

**ST. XAVIER'S COLLEGE
(AUTONOMOUS), KOLKATA**

**Bayesian approach for solving the
drawbacks of weighted sum**

**B.SC. COMPUTER SCIENCE
SEMESTER VI**

Submitted by

ANNE HILARIAN

(16-300-4-02-0504)

RAJARSI SAHA

(16-300-4-02-0529)

SANCHARI KAR CHOWDHURY

(16-300-4-02-0562)

UNDER THE GUIDANCE OF

PROF. ROMIT S. BEED

CERTIFICATE OF AUTHENTICATED WORK

This is to certify that the project report entitled **Bayesian approach for solving the drawbacks of weighted sum** submitted to Department of Computer Science ST. XAVIER'S COLLEGE (AUTONOMOUS), KOLKATA, in partial fulfilment of the requirement for the award of the degree of BACHELOR OF SCIENCE (B.SC.) is an **original** work carried out by:

Name	Roll No.	Registration No.
ANNE HILARIAN	16-300-4-02-0504	A01-2112-0752-16
RAJARSI SAHA	16-300-4-02-0529	A01-1122-0771-16
SANCHARI KAR CHOWDHURY	16-300-4-02-0562	A01-2112-0785-16

under my guidance. The matter embodied in this project is authentic and is genuine work done by the student and has not been submitted whether to this College or to any other Institute for the fulfilment of the requirement of any course of study.

Anne Hilarian

Date:

Project Guide:

Romit S. Beed

Date:

Rajarsi Saha

Date:

Sanchari Kar Chowdhury

Date:

We hereby recommend that the project report prepared by Anne Hilarian, Rajarsi Saha and Sanchari Kar Chowdhury entitled "**Bayesian approach for solving the drawbacks of weighted sum**" be accepted in partial fulfilment of the requirements for the degree of B.Sc. in Computer Science at St. Xavier's College (Autonomous), Kolkata

Head of the Department
Department of Computer Science

External Examiner

ROLES AND RESPONSIBILITIES FORM

Name of the Project: **Bayesian approach for solving the drawbacks of weighted sum**

Name of Team Members	Task and Responsibilities
Anne Hilarian	Documentation and Data Collection
Rajarsi Saha	Coding, Debugging and Design
Sanchari Kar Chowdhury	Testing and Visualization

Date:

Name and Signature of the Project Team members:

Name:

Signature:

.....

.....

.....

.....

.....

.....

Signature of the Professor:

Date:

.....

.....

ABSTRACT

This paper proposes a scientific technique to improve the weights obtained deterministically in solving optimization problems using the popular weighted sum technique. It aims to probabilistically generate the weights for conflicting objectives using a Bayesian Hierarchical model based on multinomial distribution and Dirichlet prior. The proposed technique is ideally suited for situations when one has to depend on a small dataset. The method aims at improving the existing methods of weight determination when using data driven choice of weights. This is done using appropriate probabilistic modelling techniques and ensures, on an average, much reliable results than non-probabilistic techniques.

ACKNOWLEDGEMENT

In performing our assignment, we had to take the help and guidance of some helpful people, who deserve our greatest gratitude. The completion of this assignment gives us much pleasure. We would like to extend our heartfelt gratitude to Prof. Romit S. Beed, Project Mentor, for his valuable time in giving us good advice for the assignment throughout numerous consultations.

In addition, we thank Prof. Durba Bhattacharya, for introducing us to the statistical methodologies required in the project.

Many people, especially our classmates and our faculty have given us valuable suggestions on this proposal which gave us an inspiration to improve our assignment. We thank all the people for their help directly or indirectly in completing our assignment.

TABLE OF CONTENTS

	PAGE NUMBER
1. INTRODUCTION	10
1.1. BACKGROUND	10
1.2. OBJECTIVES	11
1.3. SCOPE AND PURPOSE	11
2. SURVEY OF TECHNOLOGIES	12
2.1. THE WEIGHTED SUM METHOD	12
2.2. STATISTICAL PREREQUISITES	13
2.3. PARETO OPTIMAL SOLUTION	15
2.4. THE SHORTEST PATH PROBLEM	15
2.5. GENETIC ALGORITHM	15
3. REQUIREMENT AND ANALYSIS	17
3.1. PROBLEM DEFINITION	17
3.2. PLANNING AND SCHEDULING	19
3.3. MODEL AND METHODOLOGIES IN DETERMINATION OF WEIGHTS	20
3.4. SOFTWARE AND HARDWARE REQUIREMENTS	24
3.4.1. HARDWARE REQUIREMENTS	24
3.4.2. SOFTWARE REQUIREMENTS	24
3.5. PRELIMINARY PRODUCT DESCRIPTION	24
4. SYSTEM DESIGN	24
4.1. BASIC MODULES	25
4.1.1. MATRICES	25
4.1.2. FREQUENTIST	26
4.1.3. BAYESIAN	26
4.1.4. FREQ_GENETIC	26
4.1.5. BAY_GENETIC	26
4.1.6. CHROMOSOME	26
4.1.7. IMPORT_CODE	27
4.1.8. MATRIX	27

4.1.9.	SPARSE_GRAPH_SET1	27
4.1.10.	SPARSE_GRAPH_SET2	27
4.2.	DATA DESIGN	28
4.2.1.	DATA INTEGRITY AND CONSTRAINTS	30
4.3.	PROCEDURAL DESIGN	31
4.3.1.	LOGIC DIAGRAMS	31
4.3.2.	DATA STRUCTURES	31
4.3.3.	ALGORITHM DESIGN	32
5.	IMPLEMENTATION AND TESTING	37
5.1.	IMPLEMENTATION APPROACHES	37
5.2.	CODE DETAILS AND CODE EFFICIENCY	37
5.2.1.	CODING DETAILS	37
5.2.2.	CODE EFFICIENCY	55
5.3.	TESTING APPROACH	55
5.3.1.	UNIT TESTING	55
5.3.2.	INTEGRATED TESTING	58
5.4.	MODIFICATIONS AND IMPROVEMENTS	66
6.	RESULTS AND DISCUSSION	66
6.1.	TEST DESIGN AND REPORTS	66
7.	CONCLUSION	75
7.1.	CONCLUSION	75
7.2.	LIMITATIONS OF THE SYSTEM	75
7.3.	FUTURE SCOPE OF THE PROJECT	75
REFERENCES		76

TABLE OF FIGURE

FIGURE NO	FIGURE DESCRIPTION	PAGE
	GENETIC ALGORITHM	
1	CROSSOVER PROCESS	16
	PLANNING AND SCHEDULING	
2	PERT CHART	19
3	GNATT CHART	19
	DATA DESIGN	
4	GRAPH VISUALIZATION FOR DATASET 1	28
5	GRAPH VISUALIZATION FOR DATASET 2	29
	LOGIC DIAGRAMS	
6	DATA FLOW DIAGRAM	31
	ALGORITHM DESIGN	
7	GENETIC ALGORITHM FLOWCHART	32
8	FREQUENTIST WEIGHTS	31
9	BAYESIAN WEIGHTS	33
10	PLOT SHOWING GAIN IN EFFICIENCY	33
11	PLOT SHOWING EFFICIENCY	34
12	OPTIMUM PATH FROM FREQUENTIST WEIGHTS	34
13	OPTIMUM PATH FROM FREQUENTIST WEIGHTS (INITIAL VIEW)	35
14	OPTIMUM PATH FROM FREQUENTIST WEIGHTS (INTERMEDIATE VIEW)	35
15	OPTIMUM PATH FROM FREQUENTIST WEIGHTS (RESULT VIEW)	36
16(a)	OPTIMAL PATH PLOT – FREQUENTIST (DATA SET 1)	36
16(b)	OPTIMAL PATH PLOT – BAYESIAN (DATA SET 1)	36
	UNIT TESTING	
17(a)	BAYESIAN WEIGHT DETERMINATION (INITIAL VIEW)	56
17(b)	BAYESIAN WEIGHT DETERMINATION (RESULT VIEW)	56
18	OPTIMAL PATH DETERMINATION	57
19(a)	OPTIMAL PATH DETERMINATION (INITIAL VIEW)	58
19(b)	OPTIMAL PATH DETERMINATION (RESULT VIEW)	58
	INTEGRATED TESTING	
20	FREQUENTIST WEIGHT DETERMINATION	59
21	BAYESIAN WEIGHT DETERMINATION	59
22	GAIN IN EFFICIENCY PLOT	60
23	EFFICIENCY PLOT	60
24	EXCEL SHEET DATA	61
25	OPTIMAL PATH DETERMINATION USING FREQUENTIST WEIGHTS	62
26	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS (INITIAL VIEW)	62

27	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS (RESULT VIEW)	63
28(a)	OPTIMAL PATH PLOT FOR FREQUENTIST WEIGHT	63
28(b)	OPTIMAL PATH PLOT FOR BAYESIAN WEIGHTS	63
29	OPTIMAL PATH DETERMINATION USING FREQUENTIST WEIGHTS	64
30	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS (INITIAL VIEW)	64
31	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS (RESULT VIEW)	65
32(a)	OPTIMAL PATH PLOT FOR FREQUENTIST WEIGHT	65
32(b)	OPTIMAL PATH PLOT FOR BAYESIAN WEIGHT	65
	TEST DESIGN AND REPORT	
33	FREQUENTIST WEIGHT DETERMINATION	66
34	BAYESIAN WEIGHT DETERMINATION	67
35	OPTIMAL PATH DETERMINATION USING FREQUENTIST WEIGHTS	67
36	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS	68
37(a)	OPTIMAL PATH PLOT FOR FREQUENTIST WEIGHTS	68
37(b)	OPTIMAL PATH PLOT FOR BAYESIAN WEIGHTS	68
38	FREQUENTIST WEIGHT DETERMINATION	69
39	BAYESIAN WEIGHT DETERMINATION	69
40	RESULT FOR INVALID DESTINATION VALUE	70
41	FREQUENTIST WEIGHT DETERMINATION	70
42	BAYESIAN WEIGHT DETERMINATION	71
43	RESULT FOR NON-COMPATIBLE SAMPLE VALUES	71
44	FREQUENTIST WEIGHT DETERMINATION	72
45	BAYESIAN WEIGHT DETERMINATION	72
46	OPTIMAL PATH DETERMINATION USING FREQUENTIST WEIGHTS	73
47	OPTIMAL PATH DETERMINATION USING BAYESIAN WEIGHTS	73
48(a)	OPTIMAL PATH PLOT FOR FREQUENTIST WEIGHTS	74
48(b)	OPTIMAL PATH PLOT FOR BAYESIAN WEIGHTS	74

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

Following the introduction of the weighted sum method in 1963, it has been used extensively in solving optimization problems. However, most general treatments of optimization problems simply outline the weighted sum approach and indicate that it provides a Pareto optimal solution. Intricacies of the method and of the solutions that it yields are typically not discussed. In particular, the significance of the weights is not thoroughly explored, and thus, despite the presence of many algorithms for determining weight values, no fundamental guidelines have been presented for selecting weights. A few different, somewhat arbitrary sets of weights are considered, and the corresponding solutions are compared, but the method itself has not been studied extensively. The most common method linearly aggregates all the individual objective functions in an optimization problem into one objective by using a weight vector. Usually, a weight vector was predefined before the search, or changed during the search progressively. For instance, in using this method to solve optimization problems, weights are predefined and multiple Pareto optimal solutions are obtained by a systematic change in weights in different algorithm runs.

As a common concept, minimizing a weighted sum constitutes an independent method as well as a component of other methods. Consequently, insight into characteristics of the weighted sum method has far reaching implications.

This project proposes a logical method to improve the weights obtained deterministically in comprehending optimization issues, utilizing the prominent weighted sum technique. It probabilistically produces the weights for conflicting objective functions, utilizing a Bayesian model based on multinomial dissemination and Dirichlet.

By definition, the weighted sum reduces a positively weighted convex sum of objectives as follows:

$$\min \sum_{i=1}^n w_i \cdot f_i(x)$$
$$\sum_{i=1}^n w_i = 1 ; w_i > 0; i = 1, 2, \dots, n$$

This process involves scalarizing the conflicting objectives into a single objective function.

1.2 OBJECTIVES

- Our primary objective here is to use a probabilistic technique to obtain weights to be used for the weighted sum algorithm. Comparisons are then made with existing and traditional weight-determination techniques.
- The method works best when one has to work with a small data set.
- To demonstrate the use of the proposed technique, the project attempts to consider the objectives of distance, time and availability of parking when moving from a particular source to a destination via a number of intermediate nodes. This involves:
 - Minimizing the distance from the source to the destination.
 - Minimizing the time taken to reach the destination.
 - Maximizing the availability of the parking.

1.3 SCOPE AND PURPOSE

The purpose here is to decide on a scientific technique to improve the weights obtained deterministically in solving optimization problems using weighted sum technique. The approach here is probabilistic as compared to the usual deterministic approach to weight selection.

Examining and determining the most favourable and optimal route considering several factors.

CHAPTER 2: SURVEY OF TECHNOLOGIES

2.1 THE WEIGHTED SUM METHOD

It is observed that majority of the real-world problems generally involve optimizing multiple conflicting objectives. As these optimization formulations involve multiple objectives, the objective function is formulated as a vector and it is treated as a vector optimization or a multi-objective optimization problem (MOOP). A multi-objective problem involving multiple, conflicting objectives may be combined into a single-objective scalar function. This approach is called the weighted sum method.

Literature Review

The Weighted sum (WS) technique, a commonly used scalarizing technique in optimization algorithms, has distinctive advantages of greater efficiency and easier computational capabilities. Nevertheless, it is frequently critiqued for its inability to predict the logic behind the weight selection. It was suggested by Steuer [1] that the weights scientifically determine the decision maker's preference for a particular objective. Das and Dennis [2] offered a graphical explanation of this technique to elucidate few of its drawbacks. The delusion between the hypothetical and the realistic interpretation of the weights for the conflicting objectives made the weight selection process quite inefficient. Various approaches have been suggested for weight selection- Yoon and Hwang [3] suggested a ranking technique whereby the objectives are ranked based on their significance. The most important objective received the largest weight with gradual decrease in weights to lesser important objectives. A similar technique proposed was the categorization technique in which the conflicting objectives were grouped according to their varying degree of importance.

Selection of accurate weights leads to an algorithm's better performance. Gennert and Yuille [4] proposed a nonlinear weight determination algorithm where an optimal point is obtained that is not in the vicinity of the extreme points. Although a lot of literature is available regarding systematic selection of weights in solving optimization problems, till date a comprehensive data driven technique determining weights reflecting the relative importance of the conflicting objectives is lacking.

Most of the work in the available literature is based on deterministically fixing the weights based on some prior beliefs of information. The focus of the existing methods is towards refining the distribution of Pareto solutions provided by the WS technique [5], with less emphasis on the stability and appropriateness of the choice of weights for precise representation of the conflicting objectives. In contrast to the objective of weight determination of the existing works, which aim at choosing the set of weights which stabilises the solution set, this work proposes to frame a model which determines a much stable set of weights in comparison to that obtained deterministically. The criticisms of the existing methodologies for determination of weights have motivated this work and to propose the Bayesian model on multinomial and Dirichlet priors.

Interpretation of the weights

One often views the weights as general gauges of relative importance for each objective function. However, selecting a set of weights that reflects preference towards one objective or another can be difficult, because preferences tend to be indistinct. In addition, even with full knowledge of the objectives and satisfactory selection of weights, the final solution may not necessarily reflect the intended preferences that are supposedly incorporated in the weights. Nonetheless, the specific Pareto optimal point that is provided as the solution depends on which weights are used, so it is important to determine how the weights relate to preferences, to the Pareto optimal set, and to the individual objective functions.

2.2 STATISTICAL PREREQUISITES

Bayes' Theorem

The conditional probability of an event is a probability obtained with the additional information that some other event has already occurred. The expression $P(B/A)$ is used to denote the conditional probability of event B occurring, given that event A has already occurred. The following formula gives $P(B/A)$:

$$P(B|A) = \frac{P(A \text{ and } B)}{P(A)}$$

In addition to the above formal rule, conditional probability can also be explained as follows:

*The conditional probability of **B** given **A** can be found by assuming that event **A** has occurred and, working under that assumption, calculating the probability that event **B** will occur.*

In addition, we have two key terms in Bayesian Statistics: Prior probability distribution and posterior probability distribution.

A **prior probability** is an initial probability value originally obtained before any additional information is obtained.

A **posterior probability** is a probability value that has been revised by using additional information that is later obtained.

Bayesian vs Frequentist Approach

Unlike frequentist approach which doesn't quantify the uncertainty in fixed but unknown values of the parameters, Bayesian approach defines probability distributions over possible values of a parameter. Let x denote the data and θ be the parameter of interest which is unknown. Let $\theta \in \Theta$ be the parametric space. Under Bayesian approach one can quantify the prior belief about θ by defining a prior probability distribution over Θ , the set of possible values of θ . The newly collected data makes the probability distribution over Θ narrower by updating the prior distribution to posterior distribution of θ using Bayes' theorem which states that

$$P(\theta|Data) = \frac{P(Data|\theta) \times P(\theta)}{P(Data)}$$

$P(Data|\theta)$ is called the likelihood and $P(\theta|Data)$ is the posterior distribution of the parameter θ .

Multinomial Distribution

It is a multivariate generalization of Binomial Distribution. Suppose an experiment is conducted such that each trial has k (finite and fixed) mutually exclusive and exhaustive possible outcomes with probabilities p_1, p_2, \dots, p_k such that $p_i \geq 0 \forall i = 1(1)k$ and $\sum p_i = 1$. If X_i be the random variable indicating the number of times category i is observed over n independent trials of the experiment, then the vector $X = (X_1, X_2, \dots, X_k)$ follows a Multinomial Distribution with parameters n and (p_1, p_2, \dots, p_k) . The probability mass function of the multinomial distribution is:

$$\begin{aligned} f(X_1 = x_1, X_2 = x_2, \dots, X_k = x_k) \\ = \frac{n!}{x_1! x_2! \dots x_k!} p_1^{x_1} p_2^{x_2} \dots p_k^{x_k} (1 - p_1 - \dots - p_{k-1})^{n-x_1-\dots-x_{k-1}} \end{aligned}$$

Dirichlet Distribution

Dirichlet distribution is a multivariate generalization of Beta distribution. Dirichlet distribution of order ($k \geq 2$) with parameters $a_1, a_2, \dots, a_k > 0$ has the following probability density function:

$$g(x_1, x_2, \dots, x_k) = \frac{\Gamma(\sum_{i=1}^k a_i)}{\prod_{i=1}^k \Gamma(a_i)} \prod_{i=1}^k x_i^{a_i-1}$$

Here the x_i are continuous random variables with $x_i \geq 0 \forall i$ and $\sum x_i = 1$, that is, the support of Dirichlet distribution is the set of k -dimensional vectors whose entries belong to $(0,1)$ and add up to one. It is to be noted that the parameter vector (p_1, p_2, \dots, p_k) of the Multinomial distribution has the properties of the x_i above, and hence can be modelled using an appropriate Dirichlet distribution.

Conjugate Prior

In Bayesian probability theory, if posterior and prior probability distributions of the parameter θ belong to the same probability distribution family, the prior is then called a conjugate prior.

In other words, in the formula

$$P(\theta|Data) = \frac{P(Data|\theta) \times P(\theta)}{P(Data)}$$

if $P(\theta)$ and $P(\theta|Data)$ are in the same family of distributions,

they are called conjugate distributions. It can be shown that Dirichlet distribution acts as conjugate prior for Multinomial distribution.

2.3 PARETO OPTIMAL SOLUTION

Given a set of solutions, the non-dominated solution set is a set of all the solutions that are not dominated by any member of the solution set. The non-dominated set of the entire feasible decision space is called the **Pareto-optimal set**. The boundary defined by the set of all point mapped from the Pareto optimal set is called the **Pareto optimal front**.

2.4 THE SHORTEST PATH PROBLEM

The shortest path problem is defined as that of finding a minimum-length (cost) path between a given pair of nodes. Shortest path problem is a classical research topic. It was proposed by Dijkstra in 1959 and has been widely researched. The Dijkstra algorithm is considered as the most efficient method. It is based on the Bellman optimization theory. But when the network is very big, then it becomes inefficient since a lot of computations need to be repeated. Also, it cannot be implemented in the permitted time. In this project the shortest path problem has been used to demonstrate the application of the proposed method.

2.5 GENETIC ALGORITHM

Routing is a fundamental engineering task on the Internet. It consists in finding a path from a source to a destination host. Routing is complex in large networks because of the many potential intermediate destinations a packet might traverse before reaching its destination. The link weights are assigned by the network operator. The lower the weight, the greater the chance that traffic will get routed on that link. When one sends or receives data over the Internet, the information is divided into small chunks called packets or datagrams. As a special kind of stochastic search algorithms, genetic algorithm is a problem-solving method which is based on the concept of natural selection and genetics. In the 1970s, Holland first introduced genetic algorithms to explain the adaptive processes of natural systems and to design an artificial system, which retains the robust mechanism of natural systems.

The steps of a GA are shown:

- 1: Choose initial population
- 2: Evaluate the fitness of each individual in the population
- 3: **while** <terminating condition>**do**
- 4: Select best-ranking individuals to reproduce
- 5: Breed new generation through crossover and mutation (genetic operations) and give birth to offspring
- 6: Evaluate the individual fitnesses of the offspring
- 7: Replace worst ranked part of population with offspring
- 8: **end while**

Two main operations involved in a Genetic Algorithm are mutation and crossover.

Crossover: The crossover operation takes two parents to mate. It looks for the common points in the parents. The common nodes are where these two paths intersect. Among the common points, the program selects one of them randomly. It makes the crossover from that point. The crossover operation is illustrated as follows:

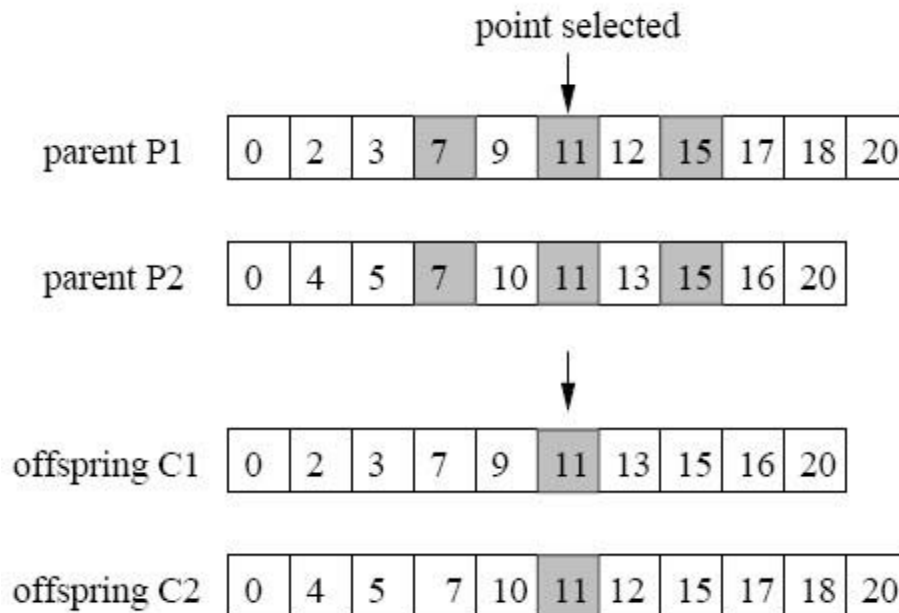


Fig. Crossover Process

Mutation: This operation is applied on the offsprings obtained as a result of the crossover process. Mutation is used to maintain genetic diversity from one generation of a population of chromosomes to next.

CHAPTER 3: REQUIREMENTS & ANALYSIS

3.1 PROBLEM DEFINITION

To demonstrate the use of the proposed method, we use our weighted sum method to solve the ever-popular shortest-path problem. For this we consider two separate graphs (dataset 1 and dataset 2) and apply the method in both these graphs- once using a set of deterministic weights and then using the corresponding probabilistic weights.

The system will give information about the most suitable and optimal path based on several factors. This paper focuses on solving optimization problems using weighted sum technique. More specifically, it focuses on a scientific technique to improve the weights obtained by frequentist approach in solving optimization problems using weighted sum technique. It focuses on a probabilistic manner to generate the weights for conflicting objective functions using a model based on multinomial distribution and Dirichlet prior. Hence in using this method to solve the shortest path problem, we deviate from the traditional method of solving the shortest path problem in the sense that, here, the optimal route(s) obtained depend on more than one factor which ultimately determines the cost of the path whereas traditionally only one factor (mostly distance) determines the cost of the path.

There is a generation of weights according to our requirements and corresponding to each of the three parameters (factors):

1. Distance from a source node to destination node.
2. Average time taken by a car to cover the distance.
3. Availability of one of the following:
 - Parking space (in case of Dataset 1)
 - Passengers (in case of Dataset 2)

Weighted Sum by Bayesian Approach

A Bayesian Network (BN) is a probabilistic reasoning technique, which to date has been used in a broad range of applications. One of the key challenges in constructing a BN is obtaining its Conditional Probability Tables (CPTs). CPTs can be learnt from data (when available), elicited from domain experts, or a combination of both. Eliciting from domain experts provides more flexibility; however, CPTs grow in size of exponentially, thus making their elicitation process very time consuming and costly. Previous work proposed a solution to this problem using the weighted sum algorithm (WSA) [9]; however no empirical results were given on the algorithm's elicitation reduction and prediction accuracy. Hence the aim of this paper is to present two empirical studies that assess the WSA's efficiency and prediction accuracy. Our results show that the estimates obtained using the WSA were highly accurate and make significant reductions in elicitation.

A Bayesian average is a method of estimating the mean of a population using outside information, especially a pre-existing belief that is factored into the calculation. This is a central feature of Bayesian interpretation. This is useful when the available data set is small.

The Bayesian approach is to write down exactly the probability we want to infer, in terms only of the data we know, and directly solve the resulting equation — which forces us to deal explicitly with all mathematical difficulties, additional assumptions and uncertainties that may

arise. One distinctive feature of a Bayesian approach is that if we need to invoke uncertain parameters in the problem, we do not attempt to make point estimates of these parameters; instead, we deal with uncertainty more rigorously, by integrating over all possible values that a parameter might assume. The solution to inverse probability problems is the grandiosely named “Bayes' theorem”, which actually is a trivial algebraic truism for two random variables X and Y :

$$P(X | Y) = \frac{P(Y | X)P(X)}{P(Y)} = \frac{P(Y | X)P(X)}{\sum_{X'} P(Y | X')P(X')}$$

That is, the probability of a particular choice of p given the data (the 'posterior probability' of p) is proportional to the probability that we would get the observed data if that p were true (the 'likelihood' of p) multiplied by the *a priori* probability of this p relative to all other possible values of p (the 'prior probability' of p). To make this come out as a probability, we divide by a summation over all possible values of p ; because p is a continuous variable, this means an integration from $p = 0$ to $p = 1$. The use of inverse probability calculations and Bayes' theorem is a second distinctive feature of Bayesian approaches.

Drawbacks of Weighted Sum

A standard technique for generating the Pareto set in multicriteria optimization problems is to minimize (convex) weighted sums of the different objectives for various different settings of the weights. However, it is well-known that this method succeeds in getting points from all parts of the Pareto set only when the Pareto curve is convex. This article provides a geometrical argument as to why this is the case.

Secondly, it is a frequent observation that even for convex Pareto curves, an evenly distributed set of weights fails to produce an even distribution of points from all parts of the Pareto set. This article aims to identify the mechanism behind this observation. Roughly, the weight is related to the slope of the Pareto curve in the objective space in a way such that an even spread of Pareto points actually corresponds to often very uneven distributions of weights. Several examples are provided showing assumed shapes of Pareto curves and the distribution of weights corresponding to an even spread of points on those Pareto curves.

3.2 PLANNING AND SCHEDULING

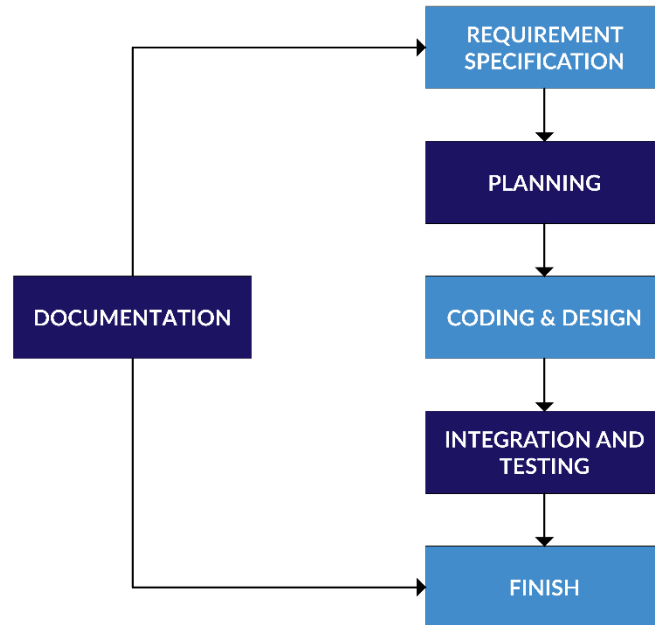


Fig.2: PERT Chart

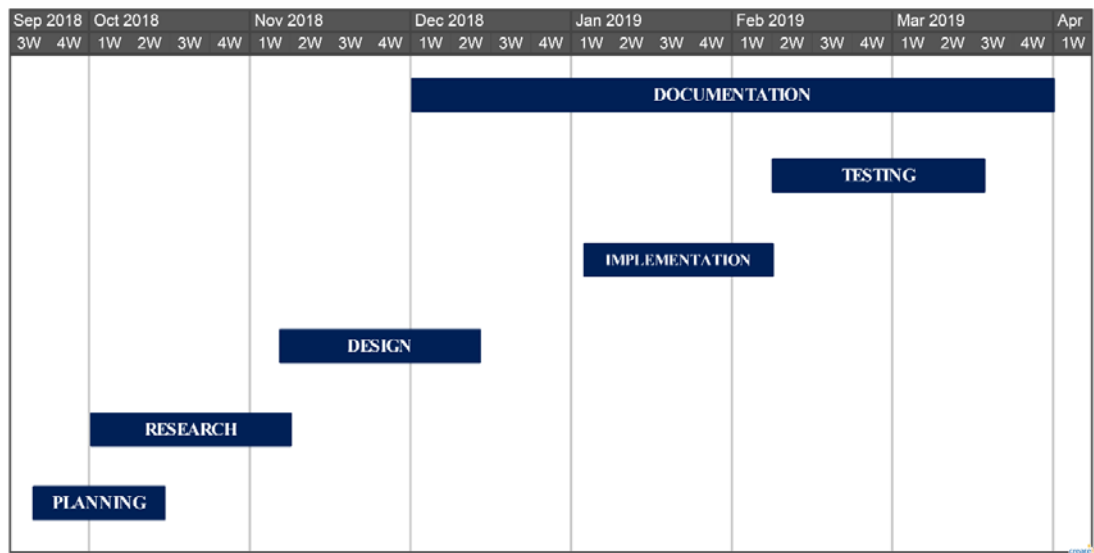


Fig.3: GNATT Chart

3.3 MODEL AND METHODOLOGIES IN DETERMINATION OF WEIGHTS

The process starts by describing the conflicting objectives in the objective space by

$$M = \{f_1(x), f_2(x), \dots, f_l(x); h_i(x) \leq / \geq 0, i=1, 2, \dots, p\}$$

Where $f_1(x), f_2(x), \dots, f_l(x)$ are the conflicting objective functions, $h_i(x) \leq / \geq 0$ denotes the set of p constraints.

The weighted sum method scalarizes the vector objective functions.

$f = (f_1(x), f_2(x), \dots, f_l(x)) \in R^l$, l -dimensional Euclidean space, using the appropriately selected vector of weights $w = (w_1, w_2, \dots, w_l) \in R^l$, such that $w_i \geq 0$ and $\sum_{i=1}^l w_i = 1$.

$$\theta = w'f = w_1f_1(x) + w_2f_2(x) + \dots + w_lf_l(x)$$

It is to be noted that $\theta \in R$, That is, a scalar. Without any loss of generality one can assume the objective functions $f_i(x) \forall i = 1, 2, \dots, l$ to be normalized.

To determine the weights, suppose one obtains data on the preferences of n individuals regarding the choice of different categories (representing different conflicting objectives). Suppose a pilot survey is conducted on individuals voting for the single most important category out of a finite number of mutually exclusive and exhaustive set of choices.

Let n_i = number of individuals who have voted for category i ($i = 1, 2, \dots, l$) represent the i^{th} objective function) in the pilot survey.

w_i s were estimated by the proportion of preference in the respective categories.

$$\hat{w}_i = p_i = n_i / n;$$

$$i = 1, 2, \dots, l$$

$$n = \sum_{i=1}^l n_i$$

The multinomial distribution is used for modeling the probability of counts in the different categories (representing the different objective functions), as the individuals vote independently for exactly one of the l categories.

Then, $(n_1, n_2, \dots, n_l) \sim \text{Multinomial}(n; w_1, w_2, \dots, w_l)$, where w_i is the population proportion of individuals (unknown to us) who will vote for category i or is the probability that a randomly selected individual votes for i^{th} category.

Probability mass function of multinomial distribution is given by:

$$f(n_1, n_2, \dots, n_l | w_1, w_2, \dots, w_l) = \frac{n!}{n_1! n_2! \dots n_l!} w_1^{n_1} w_2^{n_2} \dots w_{l-1}^{n_{l-1}} (1 - w_1 - \dots - w_{l-1})^{n - n_1 - \dots - n_{l-1}} \quad (4.1)$$

It is further assumed that, $(w_1, w_2, \dots, w_l) \sim \text{Dirichlet}(\alpha_1, \alpha_2, \dots, \alpha_l)$ having the following form of density

$$g(w_1, w_2, \dots, w_l | \alpha_1, \alpha_2, \dots, \alpha_l) = \frac{\Gamma(\sum_{i=1}^l \alpha_i)}{\prod_{i=1}^l \Gamma(\alpha_i)} \prod_{i=1}^l w_i^{\alpha_i - 1} \quad (4.2)$$

As w_i 's are continuous random variables with $w_i \geq 0 \forall i$ and $\sum w_i = 1$, here Dirichlet distribution, being a distribution over a probability simplex, is most appropriate for modelling (w_1, w_2, \dots, w_l) where $\alpha_1, \alpha_2, \dots, \alpha_l$ are the concentration parameters such that $\alpha_i > 0 \forall i = 1, 2, \dots, l$.

The marginal likelihood function is given by,

$$h(n_1, n_2, \dots, n_l | \alpha_1, \alpha_2, \dots, \alpha_l) = \int_{w_1, w_2, \dots, w_l} f(n_1, n_2, \dots, n_l | w_1, w_2, \dots, w_l) \cdot g(w_1, w_2, \dots, w_l | \alpha_1, \alpha_2, \dots, \alpha_l) dw_1 dw_2 \dots dw_l$$

$$= \frac{\Gamma(\sum_{j=1}^l \alpha_j)}{\prod_{j=1}^l \Gamma(\alpha_j)} \cdot \frac{n!}{\prod_{j=1}^l n_j!} \cdot \frac{\prod_{j=1}^l \Gamma(n_j + \alpha_j)}{\Gamma(\sum_{j=1}^l \alpha_j + n)} \quad (4.3)$$

Now it can be shown that $[w_1, w_2, \dots, w_l | n_1, n_2, \dots, n_l] \sim \text{Dirichlet}(\alpha_1 + n_1, \alpha_2 + n_2, \dots, \alpha_l + n_l)$

Posterior distribution of the weights given the data, follows Dirichlet distribution with concentration parameters $(\alpha_1 + n_1, \alpha_2 + n_2, \dots, \alpha_l + n_l)$

$$\text{Now, } E(w_i | n_1, n_2, n_3) = \frac{\alpha_i + n_i}{\sum_{i=1}^3 (\alpha_i + n_i)}$$

Hence, posterior expectations of the weights are given by,

$$\hat{w}_i^* = \frac{\hat{\alpha}_i + n_i}{\sum_{i=1}^3 (\hat{\alpha}_i + n_i)}, i = 1, 2, 3 \quad (4.4)$$

The values of $\alpha_1, \alpha_2, \dots, \alpha_l$ which maximizes (4.3) are considered as the estimates.

Hence, the objective function gets modified as follows:

$$\text{Minimization of } f = \hat{w}_1^* f_1 + \hat{w}_2^* f_2 + \hat{w}_3^* f_3$$

It is to be noted that the above modeling technique incorporates the uncertainties in determination of the weights through a stochastic hierarchical model based on multinomial distribution with Dirichlet prior.

Experiment and Results

Comparison of estimate of weights obtained under the stochastic and non-stochastic setup is as follows:
Let the results obtained from the survey be: $n_1=24$, $n_2=11$, $n_3=12$, $n=47$.

Under deterministic setup, the estimated weights are

$$W_1 = n_1/n$$

$$W_2 = n_2/n$$

$$W_3 = n_3/n$$

With error variance as follows: $V(W_i) = \text{Var}(n_i/n) = W_i (1 - W_i)/n$

As $n \sim \text{Binomial}(n, p_i)$ for $i=1, 2, 3$.

$$V(W_1) = W_1 (1 - W_1)/n = 0.005316741$$

$$V(W_2) = W_2 (1 - W_2)/n = 0.003814184$$

$$V(W_3) = W_3 (1 - W_3)/n = 0.004045346$$

Under Bayesian setup,

$$W_1^* = \frac{a_1 + n_1}{\sum_{i=1}^3 (a_i + n_i)} = 0.509$$

$$W_2^* = \frac{a_2 + n_2}{\sum_{i=1}^3 (a_i + n_i)} = 0.235$$

$$W_3^* = \frac{a_3 + n_3}{\sum_{i=1}^3 (a_i + n_i)} = 0.256$$

Although the weights seem to be close enough, if observed carefully the error variance under the stochastic setup. One can see that these observations are much better than under the non-stochastic setup

$$V(W_i^*) = \text{Var}\left(\frac{a_i + n_i}{\sum_{i=1}^3 (a_i + n_i)}\right) = \frac{n W_i^* (1 - W_i^*)}{\{\sum_{i=1}^3 (a_i + n_i)\}^2}$$

$$V(W_1^*) = 0.0003987358$$

$$V(W_2^*) = 0.0002860496$$

$$V(W_3^*) = 0.0003033859$$

Sample size	Frequentist		Bayesian	
n_i	W_i	$V(W_i)$	W_i	$V(W_i^*)$
24	0.510638	0.005316741	0.509	0.0003987358
11	0.234043	0.003814184	0.235	0.0002860496
12	0.255319	0.004045346	0.256	0.0003033859

Table: Comparison of error variance under the frequentist and Bayesian techniques for different sample sizes.

3.4 SOFTWARE AND HARDWARE REQUIREMENTS

3.4.1 HARDWARE REQUIREMENTS:

- Operating Systems: Windows 10/ Windows 7 Service pack1/ Windows Server 2016/ Windows Server 2012 R2/ Windows Server 2012
- RAM: 4 GB
- Disk Space: 2 GB of HDD space for Matlab only, 4-6 GB for a typical installation. A full installation of all MathWorks products may take upto 23 GB of disk space
- Processor: Any Intel or AMD x86-64 processor
- Browser: Internet Explorer 9 and above, Firefox, Google Chrome

3.4.2 SOFTWARE REQUIREMENTS:

- MATLAB (Any standard version)
- Python 2.7 (Preferably using Anaconda Distribution)

3.5 PRELIMINARY PRODUCT DESCRIPTION

The system aims to help a driver choose the optimal path keeping in mind the requirements and preferences of the driver. In **Dataset 1**, the aim is to provide the best path for a tourist travelling from one spot to another so as to fulfil the conditions of minimum time and distance covered as well as ensuring that the driver goes through spots with highest probability of parking the vehicle. **Dataset 2** deals with a carpool driver who wants to maximize the profit while driving through a city. Here the aim will be to pass through the places where probability of finding customers is highest while minimizing the distance and time.

CHAPTER 4: SYSTEM DESIGN

4.1 BASIC MODULES

The project is broadly divided into 4 modules:

1. **Frequentist Weight Determination Module:** This is for determining the frequentist(deterministic) weights from a given set of pre-obtained samples.
2. **Bayesian Weight Determination Module:** This is for determining the Bayesian(stochastic) weights from a given set of pre-obtained samples
3. **Optimum Path Determination Module:** This is used to obtain an optimal path from frequentist weights.
4. **Optimum Path Determination Module:** This is used to obtain an optimal path from Bayesian weights.
5. **Visualization Module:** It creates a graphical visualisation as well as a comparison between the two obtained solutions.

These are implemented using the following 5 components:

1. Matrices.xlsx
2. frequentist.m
3. bayesian.m
4. freq_genetic.py
5. bay_genetic.py
6. chromosome.py
7. import_code.py
8. matrix.py
9. sparse_graph_SET1.m
10. sparse_graph_SET2.m

Where, the **Frequentist Weight Determination Module** is implemented by the **frequentist.m**, the **Bayesian Weight Determination Module** is implemented by the **bayesian.m** and the **Optimum Path Determination Modules** are implemented by **freq_genetic.py**, **bay_genetic.py**, **chromosome.py**, **import_code.py**, **matrix.py** and **Matrix.xlsx** and finally the **Visualisation Module** is implemented by the **sparse_graph.m**.

4.1.1 Matrices.xlsx

This is a Microsoft Excel Workbook which contains 3 sheets containing data on distance, time and availability of parking space individually on each sheet. This component basically contains the data in raw format. This is actually comprised of two separate workbooks:

1. Matrices_SET1 2. Matrices_SET2

Distance Sheet – This sheet includes a Distance matrix which contains raw data on the distinct distances to be covered between various locations in an area.

Time Sheet – This sheet includes a Time matrix which contains raw data on the distinct time taken to travel between various locations in an area.

Availability Sheet – This sheet includes an Availability matrix which contains raw data on the distinct availability of parking space of the locations taken into consideration.

4.1.2 frequentist.m

This module is used for determining the frequentist weights from a given set of pre-obtained samples of survey figures. It estimates the weights of the three parameters Distance, Time and Parking Availability in a deterministic approach, i.e. by the proportion of preference in the respective categories.

4.1.3 bayesian.m

This module is used for determining the Bayesian weights from a given set of pre-obtained samples of the same survey figures. It estimates the weights of the three parameters Distance, Time and Parking Availability using the Bayesian approach considering the concentration parameters obtained by maximizing the marginal likelihood function and passing them as inputs to the Dirichlet function to obtain a random distribution of weights, called as the **Bayesian weights** henceforth.

4.1.4 freq_genetic.py

This module is used for generating an optimal solution if available using the genetic algorithm. It uses the frequentist weights obtained from the frequentist.m module. The optimal solution may or may not be the best solution as it completely depends on the frequentist weights. The solution is in the form of a path along with its fitness value.

4.1.5 bay_genetic.py

This module is used for generating an optimal solution if available using the genetic algorithm. It uses the Bayesian weights obtained from the bayesian.m module. The optimal solution may or may not be the best solution as it completely depends on the Bayesian weights. The solution is in the form of a path along with its fitness value.

4.1.6 chromosome.py

This module is used for returning a chromosome from the population of chromosomes in a vector format for easier handling inside the program. It basically returns a chromosome as an object of the Chromosome class which is part of this module.

4.1.7 import_code.py

This module is used for importing the Distance, Time and Availability of parking space data from the Microsoft Excel sheet into separate matrix forms, i.e. a 2-D array format. As a result, we obtain 3 separate 2-D arrays of Distance, Time and Availability such that it can be used in the program directly.

4.1.8 matrix.py

This module returns the 3 matrices using a single object so that it becomes easier for the main program module to recognise and assign 3 separate variables to the 3 separate matrices of Distance, Time and Availability. Also, it helps in better understanding of the program by a user as 3 different 2-D arrays are visible to the user.

4.1.9 sparse_graph_SET1.m

This module is mainly for visualising the obtained path solutions from the genetic algorithm modules for both the frequentist and Bayesian approaches. It has comprised of the data and settings for DATA SET 1 containing 20 nodes. It helps to compare the two results and check if there exists a difference in the obtained paths as we are trying to find out how the Bayesian weights are affecting the path which was obtained earlier from the deterministic approach.

4.1.10 sparse_graph_SET2.m

This module is mainly for visualising the obtained path solutions from the genetic algorithm modules for both the frequentist and Bayesian approaches. It has comprised of the data and settings for DATA SET 2 containing 39 nodes. It helps to compare the two results and check if there exists a difference in the obtained paths as we are trying to find out how the Bayesian weights are affecting the path which was obtained earlier from the deterministic approach.

4.2 DATA DESIGN

Two different data sets are taken for the purpose of testing the proposed technique.

Dataset 1

This is a graph consisting of 20 nodes. Node 1 and Node 20 are the starting node and destination node respectively. Both node 1 and 20 are major tourist attractions and the path from node 1 to node 20 contains a number of intermediate stops. If one wishes to travel from Node 1 to Node 20 by vehicle, one will have to travel through these intermediate points. The distance between two nodes may not always give the exact time needed to travel between the nodes as many other factors such as condition of the roads, slope, and altitude affect the speed of a vehicle and thereby affect the overall time required. Each node also has its average probability of parking i.e, the average availability of parking spots in that place. The objective here is to travel from node 1 to node 20 in minimum time, covering minimum distance as well as ensuring maximum probability of obtaining a parking spot in each node visited.

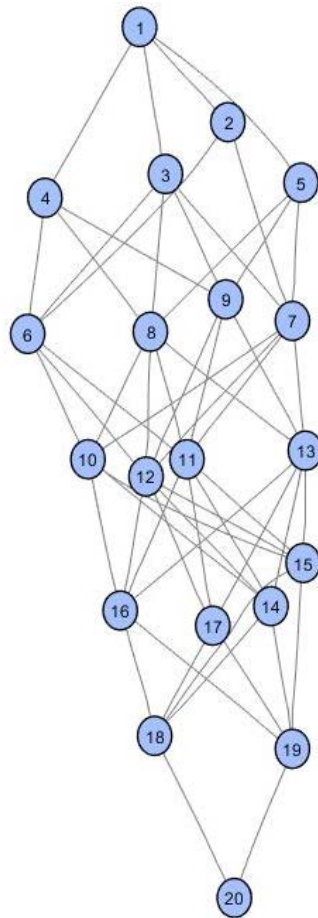


Fig.4: Graph visualization for Dataset 1

Dataset 2

This dataset considers a carpool routing mechanism. It uses a graph with 39 nodes, essentially depicting the topology of a city with 39 major locations. A carpool has to travel from location X to location Y while visiting a number of connecting points and maximize its profit. The factors to be considered in this case are distance between points, time required for a vehicle to travel that distance which is affected by traffic in the roads, the number of traffic signals encountered in the path in addition to distance between the points and the average probability of encountering prospective passengers in the chosen route. It is to be noted that in this dataset the source and destination nodes can be any two points. The objective here is to travel from source node to destination node and maximize the profit for the carpool. To maximize the profit, one will have to cover minimum distance in the minimum possible time and pick up as many numbers of passengers as possible.

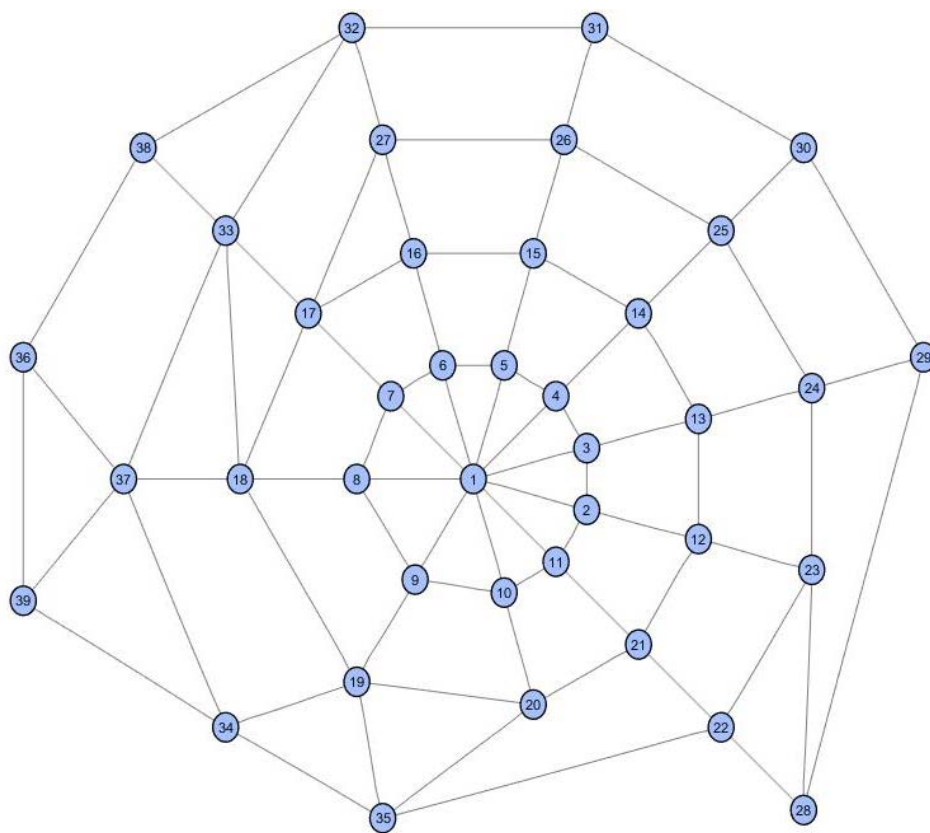


Fig.5: Graph visualization for Dataset 2

4.2.1 DATA INTEGRITY AND CONSTRAINTS

- Data Set 1 - Three matrices of Distance, Time and Parking Availability are taken each containing data about 20 locations (nodes). Thus, it comes down to the 2-D (20x20) array of distance and time and a 1-D (1x20) array of parking availability.
- Data Set 2 - Three matrices of Distance, Time and Parking Availability are taken each containing data about 39 locations (nodes). Thus, it comes down to the 2-D (39x39) array of distance and time and a 1-D (1x39) array of parking availability.
- The path (chromosome) size is taken as a counter starting from 3.
- The max no. of generations is set at 1000.
- Increasing the above numbers could be a possibility to increase the scalability of the system.

4.3 PROCEDURAL DESIGN

4.3.1 LOGIC DIAGRAMS

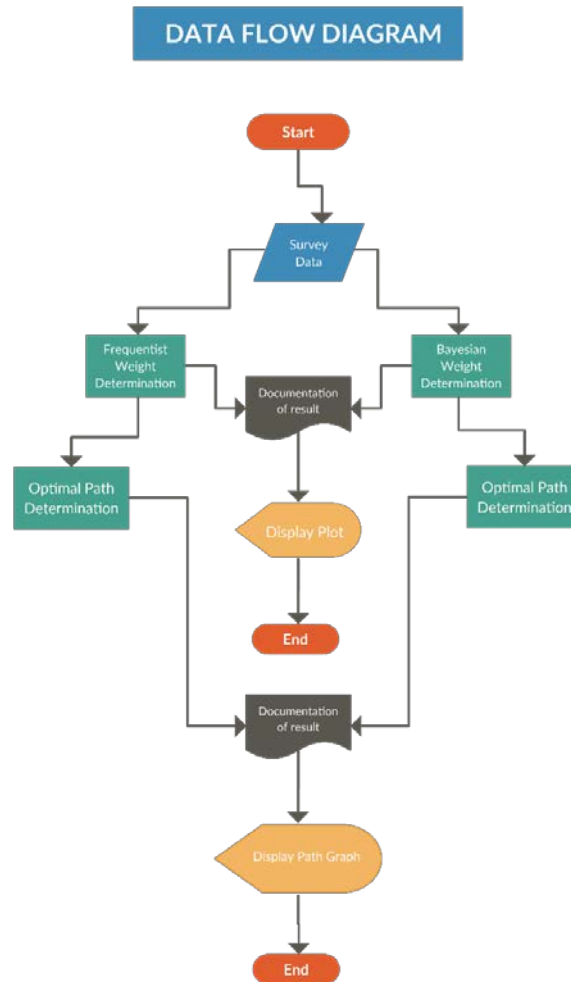


Fig.6: Data Flow Diagram

4.3.2 DATA STRUCTURES

An array is the primary data structure that is used in the MATLAB codes and a list is the primary data structure that is used in the Python codes.

The different types of arrays and lists used in the program are as follows:

- Two separate 2-D lists (matrix) for storing the distance and time taken between each node (location).
- One 1-D list for the availability of a parking space of each node (location).
- One 2-D array to store the weights generated from the Dirichlet distribution.

4.3.3 ALGORITHM DESIGN

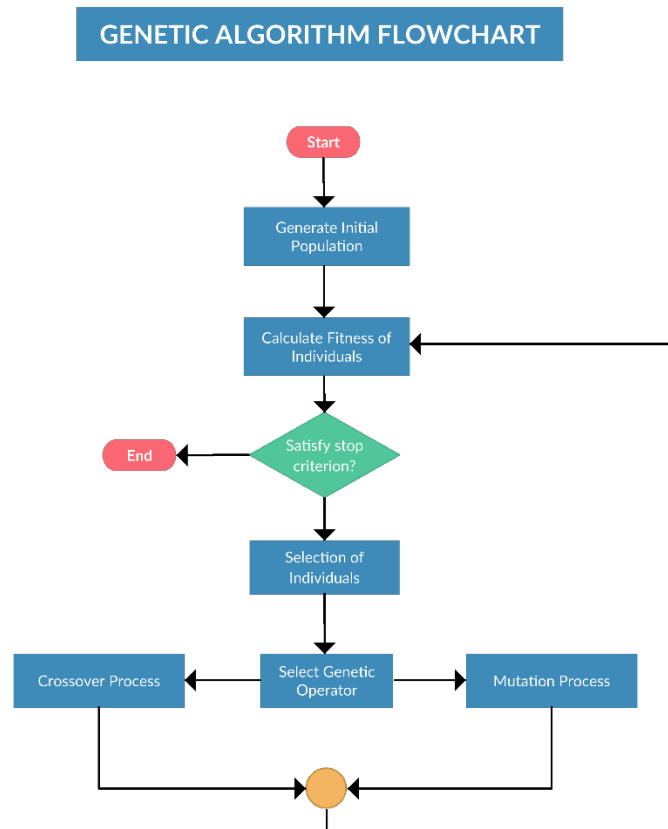


Fig.7: Genetic Algorithm Flowchart

Step 1: Finding Frequentist Weights:

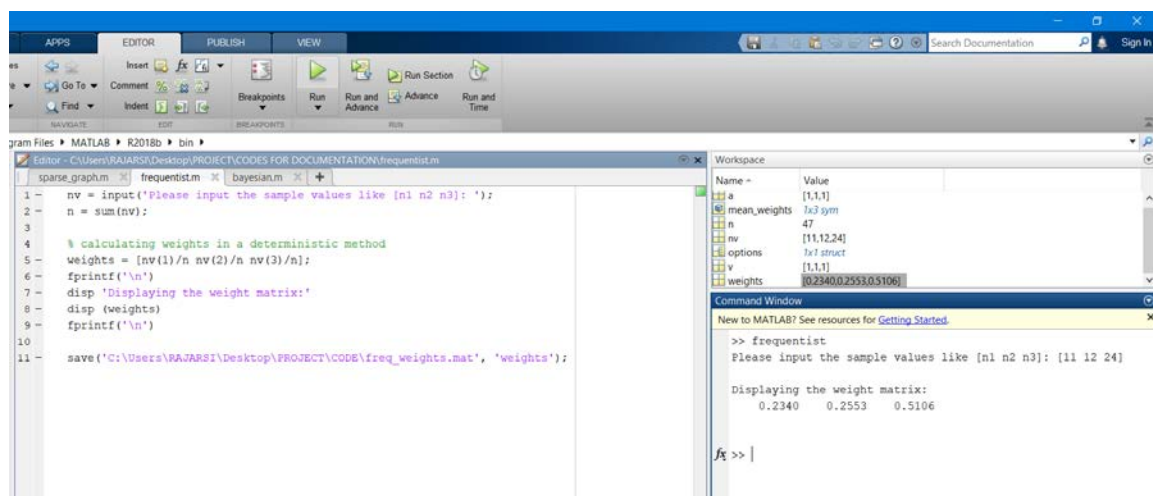


Fig.8: Frequentist Weights

Step 2: Finding Bayesian Weights:

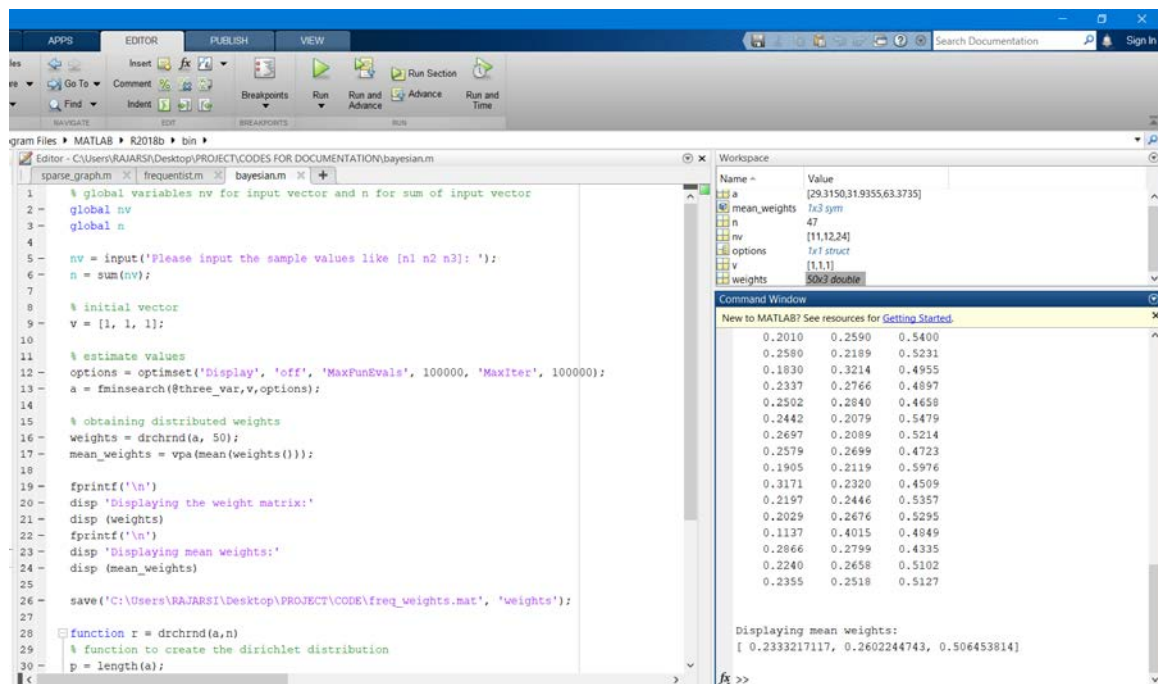


Fig.9: Bayesian Weights

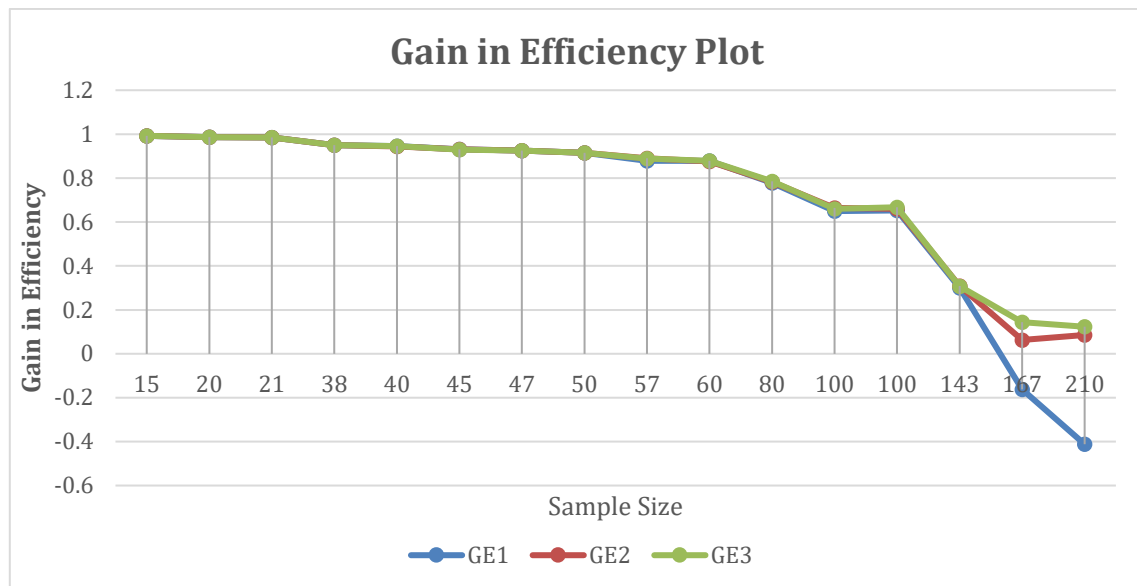


Fig.11: Plot showing gain in efficiency



Fig.10: Plot showing efficiency

Step 3: Finding Optimum path from Frequentist weights:

```

218 def print_chromosomes(self, chromosomes):
219     #function to print chromosomes
220     for chromosome in chromosomes:
221         print str(chromosome) + ' ' + str(self.fitness(chromosome))
222
223 def pretty_print(self, to_print, hint=''):
224     #function to create a print format
225     print ''
226     print '*****'
227     print hint + str(to_print)
228     print '*****'
229
230 if __name__ == '__main__':
231     #main script
232
233     #loading the weight matrix
234     weight_list = scipy.io.loadmat('freq_weights.mat')
235     weight_list = weight_list['weights']
236
237     best = None
238     #running genetic algorithm for all weights
239     for i in range(len(weight_list)):
240         gene_network = GeneNetwork(dim, chromosome_length, weight_list[i, 0, 10])
241
242         #taking 1000 generations
243         res = gene_network.start(1000, 20)
244         if best is None or best[1] > res[1]:
245             best = res
246
247         gene_network.pretty_print(res, 'Solution: ')
248     gene_network = None
249
250 print ''
251 print ''
252 #printing the best solution
253 print 'OPTIMUM SOLUTION WITH BEST FITNESS: '
254 print best
  
```

Python console output:

```

In [1]: runfile('C:/Users/RAJARS/Desktop/PROJECT/CODE/genetic.py', wdir='C:/Users/RAJARS/Desktop/PROJECT/CODE')

Solution: ([0, 2, 7, 12, 16, 18, 19], -0.37555313897616766)

OPTIMUM SOLUTION WITH BEST FITNESS:
([0, 2, 7, 12, 16, 18, 19], -0.37555313897616766)

In [2]:
  
```

Fig.12: Optimum path from Frequentist Weights

Step 4: Finding Optimum path from Bayesian Weights:

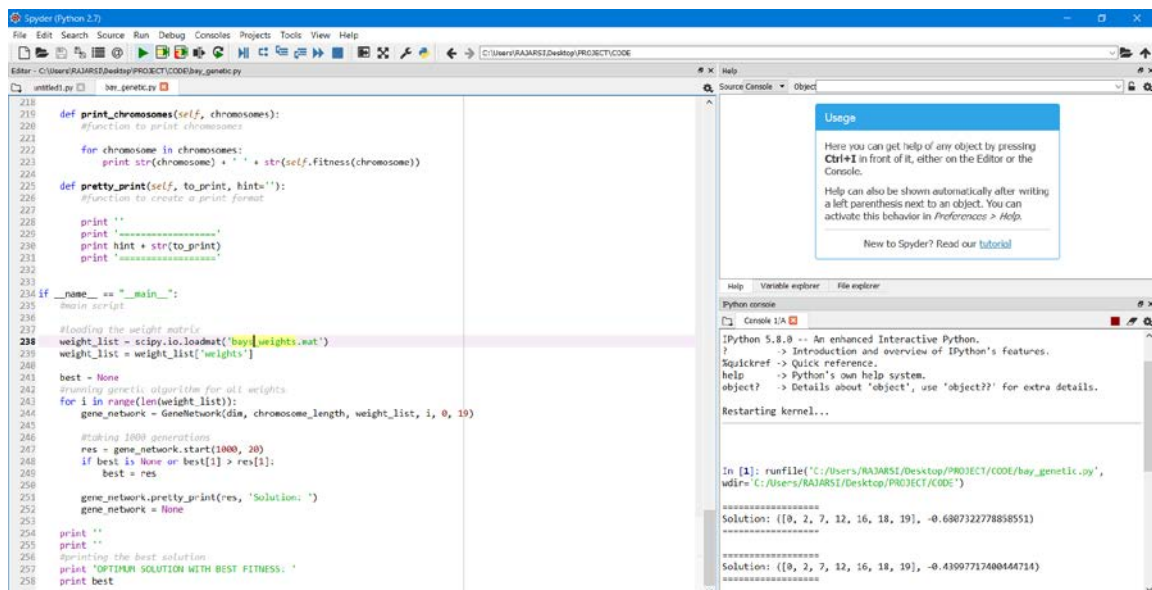


Fig.13: Optimum path from Frequentist Weights (Initial View)

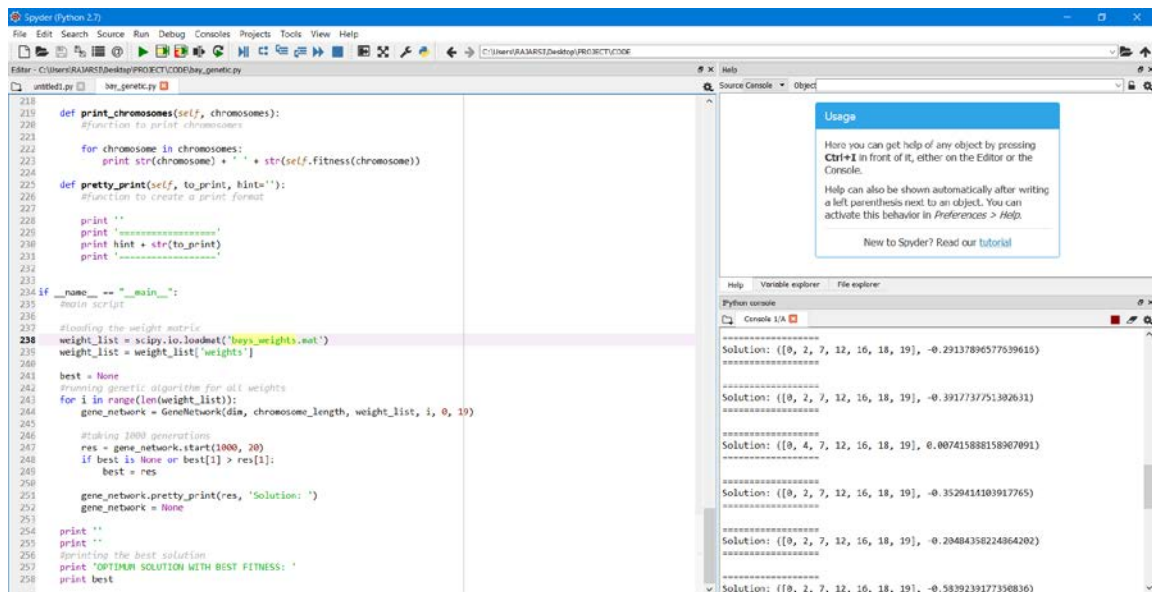


Fig.14: Optimum path from Frequentist Weights (Intermediate View)

```

218 def print_chromosomes(self, chromosomes):
219     #function to print chromosomes
220
221     for chromosome in chromosomes:
222         print str(chromosome) + ' ' + str(self.fitness(chromosome))
223
224 def pretty_print(self, to_print, hint=''):
225     #function to create a print format
226
227     print ""
228     print "*****"
229     print hint + str(to_print)
230     print "*****"
231
232
233
234 if __name__ == "__main__":
235     #main script
236
237     #loading the weight matrix
238     weight_list = scipy.io.loadmat('bay_weights.mat')
239     weight_list = weight_list['weights']
240
241     best = None
242     #running genetic algorithm for all weights
243     for i in range(len(weight_list)):
244         gene_network = GeneNetwork(dim, chromosome_length, weight_list, i, 0, 19)
245
246         #taking 1000 generations
247         res = gene_network.start(1000, 20)
248         if best is None or best[i] > res[i]:
249             best = res
250
251         gene_network.pretty_print(res, 'Solution: ')
252         gene_network = None
253
254     print ""
255     print ""
256     #printing the best solution
257     print "OPTIMUM SOLUTION WITH BEST FITNESS: "
258     print best

```

Python console output:

```

Solution: ([0, 2, 7, 12, 16, 18, 19], -0.40311323667492654)
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.39914553387121476)
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.25985422286968585)
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.7697022000293858)
OPTIMUM SOLUTION WITH BEST FITNESS:
([0, 2, 7, 12, 16, 18, 19], -0.9053980750622799)

```

Fig.15: Optimum path from Frequentist Weights (Result View)

Step 4: Visualising the two solutions through the graph using DATA SET 1:

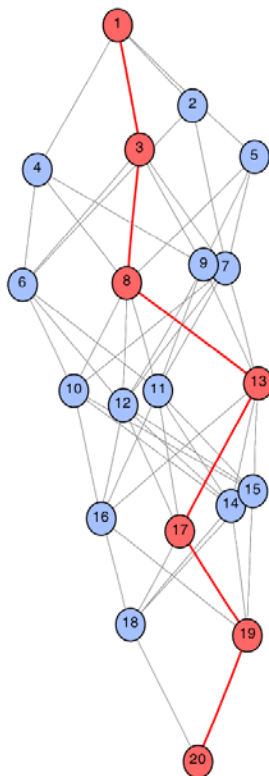


Fig.16(a): Optimal Path Plot – Frequentist (Data Set 1)

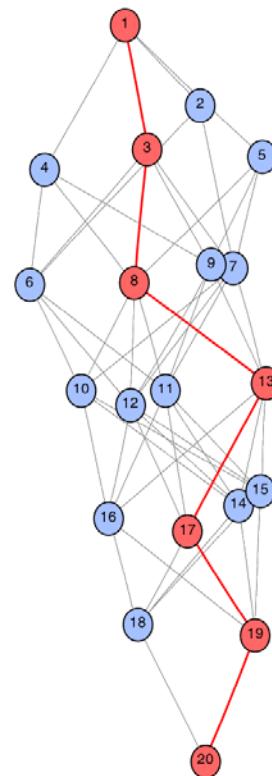


Fig.16(b): Optimal Path Plot – Bayesian (Data Set 1)

The node numbers from the graph are 1 more than the actual node values as the graph plotting program does not accept the value 0 as the initial node value.

CHAPTER 5: IMPLEMENTATION AND TESTING

5.1 IMPLEMENTATION APPROACHES

The data of either of the two maps is stored in Microsoft Excel workbooks and is loaded into the program through 2-D lists. The program proceeds by generating the frequentist weights and Bayesian weights from a previously done survey of varying sample sizes. One of the samples is taken into the program to generate the above-mentioned weights. The weights are then stored in .mat files through internal coding. These .mat files along with the distance, time and availability data is loaded into the python programs and used for determining the optimal paths using the very popular genetic algorithm. The genetic algorithm for a vast extent of weights generate mostly a constant solution. The two solutions generated, i.e. the one through frequentist weights and the other through Bayesian weights are then taken to be used for visualisation through a graph plotting program in MATLAB. Thus, a proper comparison can be done viewing the two graphs side by side. This is not an automated system, since our task is mainly to compare the results between the two obtained solutions.

5.2 CODE DETAILS AND CODE EFFICIENCY

5.2.1 CODING DETAILS

MATLAB SCRIPTS

frequentist.m

```
nv = input('Please input the sample values like [n1 n2 n3]: ');
n = sum(nv);

% calculating weights in a deterministic method
weights = [nv(1)/n nv(2)/n nv(3)/n];
fprintf('\n')
disp('Displaying the weight matrix:')
disp(weights)
fprintf('\n')

save('C:\Users\RAJARSI\Desktop\PROJECT\CODE\freq_weights.mat', 'weights');
```

bayesian.m

```
% global variables nv for input vector and n for sum of input vector
global nv
global n

nv = input('Please input the sample values like [n1 n2 n3]: ');
n = sum(nv);

% initial vector
v = [1, 1, 1];

% estimate values
options = optimset('Display', 'off', 'MaxFunEvals', 100000, 'MaxIter', 100000);
a = fminsearch(@three_var,v,options);

% obtaining distributed weights
weights = drchrnd(a, 50);
mean_weights = vpa(mean(weights()));

fprintf('\n')
disp 'Displaying the weight matrix:'
disp (weights)
fprintf('\n')
disp 'Displaying mean weights:'
disp (mean_weights)

save('C:\Users\RAJARSI\Desktop\PROJECT\CODE\bays_weights.mat', 'weights');

function r = drchrnd(a,n)
% function to create the dirichlet distribution
p = length(a);
r = gamrnd(repmat(a,n,1),1,n,p);
r = r ./ repmat(sum(r,2),1,p);
end

function func = three_var(v)
% function to create the marginal likelihood
global nv
global n

func =
(gamma(sum(v))/(gamma(v(1))*gamma(v(2))*gamma(v(3))))*(factorial(n)/(factorial(nv(1))*factorial(nv(2))*factorial(nv(3))))*((gamma(v(1)+nv(1))*gamma(v(2)+nv(2))*gamma(v(3)+nv(3)))/gamma(v(1)+nv(1)+v(2)+nv(2)+v(3)+nv(3)));
func = -func;
end
```

freq_genetic.py

```
import random
from chromosome import Chromosome
from import_code import Import_Code
import scipy.io

dim = 39
quite = False

class GeneNetwork(object):

    def __init__(self, dim, chromosome_length, weight_list, i, source,
destination):
        #initializing all member variables
        if source >= dim or destination >= dim:
            raise ValueError
        self.chromosome_length = chromosome_length
        self.dim = dim

        self.distance = matrices.distance
        self.time = matrices.time
        self.availability = matrices.availability
        self.normal_dist = self.norm_dist(self.distance)
        self.normal_time = self.norm_time(self.time)
        self.normal_avail = self.norm_avail(self.availability)

        self.source = source
        self.destination = destination
        self.population = []
        self.population_size = 0
        self.results = []
        self.best = None
        self.weight_list = weight_list
        self.i = i
        #weights from external weight list
        self.w1 = self.weight_list[i][0]
        self.w2 = self.weight_list[i][1]
        self.w3 = self.weight_list[i][2]

    def start(self, gen_max, pop_size):
        #function to start the genetic algorithm

        #from first generation
        gen = 1

        #generate initial population
        self.generate_population(pop_size)
        self.population_size = pop_size

        while gen <= gen_max:
            gen += 1
            p = 1
```

```

new_population = list()

while p <= self.population_size:
    p += 1
    #taking two chromosomes at a time
    parents = random.sample(range(self.population_size), 2)
    #calling the random crossover function
    newbie = self.crossover(self.population[parents[0]],
self.population[parents[1]])
    #calling the random mutation function
    newbie.mutate()
    fit = self.fitness(newbie)

    while not (fit > -5):
        #taking two random chromosomes at a time
        parents = random.sample(range(self.population_size), 2)
        #calling the random crossover function
        newbie = self.crossover(self.population[parents[0]],
self.population[parents[1]])
        #calling the random mutation function
        newbie.mutate()
        fit = self.fitness(newbie)

    self.results.append((newbie, fit))
    new_population.append(newbie)

    #checking for the best chromosome
    if self.best is None or self.best[1] > fit:
        self.best = (newbie, fit)

    #selecting best set of chromosomes using selection function
    self.population = self.selection(self.population,
new_population)

    return self.best

def selection(self, prev, now):
    #selection function to select best set of chromosomes

    #adding the new population to the old population
    prev.extend(now)
    #sorting the population according to the fitness values of each
chromosome
    prev.sort(lambda x, y: self.compare(self.fitness(x),
self.fitness(y)))
    #cropping out best set of choromosomes
    population = prev[:self.population_size]
    return population

def norm_dist(self, value):
    #function to calculate normalised distance values

    dist_set = set()

```



```

for row in self.distance:
    for col in row:
        dist_set.add(col)

min_dist = min(dist_set - {10000.0})
max_dist = max(dist_set - {10000.0})
array = [[0 for x in range(dim)] for y in range(dim)]

for i in range(dim):
    for j in range(dim):
        if self.distance[i][j] == 10000.0:
            array[i][j] = 10000.0
        else:
            array[i][j] = (self.distance[i][j] -
min_dist)/(max_dist - min_dist)

return array

def norm_time(self, value):
    #function to calculate normalised time values

    time_set = set()
    for row in self.time:
        for col in row:
            time_set.add(col)

    min_time = min(time_set - {10000.0})
    max_time = max(time_set - {10000.0})
    array = [[0 for x in range(dim)] for y in range(dim)]

    for i in range(dim):
        for j in range(dim):
            if self.time[i][j] == 10000.0:
                array[i][j] = 10000.0
            else:
                array[i][j] = (self.time[i][j] - min_time)/(max_time -
min_time)

    return array

def norm_avail(self, value):
    #function to calculate normalised availability values

    avail_set = set()
    for item in self.availability:
        avail_set.add(item)

    min_avail = min(avail_set - {10000.0})
    max_avail = max(avail_set - {10000.0})
    array = [0 for x in range(dim)]

    for i in range(dim):
        if self.availability[i] == 10000.0:

```

```

        array[i] = 10000.0
    else:
        array[i] = (self.availability[i] - min_avail)/(max_avail -
min_avail)

    return array

def generate_population(self, n):
    #function to generate initial population

    chromosomes = list()
    for i in range(n):
        #calling chromosome generating function
        chromosomes.append(self._gen_chromosome())
    self.population = chromosomes

def _gen_chromosome(self):
    #function to generate chromosomes

    #generating random chromosomes from list excluding source and
destination
    chromosome = random.sample(list(set(range(self.dim)) -
{self.source, self.destination}), self.chromosome_length - 2)

    chromosome.insert(0, self.source)
    chromosome.append(self.destination)

    return Chromosome(chromosome)

def crossover(self, mother, father):
    #function to create new chromosome using crossover method

    mother_list = mother.get()
    father_list = father.get()

    #choosing a random point to make the crossover happen
    cut = random.randint(0, self.chromosome_length - 1)
    child = mother_list[0:cut] + father_list[cut:]

    return Chromosome(child)

def fitness(self, chromosome):
    #function to calculate fitness value of each chromosome

    chromosome_list = chromosome.get()

    #returning fitness value
    return self.w3*self.sum_value(self.normal_dist, chromosome_list) +
self.w1*self.sum_value(self.normal_time, chromosome_list) -
self.w2*self.sum1_value(self.normal_avail, chromosome_list)

def sum_value(self, value, chrom_list):
    #function to calculate sum

```

```

        return sum([value[i][j] for i, j in zip(chrom_list[:-1],
chrom_list[1:])])

    def sum1_value(self, value, chrom_list):
        #function to calculate sum

        return sum([value[j] for i, j in zip(chrom_list[:-1],
chrom_list[1:])])

    def compare(self, item1, item2):
        #function to compare fitness values

        sign = item1 - item2

        if sign < 0:
            return -1
        else:
            return 1

    def print_chromosomes(self, chromosomes):
        #function to print chromosomes

        for chromosome in chromosomes:
            print str(chromosome) + ' ' + str(self.fitness(chromosome))

    def pretty_print(self, to_print, hint=''):
        #function to create a print format

        print ''
        print '======'
        print hint + str(to_print)
        print '======'

if __name__ == "__main__":
    #main script

    #loading the weight matrix
    weight_list = scipy.io.loadmat('freq_weights.mat')
    weight_list = weight_list['weights']

    imp = Import_Code()
    matrices = imp.create_matrices()

    #taking inputs
    source = input('Enter source: ')
    destination = input('Enter destination: ')

    chromosome_length = 3
    best = None

    #running genetic algorithm for all weights

```

```

for i in range(len(weight_list)):
    while not best or best[1] > 10:
        gene_network = GeneNetwork(dim, chromosome_length, weight_list,
i, source, destination)

        #taking 1000 generations
        res = gene_network.start(1000, 50)
        if best is None or best[1] > res[1]:
            best = res
            chromosome_length += 1

        gene_network.pretty_print(res, 'Solution: ')
        gene_network = None
        chromosome_length = 3

print ''
print ''
#printing the best solution
print 'OPTIMUM SOLUTION WITH BEST FITNESS: '
print best

```

bay_genetic.py

```
import random
from chromosome import Chromosome
from import_code import Import_Code
import scipy.io

dim = 39
quite = False

class GeneNetwork(object):

    def __init__(self, dim, chromosome_length, weight_list, i, source,
destination):
        #initializing all member variables
        if source >= dim or destination >= dim:
            raise ValueError
        self.chromosome_length = chromosome_length
        self.dim = dim

        self.distance = matrices.distance
        self.time = matrices.time
        self.availability = matrices.availability
        self.normal_dist = self.norm_dist(self.distance)
        self.normal_time = self.norm_time(self.time)
        self.normal_avail = self.norm_avail(self.availability)

        self.source = source
        self.destination = destination
        self.population = []
        self.population_size = 0
        self.results = []
        self.best = None
        self.weight_list = weight_list
        self.i = i
        #weights from external weight list
        self.w1 = self.weight_list[i][0]
        self.w2 = self.weight_list[i][1]
        self.w3 = self.weight_list[i][2]

    def start(self, gen_max, pop_size):
        #function to start the genetic algorithm

        #from first generation
        gen = 1

        #generate initial population
        self.generate_population(pop_size)
        self.population_size = pop_size

        while gen <= gen_max:
            gen += 1
            p = 1
```

```

new_population = list()

while p <= self.population_size:
    p += 1
    #taking two chromosomes at a time
    parents = random.sample(range(self.population_size), 2)
    #calling the random crossover function
    newbie = self.crossover(self.population[parents[0]],
self.population[parents[1]])
    #calling the random mutation function
    newbie.mutate()
    fit = self.fitness(newbie)

    while not (fit > -5):
        #taking two random chromosomes at a time
        parents = random.sample(range(self.population_size), 2)
        #calling the random crossover function
        newbie = self.crossover(self.population[parents[0]],
self.population[parents[1]])
        #calling the random mutation function
        newbie.mutate()
        fit = self.fitness(newbie)

    self.results.append((newbie, fit))
    new_population.append(newbie)

    #checking for the best chromosome
    if self.best is None or self.best[1] > fit:
        self.best = (newbie, fit)

    #selecting best set of chromosomes using selection function
    self.population = self.selection(self.population,
new_population)

    return self.best

def selection(self, prev, now):
    #selection function to select best set of chromosomes

    #adding the new population to the old population
    prev.extend(now)
    #sorting the population according to the fitness values of each
chromosome
    prev.sort(lambda x, y: self.compare(self.fitness(x),
self.fitness(y)))
    #cropping out best set of choromosomes
    population = prev[:self.population_size]
    return population

def norm_dist(self, value):
    #function to calculate normalised distance values

    dist_set = set()

```

```

for row in self.distance:
    for col in row:
        dist_set.add(col)

min_dist = min(dist_set - {10000.0})
max_dist = max(dist_set - {10000.0})
array = [[0 for x in range(dim)] for y in range(dim)]

for i in range(dim):
    for j in range(dim):
        if self.distance[i][j] == 10000.0:
            array[i][j] = 10000.0
        else:
            array[i][j] = (self.distance[i][j] -
min_dist)/(max_dist - min_dist)

return array

def norm_time(self, value):
    #function to calculate normalised time values

    time_set = set()
    for row in self.time:
        for col in row:
            time_set.add(col)

    min_time = min(time_set - {10000.0})
    max_time = max(time_set - {10000.0})
    array = [[0 for x in range(dim)] for y in range(dim)]

    for i in range(dim):
        for j in range(dim):
            if self.time[i][j] == 10000.0:
                array[i][j] = 10000.0
            else:
                array[i][j] = (self.time[i][j] - min_time)/(max_time -
min_time)

    return array

def norm_avail(self, value):
    #function to calculate normalised availability values

    avail_set = set()
    for item in self.availability:
        avail_set.add(item)

    min_avail = min(avail_set - {10000.0})
    max_avail = max(avail_set - {10000.0})
    array = [0 for x in range(dim)]

    for i in range(dim):
        if self.availability[i] == 10000.0:

```

```

        array[i] = 10000.0
    else:
        array[i] = (self.availability[i] - min_avail)/(max_avail -
min_avail)

    return array

def generate_population(self, n):
    #function to generate initial population

    chromosomes = list()
    for i in range(n):
        #calling chromosome generating function
        chromosomes.append(self._gen_chromosome())
    self.population = chromosomes

def _gen_chromosome(self):
    #function to generate chromosomes

    #generating random chromosomes from list excluding source and
destination
    chromosome = random.sample(list(set(range(self.dim)) -
{self.source, self.destination}), self.chromosome_length - 2)

    chromosome.insert(0, self.source)
    chromosome.append(self.destination)

    return Chromosome(chromosome)

def crossover(self, mother, father):
    #function to create new chromosome using crossover method

    mother_list = mother.get()
    father_list = father.get()

    #choosing a random point to make the crossover happen
    cut = random.randint(0, self.chromosome_length - 1)
    child = mother_list[0:cut] + father_list[cut:]

    return Chromosome(child)

def fitness(self, chromosome):
    #function to calculate fitness value of each chromosome

    chromosome_list = chromosome.get()

    #returning fitness value
    return self.w3*self.sum_value(self.normal_dist, chromosome_list) +
self.w1*self.sum_value(self.normal_time, chromosome_list) -
self.w2*self.sum1_value(self.normal_avail, chromosome_list)

def sum_value(self, value, chrom_list):
    #function to calculate sum

```



```

        return sum([value[i][j] for i, j in zip(chrom_list[:-1],
chrom_list[1:])])

    def sum1_value(self, value, chrom_list):
        #function to calculate sum

        return sum([value[j] for i, j in zip(chrom_list[:-1],
chrom_list[1:])])

    def compare(self, item1, item2):
        #function to compare fitness values

        sign = item1 - item2

        if sign < 0:
            return -1
        else:
            return 1

    def print_chromosomes(self, chromosomes):
        #function to print chromosomes

        for chromosome in chromosomes:
            print str(chromosome) + ' ' + str(self.fitness(chromosome))

    def pretty_print(self, to_print, hint=''):
        #function to create a print format

        print ''
        print '======'
        print hint + str(to_print)
        print '======'

if __name__ == "__main__":
    #main script

    #loading the weight matrix
    weight_list = scipy.io.loadmat('bays_weights.mat')
    weight_list = weight_list['weights']

    imp = Import_Code()
    matrices = imp.create_matrices()

    #taking inputs
    source = input('Enter source: ')
    destination = input('Enter destination: ')

    chromosome_length = 3

    #running genetic algorithm for all weights
    for i in range(len(weight_list)):

```

```

    best = None
    while not best or best[1] > 10:
        gene_network = GeneNetwork(dim, chromosome_length, weight_list,
i, source, destination)

        #taking 1000 generations
        res = gene_network.start(1000, 50)
        if best is None or best[1] > res[1]:
            best = res
            chromosome_length += 1

    gene_network.pretty_print(res, 'Solution: ')
    gene_network = None
    chromosome_length = 3

print ''
print ''
#printing the best solution
print 'OPTIMUM SOLUTION WITH BEST FITNESS: '
print best

```

chromosome.py

```
import random

class Chromosome(object):
    #Chromosome class to convert chromosome to vector
    def __init__(self, vector):
        self.vector = vector

    def __repr__(self):
        return str(self.vector)

    def get(self):
        return self.vector

    def size(self):
        return len(self.vector)

    def path(self):
        return sum(self.vector)

    def mutate(self):
        #function to mutate a chromosome at 3 random points

        pos = random.randint(1, self.size() - 2)
        self.vector[pos] = random.randint(0, 19)

        pos = random.randint(1, self.size() - 2)
        self.vector[pos] = random.randint(0, 19)

        pos = random.randint(1, self.size() - 2)
        self.vector[pos] = random.randint(0, 19)
```

import_code.py

```
import xlrd
from matrix import Matrix

class Import_Code(object):
    #class for importing distance, time and availability matrices from excel
    file

    def create_matrices(self):
        #function to create matrices

        wb = xlrd.open_workbook('Matrices.xlsx')
        sheet1 = wb.sheet_by_index(0)
        sheet2 = wb.sheet_by_index(1)
        sheet3 = wb.sheet_by_index(2)

        matrices = Matrix([[0 for i in xrange(sheet1.ncols)] for i in
xrange(sheet1.nrows)], [[0 for i in xrange(sheet2.ncols)] for i in
xrange(sheet2.nrows)], [0 for i in xrange(sheet3.ncols)])

        for i in range(sheet1.nrows):
            for j in range(sheet1.ncols):
                matrices.time[i][j] = sheet1.cell_value(i,j)

        for i in range(sheet2.nrows):
            for j in range(sheet2.ncols):
                matrices.distance[i][j] = sheet2.cell_value(i,j)

        for i in range(sheet3.ncols):
            matrices.availability[i] = sheet3.cell_value(0,i)

        return matrices
```

matrix.py

```
class Matrix(object):
    #class to convert time, distance and availability into a matrix

    def __init__(self, time, distance, availability):

        self.distance = distance
        self.time = time
        self.availability = availability
```

sparse_graph_SET1.m

```
% taking edges
A=[0 0 0 0 0 1 1 2 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 6 7 7 7 7 8 8 8 9 9 9 10
10 10 10 11 11 11 11 12 12 12 12 13 13 14 14 15 15 16 16 17 18 19];
B=[0 1 2 3 4 5 6 5 6 7 8 5 7 8 6 7 8 9 10 11 9 10 11 12 9 10 11 12 10 11 12
13 14 15 13 14 15 16 13 14 15 16 13 14 15 16 17 18 17 18 17 18 17 18 19 19
19];
A = A + 1;
B = B + 1;

% taking path obtained
path = input('Enter path like [x1, x2,..., xn]: ');
path = path + 1;

W=[1000 52 61 8 16 78 41 84 63 2 99 71 223 73 55 44 88 11 22 33 21 32 43 54
74 85 96 14 64 75 35 66 55 44 91 97 73 19 45 65 25 85 73 37 87 16 86 84 74
76 2 6 7 9 52 25 1000];

%taking node ids
ids={'1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16'
,'17','18','19','20'};

% creating sparse matrix
DG=sparse(A,B,W);

% creating and setting properties of graph
bg=biograph(DG,ids,'ShowWeights','off','LayoutType','equilibrium','ID','Equal weight
graph','EdgeType','curved','ShowArrows','off','NodeAutoSize','off','LayoutScale',2.5,'Label','Graph with equal weights','Description','Graph with
equal weights');
G=view(bg)
set(G.Nodes,'Color',[0.65 0.7559 1],'Shape','Ellipse','Size',[45
45],'LineColor',[0 0 0],'FontSize',20)
set(G.Nodes(path),'Color',[1 0.4 0.4])
edges=getedgesbynodeid(G,get(G.Nodes(path),'ID'));
set(edges,'LineColor',[1 0 0])
set(edges,'LineWidth',1.5)
```

sparse_graph_SET1.m

```
% taking edges
A=[2 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 10 11 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 21 12 13 14 15 16 22 23 24 25 26 27 22 23 24 25 26 27
28 29 30 31 17 32 18 19 34 19 18 37 36 36 34 36 39 38 37 35 38 35];
B=[1 3 4 5 6 7 8 9 10 11 3 4 5 6 7 8 9 10 11 2 12 13 14 15 16 17 18 19 20
21 13 14 15 16 17 18 19 20 21 12 22 23 24 25 26 27 23 24 25 26 27 17 28 28
29 30 31 32 29 30 31 32 33 33 33 34 35 35 37 33 37 38 39 39 37 33 34 20 32
22];

% taking path obtained
path = input('Enter path like [x1, x2,..., xn]: ');
path = path + 1;

W=ones(1,80);

%taking node ids
ids={'1','2','3','4','5','6','7','8','9','10','11','12','13','14','15','16'
,'17','18','19','20','21','22','23','24','25','26','27','28','29','30','31'
,'32','33','34','35','36','37','38','39'};

% creating sparse matrix
DG=sparse(A,B,W);

% creating and setting properties of graph
bg=biograph(DG,ids,'ShowWeights','off','LayoutType','radial','ID','Equal
weight
graph','EdgeType','curved','ShowArrows','off','NodeAutoSize','off','LayoutS
cale',2.5,'Label','Graph with equal weights','Description','Graph with
equal weights');
G=view(bg)
set(G.Nodes,'Color',[0.65 0.7559 1],'Shape','Ellipse','Size',[45
45],'LineColor',[0 0 0],'FontSize',20)
set(G.Nodes(path),'Color',[1 0.4 0.4])
edges=getedgesbynodeid(G,get(G.Nodes(path),'ID'));
set(edges,'LineColor',[1 0 0])
set(edges,'LineWidth',1.5)
```

5.2.1 CODE EFFICIENCY

1. For the best case of the above procedure, starting the genetic algorithm using a fixed chromosome length would be much faster as compared to variable chromosome length as the program will have to keep running the algorithm for multiple chromosome lengths, the result of which is wastage of program running time.
2. A reduction in the population size will also decrease the algorithm runtime but on the other hand would reduce the chances of getting proper output or obtaining valid solutions. Thus, a population size will have to be kept an optimum level such that we attain maximum code efficiency
3. The individual modules of crossover, mutation and also selection from the genetic algorithm as well as the functions to obtain normalised distance, time and availability values are themselves complex in nature.
4. Finally, a complex map with more edges than a simpler one with less edges will again increase the time complexity of the genetic algorithm.

5.3 TESTING APPROACH

5.3.1 UNIT TESTING

The separate units of Bayesian weight determination, optimal path determination using genetic algorithm with the frequentist as well as the Bayesian weights were tested and the results obtained were as follows. These units being an essential part of the algorithm were tested to observe the correctness.

Results for Bayesian weight determination:

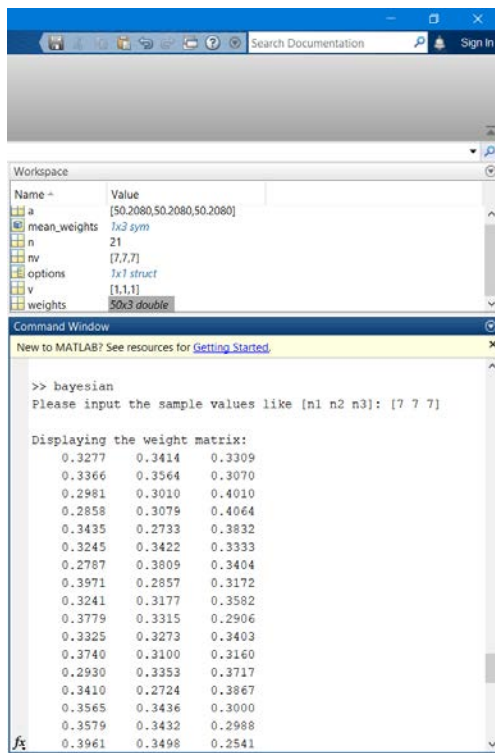


Fig.17(a): Bayesian weight determination
(Initial View)

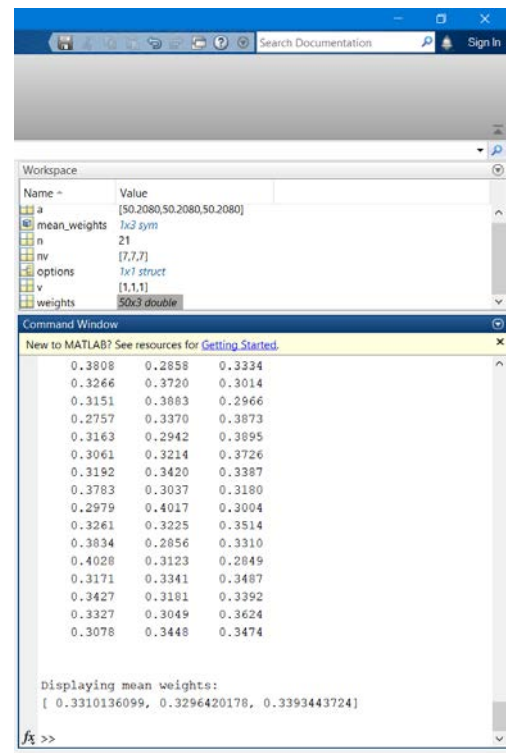
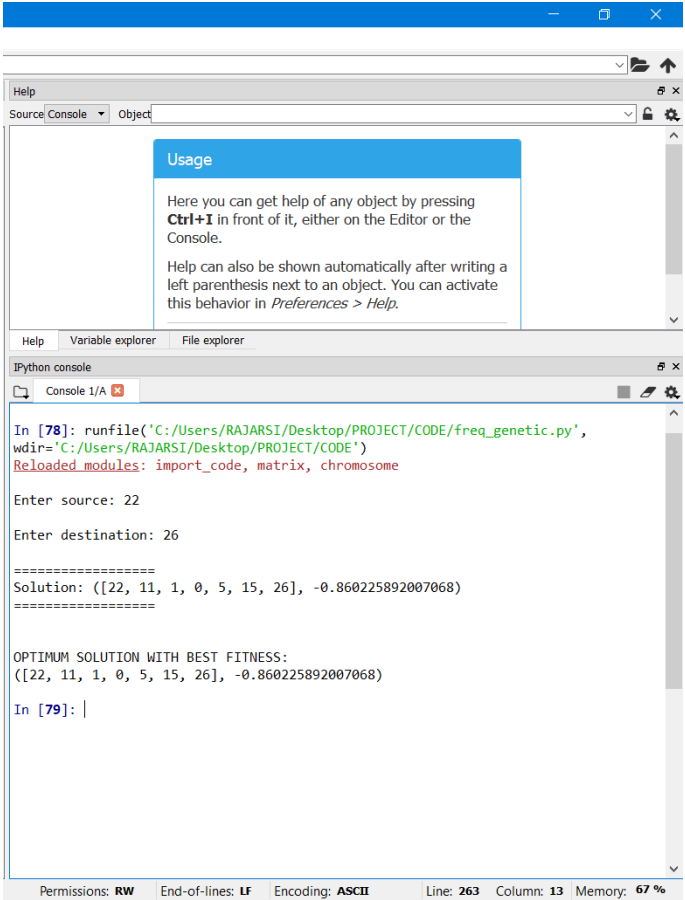


Fig.17(b): Bayesian weight determination
(Result View)

Results for optimal path determination with frequentist weights using genetic algorithm:



```
In [78]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 22
Enter destination: 26
=====
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.860225892007068)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([22, 11, 1, 0, 5, 15, 26], -0.860225892007068)

In [79]: |
```

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 263 Column: 13 Memory: 67 %

Fig.18: Optimal path determination

Results for optimal path determination with Bayesian weights using genetic algorithm:

```

Help
Source Console Object

Usage
Here you can get help of any object by pressing
Ctrl+I in front of it, either on the Editor or the
Console.
Help can also be shown automatically after writing a
left parenthesis next to an object. You can activate
this behavior in Preferences > Help.

Help Variable explorer File explorer

Python console
Users\RA3ARSI\Desktop\PROJECT\CODE*
Reloaded modules: import_code, matrix, chromosome

Enter source: 22
Enter destination: 26

Solution: ([22, 11, 1, 0, 5, 15, 26], -0.8604658013464905)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.6345580095927345)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.7159364150860839)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.8781303498915837)
Solution: ([22, 11, 1, 0, 5, 15, 26], -1.001409470948403)
  
```

Fig.19(a): Optimal path determination (Initial View)

```

Help
Source Console Object

Usage
Here you can get help of any object by pressing
Ctrl+I in front of it, either on the Editor or the
Console.
Help can also be shown automatically after writing a
left parenthesis next to an object. You can activate
this behavior in Preferences > Help.

Help Variable explorer File explorer

Python console
Console 1/4

Solution: ([22, 11, 1, 0, 5, 15, 26], -1.1085761897514954)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.7078278872449721)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.8019556428143091)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.9669171500857245)
Solution: ([22, 11, 1, 0, 5, 15, 26], -0.719428716676898)

OPTIMUM SOLUTION WITH BEST FITNESS:
([22, 11, 1, 0, 5, 15, 26], -0.719428716676898)

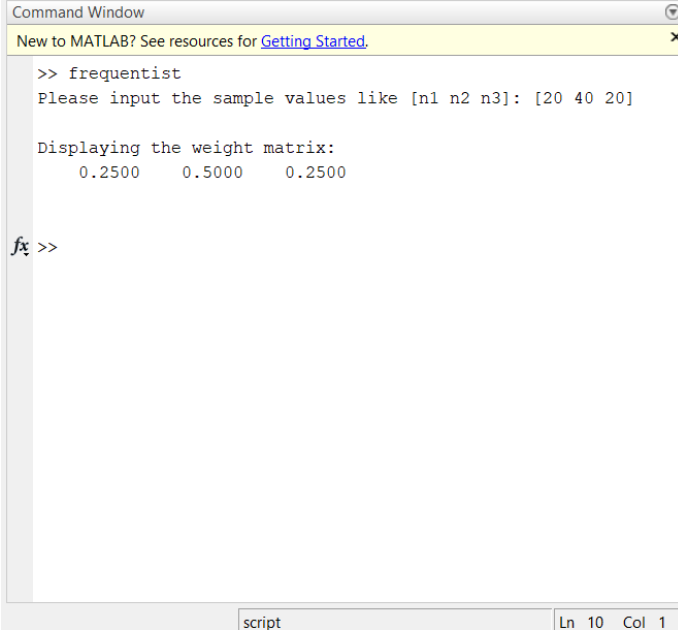
In [2]:
  
```

Fig.19(b): Optimal path determination (Result View)

5.3.1 INTEGRATED TESTING

Once each module was compiled and executed separately, they were integrated to check whether they produced the desired output. Keeping in mind the objectives of the system, the results obtained from entering different sample values from the survey were observed and documented.

Displaying results for sample input: 20 40 20



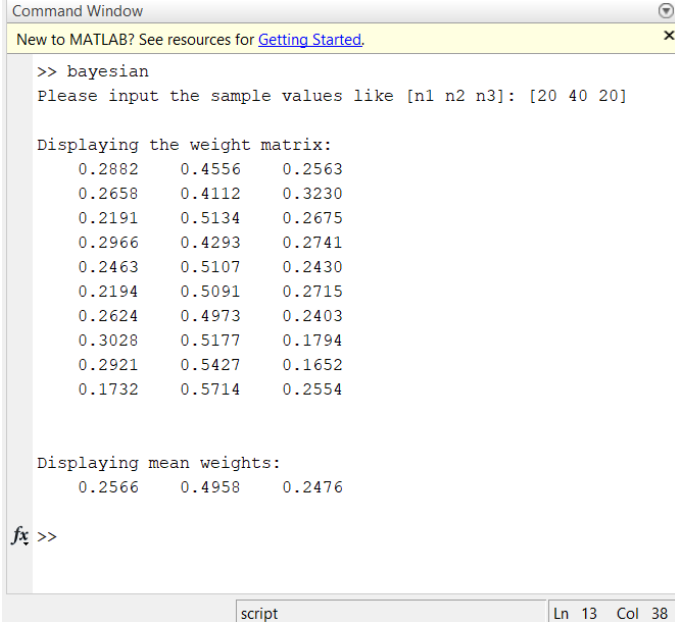
```
Command Window
New to MATLAB? See resources for Getting Started.
>> frequentist
Please input the sample values like [n1 n2 n3]: [20 40 20]

Displaying the weight matrix:
    0.2500    0.5000    0.2500

fx >>
```

The screenshot shows a MATLAB Command Window. At the top, there is a yellow banner with the text "New to MATLAB? See resources for [Getting Started.](#)". Below this, the user has entered the command `>> frequentist`. The prompt "Please input the sample values like [n1 n2 n3]: [20 40 20]" is displayed. The output shows "Displaying the weight matrix:" followed by a 1x3 matrix: `0.2500 0.5000 0.2500`. The status bar at the bottom indicates "script" and "Ln 10 Col 1".

Fig.20: Frequentist weight determination



```
Command Window
New to MATLAB? See resources for Getting Started.
>> bayesian
Please input the sample values like [n1 n2 n3]: [20 40 20]

Displaying the weight matrix:
    0.2882    0.4556    0.2563
    0.2658    0.4112    0.3230
    0.2191    0.5134    0.2675
    0.2966    0.4293    0.2741
    0.2463    0.5107    0.2430
    0.2194    0.5091    0.2715
    0.2624    0.4973    0.2403
    0.3028    0.5177    0.1794
    0.2921    0.5427    0.1652
    0.1732    0.5714    0.2554

Displaying mean weights:
    0.2566    0.4958    0.2476

fx >>
```

The screenshot shows a MATLAB Command Window. At the top, there is a yellow banner with the text "New to MATLAB? See resources for [Getting Started.](#)". Below this, the user has entered the command `>> bayesian`. The prompt "Please input the sample values like [n1 n2 n3]: [20 40 20]" is displayed. The output shows "Displaying the weight matrix:" followed by a 10x3 matrix of values. Below this, it shows "Displaying mean weights:" followed by a 1x3 matrix: `0.2566 0.4958 0.2476`. The status bar at the bottom indicates "script" and "Ln 13 Col 38".

Fig.21: Bayesian weight determination

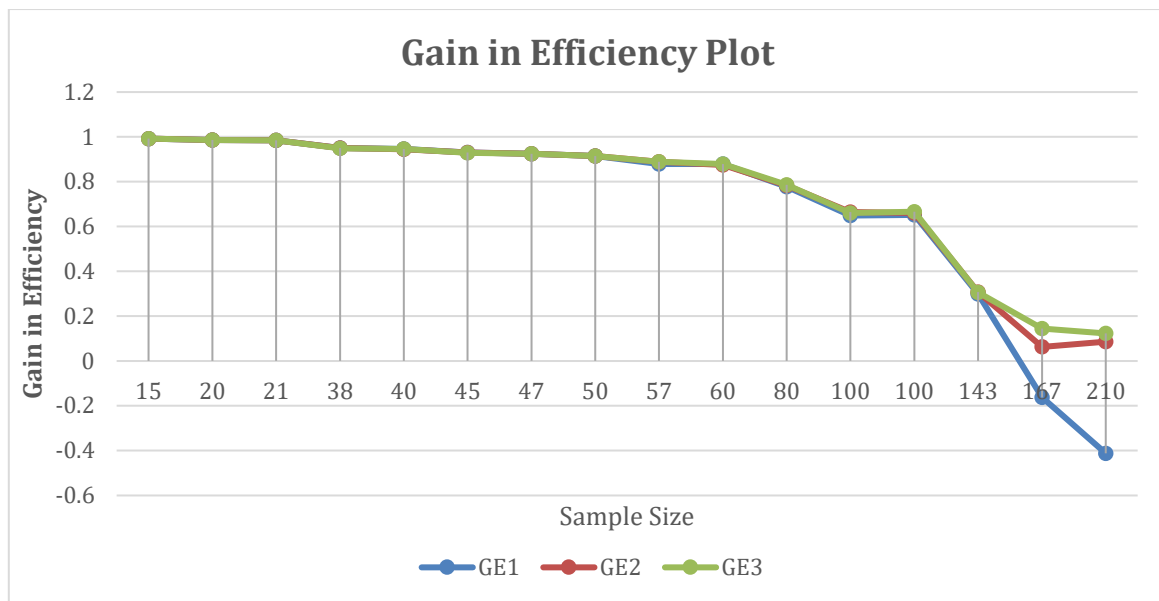


Fig.22: Gain in Efficiency Plot

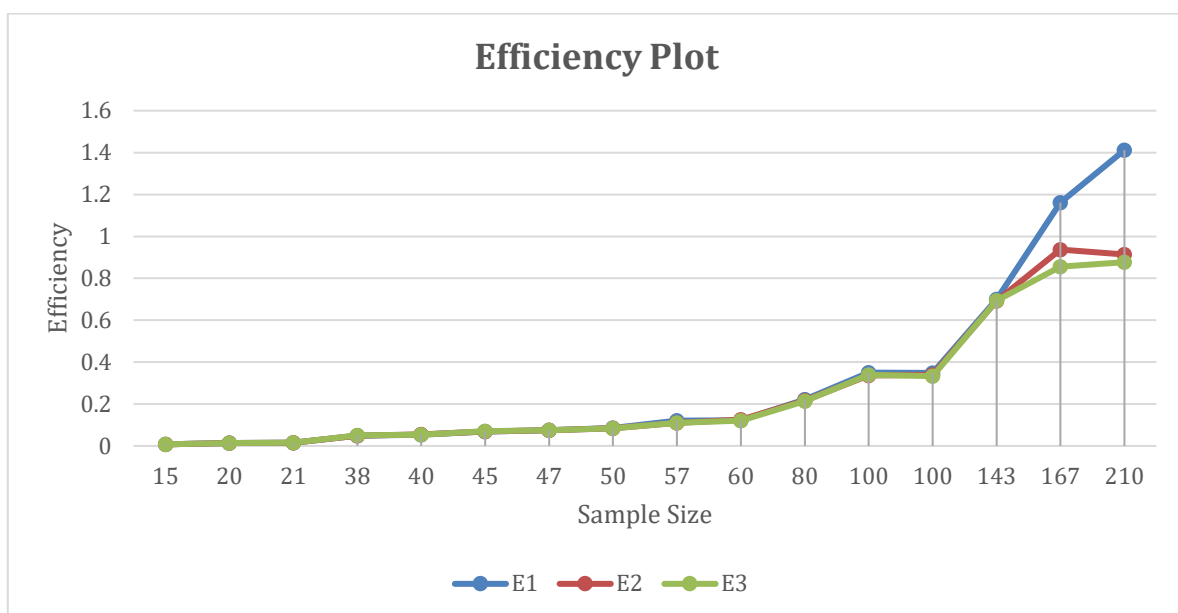


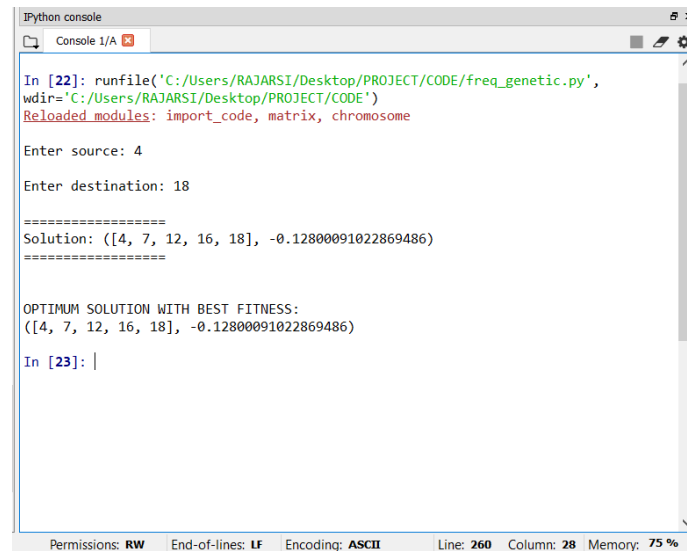
Fig.23: Efficiency Plot

The Graph Plots have been done for values not only for sample 20 30 50, but also for other values. The data from which the graphs have be plotted is as follows:

Sample Size	Input Values	Alphas	Frequentist Weights	Bayesian Weights	Frequentist Error Variance	Bayesian Error Variance	Difference in Error Variance	Gain in Efficiency	Efficiency
15	3 5 7	31.52324 52.20833 72.89243	0.2 0.333333 0.466667	0.207464 0.330588 0.461948	0.010667 0.014815 0.016593	8.37E-05 0.000113 0.000127	0.010583 0.014702 0.016466	0.99715 0.992393 0.992372	0.00785 0.007607 0.007628
20	5 10 5	38.08025 75.56361 38.03014	0.25 0.5 0.25	0.245119 0.491572 0.263309	0.009375 0.0125 0.009375	0.000126 0.00017 0.000132	0.009249 0.01233 0.009243	0.986598 0.986424 0.989551	0.013402 0.013576 0.014049
21	7 7 7	50.20859 50.20739 50.20801	0.333333 0.333333 0.333333	0.344642 0.325333 0.330024	0.010582 0.010582 0.010582	0.000161 0.000156 0.000158	0.010421 0.010426 0.010424	0.984783 0.985212 0.985103	0.015217 0.014788 0.014897
38	10 25 3	35.27795 87.42795 10.9231	0.263158 0.657895 0.078947	0.268439 0.649946 0.081615	0.005103 0.005923 0.001914	0.000253 0.000294 9.67E-05	0.004849 0.005629 0.001817	0.95035 0.950443 0.949466	0.04965 0.049557 0.050534
40	20 12 8	65.56367 39.53709 26.52325	0.5 0.3 0.2	0.488851 0.312948 0.198701	0.00625 0.00525 0.004	0.000339 0.000292 0.000216	0.005911 0.004958 0.003784	0.945706 0.944383 0.946047	0.054294 0.055617 0.053953
45	12 30 3	33.86725 83.92302 8.833734	0.266667 0.666667 0.066667	0.270333 0.661061 0.068806	0.004346 0.004938 0.001383	0.000301 0.000342 9.76E-05	0.004044 0.004596 0.001285	0.930654 0.930682 0.929398	0.069346 0.069318 0.070602
47	11 12 24	29.31499 31.93549 63.37352	0.234043 0.255319 0.510638	0.233745 0.25564 0.509616	0.003814 0.004045 0.005317	0.000286 0.000304 0.000399	0.003528 0.003741 0.004918	0.92507 0.924749 0.924998	0.07493 0.075251 0.075002
50	17 10 23	41.34247 24.52303 55.7585	0.34 0.2 0.46	0.341111 0.199218 0.459671	0.004488 0.0032 0.004968	0.000382 0.000271 0.000422	0.004106 0.002929 0.004546	0.91499 0.915373 0.915133	0.08501 0.084627 0.084867
57	2 30 25	4.452601 60.0483 50.12309	0.035088 0.526316 0.438596	0.038554 0.516842 0.444604	0.00594 0.004374 0.00432	7.17E-05 0.000483 0.000478	0.000522 0.003891 0.003842	0.879234 0.889514 0.889381	0.120766 0.110486 0.110619
60	16 18 26	29.77747 33.69759 48.15395	0.266667 0.3 0.433333	0.264721 0.311521 0.423758	0.003259 0.0035 0.004093	0.000396 0.000437 0.000497	0.002863 0.003063 0.003595	0.878349 0.875174 0.87846	0.121651 0.124826 0.12154
80	20 40 20	23.02991 45.56418 23.02991	0.25 0.5 0.25	0.258011 0.496802 0.245186	0.002344 0.003125 0.002344	0.00052 0.000679 0.000503	0.001824 0.002446 0.001841	0.77815 0.782727 0.785534	0.22185 0.217273 0.214466
100	20 30 50	14.97484 20.88759 35.76157	0.2 0.3 0.5	0.208879 0.294404 0.496717	0.0016 0.0021 0.0025	0.000561 0.000705 0.000849	0.001039 0.001395 0.001651	0.64936 0.664166 0.660511	0.35064 0.335834 0.339489
100	23 48 29	16.62716 34.16083 20.83601	0.23 0.48 0.29	0.238169 0.480657 0.281174	0.001771 0.002496 0.002059	0.000616 0.000847 0.000686	0.001155 0.001649 0.001373	0.652168 0.660462 0.666737	0.347832 0.339538 0.333263
135	10 90 35	3.084675 23.93171 9.607614	0.666667 0.259259 0.101922	0.642316 0.255762 0.101922	0.000508 0.001646 0.001423	0.00042 0.001053 0.000872	8.85E-05 0.000593 0.00055	0.174245 0.360306 0.386719	0.825755 0.639694 0.613281
143	39 45 59	7.868715 9.083338 11.67195	0.314685 0.442587 0.272727	0.276756 0.312816 0.410428	0.001387 0.001508 0.001695	0.000972 0.001044 0.001175	0.000415 0.000464 0.00052	0.299397 0.307992 0.306844	0.700603 0.692008 0.693156
167	20 90 57	0.813387 2.472712 1.337901	0.538972 0.341317 0.152666	0.563765 0.283568 0.00631	0.001488 0.001346 0.000733	0.001394 0.001152 -0.0001	9.36E-05 0.000194 -0.0001	0.06288 0.144392 0.14392	0.93712 0.855608 0.855608
210	40 80 90	1 1 1	0.380952 0.428571 0.190476	0.338657 0.33184 0.329503	0.000734 0.001123 0.001166	0.001037 0.001026 0.001023	-0.0003 9.67E-05 0.000144	-0.41187 0.08611 0.123099	1.411873 0.91389 0.876901

Fig.24: Excel sheet data

Program Run for Data Set 1



```
IPython console
Console 1/A

In [22]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 4

Enter destination: 18

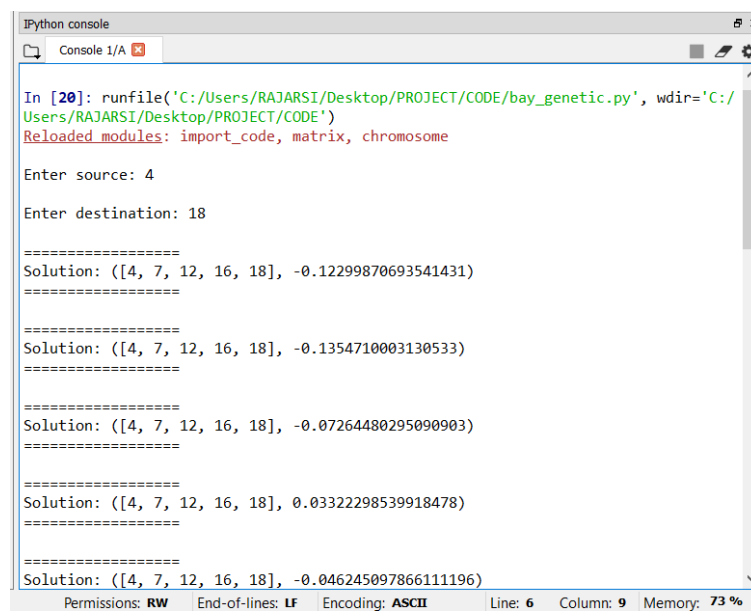
=====
Solution: ([4, 7, 12, 16, 18], -0.12800091022869486)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([4, 7, 12, 16, 18], -0.12800091022869486)

In [23]: |

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 260 Column: 28 Memory: 75 %
```

Fig.25: Optimal Path determination using frequentist weights



```
IPython console
Console 1/A

In [20]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/bay_genetic.py', wdir='C:/
Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 4

Enter destination: 18

=====
Solution: ([4, 7, 12, 16, 18], -0.12299870693541431)
=====

=====
Solution: ([4, 7, 12, 16, 18], -0.1354710003130533)
=====

=====
Solution: ([4, 7, 12, 16, 18], -0.07264480295090903)
=====

=====
Solution: ([4, 7, 12, 16, 18], 0.03322298539918478)
=====

=====
Solution: ([4, 7, 12, 16, 18], -0.046245097866111196)
=====

Permissions: RW End-of-lines: LF Encoding: ASCII Line: 6 Column: 9 Memory: 73 %
```

Fig.26: Optimal Path determination using Bayesian weights (Initial View)

```

Python console
Console 1/A
Solution: ([4, 7, 12, 16, 18], -0.13931048074987823)
=====
Solution: ([4, 7, 12, 16, 18], -0.11037507685136633)
=====
Solution: ([4, 7, 12, 16, 18], -0.29994988023827807)
=====
Solution: ([4, 7, 12, 16, 18], 0.03877442740619946)
=====
Solution: ([4, 7, 12, 16, 18], -0.07438267096205758)
=====
OPTIMUM SOLUTION WITH BEST FITNESS:
([4, 7, 12, 16, 18], -0.07438267096205758)
In [21]:
Permissions: RW End-of-lines: LF Encoding: ASCII Line: 6 Column: 9 Memory: 73 %

```

Fig.27: Optimal Path determination using Bayesian weights (Result View)

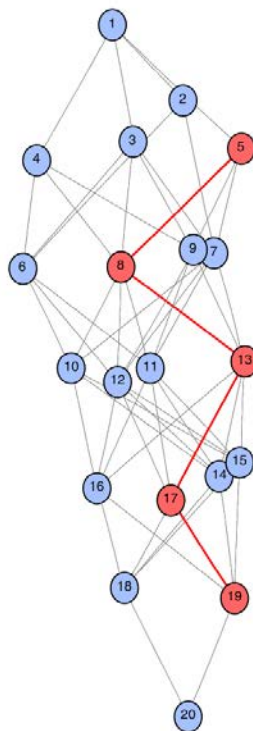


Fig.28(a): Optimal Path Plot for frequentist weight

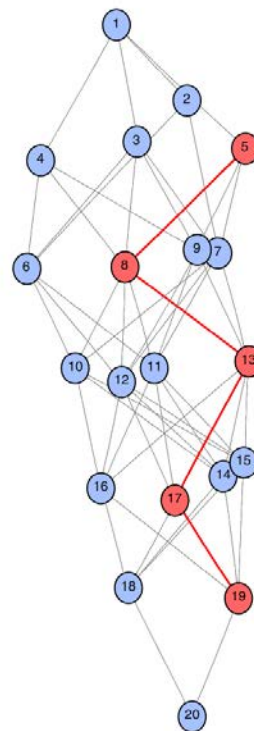
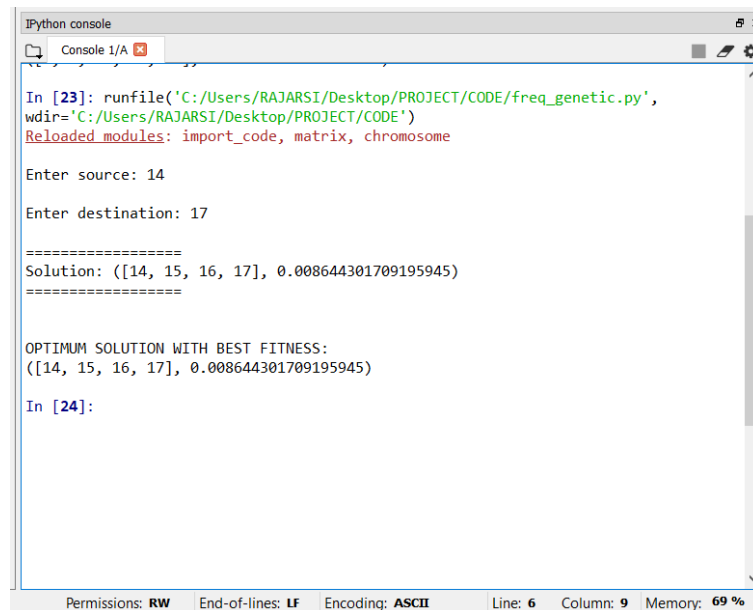


Fig.28(b) Optimal Path Plot for Bayesian weights

The node numbers from the graph are 1 more than the actual node values as the graph plotting program does not accept the value 0 as the initial node value.

Program Run for Data Set 2



```
IPython console
Console 1/A

In [23]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 14

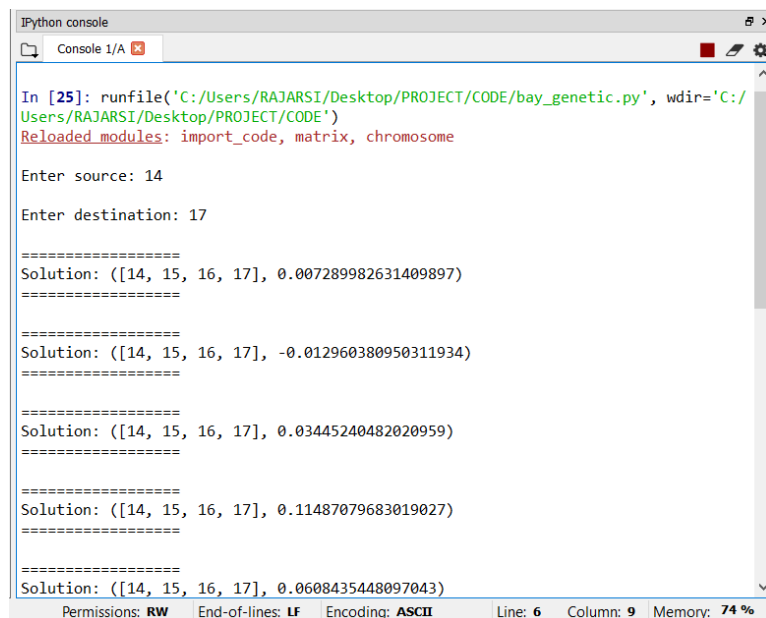
Enter destination: 17

=====
Solution: ([14, 15, 16, 17], 0.008644301709195945)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([14, 15, 16, 17], 0.008644301709195945)

In [24]:
```

Fig.29: Optimal Path determination using frequentist weights



```
IPython console
Console 1/A

In [25]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/bay_genetic.py', wdir='C:/
Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 14

Enter destination: 17

=====
Solution: ([14, 15, 16, 17], 0.007289982631409897)
=====

=====
Solution: ([14, 15, 16, 17], -0.012960380950311934)
=====

=====
Solution: ([14, 15, 16, 17], 0.03445240482020959)
=====

=====
Solution: ([14, 15, 16, 17], 0.11487079683019027)
=====

=====
Solution: ([14, 15, 16, 17], 0.0608435448097043)
=====
```

Fig.30: Optimal Path determination using Bayesian weights (Initial View)


```

IPython console
Console 1/A

=====
Solution: ([14, 15, 16, 17], -0.0055835010307968)
=====

=====
Solution: ([14, 15, 16, 17], 0.06794975096543387)
=====

=====
Solution: ([14, 15, 16, 17], -0.09394335867283696)
=====

=====
Solution: ([14, 15, 16, 17], 0.11522723311069075)
=====

=====
Solution: ([14, 15, 16, 17], 0.057211101274543735)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([14, 15, 16, 17], 0.057211101274543735)

In [26]:
Permissions: RW End-of-lines: LF Encoding: ASCII Line: 6 Column: 9 Memory: 75 %

```

Fig.31: Optimal Path determination using Bayesian weights (Result View)

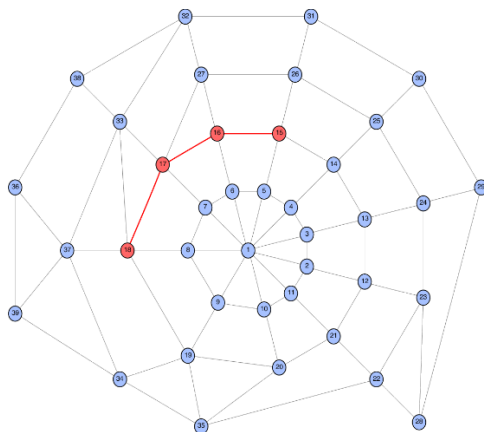


Fig.32(a): Optimal Path Plot for frequentist weight

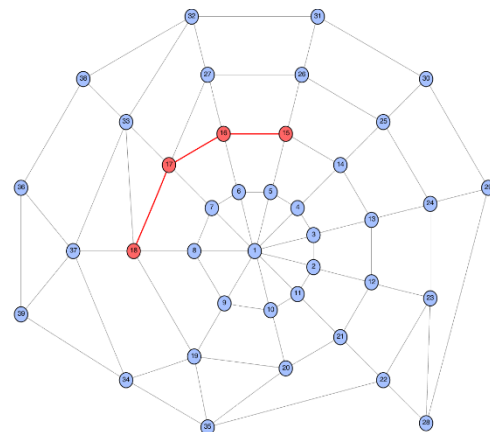


Fig.32(b): Optimal Path Plot for Bayesian weight

The node numbers from the graph are 1 more than the actual node values as the graph plotting program does not accept the value 0 as the initial node value.

5.4 MODIFICATIONS AND IMPROVEMENTS

Domain: The initial map used was found to be restrictive and did not produce enough variation in the routes. Also, the number of edges were small. Thus, the interconnection of the map was increased by increasing the intermediate links between the nodes.

Genetic Algorithm: Initially it was observed that the algorithm was producing solutions for a fixed chromosome length. Thus, it was changed and a variable counter was set to control the chromosome length to produce a solution with the minimum nodes.

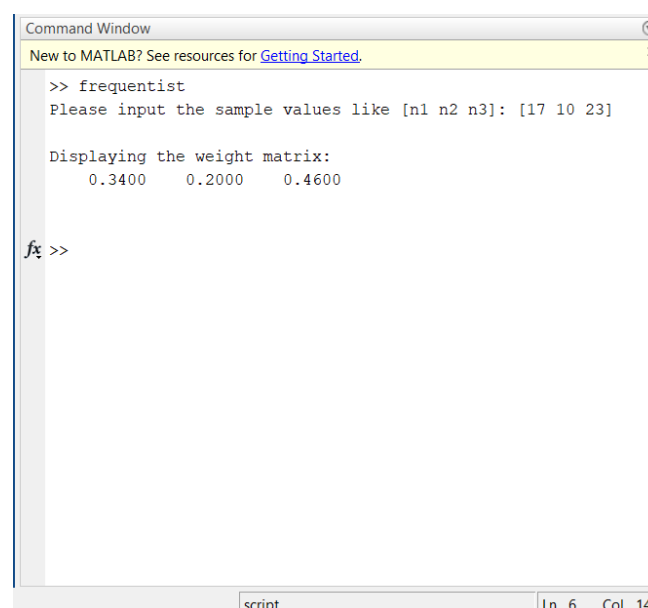
Visualisation: The optimal path solutions that were being obtained couldn't be visualised initially due to lack of knowledge on software. Thus, MATLAB was used later on during the progress of project to plot the optimal solution in a graph to visualize and compare the solutions.

CHAPTER 6: RESULTS AND DISCUSSION

6.1 TEST DESIGN AND REPORTS

The test cases have been designed keeping in mind the restraints and limitations of the project. To make error detection and their correction easy and comprehensive, test cases were selected accordingly. Based on the test cases the dimension of the structural element used in the program has been decided through trial and error method.

TEST CASE 1: Input Values: $n1 = 17$, $n2 = 10$, $n3 = 23$ and DATA SET 2 where the weights give us valid solutions for both frequentist and Bayesian approaches.

A screenshot of the MATLAB Command Window. At the top, there is a yellow banner that says "New to MATLAB? See resources for [Getting Started](#)". Below this, the user has entered the command ">> frequentist". The window then displays the prompt "Please input the sample values like [n1 n2 n3]: [17 10 23]". After the user enters these values, the window displays "Displaying the weight matrix:" followed by a 1x3 matrix: 0.3400 0.2000 0.4600. At the bottom of the window, there is a status bar that says "script" and "Ln 6 Col 14".

```
Command Window
New to MATLAB? See resources for Getting Started
>> frequentist
Please input the sample values like [n1 n2 n3]: [17 10 23]

Displaying the weight matrix:
    0.3400    0.2000    0.4600

fx >>
```

Fig.33: Frequentist weight determination

```

Command Window
New to MATLAB? See resources for Getting Started.
>> bayesian
Please input the sample values like [n1 n2 n3]: [17 10 23]

Displaying the weight matrix:
    0.3470    0.1842    0.4687
    0.3472    0.1764    0.4765
    0.2893    0.1981    0.5126
    0.3533    0.1930    0.4537
    0.3672    0.1566    0.4761
    0.2893    0.2642    0.4464
    0.4066    0.1545    0.4389
    0.3300    0.2170    0.4530
    0.3213    0.2334    0.4453
    0.4144    0.1194    0.4662

Displaying mean weights:
    0.3466    0.1897    0.4637

fx >>
script / three_var      Ln 36  Col 29

```

Fig.34: Bayesian weight determination

```

IPython console
Console 1/A
In [38]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 16
Enter destination: 24
=====
Solution: ([16, 15, 14, 13, 24], -0.08656258631423042)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([16, 15, 14, 13, 24], -0.08656258631423042)

In [39]:
missions: RW  End-of-lines: LF  Encoding: ASCII  Line: 73  Column: 46  Memory: 64 %

```

Fig.35: Optimal Path determination using frequentist weights

```

Python console
Console 1/A
Solution: ([16, 15, 14, 13, 24], -0.21094957291126676)
=====
Solution: ([16, 15, 14, 13, 24], -0.015139202178322764)
=====
Solution: ([16, 15, 14, 13, 24], -0.12123929628776164)
=====
Solution: ([16, 15, 14, 13, 24], -0.15547668792652247)
=====
Solution: ([16, 15, 14, 13, 24], 0.06359913830406025)
=====
OPTIMUM SOLUTION WITH BEST FITNESS:
([16, 15, 14, 13, 24], 0.06359913830406025)
In [41]: |

```

Fig.36: Optimal Path determination using Bayesian weights

Thus, the solution received from both frequentist and Bayesian weights is: [16, 15, 14, 13, 24]. The graphs for both are given below.

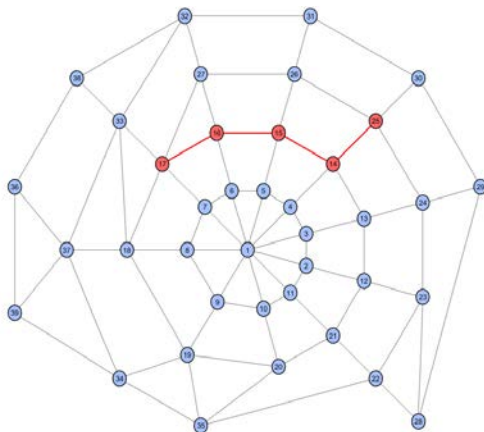


Fig.37(a): Optimal Path Plot for frequentist weights

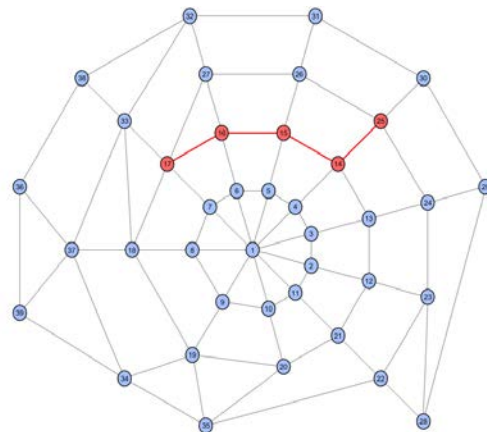
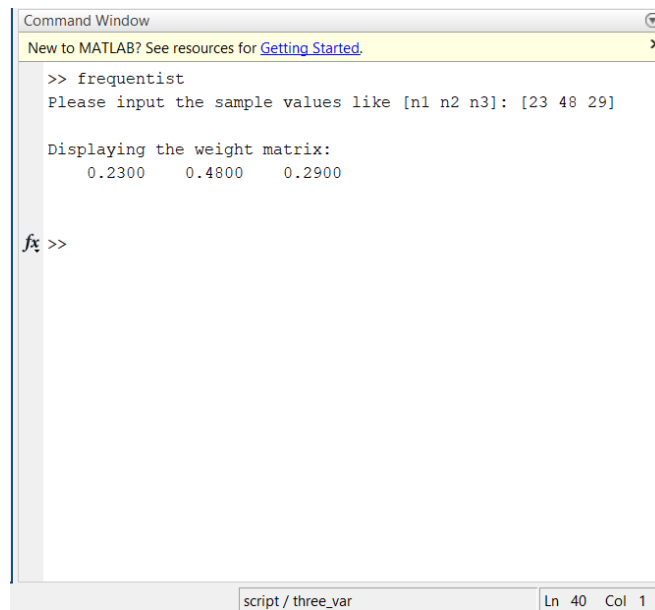


Fig.37(b): Optimal Path Plot for Bayesian weights

The node numbers from the graph are 1 more than the actual node values as the graph plotting program does not accept the value 0 as the initial node value.

TEST CASE 2: Input Values: $n1 = 23$, $n2 = 48$, $n3 = 29$ and DATA SET 2 which leads to a partial success.



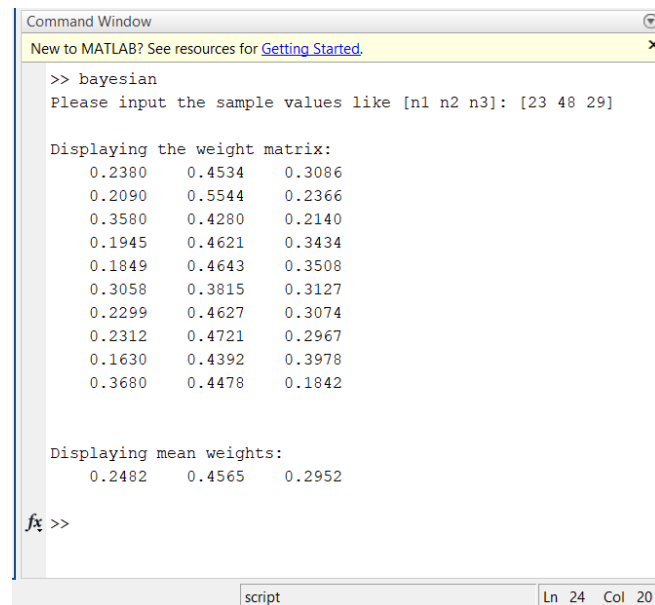
```
Command Window
New to MATLAB? See resources for Getting Started.
>> frequentist
Please input the sample values like [n1 n2 n3]: [23 48 29]

Displaying the weight matrix:
    0.2300    0.4800    0.2900

fx >>
```

script / three_var Ln 40 Col 1

Fig.38: Frequentist weight determination



```
Command Window
New to MATLAB? See resources for Getting Started.
>> bayesian
Please input the sample values like [n1 n2 n3]: [23 48 29]

Displaying the weight matrix:
    0.2380    0.4534    0.3086
    0.2090    0.5544    0.2366
    0.3580    0.4280    0.2140
    0.1945    0.4621    0.3434
    0.1849    0.4643    0.3508
    0.3058    0.3815    0.3127
    0.2299    0.4627    0.3074
    0.2312    0.4721    0.2967
    0.1630    0.4392    0.3978
    0.3680    0.4478    0.1842

Displaying mean weights:
    0.2482    0.4565    0.2952

fx >>
```

script Ln 24 Col 20

Fig.39: Bayesian weight determination

```

IPython console
Console 1/A
File "<ipython-input-3-695c4d37e7c1>", line 1, in <module>
    runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
    wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')

File "C:\ProgramData\Anaconda2\lib\site-packages\spyder_kernels\customize
\spydercustomize.py", line 704, in runfile
    execfile(filename, namespace)

File "C:\ProgramData\Anaconda2\lib\site-packages\spyder_kernels\customize
\spydercustomize.py", line 93, in execfile
    exec(compile(scripttext, filename, 'exec'), glob, loc)

File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 251,
in <module>
    gene_network = GeneNetwork(dim, chromosome_length, weight_list, i,
    source, destination)

File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 14, in
_init_
    raise ValueError

ValueError

In [4]:
In [4]:

```

Fig.40: Result for invalid destination value

This case did not produce any result and was a partial success as the destination value entered was more than what existed in the dataset. Therefore, a **ValueError was raised** in the above test case. Since, this dataset only has 39 nodes, the source or destination cannot be higher than 39. Moreover, the source and destination values cannot be lesser than 0 as no such node exists.

TEST CASE 3: Input Values: $n1 = 10$, $n2 = 90$, $n3 = 35$ and DATA SET 2 which leads to a partial success.

```

Command Window
New to MATLAB? See resources for Getting Started.
>> frequentist
Please input the sample values like [n1 n2 n3]: [10 90 35]

Displaying the weight matrix:
    0.0741    0.6667    0.2593

fx >>

```

Fig.41: Frequentist weight determination

```

Command Window
New to MATLAB? See resources for Getting Started.
>> bayesian
Please input the sample values like [n1 n2 n3]: [10 90 35]

Displaying the weight matrix:
    0.2243    0.5996    0.1760
    0.0615    0.6726    0.2659
    0.0769    0.6787    0.2445
    0.1417    0.6989    0.1594
    0.0520    0.5537    0.3942
    0.0326    0.7149    0.2525
    0.1152    0.6240    0.2609
    0.1614    0.5290    0.3096
    0.1130    0.4706    0.4164
    0.1417    0.5788    0.2794

Displaying mean weights:
    0.1120    0.6121    0.2759

fx >>
script Ln 21 Col 15

```

Fig.42: Bayesian weight determination

```

JPython console
Console 1/A
File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 254,
in <module>
    res = gene_network.start(1000, 50)
File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 45, in
start
    self.generate_population(pop_size)
File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 162,
in generate_population
    chromosomes.append(self._gen_chromosome())
File "C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py", line 169,
in _gen_chromosome
    chromosome = random.sample(list(set(range(self.dim)) - {self.source,
self.destination}), self.chromosome_length - 2)
File "C:\ProgramData\Anaconda2\lib\random.py", line 325, in sample
    raise ValueError("sample larger than population")
ValueError: sample larger than population

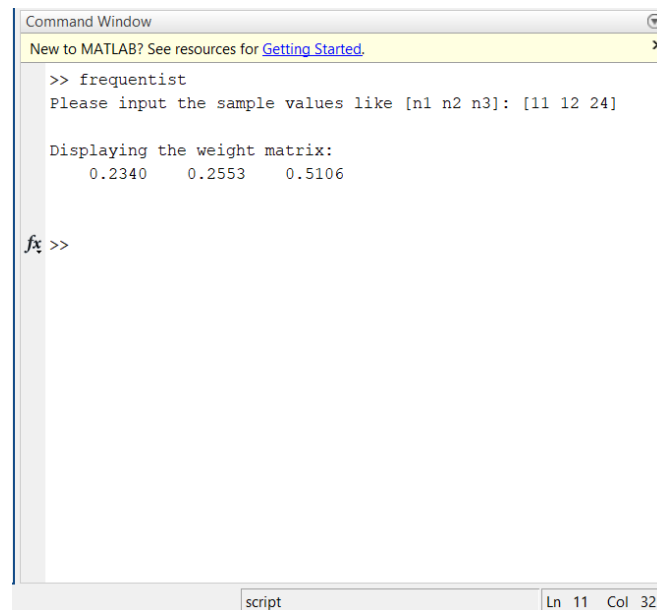
In [35]:
In [35]:
emissions: RW End-of-lines: LF Encoding: ASCII Line: 43 Column: 9 Memory: 68 %

```

Fig.43: Result for non-compatible sample values

The weights produced by this sample set does not produce a valid solution and sets the program into an almost infinite loop till the above error, i.e. **ValueError: sample larger than population** sets in to break the program run.

TEST CASE 4: Input Values: $n1 = 11$, $n2 = 12$, $n3 = 24$ and DATA SET 1 where the weights give us valid solutions for both frequentist and Bayesian approaches.



The image shows a MATLAB Command Window with a yellow header bar that says "New to MATLAB? See resources for [Getting Started](#)". The command prompt shows the execution of the 'frequentist' function. It prompts the user to input sample values like [n1 n2 n3]: [11 12 24]. It then displays the weight matrix as a 1x3 row vector.

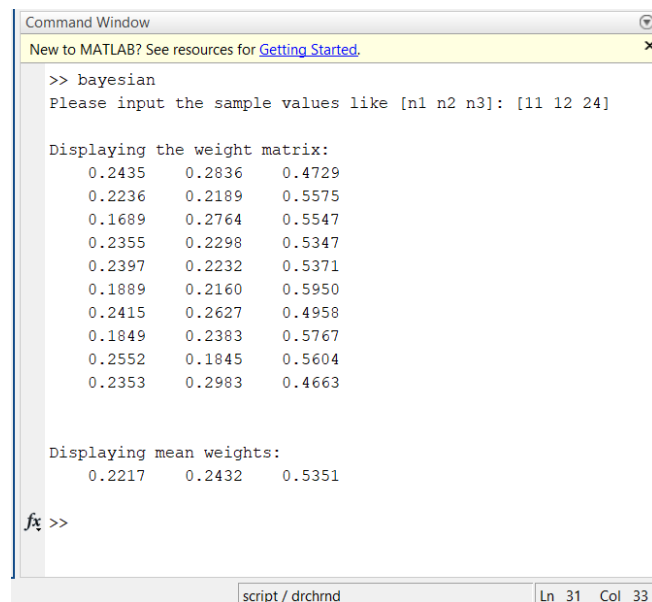
```
>> frequentist
Please input the sample values like [n1 n2 n3]: [11 12 24]

Displaying the weight matrix:
    0.2340    0.2553    0.5106

fx >>
```

At the bottom of the window, the status bar indicates "script" and "Ln 11 Col 32".

Fig.44: Frequentist weight determination



The image shows a MATLAB Command Window with a yellow header bar that says "New to MATLAB? See resources for [Getting Started](#)". The command prompt shows the execution of the 'bayesian' function. It prompts the user to input sample values like [n1 n2 n3]: [11 12 24]. It then displays the weight matrix as a 10x3 matrix. Below this, it displays the mean weights as a 1x3 row vector.

```
>> bayesian
Please input the sample values like [n1 n2 n3]: [11 12 24]

Displaying the weight matrix:
    0.2435    0.2836    0.4729
    0.2236    0.2189    0.5575
    0.1689    0.2764    0.5547
    0.2355    0.2298    0.5347
    0.2397    0.2232    0.5371
    0.1889    0.2160    0.5950
    0.2415    0.2627    0.4958
    0.1849    0.2383    0.5767
    0.2552    0.1845    0.5604
    0.2353    0.2983    0.4663

Displaying mean weights:
    0.2217    0.2432    0.5351

fx >>
```

At the bottom of the window, the status bar indicates "script / drchrnd" and "Ln 31 Col 33".

Fig.45: Bayesian weight determination


```

Python console
Console 1/A

In [3]: runfile('C:/Users/RAJARSI/Desktop/PROJECT/CODE/freq_genetic.py',
wdir='C:/Users/RAJARSI/Desktop/PROJECT/CODE')
Reloaded modules: import_code, matrix, chromosome

Enter source: 0

Enter destination: 19

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.37555313897616766)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([0, 2, 7, 12, 16, 18, 19], -0.37555313897616766)

In [4]:

```

emissions: **RW** End-of-lines: **LF** Encoding: **ASCII** Line: 6 Column: 9 Memory: 61 %

Fig.46: Optimal Path determination using frequentist weights

```

Python console
Console 1/A

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.2393174445283453)
=====

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.40141202878842186)
=====

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.32162491791478987)
=====

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.11217092841470888)
=====

=====
Solution: ([0, 2, 7, 12, 16, 18, 19], -0.5330744455663105)
=====

OPTIMUM SOLUTION WITH BEST FITNESS:
([0, 2, 7, 12, 16, 18, 19], -0.5330744455663105)

In [6]:

```

emissions: **RW** End-of-lines: **LF** Encoding: **ASCII** Line: 6 Column: 9 Memory: 61 %

Fig.47: Optimal Path determination using Bayesian weights

Thus, the solution received from both frequentist and Bayesian weights is: [0, 2, 7, 12, 16, 18, 19]. The graphs for both are given below.

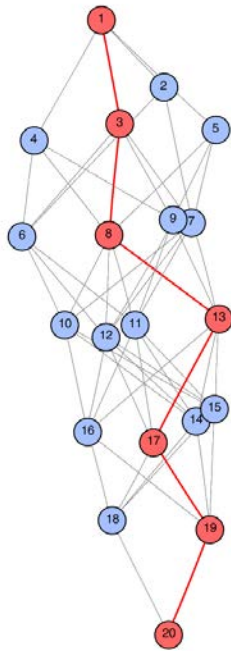


Fig.48(a): Optimal Path Plot for frequentist weights

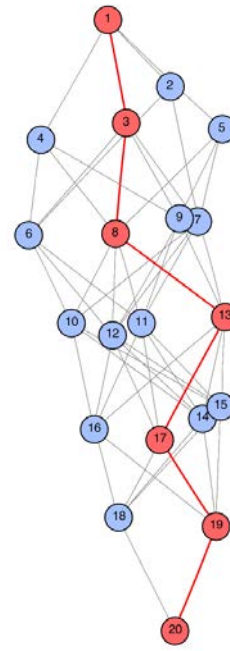


Fig.48(b): Optimal Path Plot for Bayesian weights

The node numbers from the graph are 1 more than the actual node values as the graph plotting program does not accept the value 0 as the initial node value.

CHAPTER 7: CONCLUSION

7.1 CONCLUSION

The proposed method of weight improvement works best with small data sets as observed in the results. For larger data sets, this method is at par with the existing and traditional method of weighted sum used in optimization problems. Our proposed work focuses on the improvement of determining weights in order to determine an optimal path based on several factors. The improvement in weights is obtained using a scientific, probabilistic approach hence overcoming the limitations of a weighted sum. There is still scope for future work even after satisfactory modifications.

7.2 LIMITATIONS OF THE SYSTEM

Along with its own advantages and after modifications, the system still holds on to certain limitations which provides future work prospects, few of them being:

- Works on a limited and small data set.
- Based on population and number of intermediate nodes the optimal path may vary.
- The functionality and efficiency of the system can be further improved by using real world live data.

7.3 FUTURE SCOPE OF THE PROJECT

In this project, we have successfully solved a major drawback of the popular weighted sum method using a Bayesian method based on Multinomial distribution and Dirichlet prior. As mentioned before, the method works best for small datasets. Thus, when one has to work with a small dataset, it is advisable to use the proposed technique. Future work may also explore the results of this technique when using objective functions.

REFERENCES

1. Steuer, Multiple Criteria Optimization: Theory, Computation, and Application. Malabar: Robert E. Krieger Publishing. 1989.
2. Das, I.; Dennis, J.E: A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems. Struct. Optim. 14, 63-69, 1997.
3. Yoon, K.P; Hwang, C.-L.: Multiple Attribute Decision Making, An Introduction. London: Sage Publications, 1995.
4. M. A. Gennert and A. L. Yuille, Determining the optimal weights in multiple objective function optimization. In Second International conference on computer vision, pages 87-89, Los Alamos, CA, September 1998. IEEE.
5. Eckart Zitzler, Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications.
6. Bayes' Theorem by Mario F. Triola
7. Introduction to the Dirichlet Distribution and Related Processes Bela A. Frigyi, Amol Kapila, and Maya R. Gupta Department of Electrical Engineering, University of Washington, Seattle, WA 98195
8. Genetic Algorithm Finding the Shortest Path in Networks Bilal Gonen Department of Computer Science and Engineering, University of Nevada, Reno, Reno, Nevada, U.S.A.
9. The weighted sum method for multi-objective optimization: new insights R. Timothy Marler · Jasbir S. Arora