
25. Anand Venkataraman (1999); Introduction to Backpropagation, IEEE Service Center.

APPENDICES

Appendix A

Program Codes

A.1 Elevator Simulator Code

```
// Module 1

// Technical
// Description: This module is the main module.
//*****

#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <dos.h>

#include "main.h"
#include "elevsupp.h"
#include "elevsim.h"

elevsim::elevsim()
{
    int index;
    for (index=0;index<MAX_ELEVATORS;index++)
    {
        elevator_rack[index] = NULL;
        // initialize at something other than the
        // lobby
        last_floor[index] = TOP_FLOOR;
    }
    for (index=0;index<MAX_FLOORS;index++)
    {
        floor_passengers[index] = 0;
    }

    // initialize at something other than zero

    last_max_flow_rate =
    last_min_flow_rate = 1;

    // initialize the rest
```

```

min_flow_rate = 0;
max_flow_rate = 0;
total_passengers = 0;
min_duration = 0;
max_duration = 0;
}
elevsim::~~elevsim()
{
    int index;
    for (index=0;index<MAX_ELEVATORS;index++)
    {
        if (elevator_rack[index] != NULL)
            delete elevator_rack[index];
        elevator_rack[index] = NULL;
    }
}
void elevsim::run(void)
{
    int number_of_elevators = 1;
    int report_option = MENU_PRINT_TO_SCREEN;
    int menu_choice = MENU_BEGIN_SIMULATION;
    int floor_index = 0;
    int index = 0;
    unsigned short floor_num = 0; /* floor index */
    unsigned long *floor_array = NULL;
    unsigned short elevator_state = IDLE_STATE;

    while (TRUE)
    {
        menu_choice = menu.display_menu();

        switch (menu_choice)
        {
            case MENU_BEGIN_SIMULATION:
                // clean out rack - in case simulation is run more than
                once
                for (index=0;index<MAX_ELEVATORS;index++)
                {
                    if (elevator_rack[index] != NULL)
                        delete elevator_rack[index];
                    elevator_rack[index] = NULL;
                }

                number_of_elevators = menu.display_number_of_elevators_prompt();
                if (number_of_elevators != 0)

```

```

{
                                                    /* create the elevators */
for (index=0;index<MAX_ELEVATORS;index++)
{
    if (index < number_of_elevators)
        elevator_rack[index] = new elevator(index+1);
    else
        elevator_rack[index] = NULL;
}

                                                    /* set up graphics screen */
clrscr();
disp_out.display_static_building(number_of_elevators);

                                                    /* run until the keyboard is hit */
fflush(stdin);
while (!kbhit())
{
    min_duration = 0;
    max_duration = 0;
    total_passengers = 0;
    for (index=0;index<MAX_ELEVATORS;index++)
    {
        if (elevator_rack[index] != NULL)
        {
            elevator_rack[index]->run();
                                                    /* find the smallest duration of all
                                                    elevators */
            if (min_duration == 0) // first time
                min_duration = elevator_rack[index]->current_duration();
            else if (min_duration > elevator_rack[index]->current_duration())
                min_duration = elevator_rack[index]->current_duration();
                                                    /* find the largest duration of all elevators
                                                    */
            if (max_duration == 0) // first time
                max_duration = elevator_rack[index]->current_duration();
            else if (max_duration < elevator_rack[index]->current_duration())
                max_duration = elevator_rack[index]->current_duration();
            total_passengers += elevator_rack[index]->current_total_passenger_count();
            floor_num = elevator_rack[index]->current_floor();
            elevator_state = elevator_rack[index]->current_state();
                                                    /* draw passengers entering */
            if (elevator_state == LOAD_STATE)
                disp_out.display_passenger_entry();
        }
    }
}
}

```

```

/* draw passengers exiting */
if (elevator_state == UNLOAD_STATE)
    disp_out.display_passenger_exit(floor_num);

/* draw the elevator at the appropriate floor
*/
if (floor_num != last_floor[index])
{
    /* clear out old floor */
    disp_out.display_dynamic_elevator(index,last_floor[index],FALSE);
    /* redraw new floor */
    disp_out.display_dynamic_elevator(index,floor_num,TRUE);
    last_floor[index] = floor_num;
} /* end of floor change */
} /* end of valid elevator */
} /* end of elevator index loop */
/* NOTE: since total pass is constant, max flow uses min
duration
and vice versa. */
min_flow_rate = flow_rate_from_secs(max_duration,total_passengers);
max_flow_rate = flow_rate_from_secs(min_duration,total_passengers);
if (max_flow_rate != last_max_flow_rate)
{
    disp_out.display_passenger_per_hour(max_flow_rate);
    last_max_flow_rate = max_flow_rate;
}
// optional delay here - allows for better display on
fast cpu
delay(125);
} /* end of no keyboard hit */
(void)getch(); /* gather the keyboard hit
} /* end of valid number of elevators
*/
break;
case MENU_PRINT_REPORTS:
    report_option = menu.display_print_reports_options();
    switch(report_option)
    {
        case MENU_PRINT_TO_SCREEN:
            /* initialize the floor count array
for(floor_index=0;floor_index<MAX_FLOORS;floor_index++)
{
    floor_passengers[floor_index] = 0;
}

```

```

        clrscr();

        for (index=0;index<MAX_ELEVATORS;index++)
        {
            if (elevator_rack[index] != NULL)
            {
                sreport.print_elevator_report(
                    elevator_rack[index]->current_id(),
                    elevator_rack[index]-
>current_total_passenger_count(),
                    elevator_rack[index]->current_duration(),
                    elevator_rack[index]-
>passenger_flow_rate(),
                    elevator_rack[index]-
>pass_on_floor_array()
                );
                fflush(stdin);
                (void)getch();
                clrscr();

                /* summarize the data */
                // totalize the floor breakdown
                floor_array = elevator_rack[index]-
>pass_on_floor_array();

                for(floor_index=0;floor_index<MAX_FLOORS;floor_index++)
                {
                    floor_passengers[floor_index]+=
floor_array[floor_index];
                }
            }
        }
        /* end of valid elevator */
        /* end of elevator loop */
        sreport.print_summary_report(number_of_elevators,
            total_passengers,min_duration,max_duration,
            min_flow_rate,max_flow_rate,floor_passengers);

        fflush(stdin);
        (void)getch();
        clrscr();
        break;
    case MENU_PRINT_TO_PRINTER:
        /* initialize the floor count array
        for(floor_index=0;floor_index<MAX_FLOORS;floor_index++)
        {
            floor_passengers[floor_index] = 0;

```

```

    }
    for (index=0;index<MAX_ELEVATORS;index++)
    {
        if (elevator_rack[index] != NULL)
        {
            preport.print_elevator_report(
                elevator_rack[index]->current_id(),
                elevator_rack[index]->current_total_passenger_count(),
                elevator_rack[index]->current_duration(),
                elevator_rack[index]->passenger_flow_rate(),
                elevator_rack[index]->pass_on_floor_array()
            );

            /* summarize the data */

            // totalize the floor breakdown
            floor_array = elevator_rack[index]->pass_on_floor_array();
            for(floor_index=0;floor_index<MAX_FLOORS;floor_index++)
            {
                floor_passengers[floor_index] += floor_array[floor_index];
            }
        }
        /* end of valid elevator */
    }
    /* end of elevator loop */
    preport.print_summary_report(number_of_elevators,
        total_passengers,min_duration,max_duration,
        min_flow_rate,max_flow_rate,floor_passengers);
    break;
default:
    break;
}
break;
case MENU_EXIT:
default:
    return;
}
/* end of menu choices */
}
/* end of forever loop */
}
// end of run

```

// Module 2

```

// Technical Description: This class is responsible for the user display.
//*****
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <dos.h>

```



```

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#include "main.h"
#include "display.h"
#include "output.h"

display_graphics::display_graphics()
{
    old_flow[0] = 0;
    floor_height = ((VIDEO_MAX_Y/2)/MAX_FLOORS);
    floor_width = (VIDEO_MAX_X/3);
    shaft_width = (VIDEO_MAX_X/32);
    building_x_start = (VIDEO_MAX_X/3); /* left side of screen */
    building_y_start = (VIDEO_MAX_Y/6); /* top of screen */
}

void display_graphics::display_passenger_per_hour(long passperhour)
{
    char string[80];
    int text_x = {(int)((long)VIDEO_MAX_X * 100L/160L)}; // quicker than /1.6
    int text_y = {(VIDEO_MAX_Y/12)};

    disp_out.Set_Text_Size(1);
    disp_out.Set_Text_Color(BLACK);
    disp_out.Draw_Text(text_x,text_y,old_flow);

    sprintf(string,"%05ld",passperhour);
    disp_out.Set_Text_Size(1);
    disp_out.Set_Text_Color(WHITE);
    disp_out.Draw_Text(text_x,text_y,string);
    strcpy(old_flow,string);
}

void display_graphics::display_static_building(int number_of_elevators)
{
    int floor_index;
    int elev_index;
    int floor_left,floor_right,floor_top,floor_bottom;
    int shaft_line;
    int x_start; /* left side of screen */
    int y_start; /* top of screen */
    char string[80];
    int text_x = {(VIDEO_MAX_X/4)};

```

```

int text_y = {0};

x_start = building_x_start;
y_start = building_y_start;

disp_out.Set_Text_Size(4);
disp_out.Set_Text_Color(LIGHTGREEN);
sprintf(string,"ELEVATOR SIMULATION");
disp_out.Draw_Text(text_x,text_y,string);

for(floor_index=0;floor_index<MAX_FLOORS;floor_index++)
{
    /* draw the floor of the building */
    floor_left = x_start;
    floor_right = x_start + floor_width;
    floor_top = (floor_index*floor_height) + y_start;
    floor_bottom = (floor_index*floor_height) + y_start + floor_height;
    disp_out.Draw_Rectangle(floor_left,floor_top,floor_right,floor_bottom);

    /* draw the elevator shafts */
    for(elev_index=0;elev_index<number_of_elevators;elev_index++)
    {
        shaft_line = x_start + (elev_index*shaft_width) + shaft_width;
        disp_out.Draw_Line(shaft_line,floor_top,shaft_line,floor_bottom);
    }
    /* end of elevator loop */
}
/* end of floor loop */

// passenger per hour heading

text_x = (VIDEO_MAX_X/4);
text_y = (VIDEO_MAX_Y/12);
sprintf(string,"Passengers per hour: ");
disp_out.Set_Text_Size(1);
disp_out.Set_Text_Color(WHITE);
disp_out.Draw_Text(text_x,text_y,string);

// show escape sequence

text_x = (VIDEO_MAX_X/4);
text_y = ((MAX_FLOORS*floor_height)+y_start);// just below the building
sprintf(string,"Press any key to return to menu");
disp_out.Set_Text_Size(1);
disp_out.Set_Text_Color(YELLOW);
disp_out.Draw_Text(text_x,text_y,string);

}
/* end of method */

```

```

void display_graphics::display_dynamic_elevator(int elevator_num,
                                               int floor_num,
                                               int display)
{
    /* note: elevator number is the elevator index floor number
    is the floor index display is true if you want color, false if
    you want clear */

    int x_start,y_start,floor_top,floor_bottom;
    int shaft_right,shaft_left;
    int lobby_bottom;

    x_start = building_x_start;
    y_start = building_y_start;

    /* since floors are inverse with graphics, we need to find the
    bottom floor coordinates */
    lobby_bottom = ((MAX_FLOORS-1)*floor_height) + y_start;

    /* calculate the elevator top and bottom */

    floor_top
    = lobby_bottom - (floor_num*floor_height) + 5;
    floor_bottom
    = lobby_bottom - (floor_num*floor_height) + floor_height - 5;

    /* calculate the elevator sides */
    shaft_left = x_start + (elevator_num*shaft_width) + 5;
    shaft_right = x_start + (elevator_num*shaft_width) + shaft_width - 5;

    /* set the color if displaying or unset the color if not */
    if (display)
        disp_out.Set_Line_Color(CYAN);
    else
        disp_out.Set_Line_Color(BLACK);
    disp_out.Draw_Rectangle(shaft_left,floor_top,shaft_right,floor_bottom);
}

void display_graphics::display_passenger_exit(int floor)
{
    /* note: floor number is the floor index */

    int index;
    int x_start,y_start,floor_top,floor_bottom;
    int floor_left,floor_right;
    int center,width;

```

```

int lobby_bottom;
int animate;

x_start = building_x_start;
y_start = building_y_start;

/* since floors are inverse with graphics, we need to find the
   bottom floor coordinates */
lobby_bottom = ((MAX_FLOORS-1)*floor_height) + y_start;

/* calculate the floor top and
   bottom */
floor_top
= lobby_bottom - (floor*floor_height) + 2;
floor_bottom
= lobby_bottom - (floor*floor_height) + floor_height - 2;

/* calculate the floor sides */
floor_left = x_start + (MAX_ELEVATORS*shaft_width) + 2;
floor_right = x_start + floor_width - 2;

/* person animation */
animate=15;
for (index=0;index<animate;index++)
{
    width = ((floor_right-floor_left)/animate);
    center = (floor_left+(width*index)+(width/2));
    disp_out.Set_Line_Color(WHITE);
    Draw_Person(floor_bottom,floor_top,center,width);

    // optional delay here - allows for better display on fast cpu
    //delay(75);

    disp_out.Set_Line_Color(BLACK);
    Draw_Person(floor_bottom,floor_top,center,width);
}
}

void display_graphics::Draw_Person (int bottom,int top,int center,int width)
{
    int x1,x2,y1,y2;
    int height;
    int quad;
    int radius;

```

```

height = bottom - top;
quad = height/8;

                                                                    /* head */
y1 = bottom-(quad*6);
x1 = center;
radius = quad;
disp_out.Draw_Circle(x1,y1,radius);

                                                                    /* body */
y1 = bottom-(quad*5);
x1 = center;
y2 = bottom-(quad*2);
x2 = center;
disp_out.Draw_Line(x1,y1,x2,y2);

                                                                    /* arms */
y1 = bottom-(quad*4);
x1 = center+(width/3);
y2 = bottom-(quad*4);
x2 = center-(width/3);
disp_out.Draw_Line(x1,y1,x2,y2);

                                                                    /* left leg */
y1 = bottom;
x1 = center-(width/2);
y2 = bottom - (quad*2);
x2 = center;
disp_out.Draw_Line(x1,y1,x2,y2);

                                                                    /* right leg */
y1 = bottom;
x1 = center+(width/2);
y2 = bottom-(quad*2);
x2 = center;
disp_out.Draw_Line(x1,y1,x2,y2);
}

void display_graphics::display_passenger_entry(void)
{
                                                                    /* note: floor number is the floor index */
int index,count;
int x_start,y_start,floor_top,floor_bottom;
int floor_left,floor_right;
int center,width;

```

```

int lobby_bottom;
int animate;
int floor = LOBBY_FLOOR; /* only enter on lobby floor */

x_start = building_x_start;
y_start = building_y_start;

/* since floors are inverse with graphics, we need to find the
   bottom floor coordinates */
lobby_bottom = ((MAX_FLOORS-1)*floor_height) + y_start;

/* calculate the floor top and bottom */
floor_top
    = lobby_bottom - (floor*floor_height) + 2;
floor_bottom
    = lobby_bottom - (floor*floor_height) + floor_height - 2;

/* calculate the floor sides */
floor_left = x_start + (MAX_ELEVATORS*shaft_width) + 2;
floor_right = x_start + floor_width - 2;

/* person animation */
animate=15;
for (count=0;count<animate;count++)
{
    index = animate-(count+1);
    width = ((floor_right-floor_left)/animate);
    center = (floor_left+(width*index)+(width/2));
    disp_out.Set_Line_Color(WHITE);
    Draw_Person(floor_bottom,floor_top,center,width);

    // optional delay here - allows for better display on fast CPU
    //delay(75);

    disp_out.Set_Line_Color(BLACK);
    Draw_Person(floor_bottom,floor_top,center,width);
}
}

int menu_and_prompts::display_menu()
{
    char choice;
    int number = MENU_INVALID_CHOICE;

    while (number == MENU_INVALID_CHOICE)

```

```

{
clrscr();
cout << " Elevator Simulation"
    << "\n\n"
    << "1. Begin Elevator Simulation" << '\n'
    << "2. Print Report" << '\n'
    << "3. Exit" << '\n'
    << "\n\n"
    << "Enter your selection: ";
fflush(stdin);
choice = getche();

/* bounds check - is it a valid menu choice? */
if (isdigit(choice))
{
switch (choice)
{
case '1':
number = MENU_BEGIN_SIMULATION;
break;
case '2':
number = MENU_PRINT_REPORTS;
break;
case '3':
number = MENU_EXIT;
break;
default:
number = MENU_INVALID_CHOICE;
break;
}
}
/* end of digit entered */
else
number = MENU_INVALID_CHOICE;
/* end of invalid choice */
}

return number;
/* end of function */
}

int menu_and_prompts::display_number_of_elevators_prompt()
{
char choice[2];
int number = MENU_INVALID_CHOICE;

while (number == MENU_INVALID_CHOICE)
{

```

```

clrscr();

cout << " Elevator Simulation"
    << "\n\n"
    << "Begin Elevator Simulation" << '\n'
    << "(enter 0 for main menu)" << '\n'
    << '\n'
    << "Please enter the number of " << '\n'
    << "elevators you want to see (0 to " << MAX_ELEVATORS << "): ";
fflush(stdin);
choice[0] = getche();
choice[1] = 0;

/* bounds check - is it a valid menu choice? */
if (isdigit(choice[0]))
{
    number = atoi(choice);
    if ((number < 0) || (number > MAX_ELEVATORS))
        number = MENU_INVALID_CHOICE;
} /* end of digit entered */
else
    number = MENU_INVALID_CHOICE;
} /* end of invalid choice */

return number;
}
int menu_and_prompts::display_print_reports_options()
{
    char choice;
    int number = MENU_INVALID_CHOICE;
    while (number == MENU_INVALID_CHOICE)
    {
        clrscr();

        cout << " Elevator Simulation"
            << "\n\n"
            << " Print Report" << '\n'
            << "1. To the printer" << '\n'
            << "2. To the screen" << '\n'
            << "3. Return to main menu" << '\n'
            << "\n\n"
            << "Enter your selection: ";
        fflush(stdin);
        choice = getche();
    }
}

```



```

/* bounds check - is it a valid menu choice? */
if (isdigit(choice))
{
    switch(choice)
    {
        case '1':
            number = MENU_PRINT_TO_PRINTER;
            break;
        case '2':
            number = MENU_PRINT_TO_SCREEN;
            break;
        case '3':
            number = MENU_RETURN_TO_MAIN;
            break;
        default:
            number = MENU_INVALID_CHOICE;
            break;
    }
}
/* end of digit entered */
else
    number = MENU_INVALID_CHOICE;
/* end of invalid choice */
return number;
}

```

// Module 3:

// Technical Description: This class is responsible for keeping track of the current floor position, the direction and state of operation, and the number of passengers aboard and their destination.

/**

```

#include <iostream.h>
#include "main.h"
#include "elevator.h"

```

```

elevator::elevator(int id) // constructor
{
    int idx;
    total_passenger_count = 0;
    passengers_on_board = 0;

    floor = LOBBY_FLOOR;
    direction = UP_DIRECTION;
}

```

```

state = IDLE_STATE;
elevator_id = id;
poisson_interval.put_interval((MAX_FLOORS-1));
total_duration = 0.0;
for (idx=0;idx<MAX_FLOORS;idx++)
{
    pass_on_floor[idx] = 0;
}
for (idx=0;idx<MAX_PASSENGERS;idx++)
{
    pass[idx] = NULL;
}
}

void elevator::run()
{
    int i;
    unsigned short dest;
    short done;
    double duration = 0;

    switch(state)
    {
        case IDLE_STATE:
            state = LOAD_STATE;
            if (direction == DOWN_DIRECTION)
            {
                direction = UP_DIRECTION;
                duration += TURN_TIME;
            }
            break;
        case LOAD_STATE:
            passengers_on_board = 0;

            for (i=0;i<MAX_PASSENGERS;i++)
            {
                done = FALSE;
                while (!done)
                {
                    dest = (unsigned short)poisson_interval.next();
                    if ((dest > 0) && (dest < MAX_FLOORS))
                        done = TRUE;
                }
                pass[i] = new passenger(dest);
                passengers_on_board++;
            }
        }
    }

```

```

    duration += LOAD_TIME;
}
state = UP_STATE;
break;
case UP_STATE:
    floor++;
    duration += FLOOR_TIME;
    for (i=0;i<MAX_PASSENGERS;i++)
    {
        // look for matching floor
        if (pass[i]->destination() == floor)
            state = STOP_STATE;
    }
    break;
case STOP_STATE:
    state = UNLOAD_STATE;
    duration += STOP_TIME;
    break;
case UNLOAD_STATE:
    for (i=0;i<MAX_PASSENGERS;i++)
    {
        // look for matching floor
        if (pass[i]->destination() == floor)
        {
            total_passenger_count++;
            passengers_on_board--;
            duration += LOAD_TIME;
            pass_on_floor[floor]++;
        }
    }
    if (passengers_on_board > 0)
        state = UP_STATE;
    else
    {
        // all passengers are unloaded - they can be eradicated
        for (i=0;i<MAX_PASSENGERS;i++)
        {
            delete pass[i];
            pass[i]=NULL;
        }
        state = DOWN_STATE;
        duration += TURN_TIME;
    }
    break;
case DOWN_STATE:

```

```

    direction = DOWN_DIRECTION;
    floor--;
    duration += FLOOR_TIME;
    if (floor == LOBBY_FLOOR)
        state = IDLE_STATE;
    break;
default:
    break;
} /* end of switch */
total_duration += duration;

// can use this to put delays into actual simulation
// return duration;

} /* end of run */

double elevator::current_duration()
{
    return total_duration;
}
unsigned long elevator::passenger_flow_rate()
{
    double flow; // rate of passers per hour
    double hours;
    hours = total_duration/(60.0*60.0); // secs to hour
    if (hours > 0.0)
        flow = total_passenger_count/hours;
    else
        flow = 0.0;
    return (unsigned long)flow;
}
unsigned long elevator::current_total_passenger_count()
{
    return total_passenger_count;
}
unsigned short elevator::current_passengers_on_board()
{
    return ((unsigned short)passengers_on_board);
}
unsigned short elevator::current_floor()
{
    return floor;
}
unsigned short elevator::current_direction()
{
    return direction;
}

```

```

}
unsigned short elevator::current_state()
{
    return state;
}
int elevator::current_id()
{
    return elevator_id;
}

// destructor

elevator::~elevator()
{
    int i;
    // all passengers are unloaded - they can be eradicated
    for (i=0;i<MAX_PASSENGERS;i++)
    {
        if (pass[i] != NULL)
        {
            delete pass[i];
            pass[i] = NULL;
        }
    }
}
unsigned long *elevator::pass_on_floor_array()
{
    return pass_on_floor;
}

// Module 4
//
// Technical Description: This class contains literals used in the main program.
//*****

#ifndef mainFile
#define mainFile

#define TRUE (1)
#define FALSE (0)

// elevator state machine

#define IDLE_STATE (0)
#define LOAD_STATE (1)
#define UP_STATE (2)

```

```

#define STOP_STATE (3)
#define UNLOAD_STATE (4)
#define DOWN_STATE (5)

// elevator capacity
#define MAX_PASSENGERS (20)
// maximum number of elevator cars
#define MAX_ELEVATORS (4)

// stats for a 10 story building
#define LOBBY_FLOOR (0)
#define TOP_FLOOR (9)
#define MAX_FLOORS (10)

#define UP_DIRECTION (1)
#define DOWN_DIRECTION (2)

// time to move on floor at the maximum speed
#define FLOOR_TIME (1.45)
// time needed to decelerate, open and close the doors, and accelerate
#define STOP_TIME (7.19)
// time needed for stopped car to change directions
#define TURN_TIME (1.0)
// mean time for 1 passenger to enter or exit the car
#define LOAD_TIME (1.0)

// menu literals
#define MENU_INVALID_CHOICE (-1)
#define MENU_BEGIN_SIMULATION (1)
#define MENU_PRINT_REPORTS (2)
#define MENU_EXIT (3)
#define MENU_PRINT_TO_SCREEN (4)
#define MENU_PRINT_TO_PRINTER (5)
#define MENU_RETURN_TO_MAIN (6)

#define VIDEO_DRIVER VGA
#define VIDEO_MODE VGAHI
#define VIDEO_BGI_DRIVER EGAVGA_driver
#define VIDEO_MAX_X 640
#define VIDEO_MAX_Y 480

#endif

// Module 5

```



```

    line_style = style;
}
void output::Set_Line_Color(COLORS color)
{
    line_color = color;
}
screen_output::screen_output()           /* constructor */
{
    registerbgidriver(VIDEO_BGI_DRIVER);
    initgraph(&driver,&mode,"");
    textbackground((int)BLACK);
    driver = (int)VIDEO_DRIVER;
    mode = (int)VIDEO_MODE;
    line_style = SOLID_LINE;
    line_size = NORM_WIDTH;
    line_color = WHITE;
    text_style = SANS_SERIF_FONT;
    text_size = 1;
    text_color = BLACK;
    text_direction = HORIZ_DIR;
    textcolor(text_color);
}
screen_output::~screen_output()         /* destructor */
{
    closegraph();
}
void screen_output::Draw_Text(int text_x,int text_y,char *string)
{
    if (text_x >= VIDEO_MAX_X)
        text_x = VIDEO_MAX_X-1;
    if (text_y >= VIDEO_MAX_Y)
        text_y = VIDEO_MAX_Y-1;

    switch(text_style)
    {
        case SANS_SERIF_FONT:
            registerbgifont(sansserif_font);
            break;
        case GOTHIC_FONT:
            // registerbgifont(gothic_font);
        case TRIPLEX_FONT:
            // registerbgifont(triplex_font);
        case SMALL_FONT:
            // registerbgifont(small_font);
        default:

```



```

        break;
    }
    /* end of text style switch */
    settxtstyle(text_style,text_direction,text_size);
    setcolor(text_color);
    moveto(text_x,text_y);
    outtext(string);
}
void screen_output::Draw_Line(int x1,int y1,int x2,int y2)
{
    if (x1 >= VIDEO_MAX_X)
        x1 = VIDEO_MAX_X-1;
    if (y1 >= VIDEO_MAX_Y)
        y1 = VIDEO_MAX_Y-1;
    if (x2 >= VIDEO_MAX_X)
        x2 = VIDEO_MAX_X-1;
    if (y2 >= VIDEO_MAX_Y)
        y2 = VIDEO_MAX_Y-1;

    setcolor(line_color);
    setlinestyle(line_style,0,line_size);
    line(x1,y1,x2,y2);
}
void screen_output::Draw_Circle(int center_x,int center_y,int radius)
{
    if (center_x >= VIDEO_MAX_X)
        center_x = VIDEO_MAX_X-1;
    if (center_y >= VIDEO_MAX_Y)
        center_y = VIDEO_MAX_Y-1;

    setcolor(line_color);
    setlinestyle(line_style,0,line_size);
    circle(center_x,center_y,radius);
}
void screen_output::Draw_Rectangle(int left,int top,int right,int bottom)
{
    if (left >= VIDEO_MAX_X)
        left = VIDEO_MAX_X-1;
    if (top >= VIDEO_MAX_Y)
        top = VIDEO_MAX_Y-1;
    if (right >= VIDEO_MAX_X)
        right = VIDEO_MAX_X-1;
    if (bottom >= VIDEO_MAX_Y)
        bottom = VIDEO_MAX_Y-1;

    setcolor(line_color);

```

```

setlinestyle(line_style,0,line_size);
rectangle(left,top,right,bottom);
}

```

// Module 6

// Technical Description: This class is responsible for generating poisson intervals.

/**

```

#include "poisson.h"
#include <stdlib.h>
#include <math.h>
#include <time.h>

```

```

float Poisson::next()
{
    double dbl_rand,max_rand;
    do
    {
        dbl_rand = (double)rand();
        max_rand = (double)RAND_MAX;
        p = float (iat*-log10((dbl_rand)/(max_rand+1)));
    } while (!p);
    return p;
}

```

```

void Poisson::put_interval(float t)
{
    iat = t;

```

// set random seed based on system time

```

    srand((unsigned)time(NULL));
}
float Poisson::get_interval(void)
{
    return iat;
}

```

// Module 7

// Technical Description: This class is responsible for printing reports to the screen or printer.

/**

```

#include <iostream.h>

```

```

#include "main.h"
#include "report.h"

void report::print_elevator_report(int id,
                                   unsigned long tpass,
                                   double secs,
                                   unsigned long flow,
                                   unsigned long *pass)
{
    int index;

    cout << "Elevator #" << id << endl;
    cout << "Total Passengers: " << tpass << endl;
    cout << "Duration:      " << secs << " seconds" << endl;
    cout << "Flow Rate:      " << flow << " pass/hour" << endl;
    cout << endl;
    cout << "Floor" << "\t" << "Passengers" << endl;
    for (index=0;index<MAX_FLOORS;index++)
    {
        if (pass[index])
            cout << (index+1) << "\t" << pass[index] << endl;
        else
            cout << (index+1) << "\t" << 0 << endl;
    }
    cout << endl << endl;
}

void report::print_summary_report(int num_elev,
                                   unsigned long tpass,
                                   double min,
                                   double max,
                                   unsigned long minflow,
                                   unsigned long maxflow,
                                   unsigned long *pass)
{
    int index;

    cout << "      Building Summary" << endl;
    cout << endl;
    cout << "Number of Elevators: " << num_elev << endl;
    cout << "Total Passengers:  " << tpass << endl;
    cout << "Minimum Duration:  " << min << endl;
    cout << "Maximum Duration:  " << max << endl;
    cout << "Minimum Flow Rate: " << minflow << endl;
    cout << "Maximum Flow Rate: " << maxflow << endl;
    cout << endl;
}

```

```
cout << "Floor" << "\t" << "Passengers" << endl;
for (index=0;index<MAX_FLOORS;index++)
{
    if (pass[index])
        cout << (index+1) << "\t" << pass[index] << endl;
    else
        cout << (index+1) << "\t" << 0 << endl;
}
cout << endl << endl;
}
printer_report::printer_report()
{
    output_device = 0;
}
screen_report::screen_report()
{
    output_device = 0;
}
```

A.2 Back-Propagation Control Training Code

```
// Module 1

// Technical
// Description: This module is the main module.
//*****

#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

//returns a float in the range -1.0f -> 1.0f
#define RANDOM_CLAMP (((float)rand()-(float)rand())/RAND_MAX)
//returns a float between 0 & 1
#define RANDOM_NUM ((float)rand()/RAND_MAX+1)
//using namespace std

class Dendrite {
public:
    double d_weight; //weight of the neuron
    unsigned long d_points_to; //The index of the neuron of the next layer to which
//it points
//Constructor Assigning initial values
    Dendrite(double weight=0.0, unsigned long points_to=0){
        d_weight=weight;
        d_points_to=points_to; //Gives it an initial value
    }
};

class Neuron {
public:
    unsigned long n_ID; //ID of a particular neuron in a layer
    double n_value; //Value which neuron currently is holding
    double n_bias; //Bias of the neuron
    double n_delta;
    Dendrite *Dendrites; //Dendrites

//Constructor Assigning initial values
    Neuron(unsigned long ID=0,double value=0.0,double bias=0.0) {
        n_ID=ID;
        n_value=value;
        n_bias=bias;
        n_delta=0.0;
    }
}
```

```

//set dendrite from the neuron to given dendrite
void SetDendrites(int dendrite) {
    Dendrites=new Dendrite[dendrite];
    for(int i=0;i<dendrite;i++) {
        Dendrites[i].d_points_to=i; //Initialize the dendrite to attach next layer
    }
}
};
class Layer {
public:
    Neuron *Neurons; //Pointer to array of neurons
    /*Layer(unsigned long size=1) { //size is no. of neurons in it
        Neurons = new Neuron [size];
    } */
    //Initialize the layer
    void Initialize(unsigned long size) {
        Neurons = new Neuron [size];
    }
    //destructor deletes neurons from memory
    ~Layer() {
        delete Neurons;
    }
    //Give the neuron at index
    Neuron GetNeuron(unsigned long index) {
        return Neurons[index];
    }
    void SetNeuron(Neuron neuron,unsigned long index) {
        Neurons[index]=neuron;
    }
};
class Network {
public:
    double net_learning_rate; //learning rate of network
    Layer *Layers; //The total layers in a network
    unsigned long net_tot_layers; //Number of layers
    double *net_inputs; //input array
    double *net_outputs; //Output layers
    unsigned long *net_layers; //Array which tells the no of neurons in each layer
    //double GetRand(void);

    //Blank constructor
    Network() {
    }
    //Function to set various parameters of the net
    int SetData(double learning_rate, unsigned long layers[], unsigned long tot_layers)

```

```

{
    if(tot_layers<2) return(-1);

    net_learning_rate=learning_rate;
    net_layers=new unsigned long [tot_layers];    //Initializes the layers array
    Layers=new Layer[tot_layers];
    for(int i=0;i<tot_layers;i++) {
        net_layers[i]=layers[i];
        Layers[i].Initialize(layers[i]); //Initializes each layer with specific size
    }

    net_inputs=new double[layers[0]];
    net_outputs=new double[layers[tot_layers-1]];
    net_tot_layers=tot_layers;
    return 0;
}

//Function to set the inputs
int SetInputs(double inputs[]) {
    for(int i=0;i<net_layers[0];i++) {
        Layers[0].Neurons[i].n_value=inputs[i];
    }
    return 0;
}

//Randomize Weights and Biases
void RandomizeWB(void) {
    int i,j,k;
    for(i=0;i<net_tot_layers;i++) {
        for(j=0;j<net_layers[i];j++) {
            if(i!=(net_tot_layers-1)) {
                Layers[i].Neurons[j].SetDendrites(net_layers[i+1]);
                for(k=0;k<net_layers[i+1];k++) {

                    Layers[i].Neurons[j].Dendrites[k].d_weight=GetRand();

                }
                if(i!=0) Layers[i].Neurons[j].n_bias=GetRand();
            }
        }
    }
}

double * GetOutput(void) {
    double *outputs;
    int i,j,k;
    outputs=new double[net_layers[net_tot_layers-1]];
}

```

```

for(i=1;i<net_tot_layers;i++) {
    for(j=0;j<net_layers[i];j++) {
        Layers[i].Neurons[j].n_value=0;
        for(k=0;k<net_layers[i-1];k++){
            Layers[i].Neurons[j].n_value=Layers[i].Neurons[j].n_value+Layers[i-1].Neurons[k].n_value*Layers[i-1].Neurons[k].Dendrites[j].d_weight;
        }

Layers[i].Neurons[j].n_value=Layers[i].Neurons[j].n_value+Layers[i].Neurons[j].n_bias;
//Add bias
        Layers[i].Neurons[j].n_value=Limiter(Layers[i].Neurons[j].n_value);

//Squash value
    }
}
for(i=0;i<net_layers[net_tot_layers-1];i++) {
    outputs[i]=Layers[net_tot_layers-1].Neurons[i].n_value;
}
return outputs;
}
void Update(void) {
    GetOutput();
}

//Limit value between 1 and -1
double Limiter(double value) {
    return (1.0/(1+exp(-value)));
}

//return a random number between range -1 and 1 using time to seed the
//srand function
double GetRand(void) {
    time_t timer;
    struct tm *tblock;
    timer=timer(NULL);
    tblock=localtime(&timer);
    int seed=int(tblock->tm_sec+100*RANDOM_CLAMP+100*RANDOM_NUM);
//srand(tblock->tm_sec);

    srand(seed);
    return (RANDOM_CLAMP+RANDOM_NUM);
}

//Calculate sum of weights * delta. Used in
backpropagation
double SigmaWeightDelta(unsigned long layer_no, unsigned long neuron_no)
{
    double result=0.0;

```



```

//Go through all the neurons in the
next layer
    for(int i=0; i<net_layers[layer_no+1]; i++)
{
result=result+Layers[layer_no].Neurons[neuron_no].Dendrites[i].d_weight*Layers[layer_no
+1]
        .Neurons[i].n_delta;           //compute the summation

    }
    return result;
}

int Train(double inputs[], double outputs[]) {
    int i,j,k;
    double Target, Actual, Delta;
    SetInputs(inputs);           //Set the Inputs
    Update();                     //Update all the values

    for(i=net_tot_layers-1;i>0;i--) {           //Go from last layer to first
layer
        for(j=0;j<net_layers[i];j++) {           //Go through every neuron
            if(i==net_tot_layers-1) {           //Output layer
                Target=outputs[j];
                Actual=Layers[i].Neurons[j].n_value;           //Actual value
                Delta= (Target - Actual) * Actual * (1 - Actual); //Function to compute error
                Layers[i].Neurons[j].n_delta=Delta;           //Compute the Delta
                for(k=0;k<net_layers[i-1];k++) {
                    Layers[i-1].Neurons[k].Dendrites[j].d_weight+=
Delta*net_learning_rate*Layers[i-
                    1].Neurons[k].n_value;
                }
                Layers[i].Neurons[j].n_bias=Layers[i].Neurons[j].n_bias+
Delta*net_learning_rate*1;
            }
            else {
                Actual=Layers[i].Neurons[j].n_value;
                Delta=Actual * (1 - Actual)*SigmaWeightDelta(i,j);
                for(k=0;k<net_layers[i-1];k++) {
                    Layers[i-1].Neurons[k].Dendrites[j].d_weight+=
Delta*net_learning_rate*Layers[i- 1].Neurons[k].n_value;
                }
                if(i!=0)Layers[i].Neurons[j].n_bias=Layers[i].Neurons[j].n_bias
Delta*net_learning_rate*1;
            }
        }
    }
}

```

```

    }
}
} return 0;
~Network(){ delete Layers; }
};
int main()
{ Network my;
  unsigned long inp=10;
  unsigned long hid=10;
  unsigned long outp=10;
  unsigned long layers[3];
  layers[0]=inp;
  layers[1]=hid;
  layers[2]=outp;
  int i=0, j=0;
  unsigned long iter=0;
  cout<<"Enter number of training iterations : ";
  cin>>iter;

  double input[]={ 1,0,0,0,0,0,0,0,0,0};
  double *outputs;
  my.RandomizeWB();
  double tr_inp[10][10]=
  {{28,39,9,10,15,9,27,6,45,75},{18,64,41,28,21,26,50,232,94,116},{203,28,122,173,108,166,
104,36,104,88},{16,64,154,221,153,73,119,91,185,95},{126,159,27,96,217,45,79,28,188,84}
{234,78,154,87,149,144,154,102,76},{92,161,211,159,225,186,92,185,223,233},{92,126,163
,236,120,99,94,183,185,117},{168,80,35,204,49,1,100,10,109,222},{21,163,104,102,187,72,
210,236,45,169}}};

  double tr_out[10][1]={{40},{58},{58},{47},{46},{50},{54},{52},{58},{51}};
  cout<<"\nStarting training.....";
  for(i=0;i<iter;i++) {
    for(j=0;j<10;j++) {
      my.Train(tr_inp[j],tr_out[j]);
    }
  }
  cout<<"\nEnding Training. ";
  cout<<"\n\nStarting Testing...\n";
  for(j=0;j<10;j++) {
    cout<<"\n\nCase number : "<<j+1;
    my.SetInputs(tr_inp[j]);
    outputs=my.GetOutput();
    for(i=0;i<inp;i++) {
      cout<<"\nInput"<<i+1<<" : "<<tr_inp[j][i];
    }
  }
}

```

```
for(i=0;i<outp;i++) {  
    cout<<"\nOutput"<<i+1<<" : "<<outputs[i];  
}  
delete outputs;  
double *outputs;  
}  
cout<<"\n\nEnd Testing. \n\n";  
system("PAUSE");  
return 0;  
}
```

Appendix B.

B. Training Algorithm

The algorithm is as follows:

Step 0. Initialize weights. (Set to small random values)

Step 1. While stopping condition is false, do Steps 2 – 9.

Step2. For each training pair, do Steps 3 – 8.

(Feedforward)

Step3. Each input unit ($X_i, i = 1, \dots, n$) receives input signal x_i and broadcasts this signal to all units in the layer above (the hidden units).

Step 4. Each hidden unit ($Z_j, j = 1, \dots, p$) sums its weighted input signals,

$$Z_in_j = v_{oj} + \sum_{i=1}^n x_i v_{ij} \quad (B.1)$$

Applies its activation function to compute its output signal,

$$Z_j = f(z_in_j) \quad (B.2)$$

And sends this signal to all units in the layer above (output units)

Step 5. Each output unit ($Y_k, k = 1, \dots, m$) sums its weighted input signals,

$$y_in_k = w_{oj} + \sum_{j=1}^p z_j w_{jk} \quad (B.3)$$

and applies its activation function to compute its output signal,

$$y_k = f(y_in_k) \quad (B.4)$$

(Backpropagation of error)

Step 6. Each output unit ($Y_k, k = 1, \dots, m$) receives a target pattern corresponding to the input training pattern, computes its error information term,

$$\delta_k = (t_k - y_k) f'(y - in_k) \quad (B.5)$$

calculates its weight correction term (used to update w_{jk} later),

$$\Delta w_{jk} = \alpha \delta_k z_j \quad (B.6)$$

calculates its bias correction term (used to up-date w_{0k} later),

, $\Delta w_{0k} = \alpha \delta_k$ and sends δ_k to units in the layer below.

Step 7. Each hidden unit ($Z_j, j = 1, \dots, p$) sums its delta inputs (from units in the layer above),

$$\delta_{in_j} = \sum_{k=1}^m \delta_k w_{jk} \quad (B.7)$$

Multiplies by the derivatives of its activation function to calculate its error information term,

$$\delta_j = \delta_{in_j} f'(z_{in_j}) \quad (B.8)$$

Calculates its weight correction term (used to update v_{ij} later),

$\Delta w_{ij} = \alpha \delta_j x_i$ and calculates its bias correction term (used to update v_{0j} later), $\Delta v_{0j} = \alpha \delta_j$

Step 8. Each output unit ($Y_k, k = 1, \dots, m$) updates its bias and weights ($j = 0, \dots, p$):

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk} \quad (B.9)$$

Each hidden unit ($Z_j, j = 1, \dots, p$) updates its bias and weights ($i = 0, \dots, n$):

$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij} \quad (B.10)$$

Step 9. Test stopping condition.

Appendix C.

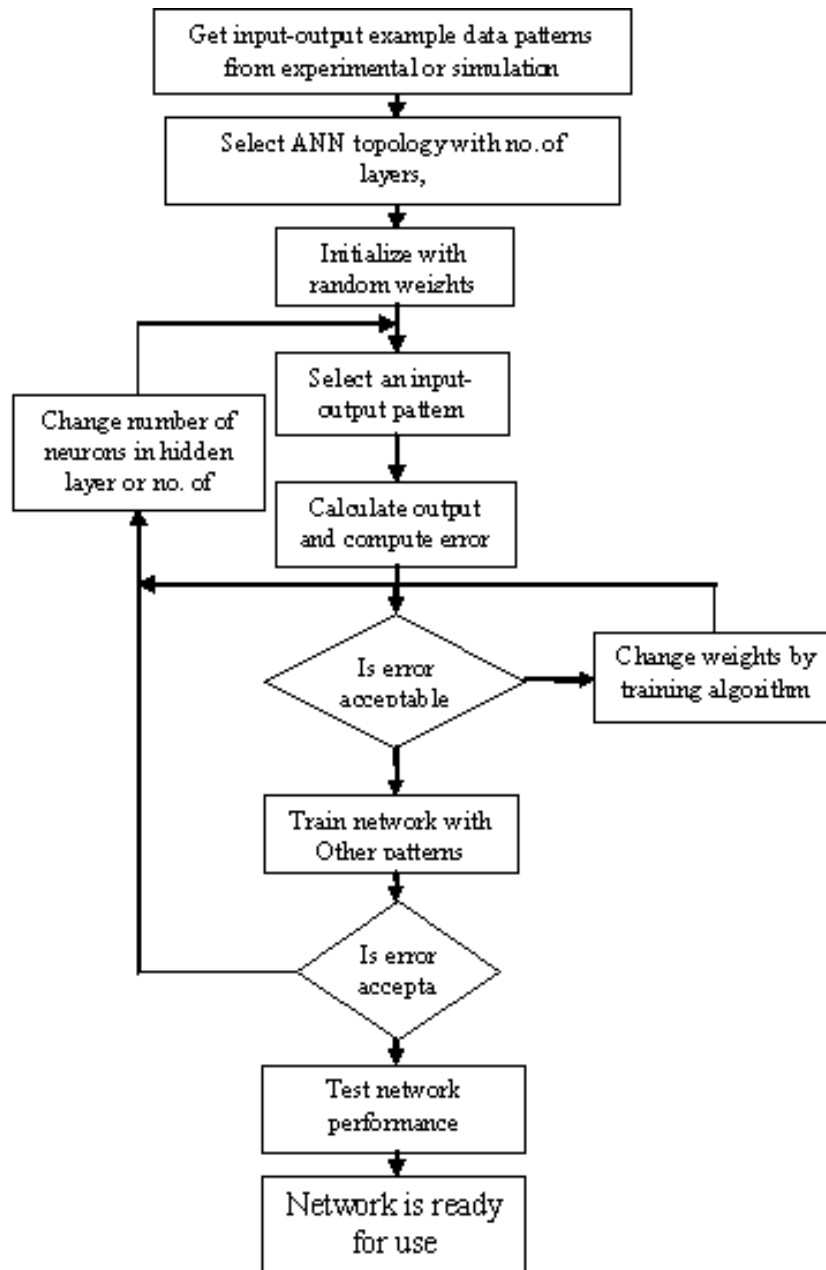


Figure B.1 Flowchart for back propagation training of feedforward neural network
The weights (and biases) were initialized to random values of between -0.5 and 0.5.