Title
A Pareto Improving Strategy for the Time-Dependent Morning Commute Problem

Author
Garcia, Reinaldo C.

Publication Date
1999-09-01

A Pareto Improving Strategy for the Time-Dependent Morning Commute Problem

by

Reinaldo Crispiniano Garcia

Graduate (Instituto Tecnológico de Aeronáutica, SP, Brazil), 1989

M.S. (Instituto Tecnológico de Aeronáutica, SP, Brazil), 1992

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering-Civil and Environmental Engineering

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Carlos Daganzo, Chair
Professor Elizabeth Deakin
Professor Samer Madanat

Fall 1999

A Pareto Improving Strategy for the Time-Dependent Morning Commute Problem

by

Reinaldo Crispiniano Garcia

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my thesis advisor Prof. Carlos Daganzo for supervising this dissertation. I cannot hope to repay Carlos for all that he has done; I can only strive to offer my colleagues and future students the same level of interest, enthusiasm, attention to detail, professionalism, good humor, and pursuit of excellence that he has demonstrated to me.

Very special thanks to Professor Elizabeth Deakin and Professor Samer Madanat for their guidance, critique and support of my research; I really appreciate their caring and friendship. Many thanks also to Professor Gordon Newell, Professor Mark Hansen, Professor Martin Wachs and Professor Mike Cassidy for providing helpful questions and comments during my qualifying examination.

One can easily conclude that Berkeley is an amazing place. During my years in Berkeley, I had the opportunity to meet the wives of some of my favorite Professors, and I definitely must mention Valery Daganzo, Barbara Newell and Rana Madanat. In my home country, Brazil, we always say that "behind any great man, there is always a wonderful woman." That saying is definitely true.

My graduate studies would not have been possible without the generous financial support of the "Centro de Aperfeiçoamento de Pessoal de Ensino Superior" (CAPES, Brazilian Federal Organization) and by the University of California Transportation Center (UCTC) fellowship program. Thanks also to the people who make the U.C. at Berkeley special, particularly Mari Cook, Catherine Cortelyou, Virginia Harris and Ann Lee.

During my stay in Berkeley, I can easily say that I have obtained great friends. Certainly, among my best friends in Berkeley, it is a honor to mention Javier Contreras (from Zaragoza, Spain), Jeanette Zollinger (from Switzerland), Joerg Appel

(from Germany), Johan Meeusen and his wife, Cathy Berx (with their son, Joachim, all from the beautiful Belgium), Jose Antonio (from Vinuelas, Spain), Juan Carlos Munoz and his wife, Paula (with their daughter, Isabel, from Chile), Katelijn Rotsaert (from Belgium), Paulo and Fernanda Miranda (with their daughter, Amanda, my "godchild", all from Brazil), Ana e Paulo Penteado (with their son, Marcos, all from Brazil), Adrian Ricardo Archilla and his wife Marcela (with their two daughters, from Argentina) and Philippe Wallart (from Switzerland) and his wife Natasha Balakova (from Russia). Before coming to the U.S., I never expected to meet such a great group of friends. Countries like Spain, Belgium, Germany...will always be in my heart.

I am also thankful to all the talented Berkeley students with whom I have had the privilege to study. Special thanks to Rob Bertini, Alan Erera, Kazuya Kawamura, Tim Lawson, Lawrence Liao, Wei Lin, Dave Lovell, Mike Mauch, Arnab Nilin, Jorge Prozzi, Karen Smilowitz, and John Windover.

Finally, my thesis would not have been possible without the support, encouragement, and patience of all my family members: my younger brother, Luis Alberto, my sister Ana Cristina and her husband (Andre), my older brother, João and his Belgian wife, Marijke Falloise (my wonderful sister-in law) and their three children (Ilya, Hendrik, my "godchild", and Elliot), and certainly, my parents, Maria and Luiz. I am really grateful to all my family members. During all these years in Berkeley, I do not know how many times I have called my parents, Maria and Luiz, asking for their guidance, love, support.... I can easily say that I have the best parents in the world, and if I can give to my future children just a fraction of their love, faith, support....I would be extremely pleased. It is to them that this dissertation is dedicated with great love and gratitude.

To my parents

Maria Eulália Crispiniano Garcia

Luiz Benedicto Coutinho Garcia

Contents

List of Figures

1 Introduction

This dissertation describes a strategy which makes all commuters better off (*i.e.* a Pareto efficient strategy) for the time-dependent morning commute problem, even if the collected revenues are not returned to the population of commuters. The proposed strategy will apply road pricing as a tool for congestion management, a practice usually called congestion pricing.

Congestion pricing involves varying the price for road use by the level of traffic congestion to encourage people to travel during less congested hours, by less congested routes, by alternative modes or not at all. Usually two types of congestion pricing schemes are considered: point and cordon pricing. Under point pricing, vehicles passing a particular point on a roadway during congested hours are charged a toll, just as with congested toll roads, bridges and tunnels. Point pricing is considered congestion pricing if the charge point is located at the entrance to a congested facility or area and if the toll is higher during hours of peak demand. Cordon pricing is a variant of point pricing in which an imaginary cordon is drawn around a congested area and charge points are established at all points of entry or exit. Charges may vary by direction traveled or vehicle type, as well as location and time of day. The application of automated charging is not essential to congestion pricing, but it has become more attractive with recent developments in technology and it makes complex congestion pricing schemes more practical.

Interest in congestion pricing has been increasing both because congestion has been growing and because the public and government officials have become frustrated with the limitations of alternative congestion remedies[1]. Even though the estimates

[1]The traditional solution to decrease the costs of increasing congestion has been to build more roads. Nowadays there seems to be academic consensus about the low feasibility of this solution, not only because it is expensive to implement it but also because it often provides only temporary relief (Mogridge, 1990).

of the costs of increasing congestion must be treated with caution, the annual cost of delay in the 10 most congested cities and suburban road systems in the US is estimated to be more than $34 billion (Financial Times, 1997)) and $308 billion (The Economist, 1997) in the whole Western Europe. One form of high congestion appears in the time-dependent morning commute problem, where commuters arrive at some bottleneck and each commuter who traverses the bottleneck must reach his destination by a specified time (his work starting time).

The major sections of this dissertation present a thorough discussion of the time-dependent morning commute problem, when the proposed strategy is implemented. Chapter 2 describes the proposed strategy in detail and presents the analytical results for the scenario where the commuters are homogeneous ("identical" commuters scenario), *i.e.* the commuters have the same desired deadline and the same willingness to pay, and value their time either in the queue or at the destination equally. Chapter 3 describes the *analytical* results for the scenario where the commuters are "heterogeneous", but along one dimension only (the commuters may have either different desired deadlines or, different values of time or, different willingness to pay). Chapter 4 extends the analytical results applying a simulation framework. In this section the commuters have different desired deadlines, different values of time *and* different willingness to pay. Finally, Chapter 5 contains a summary of the major accomplishments of this study and an outline of some areas for further research.

The remainder of this introductory chapter presents previous work. This includes both, related research and additional background for this study.

1.1 Related Research

The concept of road pricing as a policy tool to promote the efficient use of scarce public goods begins with a work by Vickrey (1963), who argued that although rev-

enues derived from vehicle taxes and other sources recovered less than one-third of the related costs, pricing strategies based on relative elasticities of demand would be profitable enough to recover all of those costs. In a later study, Vickrey (1969) analyzing an idealized model[2] of the morning commute problem, pointed out that if one charged an appropriate time-dependent toll (equal to the cost equivalent of the waiting time of the commuters), one could induce commuters to arrive at the bottleneck at such times that a queue would not form. In effect, one could convert worthless queuing time into money.

In the literature, one finds that earlier studies associated with peak traffic conditions did not take full account of the way in which commuters changed their times of departure as a function of traffic conditions (Hurdle, 1974, 1981; Fargier et al., 1979; Morin, 1980; De Palma et al., 1983; Ben Akiva et al., 1984, 1986). Such a response needs to be considered in studies of the effects of any staggering of work starting times, since this leads to changes in the desired arrival times and the associated departure times.

Further studies attempted to examine the mechanism involved in the choice of the time of undertaking a journey as a function of the required time of arrival.[3] These studies were among the first to consider deterministic time-dependent equilibrium commute problems and pointed out the likely significant of such problems for transportation planning. Subsequent work extended those papers. Smith (1984) proved the existence of equilibrium under general assumptions on the functions relating trip

[2]In Vickrey's model (1969) a fixed number of identical commuters have to travel from a single entrance (home) to a single exit (work) along a single road during the morning rush hour. There is a single bottleneck on the road with a fixed capacity or service rate and if the arrival rate at the bottleneck exceeds this capacity, a queue develops. The commuters have identical cost functions.

[3]Hendrickson and Kocur (1981) and Fargier (1981) independently formulated Vickrey's model and "assumed" that the commuters actual order of entry into the bottleneck would be the same as the desired order of departure for a Wardrop equilibrium (1952) (an equilibrium where no commuter can reduce his cost by changing his arrival time, on the assumption that costs of arriving at any other time are not altered by his change).

costs to travel time and schedule delay and, Daganzo (1985) provided a proof of the uniqueness of this equilibrium.

More recent models have tried to achieve efficiency savings, decreasing the queuing or schedule costs of the commuters. These models consider that a number of commuters must travel from home to work, between which there is a bottleneck of given capacity (Arnott et al., 1988, 1990, 1998; Laih, 1994; Berstein and Hiller, 1998). The costs of travel include queuing time and schedule delay (time early or late for work). Even though the application of congestion tolls in these models results in efficiency gains, the savings in queuing time or in schedule delay by a commuter can be more than offset by the commuter's payment in the form of tolls.

A great amount of research has been dedicated to the analysis and discussion of more complex schemes involving rationing and pricing (see Keeler, T. et al., 1975; Keeler, T. and Small, K.A., 1977; Cohen, Y., 1987; Wachs, M., 1988, 1991, 1994; Deakin, E., 1989, 1994; Gomez-Ibanez, J. and Small, K.A., 1994; Banister et al., 1996; Bristow, A. et al., 1998; Odeck, J. and Bräthen, S., 1998; Rico, A. et al., 1998). The main problems with these more elaborate policies are that they can take away the people's choice (rationing) or be regressive (pricing).

A first congestion reduction scheme having the potential for not penalizing anyone despite the tolls was suggested by Daganzo (1995). The traditional social welfare approach was modified to address the distribution of gains and losses across the population. Daganzo (1995) considered a homogeneous "steady-state" transportation system where a congestion delay was simply given by a function of its user flow. He showed that a pricing scheme with discriminating tolls that were rotated across the population could benefit everyone (unlike pure pricing alone), even if the collected revenues were not returned to the population.

A similar strategy to improve the welfare of every commuter for the "time-dependent" morning commute problem was also proposed in Daganzo (1992). He considered a scheme where passage through the bottleneck is "banned" after a certain time of the day (the start of the rush hour) for a fraction of the commuters (the "paying" commuters). Each individual commuter was designated as "paying" only on certain days; on the remaining days (s)he could travel freely with no toll. The ban remained in force even after the conclusion of the rush hour and the "paying" commuters had to pay a monetary penalty, if they missed the start of the ban. This assumption limited the benefits that could be gained from control as it discouraged commuters from delaying their arrival in order to avoid the queue. Despite this ineficiency, Daganzo (1992) showed the scheme to have the potential for benefiting everyone over a number of days.

1.2 Background

It is convenient to introduce basic ideas for the case of homogeneous commuters where the commuters have the same desired deadline, the same willingness to pay, and they value their time either in the queue or at the destination equally. These ideas are based in works by Smith (1984), Daganzo (1985), Kuwahara (1985), Newell (1987) and Arnot et al. (1988).

This thesis considers the time-dependent morning commute problem where the commuters have to pass through a single bottleneck every day to arrive at work on time. The bottleneck has a time-independent capacity, μ. For the remainder of this thesis, and without loss of generality, it will be assumed that time is measured in a system of units such that the maximum flow through the bottleneck is "1" (one), *i.e.* $\mu = 1$.

If travel times beyond the bottleneck are constant (no congestion), one can depict for each commuter (i) a desired revised deadline (w_i) for passage through the bottleneck that would allow said commuter (i) to arrive at work on time. The term "deadline" is always applied in this sense in this research. It is assumed without loss of generality that customers are labeled such that if $i < j$, $w_i < w_j$. A function $W(t)$ gives the cumulative number of commuters with (revised) deadlines prior to time t. Limiting cases for $W(t)$ are also allowed, *e.g.* $W(t)$ can jump from zero to a maximum value at some time W_0 (as if the population of commuters is finite, $i = 1, \ldots, N$, and everyone has the same desired deadline, W_0). It is also assumed that the commuting time (other than waiting in the queue) is a constant. Thus, we can eliminate this constant from our model, and may assume without loss of generality that the commuter arrives at the bottleneck as soon as he departs from home and, he arrives at his workplace immediately after passing the bottleneck.

This thesis assumes a linear dependence on time and money for each commuter's utility, but all the times (in the queue and at the destination, both early and late) are not valued "equally" by each commuter. The utility of each commuter can be expressed in any units desired, if the results are used to compare only the utilities for a specific commuter, before and after the strategy's application. Since this thesis is not concerned with aggregate welfare measures, time in queue is chosen as the measurement of scale. Note that this is appropriate even if a unit of queuing time is not equally valuable to everyone. Thus, the queuing-time-equivalent penalty of commuter i (with deadline w_i), departing the bottleneck at time t, is[4]:

$$Penalty = \begin{cases} e(w_i - t), & \text{if } t \leq w_i \text{ (the commuter departs early)} \\ L(t - w_i), & \text{if } t > w_i \text{ (the commuter departs late)} \end{cases} \quad (1)$$

where e and L are the conversion rates of earliness and lateness into queuing time, and they are assumed to satisfy: $e \leq 1$ and $L > e$.

[4]The assumption of linearity implies that doubling the commuter's earliness or lateness, doubles his penalty.

As in past studies, a state is defined as the set of arrival times to the bottleneck for all commuters i, a_i. Given this set $\{a_i\}$, we can construct the cumulative number of commuters to have arrived and departed by time t, $A(t)$ and $D(t)$. If each commuter has his own utility function involving the trade-offs between queuing time, schedule delay and toll cost (the toll charged to a commuter if a toll is applied), then for suitable convex utility functions[5] (of which Eq.(1) is an example) there will be a unique user optimal assignment specifying when each commuter will arrive at the bottleneck and when he will depart (Daganzo, 1985). Moreover, if the commuters maximize their utility, an equilibrium state exists (and is unique) whereby no commuter can improve his utility by unilaterally changing his arrival time. At such an equilibrium commuters arrive in their desired order of departure, "first-deadline-first-in" (FDFI) *i.e.* such that if $i < j$ then $a_i < a_j$. In other words, if the arrival (or departure) position of a commuter is represented by p_i, then $p_i \equiv i$ under FDFI.

It has been shown that the three curves $A(t)$, $D(t)$ and $W(t)$ must coincide when there is no queue, and while there is a queue the departure curve is a segment between two points on $W(t)$, intersecting $W(t)$ at some intermediate position p^* (see Figure 1) (Smith, 1984 and, Daganzo, 1985). Each commuter had to choose an arrival time to minimize his queuing delay plus unpunctuality cost: the commuter's departure time can be found if the arrival times of the other commuters and the service rate of the bottleneck are known. Daganzo (1985) proved that there cannot be a Wardrop equilibrium in which commuters do not leave in the wished departure order and, exploring the differentiability properties of equilibrium arrival curves (suitable convex utility functions), he also established that there can be only one arrival curve that is an equilibrium. If the commuter queues, the sum of his queuing and schedule costs

[5]To demonstrate the uniqueness of the equilibrium, Daganzo (1985) required the function to be convex and differentiable for all real arguments and, have a strict minimum.

gives his generalized cost. The following results are simplifications of those by Arnot et al. (1988) and Daganzo (1992).

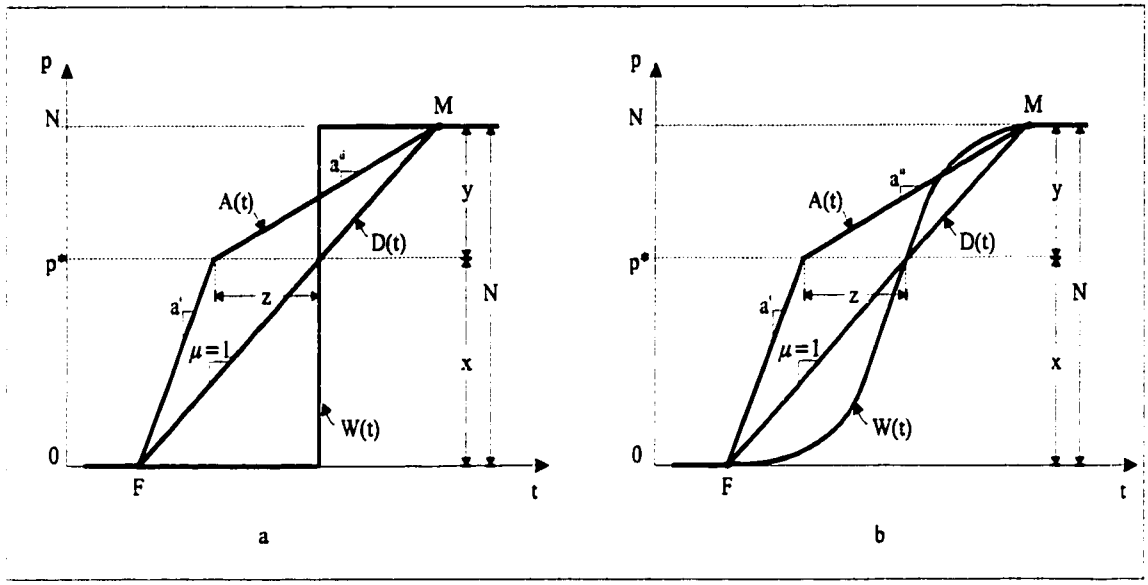

Figure 1: Cumulative arrival $A(t)$, departure $D(t)$ and desired deadline curves (commuters with the same desired deadline, W_0, Figure 1a; commuters with different desired deadlines, $W(t)$, Figure 1b).

If the commuters have the same desired deadline (case shown in Figure 1a), they must experience the same cost to be in equilibrium. The first commuter to arrive at position 0 (zero) is the commuter who experiences the largest earliness schedule cost but no queuing cost; as we progress from the commuters arriving at position 0 to position p^*, there is an "increase" in the commuter's queuing cost and an equal "decrease" in his earliness schedule cost (observe Figure 1a). The commuter arriving at position p^* is the only commuter to depart at his desired deadline, W_0 (experiencing no schedule cost), but he is the one who experiences the highest queuing cost (z). A similar analysis can be made for the commuters departing late (from positions p^* to N). It turned out that to obtain an equilibrium for the penalty function (1), the curve $A(t)$ must diverge from the low end of the segment, increasing at rate $a^i=1/(1-e)$ until reaching p^*, and at rate $a^{ii}=1/(1+L)$

thereafter; the curve $A(t)$ must also intersect the departure curve $D(t)$ at the end of the segment. Since the total number of commuters wishing to pass through the bottleneck is given by N, the geometry of the figure shows that the cost for commuter p^* is $C = z = NeL/(e+L)$.[6] This cost (C) is identical for all the commuters, since they are identical, have the same desired deadline and the system is in equilibrium. The number of commuters departing early is also represented (Figure 1) by the variable x and, the number of commuters departing late by the variable y, where from the geometry of the figure, $x = p^* = NL/(e + L)$ and $y = Ne/(e + L)$.

[6] As $a^i = 1/(1 - e) = p^*/(p^* - z)$ and $a^{ii} = 1/(1 + L) = (N - p^*)/(N - p^* + z)$, solving for p^* and z, the result follows.

2 The Strategy and Homogeneous Commuters

This chapter demonstrates that there is a tolling scheme that leads to a Pareto improving equilibrium even if the collected revenues are not returned commuters. Section2.1 describes the scheme and Section2.2 its results for the case of a population of identical commuters.

2.1 The Strategy

The proposed strategy involves the following elements:

(a) Variable Toll

A variable toll with a fixed rate τ (monetary units per unit time) applied during a fixed time window, $[W_B, W_E]$. The toll penalty for a commuter departing inside the time window is proportional to the amount of time by which the commuter "missed" the beginning or the end of the time window, whichever gives the lowest cost. Thus, the toll penalty for a commuter departing at time t is(see Figure 2):

$$Toll\ penalty\ =\ \tau\ min\left[(t - W_B), (W_E - t)\right]\ for\ t\ \in\ [W_B, W_E] \qquad (2)$$

(b) Free or Paying Commuters

Every day each commuter is classified as either "free" or "paying". The "free" commuters are allowed to use the bottleneck without paying the toll (even if they decide to pass through the bottleneck inside the time window). The "paying" commuters can avoid paying the toll only if they pass through the bottleneck outside the time window; otherwise, they have to pay the toll penalty corresponding to their passage time. The classification method is such that: (i) in the long run the fraction

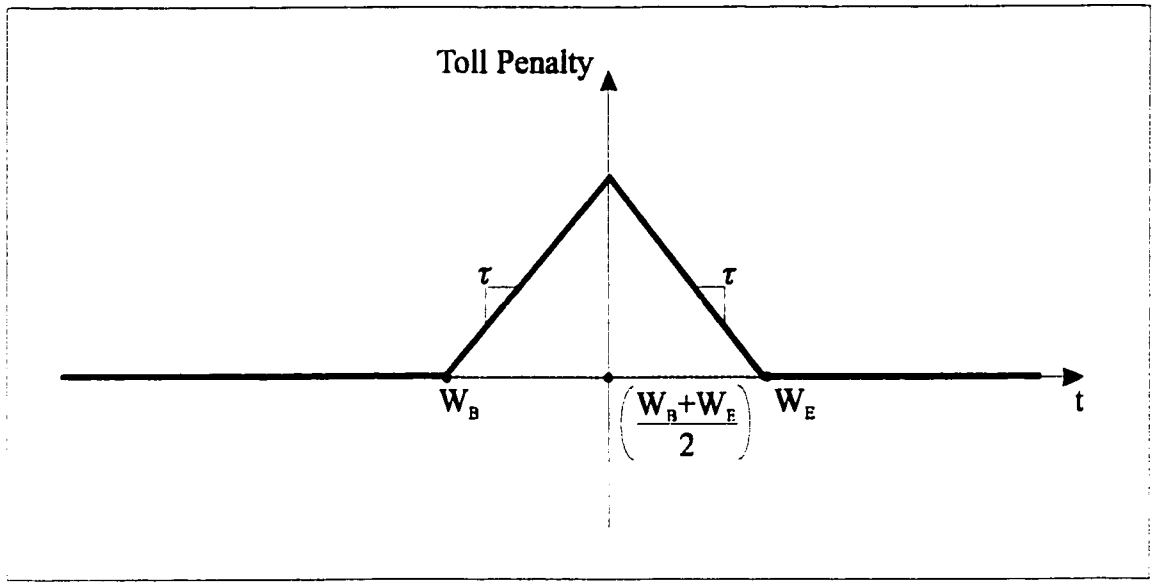

Figure 2: The toll applied as a function of time.

of days, f, that a commuter is "free", is the same for all the commuters; (ii) the fraction of "free" commuters is f every day.

(c) The System Manager

The system manager can choose four free parameters: (i) the toll rate, τ; (ii) the time window, $[W_B, W_E]$, and (iii) the fraction of "free" commuters, f.

The properties of the proposed strategy suggest that it can be easily implemented. To this end, the referred classification of the commuters as "free" or "paying" can be made by the ending of the license plates of the commuter's vehicles, with each commuter having an electronic tag attached to his car. One could also classify the commuters as "free" or "paying", depending on where they live, *e.g.* commuters living in the same downtown area would be classified as "paying" the same days of the

week[7].The corresponding toll can be charged by the use of an electronic toll facility with the corresponding toll penalty being automatically charged to any "paying" commuter whose tag is detected inside the time window. The next sections describe in detail the solutions obtained once the proposed strategy is implemented.

[7]This classification method would address the relevant issue of the *same* commuter having "more" than one car with *different* ending license plates, as this commuter would have *all* their cars classified as paying in the same days.

2.2 Homogeneous Commuters

To understand the main issues involved with the strategy's application for the time-dependent morning commute problem, it is wise to start with the simplest case, *i.e.* the one where the commuters are identical. Thus, this section presents the results obtained for the case of commuters who value their time either in the queue or at their destination equally and have the same desired deadline.

The proposed strategy aims to achieve an equilibrium pattern where the "paying" commuters depart either outside the time window or inside it but close to its borders (close to the times W_B and W_E) and, the "free" commuters depart as close as possible to their common desired deadline, W_0.

More specifically, the decision variables (f, τ, W_B and W_E) are chosen so as to achieve an equilibrium pattern with the following properties (illustrated in Figure 3):

(i) three triangular queuing episodes,

(ii) the same departure curve as without the control,

(iii) a queue that vanishes between congested episodes and inside the time window (points A and C) and,

(iv) a delay which is a (local) maximum for the commuters departing at the edges of the time window (points G and Q).

It will be shown that to obtain the equilibrium described in Figure 3 (with the "paying" commuters departing in the two extreme queuing episodes and the "free" commuters departing in the middle queuing episode), the following has also to be assumed for the slopes of the arrival curve $A(t)$ (property (v)):

$$a^i = \frac{1}{1-e} \qquad (for\ the\ segments\ FE\ and\ AD) \tag{3}$$

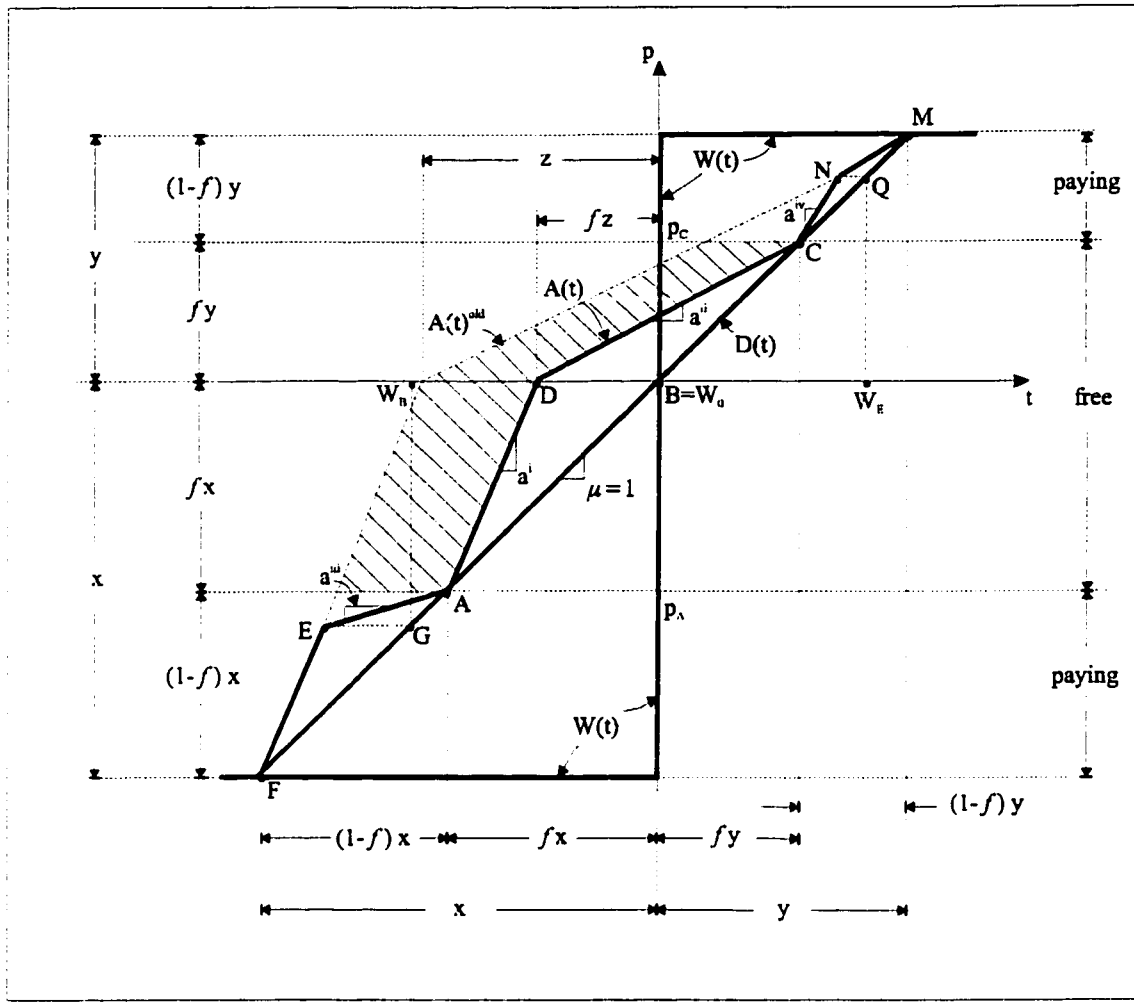

Figure 3: Equilibrium with the proposed strategy - commuters with the same desired deadline. W_0.

$$a^{ii} = \frac{1}{1+L} \qquad (for\ the\ segments\ DC\ and\ NM) \tag{4}$$

$$a^{iii} = \frac{1}{1+\tau-e} \qquad (for\ the\ segment\ EA) \tag{5}$$

$$a^{iv} = \frac{1}{1+L-\tau} \qquad (for\ the\ segment\ CN) \tag{6}$$

To obtain the described geometry (Figure 3) we cannot arbitrarily choose f, τ, W_B and W_E; a choice will be feasible only if as we construct the two extreme triangles starting from points F and M, applying properties (iv) and (v), we obtain two points

A and C such that A is below B and C is above B (where as seen in Figure 3, point B is the point on the departure curve with $t = W_0$).

To ensure the resulting pattern to be a Wardrop equilibrium (Section 1.1), two additional requirements are imposed for the middle triangle. These requirements take the form of equality conditions among the decision variables which further restricts the feasible region, but not unduly so since we started with four free variables (f, τ, W_B and W_E). The new requirements are: (vi) the break in the arrival curve in the middle triangle (as one constructs it from points A and C) should be level with the point on the departure curve with $t = W_0$ (*e.g.*, point D level with point B) and, (vii) the fraction of positions between points A and C should equal the fraction of "free" commuters, f (as shown in Figure 3, from position p_A to p_C there are fN available positions).

If the decision variables satisfy the properties and requirements above, then it can be shown that the pattern just described is a Wardrop equilibrium, where the "free" commuters depart between positions A and C (in the middle queuing episode) and the "paying" commuters depart elsewhere (in the other two queuing episodes).

To see this let's consider first the set of "paying" commuters and then the one of "free" commuters. *Paying commuters*: As everyday there are *(1-f)N* paying commuters, the geometry of Figure 3 shows that there are *(1-f)N* positions available for *(1-f)N* paying commuters. Since the departure curve did not change (property (ii)), the "paying" commuters arriving at points F or M (positions 0 and N) experience the same cost as before, z. When we progress from the commuters arriving at position 0 to position E, there is an "increase" in the commuter's queuing cost and an equal "decrease" in his earliness schedule cost; so all the positions between 0 and E share the same cost (z). When we progress from the commuters arriving at position E to position A, there is a "decrease" in the commuter's queuing and earliness cost that

precisely cancels out the "increase" in toll cost (this follows from the definition of a^{iii}, Eq.(5)). The same logic can be applied to the upper triangle for the "paying" commuters departing from positions C to N. Note as well that the middle triangle is not attractive to any of the "paying" commuters. Thus, all the "paying" commuters are in a Wardrop equilibrium where they experience the same cost as before (z). *Free commuters*: The slopes of a^{i} and a^{ii} together with property (vi) (point D level with point B), defines a geometry for the middle queuing triangle similar to the one of Figure 1. As already explained (Section 1.2), this guarantees the "free" commuters to be indifferent of departing from positions A to C. Thus, when the commuters are classified as "free", the geometry of the middle queuing triangle implies the "free" commuters to experience a cost of $C_{free} = fz$.[8] These commuters are in equilibrium because the positions correspond to the outside triangles are less attractive. The commuters' cost varies with departure time as shown in Figure 4.

since the proposed strategy designates each commuter as "free" for a fraction f of the days (and "paying" for the remainder *(1-f)* days), in the long run, the average cost per day for a typical commuter is a weighted cost ($f\,C_{free} + (1-f)\,C_{paying}$). A convenient expression is:

$$C_{avg.} = f^2 z + (1-f)z \tag{7}$$

The savings for a typical commuter are:

$$Savings = C_{paying} - C_{avg.} = zf(1-f) \tag{8}$$

Since the cost of the "paying" commuters is the same as the "before" cost.

One can see from Eq.(8) that in the traditional pure pricing approach where all the commuters are classified as "paying" (f=0) the savings are 0 (zero) for every commuter. Moreover, the savings obtained by a commuter are related to the time

[8]The middle queuing triangle in Figure 3 is similar to the queuing triangle in Figure 1, being scaled by f (the fraction of "free" commuters).

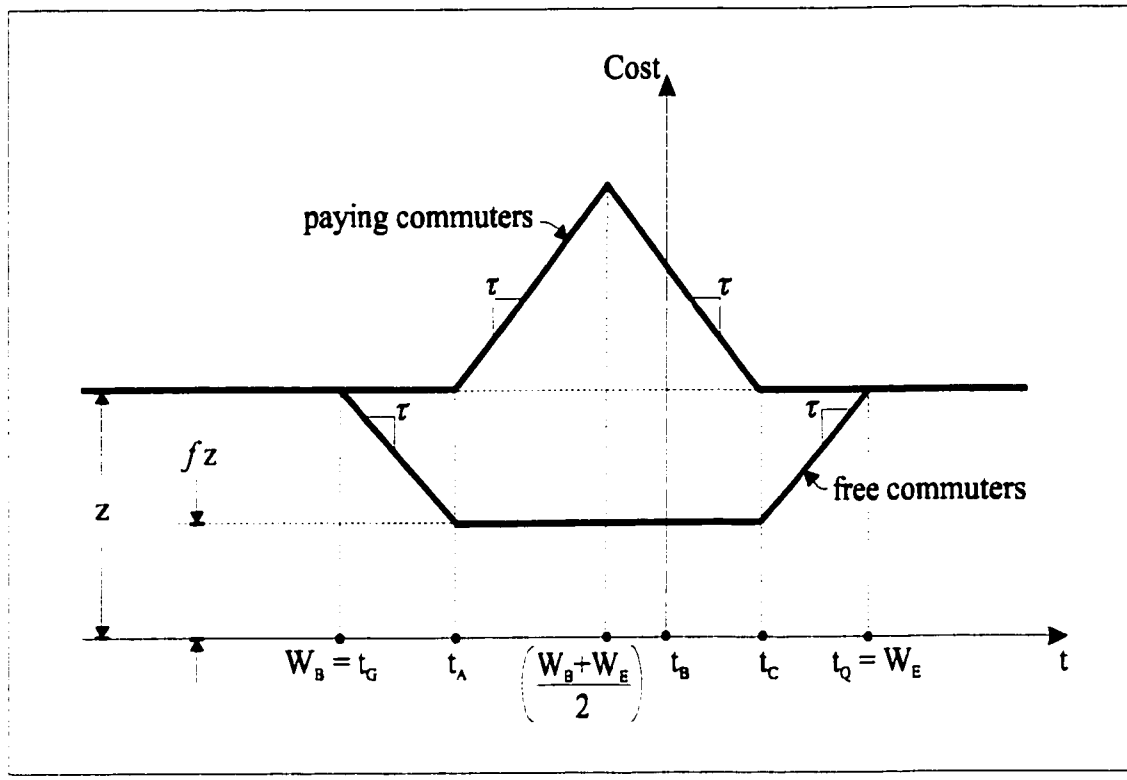

Figure 4: Variation of the user cost, including toll, with the departure time.

he is classified as "free". These savings arise from the decrease in queuing time experienced by the "free" commuters with the application of the proposed strategy. Figure 5 demonstrates the savings obtained with different values of f (small and large). We see from Eq.(8) that a maximum savings of 25% of the initial cost can be achieved, when the commuters are classified as "free" 50% of their days. These savings are obtained even without returning the collected revenues to the population.

Having shown that equitable savings can be obtained for a case with homogeneous commuters, we now extend the analysis to the more general case.

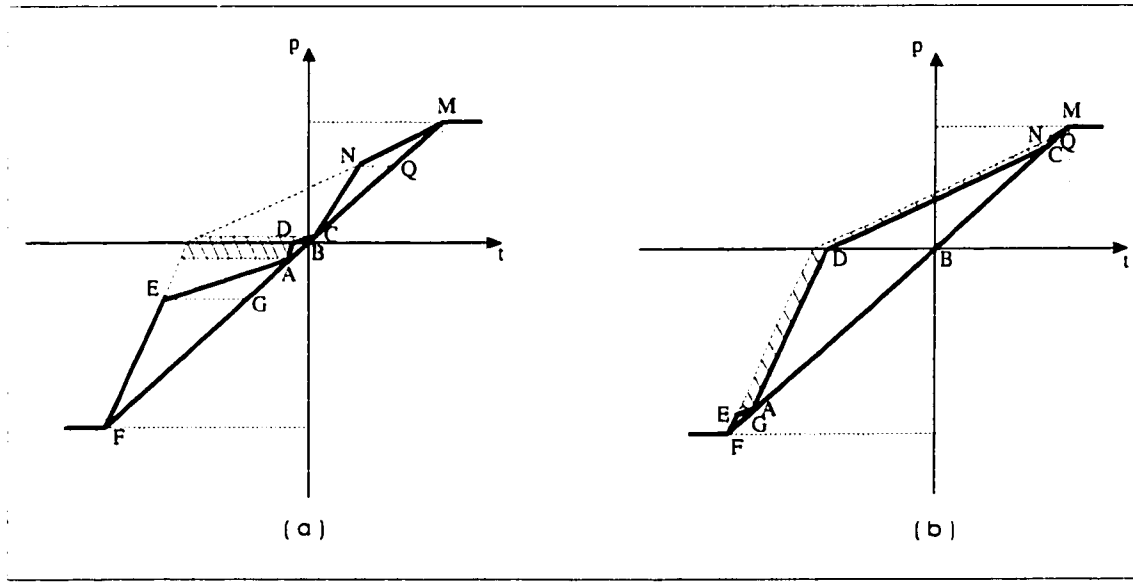

Figure 5: Illustration of the savings obtained with different values of f: (a) small: (b) large.

3 Heterogeneous Commuters - Special Cases

This chapter describes *analytical* results for the cases where the commuters are heterogeneous. Due to the complexity of solving analytically this general case in which commuters have different desired deadlines, different values of time *and* different willingness to pay (elastic demand) this chapter allows only one of these parameters vary across the population at a time. We expect that the insights gained when studying each one of these cases will help us address the general case more effectively. This will be done in Chapter4. Section3.1 below presents the case where the commuters have different desired deadlines. Section3.2 the case where the commuters have different willingness to pay (elastic demand) and section3.3 presents the case for the commuters having different values of time .

3.1 Different Desired Deadlines

We assume here (as in Smith, 1984) that the desired deadline curve *W(t)* increases in an S-shape form, *i.e.* its derivative may increase to a maximum value greater than 1, remain there for a while and finally decline from that level. We will show that if all the commuters have desired deadlines in a narrow window of prespecified width (*e.g.*, *W(t)* is moderately steep) then the solution identified in Section 2.2 continues to be an equilibrium if the customers in each class arrive in a FDFI order. This equilibrium will still be Pareto improving, and Eq.(8) will still apply. The savings in this case continue to be equal for all users independent of their desired deadlines. Figure 6 shows an equilibrium (arrival and departure curves) that satisfies the conditions of Section 2.2. It also shows a hypothetical "*W(t)*" curve with earliest and latest deadlines, $w_{min} = T_W(-x)$ and $w_{max} = T_W(y)$, that satisfy the following "steepness condition":

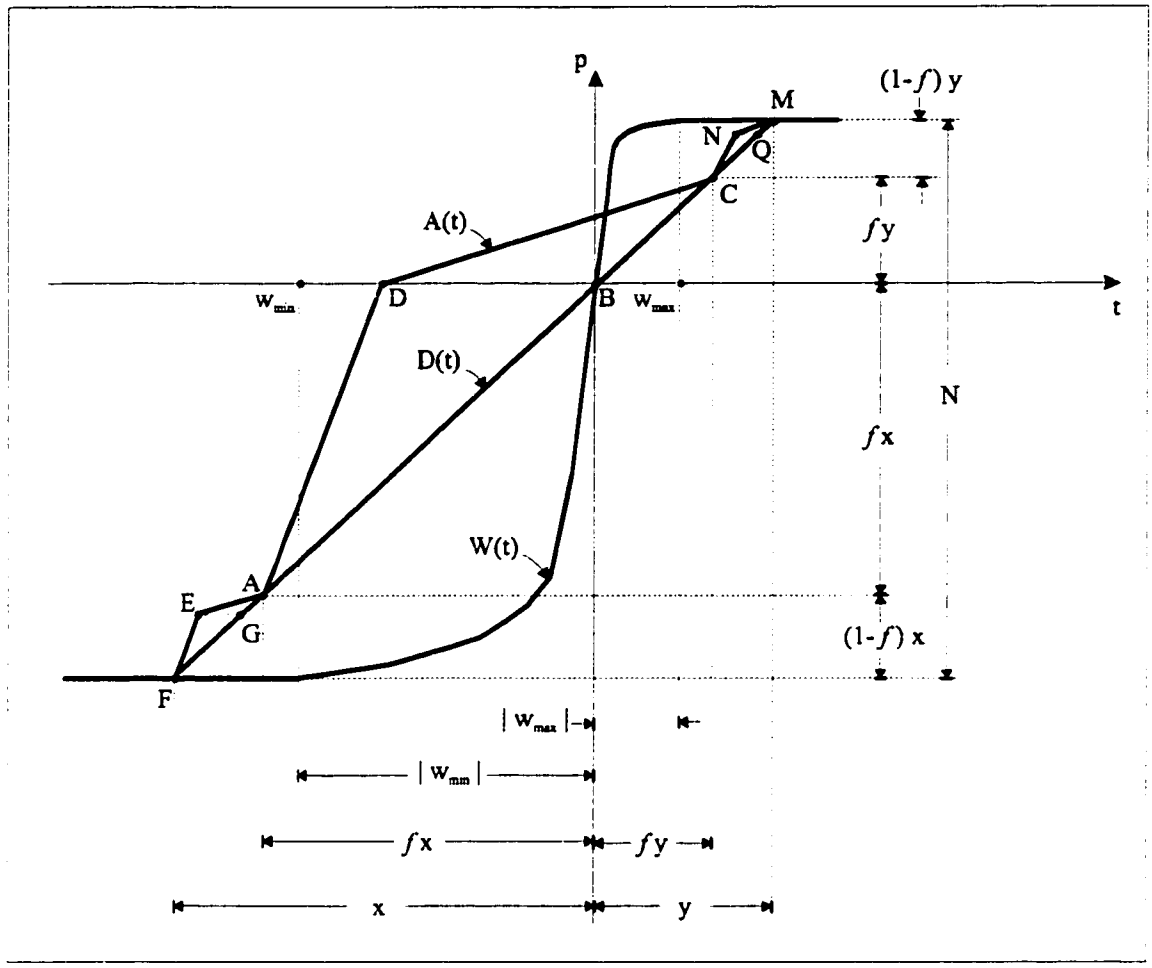

Figure 6: Equilibrium pattern of arrivals with the proposed strategy: commuters with different desired deadlines.

$$w_{min} > -x \qquad and \qquad w_{max} < y \tag{9}$$

The strategy of the form presented in Section 2.1 is Pareto improving for this problem if, as shown in the figure, the proportion of free users f is large enough to satisfy:

$$w_{min} > -fx \tag{10}$$

and

$$w_{max} < fy \tag{11}$$

Geometrically, this means that the middle queuing episode should include the time interval from w_{min} to w_{max}, as shown in Figure 6. One can easily see that a sufficiently large f (with $f < 1$) satisfying Eqs.(10-11) can always be found if Eq.(9) holds. That the equilibrium of Section 2.2 with a step *W(t)* is also an equilibrium now under class-specific FDFI follows from the following:

(a) The deadline curve of "free commuters", $W_f\ (t) = fW(t)$ (not shown in Figure 6, would pass through the queue dissipation points A and C and would divide the departure curve into two segments of similar lengths as before (see Section 2.2). Thus, since free commuters arrive in a FDFI order, they would be in equilibrium during the middle queuing episode.

(b) Because all the positions in the first queuing episode imply earliness (Eq.(10)) these positions are equivalent for all commuters. The logic behind this statement was outlined in Section 2.2. Similarly, the positions in the third queuing episode also continue to be equivalent. Recall from Section 2.2 that a user with $w_i = 0$ is indifferent to the first and third queuing episodes. It follows then that paying commuters with negative deadlines will prefer the first queuing episode and those with positive deadlines the third one. Note as well that none would opt for the middle queuing episode.

(c) There are $(1 - f)x$ paying commuters with negative deadlines who wish to arrive in the first queuing episode. Since this episode can accommodate precisely $(1 - f)x$ users, the demand for positions is met. Since the same can be said for paying commuters with positive deadlines and the third queuing episode, we see that the suggested arrival pattern is an equilibrium.

This establishes that every commuter would take the same action with a smooth deadline function as one would have taken with a step function, and that this is true for both the controlled and uncontrolled scenarios. Therefore, the improvement introduced by the control cannot depend on the shape of "W" for any user. As a result, Eq.(8) still applies. The largest benefit is obtained by maximizing Eq.(8) subject to Eqs.(10-11). Clearly, the maximum theoretical benefit (savings = $.25z$ for $f = 0.5$) can only be achieved if the tails of "W" are short ($w_{min} > (-.5x)$ and $w_{max} < .5y$). Otherwise, a smaller Pareto improvement can be obtained with a large f. Note that both the improvement for large f and the remaining control variables needed to achieve it are independent of "$W(t)$". That means that knowledge of the desired deadline curve is not necessary to devise a strategy. Moreover, it means that the proposed strategy can be introduced gradually, starting with a wide time window and a small percentage of paying commuters. The resulting effect on the arrival curve can be monitored easily. With this diagnostic, we can then decide whether narrower windows and lower values of f should be pursued.

If "$W(t)$" is so shallow that Eq.(9) is violated, there may be "free" commuters who would like to depart in the extreme queuing episodes, and this may preclude perfect class segregation with class-specific FDFI. Whether this rules out perfect Pareto improvements we do not know. In any case, and given the choice flexibility commuters have with the proposed pricing approach, we would expect a more equitable distribution of gains and losses. The next section extends the analytical analysis for the heterogeneous commuters where they have different willingness to pay (elastic demand scenario).

3.2 Heterogeneous Commuters - Elastic Demand

A very important issue when devising a strategy to provide congestion relief is that of latent demand. Even tough one can devise a strategy to decrease existing congestion for a fixed population, congestion reduction could induce nonusers to use the system, and this may decrease the likelihood of obtaining Pareto improving solution. Thus, this section shows that this is not the case.

We assume that there is a population of commuters wishing to pass through a single bottleneck everyday. The commuters have the same desired deadline and are identical in all other respects.

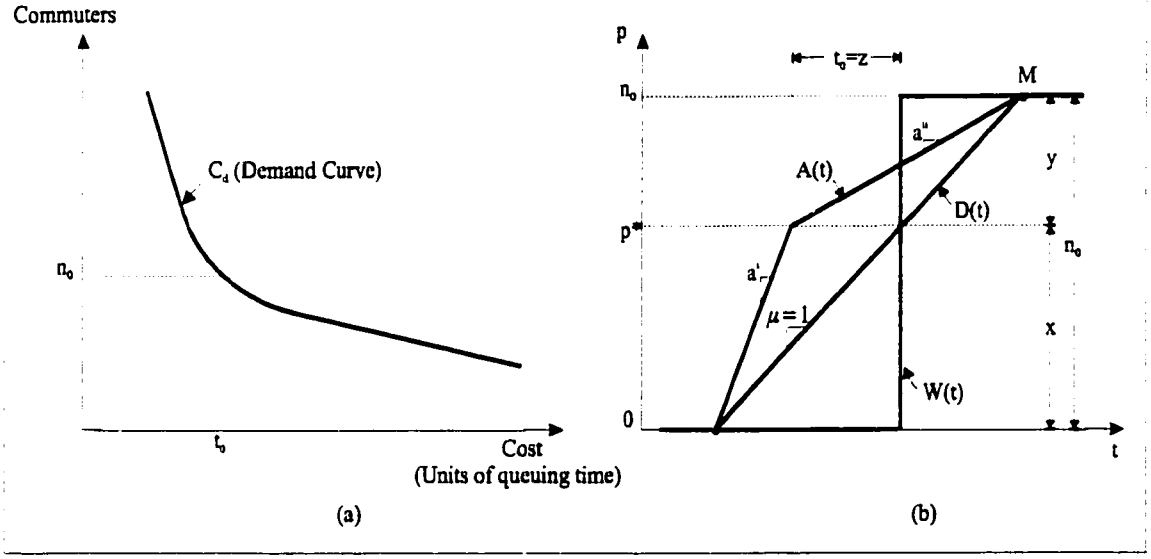

Figure 7: Elastic demand: (a)The commuters are ordered according to their willingness to pay: (b) Equilibrium before the strategy's application.

If the commuters are ordered according to their willingness to pay (Figure 7a) and one plots the commuters (#) vs. her(his) willingness to pay (in terms of queuing time), one obtains the demand curve for the problem, $C_d(t)$. This curve is assumed to

be known. Before the strategy's application, the equilibrium obtained is as described in Figure 7b[9].

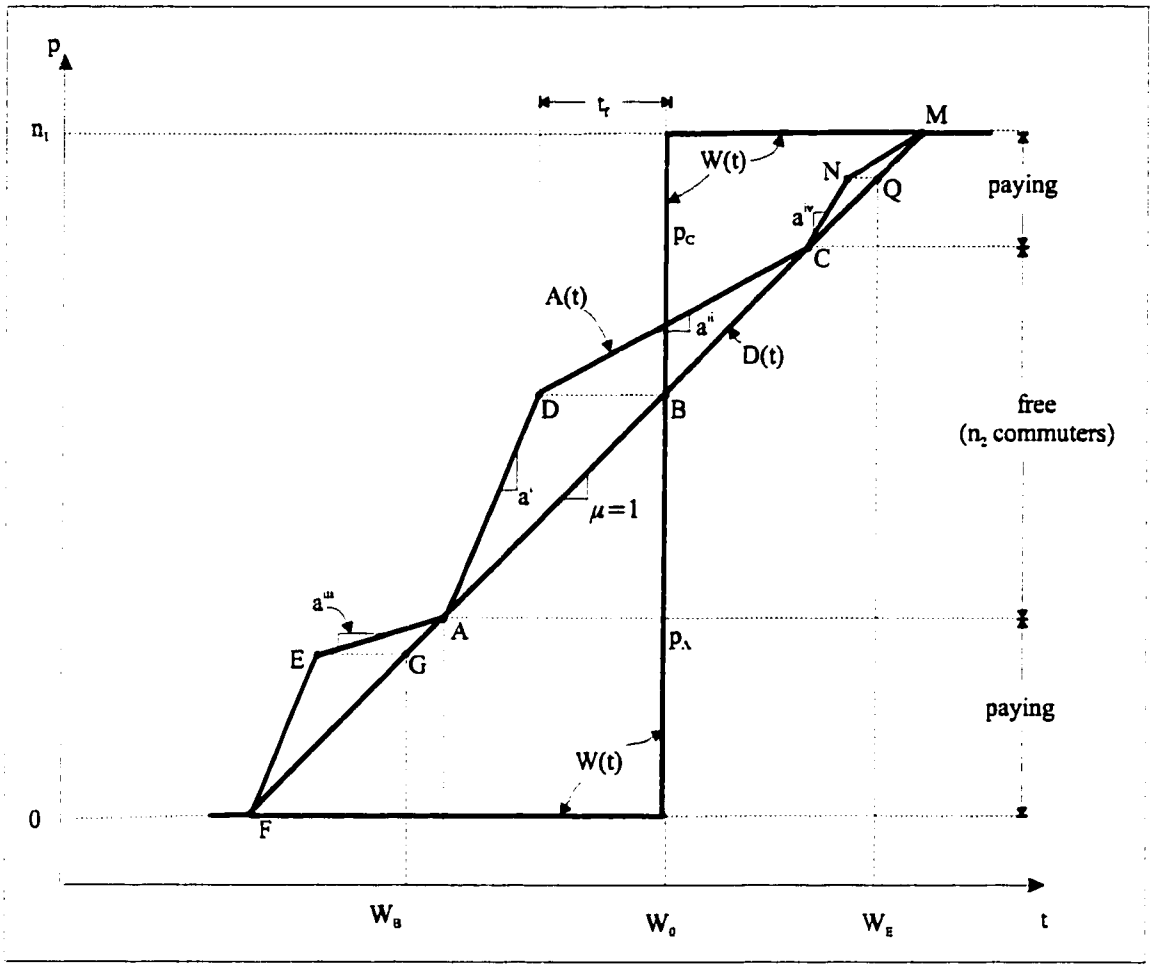

Figure 8: Equilibrium pattern of arrivals with the proposed strategy: commuters with the same desired deadline, W_0, and elastic demand.

We show below that there is a combination of the parameters (f, τ, $[W_B, W_E]$) that achieves the equilibrium described in Figure 8. The equilibrium of Figure 8 is

[9]Figure 7b is similar to Figure 1a (Section 1.2), the only difference being the number of commuters using the bottleneck, before the strategy's application (n_0 now and, "not" N as in Figure 1a). Since the demand is elastic in this section, only n_0 commuters will be willing to use the bottleneck prior to the strategy's application, *e.g.* only n_0 commuters will be willing to *pay* t_0 to pass through the bottleneck (Figure 7a).

assumed to have the following properties (note that they are the same as in (i) and (iii)-(v) of Section2.2)[10]:

(i) the arrival curve exhibits three distinct queuing episodes and the middle queuing episode is geometrically similar to the original one (before the control);

(ii) the departure curve is not shifted (either to the left or to the right);

(iii) the maximum delay in the first and third queuing episodes is experienced by the users departing at the edges of the time window;

(iv) all the free commuters depart in the middle queuing episode and all the paying commuters in the first and third.

(v) the slopes of the arrival curve are still given by (Eqs.(3-4). Section 2.2):

$$a^{i} = \frac{1}{1-e} > 1 \qquad \text{(for segments FE and AD)}$$

$$a^{ii} = \frac{1}{1+L} > 1 \qquad \text{(for segments NM and DC)}$$

To obtain the equilibrium described in Figure 8. we first choose $f \in [0, 1]$. Then the values of the critical costs when the commuters are classified as "free" (t_f) and "paying" (t_p) (see Figure 9) can be determined from:[11]:

$$fC_d(t_f) + (1-f)C_d(t_p) = t_p(e+L)/eL \tag{12}$$

[10]Property (ii) (the same departure curve as without the control (Section 2.2)) does not hold for this case because the demand is elastic (Even tough we assume the departure curve is not shifted either to the left or to the right in Figure 8. the number of commuters using the bottleneck can change, once the strategy is applied.)

[11]As a is the number of commuters using the bottleneck *only* when classified as "free" (*e.g.* $a = C_d(t_f) - C_d(t_p)$, with a fraction f of them being "free" everyday) and b is the number of commuters always using the bottleneck (*e.g.* $b = C_d(t_p)$), we can determine the total number of commuters using the bottleneck, n_1 and, the number of "free" commuters, n_2, as: $n_1 = af + b$ and $n_2 = (a+b)f$. Substituting the values of a and b as a function of the demand curve ($C_d(.)$) and observing that $t_f = n_2 eL/(e+L)$ and, $t_p = n_1 eL/(e+L)$ (Figure8), we obtain the Eqs.(12-13).

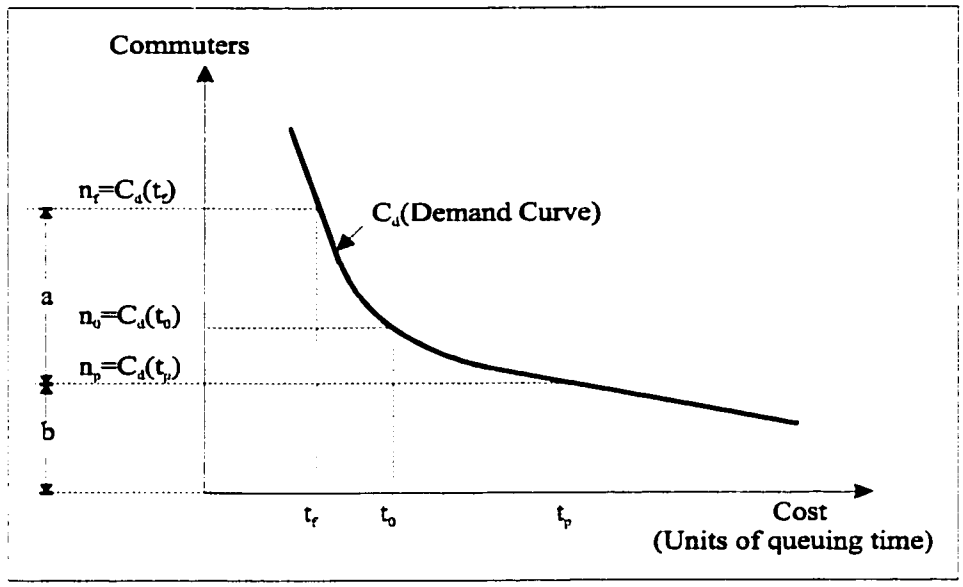

Figure 9: Elastic demand - The commuters are ordered according to their willingness to pay.

$$fC_d(t_f) = t_f(e + L)/eL \tag{13}$$

Once t_f and t_p have been determined, we know n_1 and n_2 (see Figure(8)). Therefore the points F, A, C and M of Figure 8 are also known. Finally to complete Figure 8 and make the paying commuters be in equilibrium, we need to choose a toll rate τ, such that(Eqs.(5-6)) are satisfied: *i.e.*:

$$a^{iii} = \frac{1}{1 + \tau - e} \qquad \text{(for segment AE)}$$

$$a^{iv} = \frac{1}{1 + L - \tau} \qquad \text{(for segment CN)}$$

This selection identifies points E and N. The beginning of the time window, W_B, is determined as the abscissa of the departure curve at the point G which is level with E (see Figure 8). The same logic applies to determine the end of the time window, W_E. As in the previous chapters, the "paying" commuters will depart in the two extreme queuing episodes and the "free" commuters will depart in the middle

queuing episode (This fact derives from properties (i)-(v) and from the choice of the toll rate, τ, according to Eqs.(5-6).

Having established that an equilibrium as in Figure 8 can be obtained, one has to check if it is Pareto improving. To this end, one can identify three sets of commuters: $[n_0, n_f]$, $[n_p, n_0]$, $[0, n_p]$(see Figure 9). The necessary conditions for the savings can be obtained as:

(a) Commuters from positions n_0 to n_f: these commuters certainly will be better off with the strategy's application, because once the strategy is applied they will be using the bottleneck *at least* when classified as "free" (prior to the strategy's application these commuters were *never* using the bottleneck as their willingness to pay was lower than t_0).

(b) Commuters from positions n_p to n_0: these commuters will be better off if their benefits surpluses with the strategy, S_1, (*i.e.* S_1=$f(t_i$-$t_f)$, for $i \in [n_p, n_0]$) are greater than their benefits surpluses prior to the strategy's, S_0, (*i.e.* S_0=t_i-t_0, for $i \in [n_p, n_0]$). Thus, for a Pareto Improving solution one must have positive savings: S_1- $S_0 \geq 0$, or: $f \geq (t_i$-$t_0)/(t_i$-$t_0)$. As the right hand side of the previous inequality has a maximum value when t_i is closer to t_p ($t_i \rightarrow t_p$), we obtain:

$$f \geq \frac{t_p - t_0}{t_p - t_f} \qquad \textit{(Necessary Condition for Pareto Improving)} \qquad (14)$$

(c) Commuters from positions 0 to n_p: it can be shown that these commuters will be better off if the same necessary condition Eq.(14) is satisfied [12].

[12]Similar to the case (ii) these commuters will be better off, if their savings once the strategy is applied (S_1=$f(t_i - t_f)$+(1-f)($t_i - t_p$)), for $i \in [0, n_p]$) are greater than their savings prior to the strategy's application (S_0=t_i-t_0, for $i \in [0, n_p]$), Then for a Pareto Improving solution one must have: S_1- $S_0 \geq 0$ and, the results follow.

One can see that the proposed strategy can be introduced gradually, starting with a small percentage of paying commuters. The next section extends the analytical analysis for the heterogeneous commuters where they have different values of time ("rich" and "poor" commuters).

3.3 Heterogeneous Commuters - Different Values of Time

This section addresses the case of a population of commuters with *different* values of time. The commuters are otherwise identical (as in Section 2.2).

To solve analytically this problem, we assume the existence of two classes of commuters, one with *high* values of time and another one with *low* values of time. In this research, the commuters with *low* values of time are considered to be the "poor" ones, and the commuters with *high* values of time, the "rich" ones. For this scenario, when a "toll" with rate τ is applied inside a time window, $[W_B, W_E]$, the toll penalty in units of queuing time for a "paying" commuter i departing inside the time window depends on whether the commuter is rich or poor. The penalty is given by:

$$\text{Toll Penalty} = \begin{cases} \tau_1 \min[T_d - W_B, W_E - T_d] & \text{(for "poor" commuters)} \\ \tau_2 \min[T_d - W_B, W_E - T_d] & \text{(for "rich" commuters)} \end{cases} \quad (15)$$

where $\tau_1 = \alpha_1\tau$, $\tau_2 = \alpha_2\tau$, and α_i (i=1,2) is the toll conversion rate for the two commuter types. Note that the α_i's are the reciprocal of the values of (queuing) time and that therefore must satisfy: $\alpha_1 > \alpha_2$[13].

Before the strategy's application, since money is not an issue, the equilibrium is still as in Section 1.2, Figure 1a. We want to show there is a combination of parameters (f, τ, $[W_B, W_E]$) achieving the equilibrium of Figure 10. This equilibrium has the following properties:

(i) the departure curve is the same as the original (Figure 1a) without the controls:

[13]For an infinite (n) number of classes of commuters, the commuters could be ordered from the "poorest" commuter to the "richest" one, *e.g.* with: $\alpha_1 > \alpha_2 > \alpha_3 > ... > \alpha_{n-1} > \alpha_n$, being the n_{th} commuter the "richest" commuter.

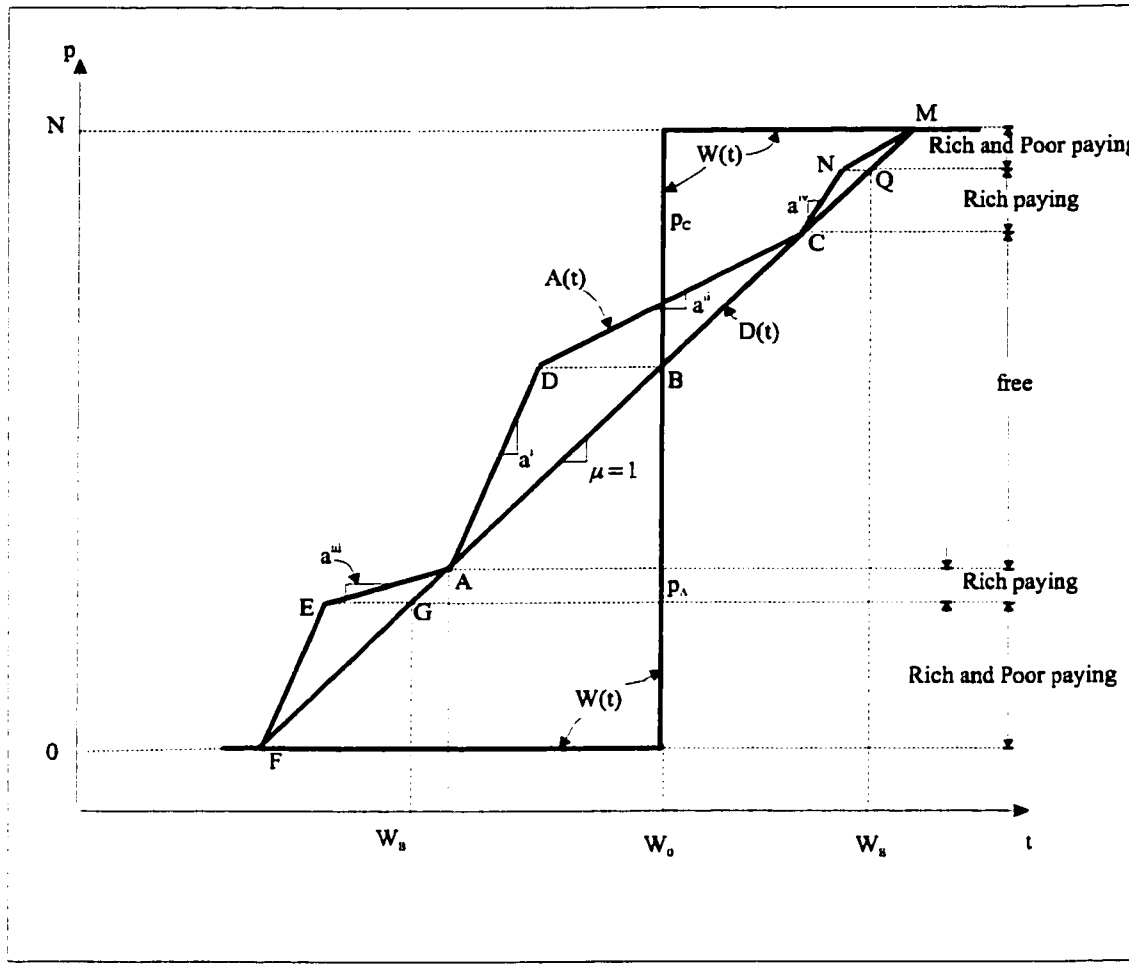

Figure 10: Equilibrium pattern of arrivals with the proposed strategy: commuters with different values of time.

(ii) the arrival curve exhibits three distinct queuing episodes and the middle queuing episode is geometrically similar to the original one (before the control);

(iii) the maximum delay in the first and third queuing episodes is experienced by the users departing at the edges of the time window;

(iv) all the free commuters depart in the middle queuing episode and all the paying commuters in the first and third queuing episodes;

(v) "poor" paying commuters avoid the toll by not passing through the bottleneck inside the time window.

Therefore, as in the previous cases, the slopes a^i (for segments FE and AD) and a^{ii} (for segments DC and NM) are fixed and given by Eqs.(3-4). It is also clear that once more the "free" commuters will be in equilibrium in the middle queuing episode (since it is geometrically similar to the queuing triangle before the strategy's application).

Now, we show that it is possible to choose the parameters of the strategy to ensure that:

(a) the "poor" paying commuters are in equilibrium while departing in the first and third queuing episodes, and outside the time window and;

(b) the "rich" paying commuters are in equilibrium departing in the first and third queuing episodes.

Since the departure curve does not change position, the points F and M have already been determined. Furthermore, once one chooses $f \in [0, 1]$ the points A and C are also determined. Then, to make the paying commuters be in equilibrium (as described in Figure 10), the toll rate τ must be such that:

$$0 < a^{iii} = \frac{1}{1 + \tau_2 - e} < 1 \qquad \text{(for segment AE)} \tag{16}$$

$$a^{iv} > \frac{1}{1 + L - \tau_2} > 1 \qquad \text{(for segment CN)} \tag{17}$$

which implies that $L/\alpha_2 < \tau < (L+1)/\alpha_2$, as: $\tau_2 = \alpha_2 \tau$.

The intersection of FE and AE determines the break in the arrival curve, and a corresponding point in the departure curve, G. The abscissa of this point is the beginning of the time window W_B. The same construction determines W_E.

As the values of the slopes for the lines AE and CN satisfy Eqs.(16-17), we can see that the equilibrium obtained will be the one described in Figure 10, with no "poor" paying commuters departing inside the time window.

Finally, to obtain the equilibrium described in Figure 10, the number of "rich" paying commuters available to depart in the first and the last queuing episodes must be enough to fill *at least* the positions spanned by the lines AE and CN. These amounts of rich paying commuters can be determined from the geometry of the curves as: (1) $(1\text{-}f)z/\tau_2$ for the positions spanned by the line AE in the first queuing episode[14], and (2) an identical amount for the third queuing episode. Therefore, as the available positions are $2(1\text{-}f)z/\tau_2$ and this number must equal or not exceed $(1\text{-}f)n_{rich}$ (with n_{rich} being the total number of "rich" commuters in the population):

$$n_{rich} \geq \frac{2z}{\tau_2} \tag{18}$$

Since we stipulated that $\tau_2 = \alpha_2\tau < (L+1)$ (see Eqs.(16-17)), a value of τ satisfying (18) can be found if n_{rich} satisfies:

$$n_{rich} \geq \frac{2z}{L+1} \tag{19}$$

If the number of *rich* commuters is so low that Eq.(18) can not be satisfied, this may suggest that some other kind of equilibrium should be pursued. Whether if Pareto improving can still be obtained we do not known, but as the commuters still have some flexibility with the proposed strategy, one could expect a more equitable distribution of gains and losses. The appendix A extends this analysis to show that it is possible to obtain a Pareto improving solution if a similar strategy is applied for more than two classes of commuters. These chapters have shown that it is possible to obtain a Pareto improving solution for the heterogeneous scenario (with only one

[14]This solution can be derived from the geometry of Figure 10, as $1/(1\text{-}e)$=FG/(FG-EG), $1/(1+\tau_2\text{-}e)$=EA/(EA+EG). As EA+EF=(1-f)x, the result just follows. The same logic can be applied for the positions spanned by the line CN.

dimension of the commuters varying). It is encouraging as in the next chapter we will study the general case.

4 Heterogeneous Commuters - General Case

This section presents the results obtained for the general scenario in which the commuters have *different* desired deadlines, *different* values of time and *different* willingness to pay. It is not easy to obtain the results analytically, therefore we applied a simulation to obtain the make the analysis. Section 4.1 describes the simulation framework applied to study the day-to-day dynamics of the present research, and Section 4.2 presents the results.

4.1 Simulation Framework

We assume that there is a population of commuters who repeat their decisions everyday. They choose their arrival time at the bottleneck based on their historical perception of the system. For the very first run of the simulation (representing the first day), we assume somewhat arbitrarily that only the commuters willing to pay more than a "standard" cost[15], z, use the bottleneck and, that they all have the *same* arrival time. Thus the first cumulative arrival curve, $A_1(t)$, is determined in this way. The rules of the bottleneck also yield the first cumulative departure curve, $D_1(t)$. For the second day, all the commuters are assumed to make their travel decisions based on these first two curves. Each commuter first calculates the arrival time which would have minimized his total cost, given curves $A_1(t)$ and $D_1(t)$. (S)he is then assumed to arrive at this time on day 2, but only if this minimum cost is lower than the commuter's willingness to pay. Once the new arrival and departure times for all the commuters have been determined in this way, we can construct the new cumulative arrival and departure curves for day 2, $A_2(t)$ and $D_2(t)$.

[15]This cost z is the equilibrium cost obtained if the commuters were identical (Section 1.2) given by: $z = neL/(e + L)$, where n is the total number of commuters in the population.

For days i=3,4,..., the process is repeated, but instead of using the curves of the previous days, the commuter is assumed to base her)his) decision on perceived curves $A_i^p(t)$ and $D_i^p(t)$ that incorporates the system's past history as follows:

$$A_1^p(t) = A_1(t) \tag{20}$$

$$D_1^p(t) = D_1(t) \tag{21}$$

$$A_i^p(t) = \alpha A_{i-1}^p(t) + (1-\alpha)A_i(t) \qquad (\text{for } i \geq 2) \tag{22}$$

$$D_i^p(t) = \alpha D_{i-1}^p(t) + (1-\alpha)D_i(t) \qquad (\text{for } i \geq 2) \tag{23}$$

where $\alpha \in [0.1]$. As one can see, Eqs.(20-23) are "smoothing" out the differences between the cumulative arrival and departure curves from one iteration of the simulation to the next. Note that all the commuters base their decisions on the same perceptions.

In our simulation, commuters change their arrival time only if the calculated cost based on these perceptions is smaller than the cost they experienced on the last day. In order to avoid unreasonably frequent changes in the commuter's arrival times; *e.g.* if they would improve the commuter's cost only by just a very low amount, we introduce randomness borrowing some ideas from the field of "simulated annealing"[16]. If a commuter observed that his new *possible* arrival time has a lower cost than his most recent one, we then calculate an annealing probability, p_{anneal}[17]. The commuter will change his arrival time if after drawing a uniform random number u ($u \in [0, 1]$), we find that $u \leq p_{anneal}$.

[16]Another important method implemented in the simulation framework is the "simulated annealing" one, being a method that has attracted significant attention as suitable for optimization or simulation problems of large scale (see Kirkpatrick et all, 1983; Otten and van Ginneken, 1989; Press et al., 1994). The essence of the process is "*slow* cooling, allowing ample time for redistribution of the *atoms* as they lose mobility". This is the technical definition of *annealing*, and it is essential for ensuring that a low energy state will be achieved. Following, we describe how the simulation annealing is applied in this research.

[17]The annealing probability is a function of the improvement in cost and the value of the present run. Thus, if the improvement in cost is "high" and the present iteration has a "low" value (*i.e.* we are at the beginning of the simulation) the probability will have a large value.

We expected that after a sufficiently large number of iterations(*i.e.* days), the system could converge to an equilibrium. Then, we could compare the new and the initial costs (before the strategy's application), for each and every commuter to see if a Pareto improving solution was obtained. Appendix B presents the flowchart of the simulation and the source code.

4.2 Simulation Results

First, we checked if the simulation would converge to an equilibrium solution for a case where the equilibrium is known. We chose for this test the case of Figure 1b where the commuters differ only with respect to their desired deadline. The initial arrival curve, $A_1(t)$, was obtained assuming all the commuters (the ones willing to pay more than z) arriving at time 0 (zero). From $A_1(t)$, we determined the first cumulative departure curve, $D_1(t)$. For this first case, we must say that the problem took a large number of iterations to achieve the equilibrium[18]. We observed the system to be quite unstable because as the commuters differ only according to their desired deadline, the commuters would take very "similar" *decisions* (they would choose basically the same arrival time to minimize their costs). Even tough there is this stability issue. Figure 11 shows that the simulation results eventually approached the expected theoretical equilibrium. We would expect this problem to diminish when the commuters disagree in more than one of their characteristics, which should be the case for the heterogeneous population we are about to study[19].

The simulation was then used to study the day-to-day dynamics of the commuters when the proposed strategy is implemented. It was assumed that a population of 160 commuters could use the bottleneck everyday, with these commuters having *different* desired deadlines, *different* values of time and *different* willingness to pay.

[18]In our example, it took 92,250 (!!) iterations to achieve the equilibrium for an α=0.65.

[19]As the differences among the commuters increase, their decisions will differ even more.

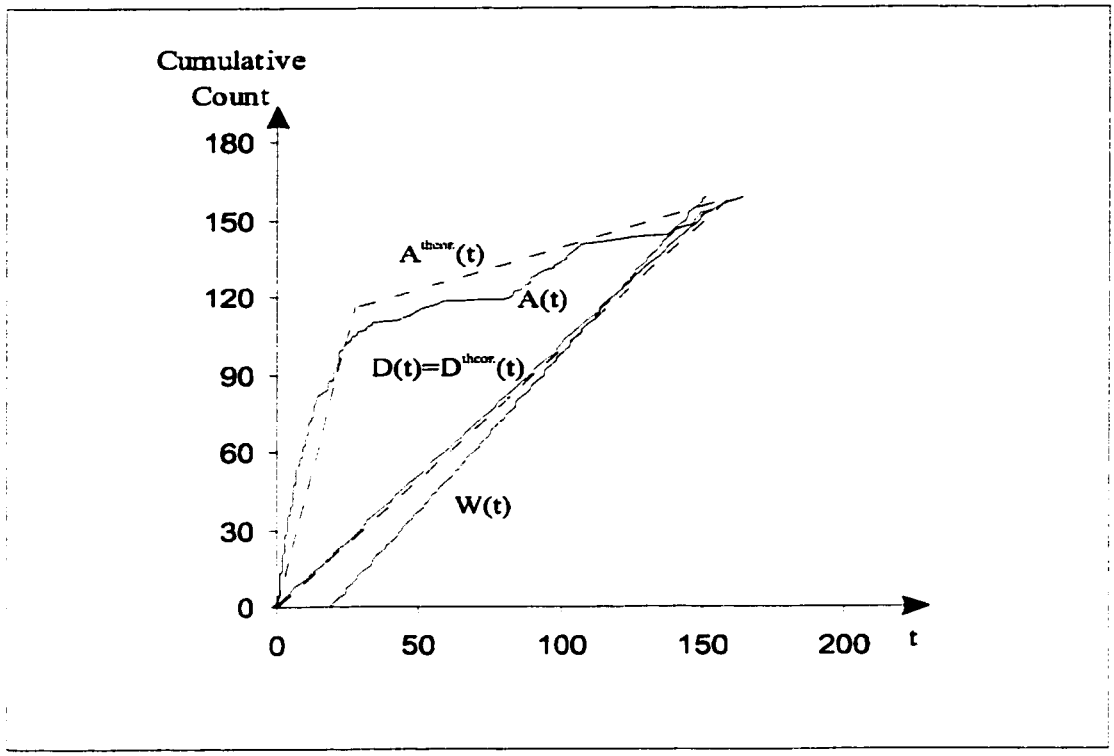

Figure 11: Equilibrium obtained with the simulation framework - "do-nothing" strategy.

Having obtained the equilibrium without applying any strategy (Figure 11), the pure pricing scheme was then studied (with all the commuters classified as "paying": f=0). For this problem, the results (given in Figure 12) show that even tough more commuters decided to use the bottleneck once the "pure" pricing policy was implemented[20], the most of the commuters who became better off were the ones with "low" values of money (the "rich" commuters). Figure 12 also presents very intuitive results as the most commuters who are worse off are the "poor" commuters for whom the trip was not very important (the ones who have "low" willingness to pay).

[20]Before the application of the strategy there were 90 commuters using the bottleneck, and after its application, 128 commuters.

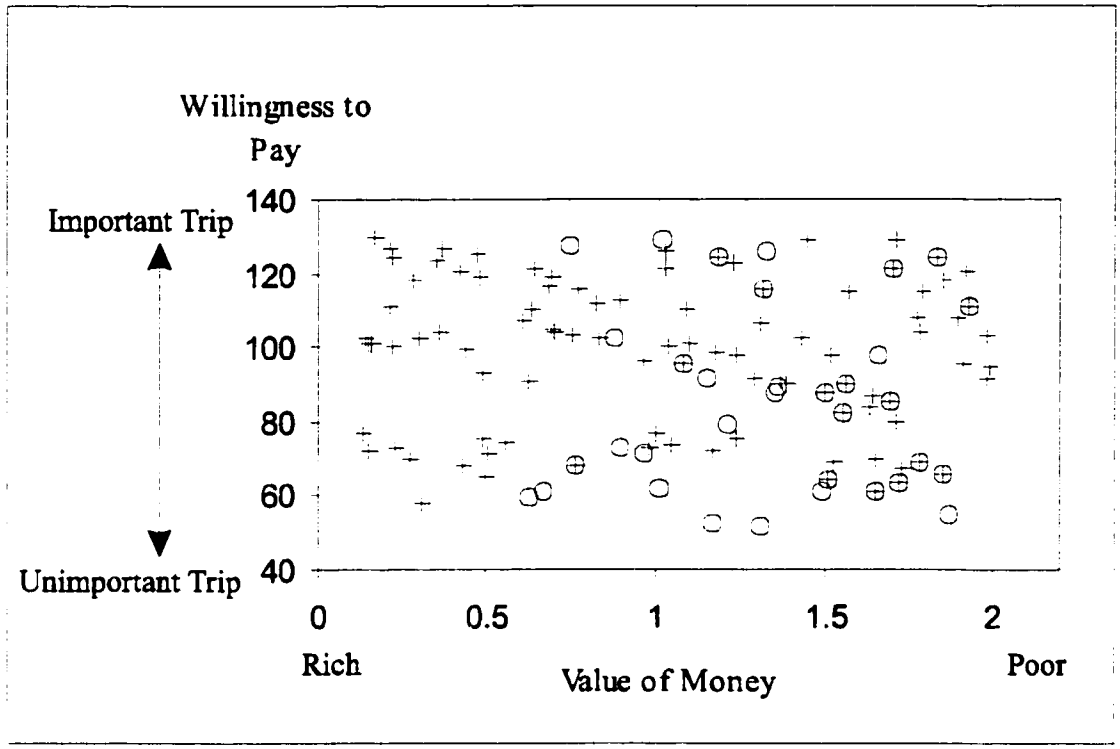

Figure 12: Application of the strategy (pure pricing, f=0): commuters better off(+): commuters worse-off(o).

Next, the scenario where half of the commuters were classified as "free" (*e.g.* f=0.5 in our strategy) was studied. For this problem, the results (see Figure 13) show that the most of the commuters who became better off were the ones with "high" willingness to pay, *e.g.* the commuters for whom the trip was important. Therefore, from our results we can conclude the proposed strategy to be a more equitable scheme than the pure pricing policy.

Further comparisons of the two schemes (f=0 and f=0.5) are described in Figure 14, where it becomes clear the superiority of the results obtained with the strategy's application. Even tough we did not obtain a Pareto improving solution for the example just described, it does *not* mean it can not be obtained. If one chooses different values for the parameters when applying the simulation, making for instance

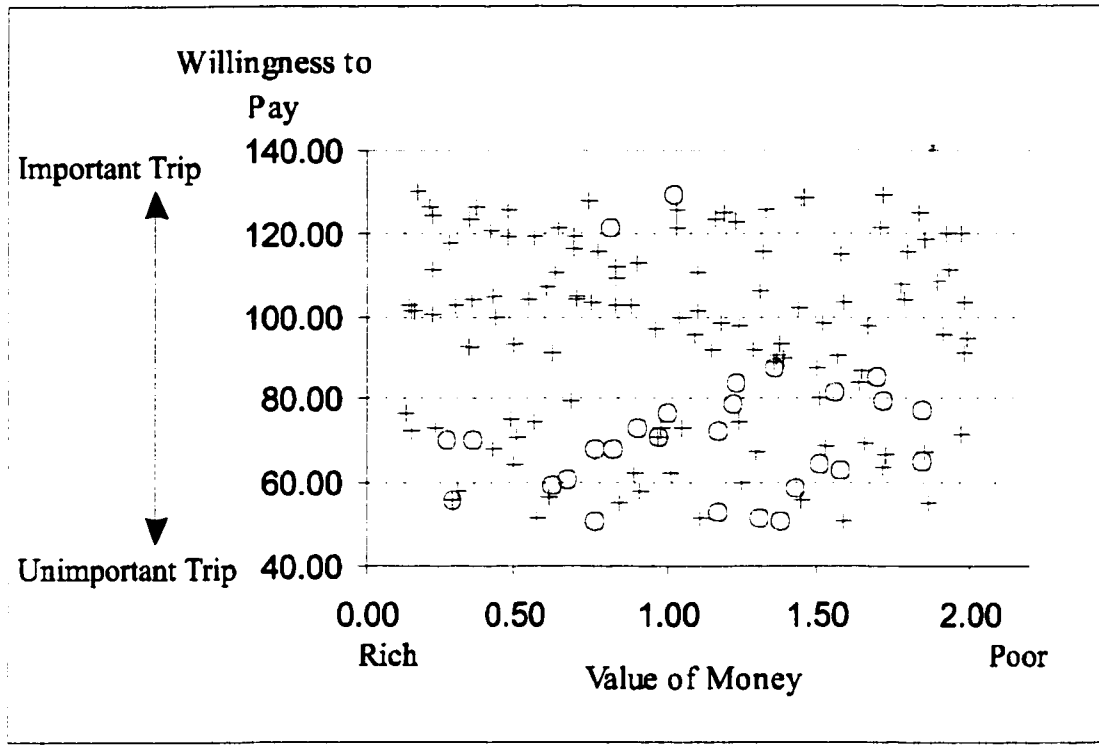

Figure 13: Application of the strategy (f=0.5): commuters better off(+); commuters worse-off(o).

f close to 1 (almost all the commuters are classified as "free"), maybe better results could have been obtained, *e.g.* almost or even *all* the commuters could be better off. Certainly, these are issues to be addressed in the future.

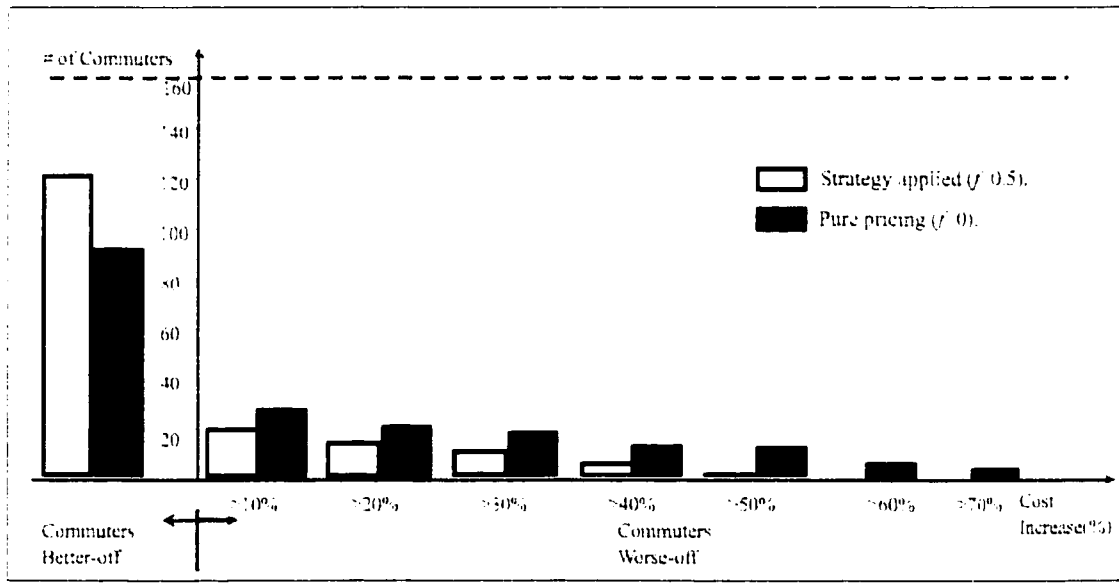

Figure 14: Distribution of the commuters: Change in cost.

5 Conclusions

The final chapter contains a summary of the research and an outline of some areas for further research.

5.1 Summary

This dissertation describes a more equitable strategy for the time-dependent morning commute problem. The first section contained a description of the problem. It presented the background and the literature review for the problem in study. It also explained why the existing schemes did not achieve any "real" monetary gains for the commuters, as what the users of the bottleneck save in queuing time, they pay in the form of tolls.

Chapter 2 offered a detailed description of the proposed strategy. It explained the tolling scheme and it also described some examples of its implementation. The simplicity of the proposed strategy suggests it can be easily implemented. The following chapters presented the results obtained with the strategy. The insights gained from the studies of the simpler scenarios were applied to the more complicated ones.

The proposed strategy made all the commuters better off for the simplest case, in which all the commuters were identical. For this case, a maximum of 25% of savings can be obtained for each commuter. This is a remarkable result as the proposed strategy benefited all the commuters even without returning the collected revenues to the payers.

Analytical results were also obtained for these cases simplified where the commuters could differ in 1 dimension only; *e.g.* they could have either *different* desired deadlines or, *different* values of time or, *different* willingness to pay. The equilib-

ria obtained with the strategy for each of these cases were described. Necessary conditions for the existence of a Pareto improving solution were also presented.

Finally, the research described a simulation framework to solve a general scenario, where commuters had *different* desired deadlines, *different* values of time and *different* willingness to pay. The results obtained suggest that the proposed strategy obtains a more equitable solution than existing schemes, such as "pure" pricing. Even tough a Pareto improving solution was not obtained for this general case, further analysis should be made before drawing definite conclusions.

5.2 Areas of Further Research

One of the main problems with congestion management is its implementation. It would be important to address how to implement the proposed strategy gradually. Once each strategy's step was implemented, can we fine tune the parameters of the strategy according to the commuters' reaction.

The present research could be extended to determine if a few exemptions should be given to the commuters, as some of them would probably use the bottleneck only during few days of the year (*i.e.* due to some medical emergency, for instance). One could also investigate how many exemptions should be given for each commuter if a Pareto improving solution was to be achieved. This exemption approach could make easier for some commuters to accept the strategy[21].

The strategy could also be combined with the use of High-Occupancy-Vehicle lanes (*e.g.* HOV lanes). As in the proposed strategy the toll charged can be quite high, some commuters could be willing to use the HOV lanes when classified as

[21]The commuters would know that even if they passed through the bottleneck when classified as "paying", *and* during the toll's application, they would "not" necessarily have to pay the toll, as they could use one of their exemptions.

"paying". Section 4.2 also showed the proposed strategy to obtain a more equitable solution than existing schemes, like "pure" pricing. One could try to determine if there is a set of the proposed strategy's parameters (f, τ, $[W_B, W_E]$) that makes all the commuters better off.

It was also assumed that there is a linear dependence on time and money for each commuter's utility. One should relax this assumption and see if a Pareto improving solution could still be obtained.

Concluding this research, we observe that the costs related to traffic congestion are becoming unbearable. New ideas have and must be developed to alleviate the negative effects of this increase in congestion costs. We hope that this research can become a valuable step in that direction.

References

Arnott, R., De Palma, A. and Lindley, R. (1988) Schedule delay and departure time decisions with heterogeneous commuters, Transportation Research Record 1197, 56-67.

Arnott, R., De Palma, A. and Lindley, R. (1990) Economics of a bottleneck, Journal of urban economics 27, 111-130.

Arnott, R., De Palma, A. and Lindsey, R. (1998) Recent developments in the bottleneck model, In: K.J. Burton and E.T. Verhoef (1998) Road pricing, traffic congestion and the environment: issues of efficiency and social feasibility Edward Elgar, Cheltenham (forthcoming).

Banister, D., Hervik, A., Braaten, S. and Pol, H. (1996) Round table 97: charging for the use of urban roads, European Commission.

Ben-Akiva, M., Cyna, M. and De Palma, A. (1984) Dynamic model of peak period congestion, Transpn. Res.-B, Vol.18B, No.4/5, pp.339-355.

Ben-Akiva, M., De Palma, A. and Kanaroglou, P. (1986) Dynamic model of peak period traffic congestion with elastic arrival rates, Transportation Science, Vol. 20, No.2, 164-181.

Berstein, D.H. and Hiller, J. (1998) User neutral congestion pricing, Presented to the 77th. TRB - Transportation Research Board, Washington, D.C., USA.

Bristow, A., Mackie, P.J., Nellthorp, J. and Jansen, G.D. (1998) Costs prices and values in the appraisal of transport projects - European principles and practice, Institute of Transportation Studies, University of Leeds and PLANCO Consulting GmbH, Presented to the 8th. WCTR - World Conference on Transport Research, Antwerpen, Belgium.

Cohen, Y. (1987) Commuter welfare under peak-period congestion tolls: who gains and who loses?, In: Commuter welfare and congestion tolls.

Daganzo, C.F. (1985) The uniqueness of a time-dependent equilibrium of arrivals at a single bottleneck, Transportation Science 19, No. 1, 29-37.

Daganzo, C.F. (1992) Restricting road use can benefit everyone - Part II: Time-of-day restrictions that encourage earlier arrivals, Working paper UCB-ITS-WP-92-8, Institute of Transportation Studies, University of California at Berkeley, Berkeley, CA, USA.

Daganzo, C.F. (1995) A Pareto optimum congestion scheme, Transportation Research B, Vol. 29B, No.21, 139-154.

Deakin, E. (1989) Toll Roads: A new direction for U.S. Highways ?, Dept. of City and Regional Planning, Institute of Transportation Studies, University of California at Berkeley, Berkeley, CA, USA.

Deakin, E. (1994) Urban transportation congestion pricing: effects on urban form , in Curbing Gridlock: Peak-Period fees to relieve traffic congestion, Special Report 242, Vol. 2, Transportation Research Board, National Research Council, Academy Press, Washington, DC.

De Palma, A., Ben-Akiva, M., Lefevre, C. and Litinas, N. (1983) Stochastic equilibrium model of peak period traffic congestion, Transportation Science, Vol.17, No.4, 430-453.

Fargier, P.H., Morin, J.M. and Ducornet, D. (1979) Reducing travel time by freeway ramp metering, especially when peak traffic demand exceeds corridor capacity, Proceedings of the International Symposium on Traffic Control Systems/C.F. Daganzo (editor), Berkeley, CA, USA.

Fargier, P.H. (1981) Influence du mecanisme de choix de l'heure de depart sur la congestion du traffic routier - approche theoretique, 223-263, paper presented at the Eight International Symposium on Transportation and Traffic Theory, Toronto, Canada.

Financial Times (1997), Americans count the cost of road congestion, 15/Nov., London, England.

Gomez-Ibanez, J. and Small, K.A. (1994) Road pricing for congestion management: a survey of international practice, Synthesis of Highway Practice 210, Transportation Research Board, National Academy Press, Washington, D.C., USA.

Hendrickson, C. and Kocur, G. (1981) Schedule delay and departure time decisions in a deterministic model, Trans. Sci. 15(1), 62-77.

Hurdle, V.F. (1974) The effects of queuing on traffic assignment in a simple road network, Proceedings of the Sixth International Symposium on Transportation and Traffic Theory, Sydney, Australia.

Hurdle, V.F. (1981) Equilibrium flows on urban freeways, Trans. Sci. 15, 255-293.

Keeler, T.E., Merewitz, L.A. and Fisher, P. (1975) The full costs of urban transport - part III: automobile costs and final intermodal cost comparisons, Institute of Urban and Regional Development, University of California at Berkeley, Berkeley, CA, USA.

Keeler, T.E. and Small, K.A. (1977) Optimal peak-load pricing, investment and service levels on urban expressways, Journal of Political Economy 85, 1-25.

Laih, C.H. (1994) Traffic problems and policies in Taipei, Taiwan, Presented at the Workshop of Regional Science and Urban Economics in Tsukuba University, Tsukuba, Ibaraki, Japan.

Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P. Science, vol. 220, pp.671-680, 1983.

Kuwahara, M. (1985) A time-dependent network analysis for highway commute traffic in a single core city, Ph.D. Thesis, Dept. of Civil Eng., Univ. of California at Berkeley, Berkeley, CA, USA.

Mogridge, M. J. (1990) Travel in towns: jam yesterday, jam today and jam tomorrow?, McMillan Reference Books, London and Basingstoke, Great Britain.

Morin, J.M. (1980) Controlling the access to saturated urban motorways corridors (Le Controle d'access dans les corridors autoroutiers urbains satures), T.E.C. (Transport Environnement Circulation) no. 39, Mar.-Apr., 44-50.

Newell, G.F. (1987) The morning commute problem for non-identical travelers.Trans. Sci. 21(2), 74-88.

Odeck, J. and Bräthen, S. (1998) The planning of toll roads - do public attitudes matter ? The case of the Oslo toll ring, Presented to the 77th. TRB - Transportation Research Board, Washington, D.C., USA.

Otten, R.H.J.M., and van Ginneken, L.P.P.P., IEEE Transactions on Computer Aided Design, vol. CAD-2, pp.215-222, 1983.

Press, W.H., Teukolsky, S.A., Flannery, B.P. and Vettering, W.T., Numerical recipes in C - The art of scientific computing, 2nd. ed., 1994, , USA.

Rico, A., Mendonza, A., Mayoral, E. and Rivera, C. (1998) Criteria for setting tariffs on toll highways in Mexico.Transportation Research Board 1558, 39-45, Washington, D.C., USA.

Smith, M.J. (1984) The existence of a time-dependent distribution of arrivals at a single bottleneck, Trans. Sci. 18, 385-394.

The Economist (1997), Jam today, road pricing tomorrow, 21-23, 6-12/Dec., London, England.

Vickrey, W.S. (1963) Pricing and resource allocation in transportation and public utilities: pricing in urban and suburban transport, Am. Econ. Rev. 53, No. 2.

Vickrey, W.S. (1969) Congestion theory and transport investment, Am. Econ. Rev. 59, 251-261.

Wachs, M. (1988) When planners lie with numbers: an exploration of data, analysis, and planning ethics, D 885, Graduate School of Architecture and Urban Planning, University of California, Los Angeles, CA, USA.

Wachs, M. (1991) Pricing as a response to congestion and air pollution in California, D 9206, Graduate School of Architecture and Urban Planning, University of California, Los Angeles, CA, USA.

Wachs, M. (1994) Will congestion pricing ever be adopted ?, Access No.4, 15-19, Spring, University of California Transportation Center, Berkeley, CA, USA.

Wardrop, J.G. (1952) Some theoretical aspects of road traffic research, Proc. Inst. Civ. Eng., Part II, 1(2), 325-362; Discussion, 362-378.

Appendix A -n classes of Values of Time

This appendix extends the analysis of Section 3.3 allowing the "poor" *paying* commuters to depart inside the time window and, it also describes the equilibrium for n classes of commuters (*i.e.* not only for two classes of commuters, "poor" and "rich".) It describes the equilibrium obtained if one uses multiple toll rates, having two time windows: $[W_{B1}, W_{E1}], [W_{B2}, W_{E2}]$ with the toll rate changing in the intervals: $[W_{B1}, W_{B2}], [W_{B2}, W_{E2}], [W_{E2}, W_{E1}]$.

Therefore, as in the previous cases, the slopes a^i (for segments FE and AD) and a^{ii} (for segments DC and NM) are fixed and given by Eqs.(3-4). It is also clear that the "free" commuters will be in equilibrium in the middle queuing episode (since it is geometrically similar to the queuing triangle before the strategy's application).

We shall show to be possible to choose the parameters of the strategy to ensure that:

(a) "rich" and "poor" paying commuters depart before the same desired deadline (making two queuing episodes),

(b) "rich" and "poor" paying commuters depart after the same desired deadline (making two more queuing episodes).

Since the departure curve does not change position, the points F and M have already been determined (the demand is inelastic). Furthermore, choosing $f \in [0, 1]$ the points A and C are also determined. Thus, to obtain the equilibrium pattern of Figure 15), the toll rate τ must be such that:

(I) for the first queuing episode, the toll rate τ_I, satisfies:

$$\frac{e}{\alpha_1} < \tau_I < \frac{e}{\alpha_2} \quad for\ [W_{B1}, W_{B2}] \tag{24}$$

(II) for the second queuing episode, the toll rate τ_{II}, satisfies:

$$\frac{L}{\alpha_2} < \tau_{II} < \frac{L+1}{\alpha_2} \quad for\ [W_{B2}, W_{E2}] \tag{25}$$

(III) for the third queuing episode, the toll rate τ_{III} , satisfies:

$$\frac{L}{\alpha_1} < \tau_{III} < min[\frac{L}{\alpha_2}, \frac{1+L}{\alpha_1}] \quad for\ [W_{E2}, W_{E1}] \tag{26}$$

Therefore, the toll with the highest rate must be charged in the middle interval ($[W_{B2}, W_{E2}]$) where a fraction of the "rich" paying commuters depart.

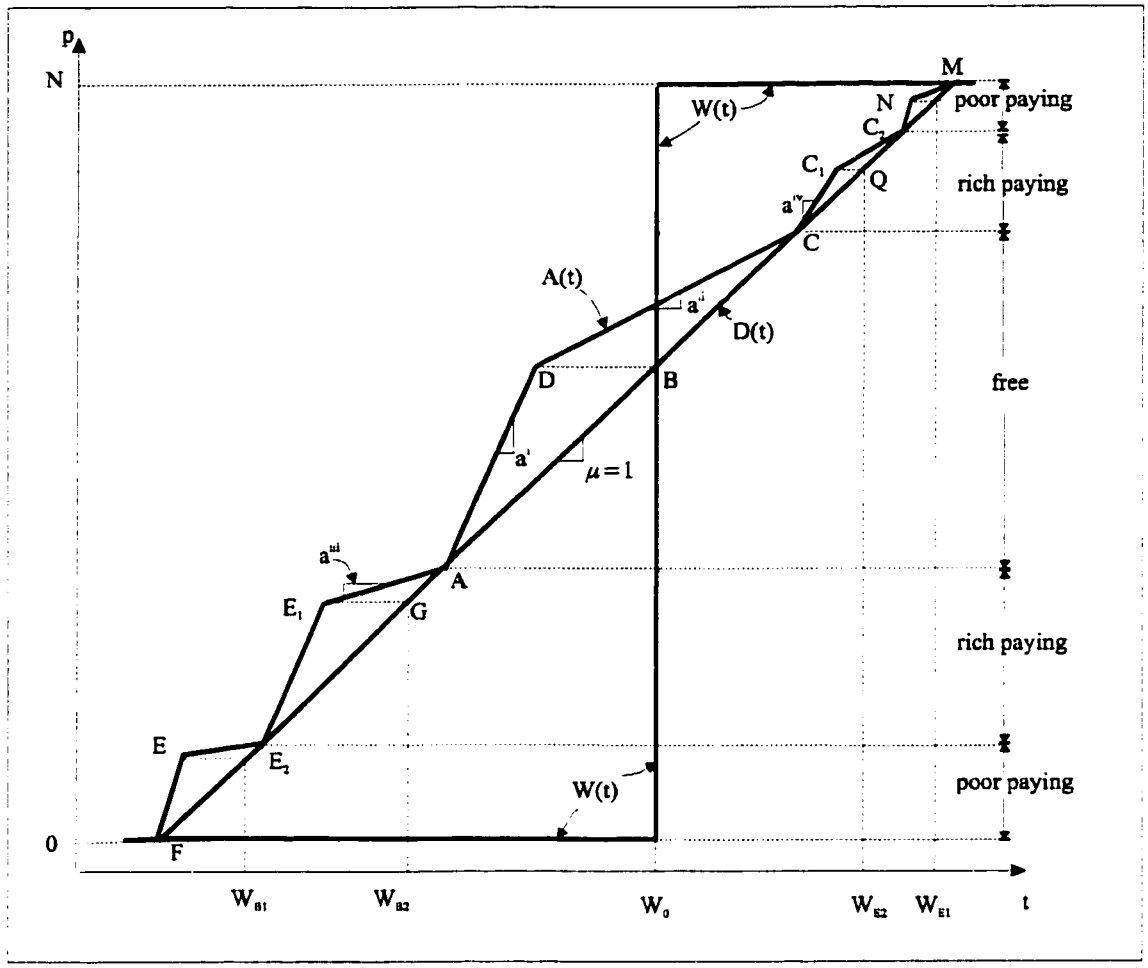

Figure 15: Equilibrium pattern of arrivals with the proposed strategy: commuters (n classes) with different values of time.

To obtain the equilibrium described in Figure 15, one must choose f, and as the departure curve does not change and the demand is inelastic, the points F, A, C and M are determined. Then, choose the toll rates $\tau_I, \tau_{II}, \tau_{III}$, satisfying the Eqs.(24-26). Knowing how many commuters exist in each class, one can determine the points E_2 and C_2 (see Figure 15), applying the same logic of Section 3.3). Finally, determine the points E_1 and C_1 applying the slopes as given in Figure 15 and, the breaks in the arrival curve determined (obtaining W_{B2} and W_{E2}). Similarly, one can determine the points E_2 and C_2 and, then W_{B1} and W_{E1}.

The derivation above can be generalized for n classes of commuters, where now there will be, $2n - 1$ queuing episodes, with the following intervals: $[W_{B1}, W_{B2}]$, $[W_{B2}, W_{B3}]$,..., $[W_{Bn-1}, W_{Bn}]$,$[W_{Bn}, W_{En}]$,$[W_{En}, W_{En-1}]$,...,$[W_{E2}, W_{E1}]$, and the

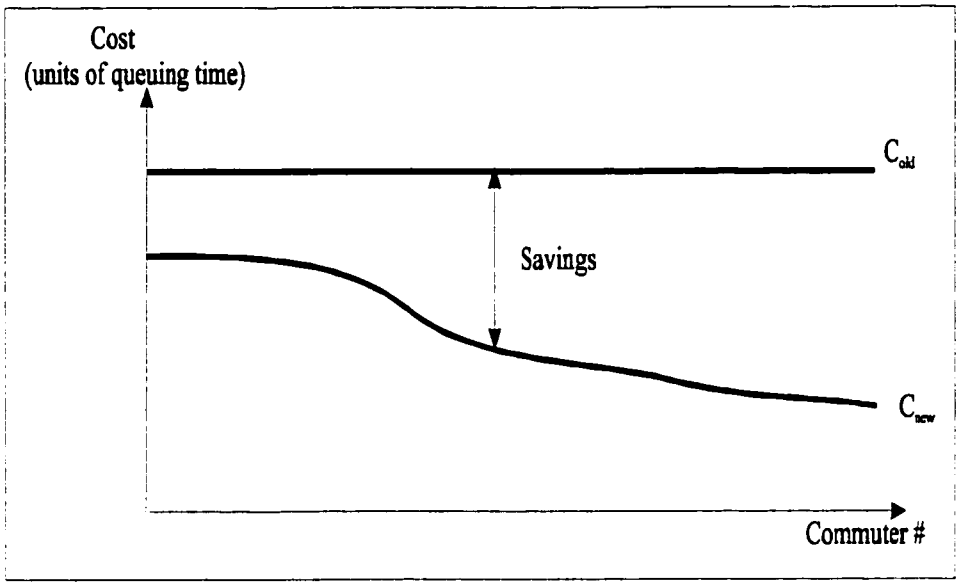

Figure 16: Savings - n classes of commuters.

slopes satisfy similar constraints to Eqs.(24-26):

$$\frac{e}{\alpha_1} < \tau_I < \frac{e}{\alpha_2} \qquad for\ [W_{B1}, W_{B2}]$$

$$\frac{e}{\alpha_2} < \tau_{II} < \frac{e}{\alpha_3} \qquad for\ [W_{B2}, W_{B3}]...$$

$$\frac{e}{\alpha_{n-1}} < \tau_{n-1} < \frac{e}{\alpha_n} \qquad for\ [W_{B(n-1)}, W_{Bn}]$$

$$\frac{L}{\alpha_n} < \tau_n < \frac{L+1}{\alpha_n} \qquad for\ [W_{B(n)}, W_{E(n)}]$$

$$\frac{L}{\alpha_{n-1}} < \tau_n < min[\frac{L}{\alpha_n}, \frac{1+L}{\alpha_{n-1}}] \qquad for\ [W_{En}, W_{E(n-1)}]...$$

$$\frac{L}{\alpha_1} < \tau_{2n-1} < min[\frac{L}{\alpha_2}, \frac{1+L}{\alpha_1}] \qquad for\ [W_{E2}, W_{E1}]$$

Therefore, it would be possible to charge a toll for an infinite number of classes where the queuing episodes for the "paying" commuters could be completely eliminated, with the "paying" commuters being charged an equivalent amount of toll for that.

The total savings for this new strategy would be given by Figure 16 with the commuters ordered from the "poorest" (lowest savings) to the "richest" (highest savings) (before the strategy's application all the commuters had the same cost: $C = z = neL/(e + L)$).

Appendix B - Algorithm and Source Code (C++)

This appendix presents the flowchart of the simulation (Figure 17) and the source code (C++).

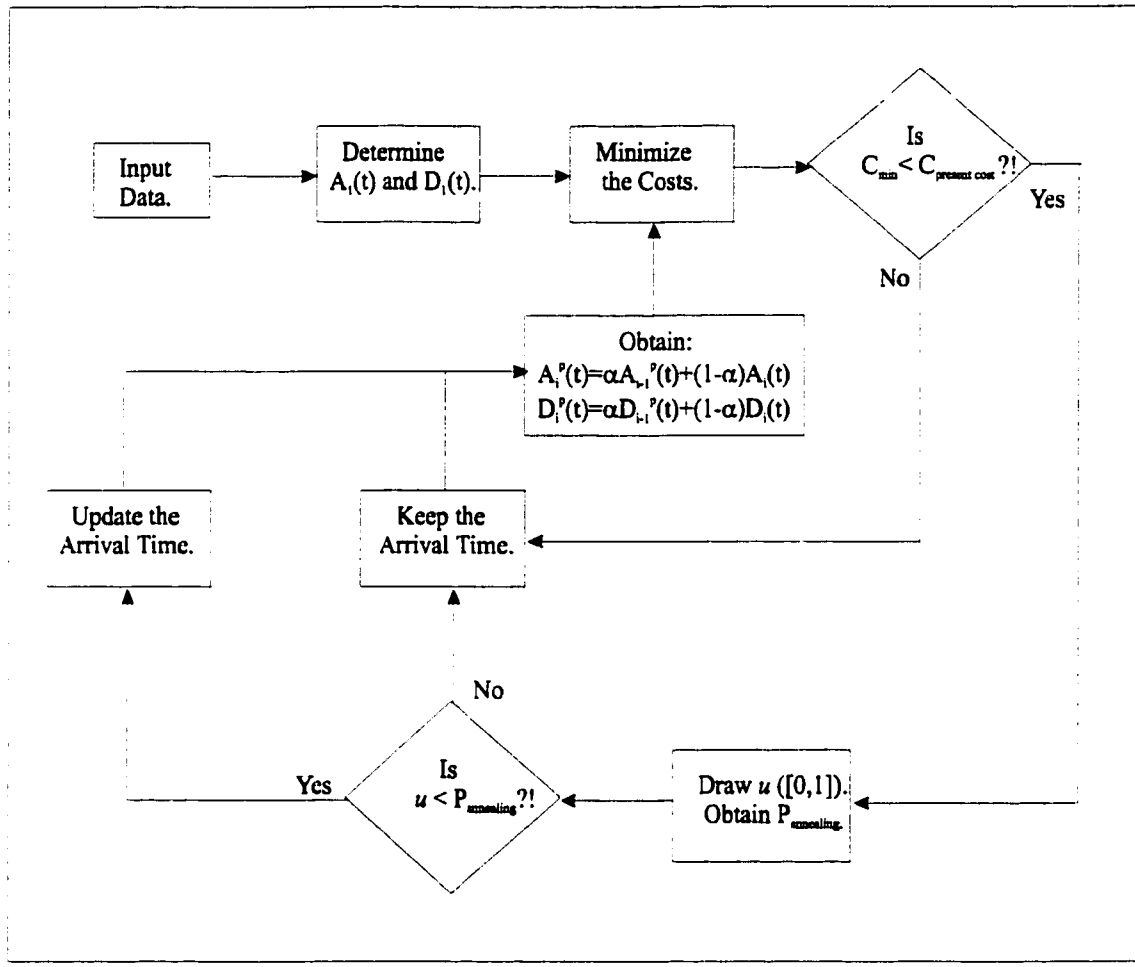

Figure 17: Flowchart for the simulation.

```
/*************************************************************
* Reinaldo C. Garcia.
* The Main Program (developed in C++) for the Thesys:
* Applying a Pareto improving strategy for the time
* dependent morning commute problem.
*************************************************************/


#include <iomanip.h>
#include <cmath>
#include <stdlib.h>
#include <string.h>
#include <numeric>
#include <functional>
#include <fstream.h>
#include <iostream.h>
#include <math.h>
#include <time.h>
const int toll_funct = 2; // IF IT IS 2, FOR INSTANCE, IT IS THE USUAL ONE,
// JUST ONE  TIME WINDOW
const int read_fileinD=1;  // TO START A NEW FILE READING IT FROM THE BEGINING:
  //POSITION, ARRIVAL TIME, DEPARTURE TIME
//AND DESIRED DEPARTURE TIME. IF
// IT IS ZERO, DO NOT READ IF 1, READ IT.
const int read_fileinE=0;
const int read_fileinF=0;
const int read_fileinG=0;
const int read_fileinH=0;
const int read_filein_total_cost=0;
const int past = 2;
const int MAX=10000;
const int MAX_uniform = 3; // IT HAS TO BE AN ODD NUMBER
const int MAX_VAR = 10; // THE MAXIMUM SIZE OF THE VARIABLES TO BE
//READ FROM A FILE INPUT - FOR 159 COMMUTERS

const int parameterdim = 80;  // 80% free - num. =4 and den.=5
const int simulations=9; // BEFORE WAS 9500
const int LIMCOMB = 20;
const int LIMITDO=1; // FOR 160 COMMUTERS
const int alfa_annealing=0.5;
const double fraction_of_rich=0.5;
const double T0 = 1;
const double alfa_power = 0.35;
const double ppf1 = 0.5;
const double p_f2 = 900;
const double alfa = 45; // ADDED AT 12/AUG/1999
const double alfai=1000000;
```

```
const int type_toll = 2;          // that's the one, with two time windows.
const double alfa_time=0.65;
const double beta_time=0.35;
class commuters
{
public:
double arriv_position;
double arrival_time; // MAYBE I WILL NEED ARRIVAL TIME FREE AND PAYING AND ALSO,
double arr_time;
int better_off;
double departure_time; // DEPARTURE TIME FREE AND PAYING
double dep_time;
double des_time;
double  com_cost;
double  com_toll_cost;
double  com_paying;
double  com_free;
double  total_cost;
double initial_cost;
double willingness_to_pay;
double difer_value_of_time;
int pos_cases;
int updateddeparture;
int use_bottleneck;
commuters * next;
};

class timesarrdep
{
public:
int arrivint_position;
double arr_time;
double dep_time;
timesarrdep * next;
};

class function_toll
{
public:
double first_value;
double sec_value;
double value;
function_toll * next;
};


double Cost(double A_t, double D_t, double Des_t, double e_rate, double l_rate,
double W_E1, double W_E2);
double C_toll(double f, double val_of_time, double D_t, double a_rate,
```

```
double t_rate1, double t_rate2, double W_B1, double W_B2, double W_E1,
double W_E2, function_toll *v_toll_start, function_toll *v_toll_point,
function_toll *v_toll_prior, int np);

double F_prob(double c1, double c2, int sim, int runs_f, double pf1, double z);
double  F_probability (int type, int runsf, double cost1, double cost2);

void main()
{
int count, count_last_willing, count_position, count_toll_funct;
int demand_elastic, i, j, n_numerator, n_denominator, first;
int dif_desired, dif_value_of_time, standardrun;

// THE SLOPE OF THE DESIRED DEADLINE CURVE: w
double w, storetotal_read, storeB, storeA_read;
double storeA1_read, storeA2_read, storeB_read;
double storeC_read, storeD_read, storeE_read;
double storeF_read, storeG_read, storeG1_read;
double storeH_read, storeI_read;
int read_file=1;
int countruns=0;
double auxarrvector[MAX_uniform];
double auxdepvector[MAX_uniform];
double auxarr_min, auxdep_min;
double a_time[MAX*2], d_time[MAX*2];
double arr_toll[MAX_uniform];
double nauxcostpay_per_time;
double nauxcostfree_per_time;
double aux1free_per_time_min, aux1pay_per_time_min;
double n_random;
long double y;
double x;
double limitdo=1.0;
double p_f1 = 30000;// ADDED AT 12/AUG/1999
double f_probconv, zaux;
function_toll * vector_toll_start, * vector_toll_prior, * vector_toll_point;
double auxarr1,auxdep1;
double user_probablity;
double x0;
double y0;
int count_time;

int jji, countjji, ctype;
int changecom1, changecom2, change_commuter_2;
double aux1_max;
double f;
double depvector[MAX];
int start;
timesarrdep * time_vector_start[simulations];
```

```
timesarrdep * time_vector_point[MAX/LIMITDO+1];
timesarrdep * time_vector_prior[simulations];

ofstream fout1("20o159_geral4_ctb99.txt");
endl;
if (read_file == 0) // 03/AUG/1999
{ // 03/AUG/1999

cout << "Enter the value for n_numerator " << endl;
cin >> n_numerator;

cout << "Enter the value for n_denominator " << endl;
cin >> n_denominator;
cout << "Enter the value for f " << endl;
cin >> f;

cout << " Do you want Linear (1) or Exponential (2) " << endl;
cin >> ctype;

} // 03/AUG/1999
else // 03/AUG/1999
{ // 03/AUG/1999

ifstream file_in("c:\\Program Files/Microsoft Visual Studio/MyProjects/10Aug99/
file1inpa_2_159.txt", ios::in);
if (!file_in)
{
cerr << "File could not be opened." << endl;
exit(1);
}

cout << "this program shows file handling.\n\n";


while(file_in >> n_numerator >> n_denominator >> f >> ctype >> dif_desired >>
dif_value_of_time >> demand_elastic >> w >> standardrun)

fout1 << n_numerator <<" --- " <<  n_denominator << endl;
fout1 << f << " --- " << ctype << " --- " << dif_desired << endl;
fout1 << "---" << dif_value_of_time << endl;
fout1 << w << " --- " << standardrun << endl;

}

commuters *pos_pointer_caspoint[MAX/parameterdim],*pos_pointer_casstart[LIMCOMB];
commuters *pos_pointer_casprior[LIMCOMB];

count=0;
```

```
/*************************************************************************
BEGINING OF if f==1
*************************************************************************/

if(f==1)
{
pos_pointer_casstart[count] = new commuters;
if (pos_pointer_casstart[count] == NULL)
{
cout << "Insufficient Memory - Program terminating \n";
// break;
}
else
{
cout << "Memory Allocated.\n";
first =0;
pos_pointer_casstart[count] -> pos_cases=1;
pos_pointer_casstart[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_casstart[count];
}

for (int i=1;i<=(n_denominator-1);i++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=1;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];

}
}
/*************************************************************************
END OF if f==1
*************************************************************************/

else
{

/*************************************************************************
BEGINING OF if f==0
*************************************************************************/
if (f==0)
{
pos_pointer_casstart[count] = new commuters;
if (pos_pointer_casstart[count] == NULL)
{
cout << "Insufficient Memory - Program terminating \n";
//break;
```

```
}
else
{
cout << "Memory Allocated.\n";
first =0;
pos_pointer_casstart[count] -> pos_cases=0;
pos_pointer_casstart[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_casstart[count];
}

for (int i=1;i<=(n_denominator-1);i++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=0;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
}

/*************************************************************************
* END OF if f==0
*************************************************************************/
}
else
{

/*************************************************************************
* BEGINING OF if 0<f<1
*************************************************************************/


/*****************************************************************
* BEGINING OF DETAIL 1: OBSERVE THAT
* THERE IS ALWAYS A START POINTER. THAT'S WHY
* THE POINTER "POINT" GOES ALWAYS FROM 1
*   (OBSERVE THE VALUES OF i IN THE FOR LOOPS
* TO A RESPECTIVE VALUE
*****************************************************************/

do
{
pos_pointer_casstart[count] = new commuters;
if (pos_pointer_casstart[count] == NULL)
{
cout << "Insufficient Memory - Program terminating \n";
break;
}
else
```

```
{
cout << "Memory Allocated.\n";
first =0;
if (count==n_numerator) pos_pointer_casstart[count] -> pos_cases=0;
else pos_pointer_casstart[count] -> pos_cases=1;
//pos_pointer_casstart[count] -> pos_cases=1;
pos_pointer_casstart[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_casstart[count];
}
cout << "the value of f test 12 is "<< f << endl;
for (i=1; i<=(n_numerator-count-1);i++)
{

pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=1;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
}

//if (count==n_numerator) i=0;

/* In order to do not go in the next if statement:
(a) It has to be the last one and;
(b) It has to be only one possible paying.
*/
if (count != n_numerator)
{
 for (j=i; j<=(i+n_denominator-n_numerator-1);j++)
 {
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=0;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
 }
}

if (count==n_numerator)
{
if ((n_denominator-n_numerator)>1)
{
for (j=i; j<=(i+n_denominator-n_numerator-2);j++)
 {
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=0;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];
```

```
pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
 }
}
else j=i;
}

cout << "the value of f is "<< f << endl;
for (i=j; i<=(n_denominator-1);i++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=1;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];

}
count++;
}while(count <=n_numerator);

/*****************************************************************
* EXEMPLIFYING THE CASE FOR 2 OUT OF 5
* (MEANING 2 FREE OUT OF 5 DAYS)
*  AS SAID ABOVE, UNTIL HERE, THE CASES OF
* 11000, 10001, 00011 ARE OBTAINED.
* NEXT THE CASES OF: 00110, 01100 ARE SOLVED.
*****************************************************************/

int pay_final_position = n_denominator-n_numerator-1;

while(count<=n_denominator-1)
{
pos_pointer_casstart[count] = new commuters;
if (pos_pointer_casstart[count] == NULL)
{
cout << "Insufficient Memory - Program terminating \n";
break;
}
else
{
cout << "Memory Allocated.\n";
first =0;
pos_pointer_casstart[count] -> pos_cases=0;
pos_pointer_casstart[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_casstart[count];
}
```

```
for (i=1; i<=(pay_final_position-1);i++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=0;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
}

for (j=i; j<=(i+n_numerator-1);j++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=1;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
}

for (i=j; i<=(n_denominator-1);i++)
{
pos_pointer_caspoint[count] = new commuters;
pos_pointer_caspoint[count] -> pos_cases=0;
pos_pointer_casprior[count] -> next = pos_pointer_caspoint[count];

pos_pointer_caspoint[count] -> next = NULL;
pos_pointer_casprior[count] = pos_pointer_caspoint[count];
}
pay_final_position--;
count++;
}
cout << "Test f1 " << endl;
cout << "The value of f is " << f << endl;
cout << "First count " << endl;
cout << "The value of count is " << count << endl;
cout << "Test f2 " << endl;
count=count-1;

}
/*****************************************************************************
* END OF 0<f<1
*****************************************************************************/
}

int limcomb; // Express the number of possible combinations
if ((f==0) || (f==1))
limcomb = 0;
else
```

```
limcomb = n_denominator-1;
cout << "The value of count is " << count << endl;
for (i=0; i<=count; i++)
{
cout << "Combination: " << i << endl;
pos_pointer_caspoint[i]=pos_pointer_casstart[i];
do
{

pos_pointer_casprior[i]=pos_pointer_caspoint[i] -> next;
cout << pos_pointer_caspoint[i] -> pos_cases;
pos_pointer_caspoint[i] =  pos_pointer_caspoint[i] ->next;
}while(pos_pointer_casprior[i] != NULL);
cout << endl;
}

commuters *finalcaspoint[MAX],*finalcasstart[MAX], *finalcasprior[MAX];

int dimension = parameterdim*n_denominator-1;
for (i=0; i<=count; i++)
{
pos_pointer_caspoint[i]=pos_pointer_casstart[i];
finalcasstart[i]=new commuters;
finalcasstart[i] -> pos_cases=(pos_pointer_caspoint[i] -> pos_cases);
finalcasstart[i] -> next = NULL;
finalcasprior[i]=finalcasstart[i];
pos_pointer_casprior[i]=pos_pointer_caspoint[i] ->next;

for (j=1; j <=dimension; )
{
if ((j%n_denominator)==0)
{
pos_pointer_caspoint[i]=pos_pointer_casstart[i];
pos_pointer_casprior[i]=pos_pointer_caspoint[i] -> next;
finalcaspoint[i]=new commuters;
if (finalcaspoint[i]== NULL)
{
cout << "Insufficient Memory - Program terminating \n";

}
else
{
cout << "Memory Allocated 1.\n";
}

finalcaspoint[i] -> pos_cases=(pos_pointer_caspoint[i] -> pos_cases);
finalcasprior[i] -> next = finalcaspoint[i];
finalcaspoint[i] -> next = NULL;
finalcasprior[i]=finalcaspoint[i];
```

```
j++;

}
do
{
pos_pointer_caspoint[i]=pos_pointer_caspoint[i] -> next;
pos_pointer_casprior[i]=pos_pointer_caspoint[i] -> next;
finalcaspoint[i]=new commuters;

if (finalcaspoint[i]== NULL)
{
cout << "Insufficient Memory - Program terminating \n";
break;
}
else
{
// cout << "Memory Allocated 2.\n";
}

     int aux =1;
 if (aux==1) aux=1;

finalcaspoint[i] -> pos_cases=(pos_pointer_caspoint[i] -> pos_cases);
finalcasprior[i] -> next = finalcaspoint[i];
finalcaspoint[i] -> next = NULL;
finalcasprior[i]=finalcaspoint[i];
j++;
}while(pos_pointer_casprior[i] !=NULL);
}
}

for (i=0; i<=(limcomb); i++)
{
cout << "Combination: " << i << endl;
finalcaspoint[i]=finalcasstart[i];
do
{

finalcasprior[i]=finalcaspoint[i] -> next;
cout << finalcaspoint[i] -> pos_cases;
finalcaspoint[i] =  finalcaspoint[i] ->next;
}while(finalcasprior[i] != NULL);
cout << endl;
}


double early_rate=double(0.8);
double ei=double(0.3);
double late_rate=double(2.0);
```

```
double li=double(3.7);
double toll_rate1=double(3); // FOR DIFFERENT VALUES OF TIME
double toll_rate2=double(0.10);
double alfa_rate;
double desired_time;
if (type_toll==2)
 {
    toll_rate1=late_rate+0.5;
toll_rate2=toll_rate1;
 }
fout1 << endl << "The early rate and late rate are " << early_rate << "    "
  << late_rate << endl;
int  Np;
double aux_cost_and_runs[MAX], commu_toll_cost[MAX],c1, c2,  WB1, WB2;
double WE1, WE2, a1, a2, a3;
double Cost(double A_t, double D_t, double Des_t, double e_rate, double l_rate);
Np = dimension;

cout << "The value of Np is " << Np << endl;
fout1 << "The value of Np is " << Np << endl;

if (read_file==0)
{
cout << " The commuters have different desired deadlines or ";
cout << " equal ones ?!"< < endl;
fout1 << " The commuters have different desired deadlines or ";
fout1 << " equal ones ?!"<< endl;
cout << " Type 1 for different or, 0 for equal. " << endl;
fout1 << " Type 1 for different or, 0 for equal. " << endl;
cin >> dif_desired;
if (dif_desired==1)
{
cout << "As you want commuters with different desired deadlines";
cout << " enter the slope of the curve. " << endl;
fout1 << "As you want commuters with different desired deadlines";
fout1 << " enter the slope of the curve. " << endl;
cin >> w;
fout1 << w << endl;
}
cout << "Type 1 for standard run (the commuters start in equilibrium" << endl;
fout1 << "Type 1 for standard run (the commuters start in equilibrium" << endl;
cout << " or 0 for starting at 0. " << endl;
fout1 << " or 0 for starting at 0. " << endl;
cin >> standardrun;
fout1 << "You type: " <<  standardrun << endl;

}
else
{
```

```
if (dif_desired==1)
{
cout << " The commuters have different desired deadlines."<< endl;
fout1 << " The commuters have different desired deadlines."<< endl;
cout << "As you want commuters with different desired deadlines, ";
cout << "the slope of the curve is: " << endl;
cout << w << endl;
fout1 << "As you want commuters with different desired deadlines,";
fout1 << " the slope of the curve is: " << endl;
fout1 << w << endl;
}
else
{
cout << " The commuters have different desired deadlines."<< endl;
fout1 << " The commuters have different desired deadlines."<< endl;

}

if (standardrun == 1)
{
cout << "The commuters start in equilibrium." << endl;
fout1 << "The commuters start in equilibrium." << endl;
}
else
{
cout << "The commuters start at 0. " << endl;
fout1 << "The commuters start at 0. " << endl;
}
}
desired_time= (Np*late_rate/(early_rate+late_rate));
c1=Np*late_rate/(early_rate+late_rate);
c2=Np*early_rate/(early_rate+late_rate);

     if (type_toll==2)
 {
        WB1=0;
  WB2=51;
   WE2=160;

 }

 else
 {
    if (type_toll==1)
{
vector_toll_start=new function_toll;
vector_toll_start -> first_value = 0;
vector_toll_start -> sec_value = 0;
```

```
vector_toll_start -> next = NULL;
vector_toll_prior = vector_toll_start;

for (i=1; i <=Np; i++)
{
vector_toll_point=new function_toll;
vector_toll_point -> first_value = 0;
vector_toll_point -> sec_value = 0;

if (i<= (Np*late_rate/(early_rate+late_rate)))
{
vector_toll_point -> value = i - (i*(1-early_rate));

}
else
{
vector_toll_point -> value = i - (1+late_rate)*(i-(Np*late_rate/
(early_rate+late_rate)))+(Np*late_rate/(early_rate+late_rate))*(1-early_rate);

}
vector_toll_point -> next = vector_toll_point;
vector_toll_point -> next = NULL;
vector_toll_prior = vector_toll_point;

}
}
else
{
if ((f!=0) && (dif_value_of_time !=1))
{
WB1 = 1/toll_rate1*(c1*(1-f)*(toll_rate1-early_rate));
WB2 = - pow(10,12);
}
else
{
if ((f==0) && (dif_value_of_time ==1))
{
WB1=Np/2-fraction_of_rich*Np/2;
WB2=0;
}
else
{
if ((f!=0) && (dif_value_of_time ==1))
{
WB1 = 1/toll_rate1*(c1*(1-f)*(toll_rate1-early_rate));
WB2 = - pow(10,12);
}
else
{
```

```
if (f==0)
{
WB1=0;
WB2 = - pow(10,12);
}
}
}

}

if ((f !=0) && (dif_value_of_time !=1))
{
WE1 = 1/(toll_rate1*(1+late_rate))*((1+toll_rate1*late_rate-late_rate*late_rate)
*(c1+f*c2)-f*c2*(1-toll_rate1+late_rate))+c1;
WE2 = pow(10,12);
}
else
{
 if ((f==0) && (dif_value_of_time ==1))
{
WE1=Np/2+fraction_of_rich*Np/2;
WE2=c1+c2;
 }
 else
 {
if ((f!=0) && (dif_value_of_time ==1))
{
WE1 = 1/(toll_rate1*(1+late_rate))*((1+toll_rate1*late_rate-late_rate*late_rate)
*(c1+f*c2)-f*c2*(1-toll_rate1+late_rate))+c1;
WE2 = pow(10,12);
}
else
{
if (f==0)
{
WE1=Np;
WB2 = - pow(10,12);
}
}
 }
}
}
 }
   cout << "The value of WB1 and WB2 are: " << WB1 << " and " << WB2 << endl;
   cout << "The value of WE1 and WE2 are: " << WE1 << " and " << WE2 << endl;

for (int iii=0; iii<= limcomb; iii++)
{
ifstream file_inB("c:\\Program Files/Microsoft Visual Studio/MyProjects/10Aug99/
```

```
file1f159.txt", ios::in);

if (!file_inB)
{
cerr << "File could not be opened." << endl;
exit(1);
}
finalcaspoint[iii]=finalcasstart[iii];

while(file_inB >> storeB)
{
cout << storeB << endl;
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> willingness_to_pay = storeB;
fout1 << " Willingness_to_pay: " << finalcaspoint[iii] -> willingness_to_pay
<< endl;
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}
}


if (dif_value_of_time != 0)
{
count_toll_funct =0;  // ADDED AT 31/AUG/99

for (int iii=0; iii<= limcomb; iii++)
{

ifstream file_inC("c:\\Program Files/Microsoft Visual Studio/MyProjects/10Aug99/
file1inpd_2_159.txt", ios::in);
if (!file_inC)
{
cerr << "File could not be opened." << endl;
exit(1);
}

finalcaspoint[iii]=finalcasstart[iii];
while(file_inC >> storeB)
{
cout << storeB << endl;

finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> difer_value_of_time = storeB;
fout1 << " The Value of Time is: " << finalcaspoint[iii] -> difer_value_of_time
<< endl;
finalcaspoint[iii] = finalcaspoint[iii] ->next;

if (toll_funct == 1)
{
```

```
}
}
}

}
else
{
for (int iii=0; iii<= limcomb; iii++)
{

finalcaspoint[iii]=finalcasstart[iii];
do
{
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> difer_value_of_time = 1;
fout1 << " The Value of Time is: "
<< finalcaspoint[iii] -> difer_value_of_time << endl;
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}while(finalcasprior[iii] != NULL);


}
}

double cumuarrival, cumudeparture;
for (i=0; i<=limcomb; i++)
{
count=0;
count_last_willing=0;  // ADDED 09/AUG/1999
count_position=0;  //ADDED 11/OCT./99
cumuarrival=0;
cumudeparture=0;
cout << "Combination: " << i << " Real toll Costs" << endl;
finalcaspoint[i]=finalcasstart[i];
do
{
finalcasprior[i]=finalcaspoint[i] -> next;

/*****************************************************************
THE NEXT STATEMENT IS EXTREMELY IMPORTANT BECAUSE THE IF STATEMENT
""MUST"" BE EXECUTED EITHER:
(i) if the willingness to pay of the commuters are greater than the
cost at the begining or,
(ii) if the demand is ""INELASTIC""  meaning that the commuters are
obliged to pass by the bottleneck.
*****************************************************************/
if (((finalcaspoint[i] -> willingness_to_pay) > (Np * early_rate * late_rate/
(early_rate+late_rate))) || (demand_elastic!=1))
```

```
{
if (count<=(Np * late_rate/(early_rate+late_rate)))
{
finalcaspoint[i] -> use_bottleneck=1;
int testfinal;
if (count>=830)
testfinal=count;

a1=(finalcaspoint[i] -> arriv_position = double(count_position));
if (standardrun==1)
{
a2=(finalcaspoint[i] -> arrival_time=0 + count * (1 - early_rate));
finalcaspoint[i] -> arr_time = a2;
}
else
{
if (standardrun==0)
{
//a2=(finalcaspoint[i] -> arrival_time=-1);  // CHNAGED AT 26/AUG/1999
a2=(finalcaspoint[i] -> arrival_time=0);
finalcaspoint[i] -> arr_time = a2;
}
else
{
a2=(finalcaspoint[i] -> arrival_time=(0 + count * (1 - ei)));
finalcaspoint[i] -> arr_time = a2;
}
}

if (dif_desired==1)
{
finalcaspoint[i] -> des_time=c1*(1-1/w)+count_last_willing/w;
desired_time=finalcaspoint[i] -> des_time;
}
else
{

finalcaspoint[i] -> des_time=c1;
desired_time=finalcaspoint[i] -> des_time;
}

cumuarrival++;
if ((cumuarrival-cumudeparture) > 0)
{
if (standardrun !=0)
{
a3=(finalcaspoint[i] -> departure_time = double(count));
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
```

```
}
else
{
a3=(finalcaspoint[i] -> departure_time =double (count));
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
}
}
else
{
a3=(finalcaspoint[i] -> departure_time = a2);
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
}
if (toll_funct == 1)
{

arr_toll[count]=0;
}
aux_cost_and_runs[count]=Cost(a2, a3, desired_time, early_rate, late_rate);
finalcaspoint[i] -> com_cost =aux_cost_and_runs[count];
finalcaspoint[i] -> com_free =aux_cost_and_runs[count];
if ((finalcaspoint[i] -> pos_cases)==0)
{
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);

finalcaspoint[i] -> com_toll_cost = commu_toll_cost[count];
finalcaspoint[i] -> com_paying =(aux_cost_and_runs[count]+
commu_toll_cost[count]);
}
else
{
commu_toll_cost[count]=0;
finalcaspoint[i] -> com_toll_cost = commu_toll_cost[count];
/*************************************************************************
*  THE NEXT STATEMENT IS NECESSARY TO MAKE SURE THAT THE COST WHEN PAYING IS
*   EVALUATED CORRECTLY  - JUST TO MAKE SUTE THAT WE ARE ALWAYS EVALUATING THE
*   TOLL COST "ONLY" WHEN THE COMMUTER IS CLASSIFIED AS PAYING, I HAVE DECIDED
*   TO LEAVE THE VARIABLE com_toll_cost IN THE CLASS commuters
*************************************************************************/
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);

finalcaspoint[i] -> com_toll_cost = commu_toll_cost[count];
```

```
finalcaspoint[i] -> com_paying =(aux_cost_and_runs[count]+
commu_toll_cost[count]);
}
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np );

finalcaspoint[i] -> total_cost =(finalcaspoint[i] -> com_free);
}
else
{
if (count <= Np)
{
finalcaspoint[i] -> use_bottleneck=1;
a1=(finalcaspoint[i] -> arriv_position= double(count_position));
if (standardrun==1)
{
a2=(finalcaspoint[i] -> arrival_time=0 + (1+late_rate)*(count - (Np*late_rate/
(early_rate+late_rate)))+ (Np*late_rate/(early_rate+late_rate))*
(1-early_rate));
finalcaspoint[i] -> arr_time = a2;
cumuarrival++;
}
else
{
if (standardrun==0)
{
// a2=(finalcaspoint[i] -> arrival_time=-1); //CHNAGED AT 26/AUG/1999
a2=(finalcaspoint[i] -> arrival_time=0);
finalcaspoint[i] -> arr_time = a2;
cumuarrival++;
}
else
{
a2=(finalcaspoint[i] -> arrival_time=0 + (1+li)*(count - (Np*late_rate/
(early_rate+late_rate)))+ (Np*late_rate/(early_rate+late_rate))*
(1-early_rate));
finalcaspoint[i] -> arr_time = a2;
}
}

if ((cumuarrival-cumudeparture) > 0)
{
if (standardrun !=0)
{
a3=(finalcaspoint[i] -> departure_time = count);
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
```

```
}
else
{
a3=(finalcaspoint[i] -> departure_time =double (count-1));
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
}
}
else
{
a3=(finalcaspoint[i] -> departure_time = a2);
finalcaspoint[i] -> dep_time = a3;
cumudeparture++;
}

if (dif_desired==1)
{
finalcaspoint[i] -> des_time=c1*(1-1/w)+count_last_willing/w;
desired_time=finalcaspoint[i] -> des_time;
}
else
{

finalcaspoint[i] -> des_time=c1;
desired_time=finalcaspoint[i] -> des_time;
}
aux_cost_and_runs[count]=Cost(a2, a3, desired_time, early_rate, late_rate);
finalcaspoint[i] -> com_cost =aux_cost_and_runs[count];
finalcaspoint[i] -> com_free =aux_cost_and_runs[count];
if ((finalcaspoint[i] -> pos_cases)==0)
{
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);

finalcaspoint[i] -> com_toll_cost = commu_toll_cost[count];
finalcaspoint[i] -> com_paying =(aux_cost_and_runs[count]+
commu_toll_cost[count]);
}
else
{
commu_toll_cost[count]=0;
finalcaspoint[i] -> com_toll_cost = commu_toll_cost[count];
/*****************************************************************************
*  THE NEXT STATEMENT IS NECESSARY TO MAKE SURE THAT THE COST WHEN PAYING IS   *
*   EVALUATED CORRECTLY  - JUST TO MAKE SUTE THAT WE ARE ALWAYS EVALUATING THE *
*   TOLL COST "ONLY" WHEN THE COMMUTER IS CLASSIFIED AS PAYING, I HAVE DECIDED *
*   TO LEAVE THE VARIBALE com_toll_cost IN THE CLASS commuters    *
```

```
*****************************************************************************/
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);
finalcaspoint[i] -> com_paying =(aux_cost_and_runs[count]+
commu_toll_cost[count]);
}
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
commu_toll_cost[count]=C_toll(f,dif_value_of_time,a3,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);
finalcaspoint[i] -> total_cost =(finalcaspoint[i] -> com_free);
}
}
count++;
count_last_willing++;
count_position++;
if (count_last_willing==950)
count_last_willing=count_last_willing;

} // IF ADDED AT 09/AUG/1999 DUE TO THE WILLINGNESS TO PAY

else
{

if (((finalcaspoint[i] -> willingness_to_pay) < (Np * early_rate * late_rate/
(early_rate+late_rate))) && (demand_elastic==1))
{
finalcaspoint[i] -> arriv_position = count_position;
finalcaspoint[i] -> use_bottleneck = 0;
finalcaspoint[i] -> arrival_time = pow(10,13);
finalcaspoint[i] -> arr_time = pow(10,13);
finalcaspoint[i] -> departure_time = pow(10,13);
finalcaspoint[i] -> dep_time = pow(10,13);
finalcaspoint[i] -> com_cost = finalcaspoint[i] -> willingness_to_pay;
finalcaspoint[i] -> com_free = finalcaspoint[i] -> willingness_to_pay;
finalcaspoint[i] -> com_toll_cost = finalcaspoint[i] -> willingness_to_pay;
finalcaspoint[i] -> com_paying = finalcaspoint[i] -> willingness_to_pay;
finalcaspoint[i] -> total_cost = finalcaspoint[i] -> willingness_to_pay;
if (dif_desired==1)
{
finalcaspoint[i] -> des_time=c1*(1-1/w)+count_last_willing/w;
desired_time=finalcaspoint[i] -> des_time;
}
else
{
finalcaspoint[i] -> des_time=c1;
desired_time=finalcaspoint[i] -> des_time;
```

```
}
count_last_willing++;
count_position++;
}
}
if ((count_last_willing-1) <= Np)
fout1 << "Commuter "<< (count_last_willing-1) << " " << setw(7) <<
finalcaspoint[i] -> com_cost << " " << setw(7) <<
finalcaspoint[i] -> com_toll_cost << " "
<< setw(7) << finalcaspoint[i] -> com_free << " " << setw(7) <<
finalcaspoint[i] -> com_paying << " " << setw(7) <<
finalcaspoint[i] -> total_cost << " " << endl;

finalcaspoint[i] =  finalcaspoint[i] ->next;
}while(finalcasprior[i] != NULL);
cout << endl;
}
cout << endl
 << "The Cost for Each Commuter is ";
for (i=0; i <= Np; i++)
{
cout << aux_cost_and_runs[i] << " ";
}

/*****************************************************************************
FREEING THE DYNAMIC ALLOCATED MEMORY FOR THE POSSIBLE CASES POINTERS
*****************************************************************************/

for (i=0; i <=limcomb; i++)
{
pos_pointer_caspoint[i] = pos_pointer_casstart[i];
do
{
pos_pointer_casprior[i] = pos_pointer_caspoint[i] -> next;
delete (pos_pointer_caspoint[i]);
pos_pointer_caspoint[i]=pos_pointer_casprior[i];
} while (pos_pointer_casprior[i] != NULL);
}

/*****************************************************************************
ENDING THE DYNAMIC ALLOCATED MEMORY FOR THE POSSIBLE CASES POINTERS
*****************************************************************************/
    for (i=0;i <=limcomb; i++)
{
       count=0;
   finalcaspoint[i]=finalcasstart[i];
   do
   {
            finalcaspoint[i] -> better_off=2;
```

```
    finalcasprior[i]=finalcaspoint[i] -> next;
if (((finalcaspoint[i] -> willingness_to_pay) > (Np * early_rate * late_rate/
(early_rate+late_rate))) || (demand_elastic!=1))
            {
                if (count <Np * late_rate * late_rate/(early_rate+late_rate))
{
a2=0+count*(1-early_rate);
if (count==0)
a3=a2;
else a3=a3+1;
desired_time=finalcaspoint[i] -> des_time;
finalcaspoint[i] -> initial_cost =Cost(a2,a3,desired_time,early_rate,late_rate);
}
else
{
                    if (count <= Np)
 {
a2=0 + (1+late_rate)*(count - (Np*late_rate/(early_rate+late_rate)))+
(Np*late_rate/(early_rate+late_rate))*(1-early_rate);
                        a3=a3+1;
    desired_time=finalcaspoint[i] -> des_time;
  finalcaspoint[i] -> initial_cost =Cost(a2,a3,desired_time,early_rate,
  late_rate);
 }

}
               finalcaspoint[i] -> use_bottleneck=1;
count++;
}
else
{
finalcaspoint[i] -> initial_cost=finalcaspoint[i] -> willingness_to_pay;
finalcaspoint[i] -> use_bottleneck=2;
}
finalcaspoint[i]=finalcaspoint[i] -> next;
   }while (finalcasprior[i]!=NULL);;
}
/*****************************************************************************
STARTING THE SIMULATION
*****************************************************************************/

if (read_fileinD == 1)
{
for (int iii=0; iii<= 0; iii++)
{
ifstream file_inD("c:\\Program Files/Microsoft Visual Studio/MyProjects/
14_6Sep99/read_file_014011_ctb99.txt", ios::in);
if (!file_inD)
{
```

```
cerr << "File could not be opened." << endl;
exit(1);
}
finalcaspoint[iii]=finalcasstart[iii];

while(file_inD >> storeA_read >> storeA1_read >> storeA2_read >> storeB_read
>> storeC_read >> storeD_read >> storeE_read >> storeF_read >> storeH_read >>
storeG_read >> storeG1_read >> storeI_read)
{
finalcaspoint[iii] -> arriv_position = storeA_read;
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> willingness_to_pay = storeA1_read;
finalcaspoint[iii] -> com_free = storeB_read;
finalcaspoint[iii] -> com_paying = storeC_read;
finalcaspoint[iii] -> total_cost = storeD_read;
finalcaspoint[iii] -> arr_time = storeE_read;
finalcaspoint[iii] -> dep_time = storeF_read;
finalcaspoint[iii] -> pos_cases = storeH_read;
finalcaspoint[iii] -> des_time = storeG_read;
finalcaspoint[iii] -> use_bottleneck = storeG1_read;
finalcaspoint[iii] -> better_off = storeI_read;
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}
}
}

fout1 << endl <<  "Tempotary for the initial cost " << endl;

for (int iiii=0; iiii<= limcomb; iiii++)
{
count=0;
fout1 << "Combination: " << iiii << " The commuter and toll costs are " << endl;
finalcaspoint[iiii]=finalcasstart[iiii];
do
{
finalcasprior[iiii]=finalcaspoint[iiii] -> next;
if ((finalcaspoint[iiii] -> arr_time < 0.000001) &&
(finalcaspoint[iiii] -> arr_time > (-0.000001)))
finalcaspoint[iiii] -> arr_time = 0;
if ((finalcaspoint[iiii] -> dep_time < 0.000001) &&
(finalcaspoint[iiii] -> dep_time > (-0.000001)))
finalcaspoint[iiii] -> dep_time = 0;
fout1 << " " << setw(16)<< finalcaspoint[iiii] -> total_cost << setw(13) <<endl;
count++;
finalcaspoint[iiii] =  finalcaspoint[iiii] ->next;
}while(finalcasprior[iiii] != NULL);
cout << "  "; //endl;
}
```

```
if (read_fileinE == 1)
{
for (int iii=1; iii<= 1; iii++)
{
ifstream file_inE("c:\\Program Files/Microsoft Visual Studio/MyProjects/
14_6Sep99/read_file_01409_ctb99.txt", ios::in);

if (!file_inE)
{
cerr << "File could not be opened." << endl;
exit(1);
}

finalcaspoint[iii]=finalcasstart[iii];
while(file_inE >> storeA_read >> storeA1_read >> storeA2_read >> storeB_read >>
storeC_read >> storeD_read >> storeE_read >> storeF_read >> storeH_read >>
storeG_read >> storeG1_read >> storeI_read)
{
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> arriv_position = storeA_read;
finalcaspoint[iii] -> willingness_to_pay = storeA1_read;
finalcaspoint[iii] ->  initial_cost = storeA2_read;
finalcaspoint[iii] -> com_free = storeB_read;
finalcaspoint[iii] -> com_paying = storeC_read;
finalcaspoint[iii] -> total_cost = storeD_read;
finalcaspoint[iii] -> arr_time = storeE_read;
finalcaspoint[iii] -> dep_time = storeF_read;
finalcaspoint[iii] -> pos_cases = storeH_read;
finalcaspoint[iii] -> des_time = storeG_read;
finalcaspoint[iii] -> use_bottleneck = storeG1_read;
finalcaspoint[iii] -> better_off = storeI_read;
finalcaspoint[iii] = finalcaspoint[iii] ->next;

}
}
}

if (read_fileinF == 1)
{


for (int iii=2; iii<= 2; iii++)
{
ifstream file_inF("c:\\Program Files/Microsoft Visual Studio/MyProjects/
14_6Sep99/read_file_01429_ctb99.txt", ios::in);

if (!file_inF)
{
cerr << "File could not be opened." << endl;
```

```
exit(1);
}
finalcaspoint[iii]=finalcasstart[iii];

while(file_inF >> storeA_read >> storeB_read >> storeC_read >> storeD_read >>
storeE_read >> storeF_read >> storeH_read >> storeG_read >> storeI_read)
{
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> arriv_position = storeA_read;
finalcaspoint[iii] -> com_free = storeB_read;
finalcaspoint[iii] -> com_paying = storeC_read;
finalcaspoint[iii] -> total_cost = storeD_read;
finalcaspoint[iii] -> arr_time = storeE_read;
finalcaspoint[iii] -> dep_time = storeF_read;
finalcaspoint[iii] -> pos_cases = storeH_read;
finalcaspoint[iii] -> des_time = storeG_read;
finalcaspoint[iii] -> better_off = storeI_read;
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}
}
}
if (read_fileinG == 1)
{

for (int iii=3; iii<= 3; iii++)
{
ifstream file_inG("c:\\Program Files/Microsoft Visual Studio/MyProjects/
14_6Sep99/read_file_014210_ctb99.txt", ios::in);

if (!file_inG)
{
cerr << "File could not be opened." << endl;
exit(1);
}
finalcaspoint[iii]=finalcasstart[iii];

while(file_inG >> storeA_read >> storeB_read >> storeC_read >> storeD_read >>
storeE_read >> storeF_read >> storeH_read >> storeG_read >> storeI_read)
{
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> arriv_position = storeA_read;
finalcaspoint[iii] -> com_free = storeB_read;
finalcaspoint[iii] -> com_paying = storeC_read;
finalcaspoint[iii] -> total_cost = storeD_read;
finalcaspoint[iii] -> arr_time = storeE_read;
finalcaspoint[iii] -> dep_time = storeF_read;
finalcaspoint[iii] -> pos_cases = storeH_read;
finalcaspoint[iii] -> des_time = storeG_read;
finalcaspoint[iii] -> better_off = storeI_read;
```

```
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}
}
}
if (read_fileinH == 1)
{
for (int iii=4; iii<= 4; iii++)
{
ifstream file_inH("c:\\Program Files/Microsoft Visual Studio/MyProjects/
14_6Sep99/read_file_014211_ctb99.txt", ios::in);

if (!file_inH)
{
cerr << "File could not be opened." << endl;
exit(1);
}


finalcaspoint[iii]=finalcasstart[iii];

while(file_inH >> storeA_read >> storeB_read >> storeC_read >> storeD_read >>
storeE_read >> storeF_read >> storeH_read >> storeG_read >> storeI_read)
{
finalcasprior[iii]=finalcaspoint[iii] -> next;
finalcaspoint[iii] -> arriv_position = storeA_read;
finalcaspoint[iii] -> com_free = storeB_read;
finalcaspoint[iii] -> com_paying = storeC_read;
finalcaspoint[iii] -> total_cost = storeD_read;
finalcaspoint[iii] -> arr_time = storeE_read;
finalcaspoint[iii] -> dep_time = storeF_read;
finalcaspoint[iii] -> pos_cases = storeH_read;
finalcaspoint[iii] -> des_time = storeG_read;
finalcaspoint[iii] -> better_off = storeI_read;
finalcaspoint[iii] = finalcaspoint[iii] ->next;
}
}

}

for (int ii=0; ii<= limcomb; ii++)
{
count=0;
fout1 << "THE DATA AS WAS ENTERED." << endl << endl;
fout1 << "Combination: " << ii << " The commuter and toll costs are " << endl;
finalcaspoint[ii]=finalcasstart[ii];
fout1 << "Commuter - Costs:  Willingness to Pay  " <<  " Value of Time "
<< endl;
do
{
```

```
finalcasprior[ii]=finalcaspoint[ii] -> next;
fout1 <<  finalcaspoint[ii] -> willingness_to_pay  << setw(20) <<
finalcaspoint[ii] -> difer_value_of_time <<  setw(20) <<
finalcaspoint[ii] -> better_off << endl;
finalcaspoint[ii] =  finalcaspoint[ii] ->next;
count++;
}while(finalcasprior[ii] != NULL);

cout << "  "; //endl;

}

for ( ii=0; ii<= limcomb; ii++)
{
count=0;
fout1 << "THE DATA AS WAS ENTERED." << endl << endl;
fout1 << "Combination: " << ii << " The commuter and toll costs are " << endl;
finalcaspoint[ii]=finalcasstart[ii];
fout1 << "Commuter - Costs:    Willingness to Pay   Initial Cost    Free "
<<  "      Paying    " << "   Total    " << "Arr. Time" <<"  Dep. Time" <<
" Poss.Cases " <<" Desired Time"<<" Use Bottleneck  Better off " << endl;
do
{
fout1 << " " << count << "   ";
finalcasprior[ii]=finalcaspoint[ii] -> next;
if ((finalcaspoint[ii] -> arr_time < 0.000001) &&
(finalcaspoint[ii] -> arr_time > (-0.000001)))
finalcaspoint[ii] -> arr_time = 0;
if ((finalcaspoint[ii] -> dep_time < 0.000001) &&
(finalcaspoint[ii] -> dep_time > (-0.000001)))
finalcaspoint[ii] -> dep_time = 0;
//fout1 << "  " << setw(10) << finalcaspoint[ii] -> com_free
<< setw(13) << finalcaspoint[ii] -> com_paying
fout1 << "    " << setw(13) << finalcaspoint[ii] -> willingness_to_pay
<< setw(20) <<  finalcaspoint[ii] -> initial_cost << setw(20)
<< finalcaspoint[ii] -> com_free << setw(13)
<< finalcaspoint[ii] -> com_paying
<< setw(13) << finalcaspoint[ii] -> total_cost << setw(16)
<< finalcaspoint[ii] -> arr_time << setw(13)
  // << finalcaspoint[ii] -> arrival_timef << setw(13)
  << finalcaspoint[ii] -> arrival_timep << setw(13)
   << finalcaspoint[ii] -> dep_time << setw(13) <<
   finalcaspoint[ii] -> pos_cases << setw(15) <<
   finalcaspoint[ii] -> des_time << setw(15) <<
   finalcaspoint[ii] -> use_bottleneck << setw(15)
   << finalcaspoint[ii] ->better_off << endl;
finalcaspoint[ii] =  finalcaspoint[ii] ->next;
count++;
}while(finalcasprior[ii] != NULL);
```

```
cout << "  "; //endl;
}

fout1 <<  endl << "END OF THE INPUT DATA." << endl << endl << endl;

double epsilon = double(0.01);
int converge = 1;
int indpast=0;
int runs = 0;
int runstotal=7550;
double auxarr, auxdep, nauxcostfree, nauxcostpay, newauxcost, nauxtollcost;
commuters *finalauxpoint[MAX], *finalauxprior[MAX];
double arrvector[MAX];
double limit_total;

do
{
cout << endl << "The value of this run is " << runs << endl;
converge=1;
if (indpast > 4) indpast =0;
for (i=0; i <= limcomb; i++)
{
cumuarrival=0;
cumudeparture=0;
count=0;
finalcaspoint[i]=finalcasstart[i];
do
{
finalcasprior[i]=finalcaspoint[i] -> next;

if ((finalcaspoint[i] -> arr_time) < pow(10,12))
{
arrvector[count]=(finalcaspoint[i] -> arr_time);
}
finalcaspoint[i]=finalcaspoint[i] -> next;
count++;
}while (finalcasprior[i] != NULL);

int si,sj;
double as;
for (sj=1; sj <=Np; sj++)
{
as=arrvector[sj];
si=sj-1;
while ((si>=0) && (arrvector[si] > as))
{
arrvector[si+1]=arrvector[si];
si--;
```

```
}
arrvector[si+1]=as;
}

for (int j=0; j <= Np; j++)
{
if (j==0) depvector[j]=arrvector[j];
else
{
if ((arrvector[j]-depvector[j-1]) >= 1)
depvector[j] =arrvector[j];
else depvector[j]=depvector[j-1] + 1;
}
}

double jtype[MAX];
for (j=0; j<= Np; j++)
jtype[j]=0;
double auxarrvector[MAX_uniform];
double auxdepvector[MAX_uniform];
double verticalaux;
double m;

int changetime = 1;
start=0;
for (j=0; j <= Np; j++)
{


int count_total_arr;
if (j==0)
count_total_arr=0;
if (j==0)
{
verticalaux=0;
if (arrvector[j] > 0.5)
{
if (Np <=200)
// auxarr=-50;

auxarr=c1*(1-1/w)-75;
// auxarr=-1   CHANGED AT 16/SEP/1999;
else
{
if (Np <=1100)
auxarr=-70; // for 440 commuters
else
auxarr=-10000;
}
```

```
}
else
{
if (Np <=200)
// auxarr=-50;
auxarr=c1*(1-1/w)-75;
// auxarr=arrvector[j]-1;  CHANGED AT 16/SEP/1999;
else
{
if (Np <=1100)
// auxarr=-800;
// auxarr=-300;
auxarr=-70; // for 440 commuters
else
auxarr=-10000;
}
}
if (arrvector[j] != arrvector[j+1])
{
// limitdo=(arrvector[j+1]-arrvector[j])/Np + 0.1;
// limitdo=0.2;  // ADDED FRIDAY AT 3AM, 02/APRIL;
limitdo=1.0;
// limitdo=1/(w); // FOR 160 COMMUTERS - DDED THURSDAY 12/AUG/1999
//limitdo=10;
}else
// limitdo=0.2; // added 2/aprillimitdo=limitdo;
limitdo=1.0;
//limitdo=1/(w); //  FOR 160 COMMUTERS - ADDED THURSDAY 12/AUG/1999
// limitdo=10;// FOR 1000 COMM.
}
else
{

if (arrvector[j] != arrvector[j-1])
{
limitdo=1.0;
}
else //limitdo=0.2; // //  FOR 160 COMMUTERS - ADDED THURSDAY 12/AUG/1999
limitdo=1.0;
}

double limitwhile;
if (j==0)
limitwhile=arrvector[j];
else
{
if (j==Np)
{
if (Np <=200)
```

```
{
// limitwhile=400;
if (dif_desired==1)
{
limitwhile=(c1+c2/w)+160;
}
else limitwhile=400;

// limitwhile=depvector[j]+1;  CHANGED AT 16/SEP/1999;
}
else
{
if (Np <=1100)
// limitwhile=1900;
// limitwhile=1500;
limitwhile=600; // for 440 commuters
else
limitwhile=10000;
}
}

else
limitwhile=arrvector[j+1];
}

if (Np <=200)
// limit_total=400;
{
if (dif_desired==1)
{
limit_total=(c1+c2/w)+160;
}
else limit_total=400;

}
else
{
if (Np <=1100)
limit_total=600; // for 440 commuters
else
limit_total=10000;
}


if (auxarr <= limit_total)  // new if added at 16/sep/99 - if16
{
do
{
for (int jji=0; jji <= (MAX_uniform-1); jji++)
```

```
{
int found=0;
if (j==0)
{
if (((auxarr+jji*limitdo) <= (arrvector[j]+epsilon)) && (found==0))
{
auxdep=auxarr+jtype[j];
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;
if (count_total_arr==0)
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector start 1. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs]=new timesarrdep;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 1. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
```

```
}
}

else
{
if ((((auxarr+jji*limitdo) > arrvector[j]) && ((auxarr+jji*limitdo)
<= (arrvector[j+1]+epsilon))) && (found==0))
{
m=(depvector[j+1]-depvector[j])/(arrvector[j+1]-arrvector[j]);
if (m >= 1)
{
verticalaux=depvector[j]+m*(auxarr+jji*limitdo-arrvector[j]);
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;

if (count_total_arr==0)
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector start 2. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 2. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
```

```
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
else
{
verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo:

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
```

```
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
}
else
{
  verticalaux++;
int jjji=j+2;
int found=0;
do
{
if (((auxarr+jji*limitdo) <= (arrvector[jjji]+epsilon)) && (found==0))
{
m=(depvector[jjji]-depvector[jjji-1])/(arrvector[jjji]-arrvector[jjji-1]);
if (m >=1)
{
verticalaux=m*(auxarr+jji*limitdo-arrvector[jjji-1])+depvector[jjji-1];

auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;
auxdep=verticalaux;
found=1;

if (count_total_arr==0)
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 4. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
```

```
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 4. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}    // if m >=1
else  // SO IF M<=1
{
if ((auxarr+jji*limitdo) >= depvector[jjji-1])
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
```

```
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
else
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[jjji-1];

if ((count_total_arr == 0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
```

```
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
}

}

jjji++;
} while ((found==0) && (jjji <= Np));

if ((found==0) && ((auxarr+jji*limitdo) <= (depvector[Np]+epsilon)))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[Np];

if (count_total_arr==0)
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 5. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
```

```
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 5. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
else
{
if ((found==0) && ((auxarr+jji*limitdo) > depvector[Np]))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if (count_total_arr==0)
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector start 6. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
```

```
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 6. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
}
}
}
}

//   THIS MUST BE THE END OF IF J==0 - LINE 1435

else
{
if (j==1)
j=1;
if (j < Np)
{
if (auxarr>=1.0)
auxarr=auxarr;
if (((auxarr+jji*limitdo > arrvector[j-1]) && (auxarr+jji*limitdo <=
(arrvector[j]+epsilon))) && (found==0))
{
m=(depvector[j]-depvector[j-1])/(arrvector[j]-arrvector[j-1]);

if (m >= 1)
```

```
{
verticalaux=depvector[j]+m*(auxarr+jji*limitdo-arrvector[j]);
// NO -> +verticalaux;  ?!?!
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 7. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}   // IF (M >= 1)
else
{
if ((auxarr+jji*limitdo) >=depvector[j-1])
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
```

```
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
else
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[j-1];

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
```

```
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}

}

}
if (((auxarr+jji*limitdo) <= (arrvector[j-1]+epsilon)) && (found==0))
{
verticalaux--;
if (j >=2)
{
int jjji=j-2;
int found=0;
do
{
if ((auxarr+jji*limitdo) > arrvector[jjji])
{
m=(depvector[jjji+1]-depvector[jjji])/(arrvector[jjji+1]-arrvector[jjji]);
verticalaux=m*(auxarr+jji*limitdo-arrvector[jjji+1])+verticalaux+depvector[j];
auxarrvector[jji]=auxarr+jji*limitdo;
```

```
auxdepvector[jji]=verticalaux;
auxdep=verticalaux;


time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 8. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
if (found==0) verticalaux--;
jjji--;
} while ((found==0) && (jjji >= 0));

if ((found==0) && ((auxarr+jji*limitdo) <= (depvector[0]+epsilon)))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 9. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
}

}
```

```
}
if (j <= 1)
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 10. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}

}
if ((((auxarr+jji*limitdo) > (arrvector[j])) && ((auxarr+jji*limitdo) <=
(arrvector[j+1]+epsilon))) && (found==0))
{
m=(depvector[j+1]-depvector[j])/(arrvector[j+1]-arrvector[j]);

if (m>=1)
{
verticalaux=depvector[j]+m*(auxarr+jji*limitdo-arrvector[j]);
// NO -> +verticalaux;  ?!?!
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 11. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
```

```
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

else
{
if ((auxarr+jji*limitdo) >= depvector[j])
{
verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
```

```
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
else
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[j];

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
```

```
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
}

}
if (((auxarr+jji*limitdo) >= arrvector[j+1]) && (found==0))
{
verticalaux++;
if (j <=(Np-2))
{
int jjji=j+2;
int found=0;
do
{
if ((auxarr+jji*limitdo) < arrvector[jjji])
{
m=(depvector[jjji]-depvector[jjji-1])/(arrvector[jjji]-arrvector[jjji-1]);
//verticalaux=m*(auxarr+jji*limitdo-arrvector[jjji-1])+
verticalaux+depvector[j-1];

if (m >=1)
{
verticalaux=m*(auxarr+jji*limitdo-arrvector[jjji-1])+depvector[jjji-1];
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;
auxdep=verticalaux;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 12. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
```

```
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

else
{
if ((auxarr+jji*limitdo) >= depvector[jjji-1])
{
verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
```

```
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
else
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[jjji-1];

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
```

```
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}
}
}
if (found==0) verticalaux++;
jjji++;
} while ((found==0) && (jjji <= Np));

if ((found==0) && ((auxarr+jji*limitdo) <= (depvector[Np]+epsilon)))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[Np];

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 13. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}
else
{
if ((found==0) && ((auxarr+jji*limitdo) > depvector[Np]))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;
```

```
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 14. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}
}
}

if (j==(Np-1))
{
if ((found==0) && ((auxarr+jji*limitdo) <= (depvector[Np]+epsilon)))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[Np];

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 13. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}
else
```

```
{
if ((found==0) && ((auxarr+jji*limitdo) > depvector[Np]))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 14. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}
}
}
}
}
}
if (j==Np)
{
if (((auxarr+jji*limitdo) > depvector[j]) && (found==0))
{
auxdep=auxarr+jji*limitdo+jtype[j];
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 15. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
```

```
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
else
{
if ((((auxarr+jji*limitdo) > arrvector[j])&&((auxarr+jji*limitdo) <=
(depvector[j]+epsilon))) && (found==0))
{
auxdep=depvector[j];
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[j];

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 16. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}

}
else
{
if ((((auxarr+jji*limitdo) > arrvector[j-1]) && ((auxarr+jji*limitdo) <=
(arrvector[j]+epsilon))) && (found==0))
{
m=(depvector[j]-depvector[j-1])/(arrvector[j]-arrvector[j-1]);

if (m >=1)
{
verticalaux=depvector[j]+m*(auxarr+jji*limitdo-arrvector[j]);
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;
auxdep=verticalaux;
```

```
time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 17. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
else
{
if ((auxarr+jji*limitdo) > depvector[j-1])
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
}
else
```

```
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}

}
else
{

verticalaux=auxarr+jji*limitdo;
auxdep=verticalaux;
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=depvector[j-1];

if ((count_total_arr==0) && (found==0))
{
time_vector_start[runs]=new timesarrdep;
if (time_vector_start[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector strat 3. " << endl;
break;
}
else
{
time_vector_start[runs] -> arrivint_position = count_total_arr;
time_vector_start[runs] -> arr_time = auxarrvector[jji];
time_vector_start[runs] -> dep_time = auxdepvector[jji];
time_vector_start[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_start[runs];
count_total_arr++;
found=1;
start++;
}
```

```
}
else
{

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 3. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
}

}
}
else
{
verticalaux--;
int jjji=j-2;
int found=0;
do
{
if ((auxarr+jji*limitdo) > arrvector[jjji])
{
m=(depvector[jjji+1]-depvector[jjji])/(arrvector[jjji+1]-arrvector[jjji]);
verticalaux=m*(auxarr+jji*limitdo-arrvector[jjji+1])+verticalaux+depvector[Np];
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=verticalaux;
auxdep=verticalaux;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 18. " << endl;
break;
}
else
{
```

```
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
found=1;
}
}
if (found==0) verticalaux--;
jjji--;
}while ((found==0) && (jjji >=0));
if ((found==0) && ((auxarr+jji*limitdo) <= (arrvector[0]+epsilon)))
{
auxarrvector[jji]=auxarr+jji*limitdo;
auxdepvector[jji]=auxarr+jji*limitdo;
auxdep=auxarr+jji*limitdo;

time_vector_point[runs] = new timesarrdep;
if (time_vector_point[runs] == NULL)
{
cout << "Insuff. Memory - Program Term. - Time vector point 19. " << endl;
break;
}
else
{
time_vector_point[runs] -> arrivint_position = count_total_arr;
time_vector_point[runs] -> arr_time = auxarrvector[jji];
time_vector_point[runs] -> dep_time = auxdepvector[jji];
time_vector_prior[runs] -> next = time_vector_point[runs];
time_vector_point[runs] -> next = NULL;
time_vector_prior[runs] = time_vector_point[runs];
count_total_arr++;
}

}
}
}
}
}
}
int auxaux1=1;
if (time_vector_point[runs]->arr_time>=0.8)
auxaux1=1;
auxarr=auxarr+MAX_uniform*limitdo;
} while(auxarr <= (limitwhile+0.000001));
}  //if19 - closing: added at 16/sep/99
}
```

```
changecom1=0;
time_vector_point[runs]=time_vector_start[runs];
count=0;

// if ((runstotal==0) | (runstotal==7576) | (runstotal==30099))
if ((runstotal==0) | (runstotal==7550))
// if (runstotal==0)
{
do
{
time_vector_prior[runs]=time_vector_point[runs] -> next;
a_time[count]=time_vector_point[runs] ->arr_time;
d_time[count]=time_vector_point[runs] ->dep_time;
time_vector_point[runs] =time_vector_point[runs]  -> next;
count++;
}while(time_vector_prior[runs] != NULL);
}

int MAX_sj;
MAX_sj=count-1;

fout1 << endl << " The values of a_time and d_time are: " << endl << endl;
count=0;

if (runstotal==0)
{
for (count=0; count <= MAX_sj; count++)
{

fout1 << a_time[count] << "   " <<  d_time[count] << endl;

count++;
}
}

time_vector_point[runs]=time_vector_start[runs];
count=0;
if (runstotal >=1)
{
do
{
time_vector_prior[runs]=time_vector_point[runs] -> next;
time_vector_point[runs] ->arr_time= alfa_time*(a_time[count])+beta_time*
(time_vector_point[runs] ->arr_time);
time_vector_point[runs] ->dep_time= alfa_time*(d_time[count])+beta_time*
(time_vector_point[runs] ->dep_time);
time_vector_point[runs] =time_vector_point[runs]  -> next;
count++;
```

```
}while(time_vector_prior[runs] != NULL);
}

double aux1;
finalcaspoint[i]=finalcasstart[i];
do
{

finalcasprior[i]=finalcaspoint[i] -> next;
cout << "The commuter is and the run is " << finalcaspoint[i] -> arriv_position
<< " - " << runstotal << endl;
desired_time=finalcaspoint[i] -> des_time;
changecom2=0;
aux1_max=0;
aux1=0;

/*************************************************************************
   THE NEXT STATEMENT FOR THE change_commuter_2 VARIABLE IS EXTREMELLY
   IMPORTANT BECAUSE IT WILL SAY IF THE COMMUTERS WILL PASS THROUGH
   OR NOT BY THE BOTTLENECK, IF THEY ARE EITHER PAYING OR FREE
*************************************************************************/

change_commuter_2=0;
double a1;
int b1;
srand(time(0));
a1=double (rand() % Np * 100 / Np ) /100;
if (a1<=(1-aux_cost_and_runs[runs]))
b1=1;
else
b1=0;


time_vector_point[runs] = time_vector_start[runs];
count_time=0;
nauxcostfree=0;
nauxcostpay=0;
do
{
if (time_vector_point[runs] == NULL)
cout << "Problem already." << endl;
time_vector_prior[runs] = time_vector_point[runs] -> next;
auxarrvector[count_time] = (time_vector_point[runs] -> arr_time);
auxdepvector[count_time] = (time_vector_point[runs] -> dep_time);
auxarr1=auxarrvector[count_time];
auxdep1=auxdepvector[count_time];
newauxcost=Cost(auxarr1,auxdep1,desired_time,early_rate,late_rate);
nauxcostfree=newauxcost+nauxcostfree;
if ((finalcaspoint[i] -> pos_cases) ==0)
```

```
{
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
nauxtollcost=C_toll(f,dif_value_of_time, auxdep1,alfa_rate,toll_rate1,toll_rate2,
WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point, vector_toll_prior, Np);
nauxcostpay=nauxtollcost+newauxcost+nauxcostpay;
}
count_time++;
time_vector_point[runs] = time_vector_point[runs] -> next;
} while ((count_time <= (MAX_uniform-1)) && (time_vector_prior[runs] != NULL));


int test5=0;
int position5;
do
{

if ((time_vector_point[runs] == NULL) && (test5==1))
{
cout << "The positiopn was: " << position5 << endl;
cout << "Problem already 2." << endl;
}
if (time_vector_point[runs] == NULL)
cout << "Problem already 2." << endl;
time_vector_prior[runs]=time_vector_point[runs] -> next;
auxarr=auxarrvector[(MAX_uniform-1)/2];
auxdep=auxdepvector[(MAX_uniform-1)/2];

if ((finalcaspoint[i] -> pos_cases) ==0)
{
nauxcostpay_per_time=nauxcostpay/MAX_uniform;
aux1=finalcaspoint[i]->com_paying-nauxcostpay_per_time;
if (((finalcaspoint[i] -> willingness_to_pay) > nauxcostpay_per_time) |
(demand_elastic != 1))
{
if (aux1_max < aux1)
{
auxarr_min = auxarr;
auxdep_min = auxdep;
aux1pay_per_time_min = nauxcostpay_per_time;
aux1_max = aux1;
}
}
}
else
{
nauxcostfree_per_time=nauxcostfree/MAX_uniform;
aux1=finalcaspoint[i]->com_free-nauxcostfree_per_time;

if (((finalcaspoint[i] -> willingness_to_pay) > nauxcostfree_per_time) |
```

```
(demand_elastic != 1))
{
if (aux1_max < aux1)
{
auxarr_min = auxarr;
auxdep_min = auxdep;
aux1free_per_time_min = nauxcostfree_per_time;
aux1_max = aux1;
}

}
}

if ((time_vector_point[runs] -> arrivint_position) >= MAX_uniform)
{
auxarr1=auxarrvector[0];
auxdep1=auxdepvector[0];
newauxcost=Cost(auxarr1,auxdep1,desired_time,early_rate,late_rate);
nauxcostfree=nauxcostfree - newauxcost;
if ((finalcaspoint[i] -> pos_cases) == 0)
{
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
nauxtollcost=C_toll(f,dif_value_of_time,auxdep1,alfa_rate,toll_rate1,
toll_rate2,WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point,
vector_toll_prior, Np);
nauxcostpay=nauxcostpay - newauxcost - nauxtollcost;
}

for (int jji=0; jji < (MAX_uniform-1); jji++)
{
auxarrvector[jji] = auxarrvector[jji+1];
auxdepvector[jji] = auxdepvector[jji+1];
countjji=jji;
}
}
countjji++;
if (time_vector_prior[runs]  == NULL)
{
test5=1;
position5=time_vector_point[runs] -> arrivint_position;
}
if (time_vector_point[runs] != NULL)
{
auxarrvector[countjji]=time_vector_point[runs] -> arr_time;
auxdepvector[countjji]=time_vector_point[runs] -> dep_time;
auxarr1=auxarrvector[countjji];
auxdep1=auxdepvector[countjji];
newauxcost=Cost(auxarr1,auxdep1,desired_time,early_rate,late_rate);
nauxcostfree=newauxcost+nauxcostfree;
```

```
if ((finalcaspoint[i] -> pos_cases) ==0)
{
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
// nauxtollcost=C_toll(auxdep1,alfa_rate,toll_rate,WB,WE);
nauxtollcost=C_toll(f,dif_value_of_time, auxdep1,alfa_rate,toll_rate1,toll_rate2,
WB1,WB2,WE1,WE2, vector_toll_start, vector_toll_point, vector_toll_prior, Np);
nauxcostpay=nauxtollcost+newauxcost+nauxcostpay;
}
}
time_vector_point[runs] = time_vector_point[runs] -> next;

}while(time_vector_prior[runs] != NULL);
if ((finalcaspoint[i] -> pos_cases) ==0)
{
if (((finalcaspoint[i] -> willingness_to_pay) > aux1pay_per_time_min) |
(demand_elastic != 1))
{
/*
VERY IMPORTANT THE NEXT LINE - IT WAS ADDED AT 11/AUG/99
*/
if (((aux1_max/(finalcaspoint[i]->com_paying+0.000001)) > epsilon) |
(finalcaspoint[i]-> arr_time > pow(10,12)))
{
double cpay1,cpay2;
double user_probability, Prob_annealing;
cpay1 = finalcaspoint[i]->com_paying;
cpay2=  aux1pay_per_time_min;

user_probability = F_probability(ctype,runstotal,cpay1, cpay2);
Prob_annealing=F_prob(cpay1,cpay2,simulations,runstotal,p_f1,c1*early_rate);
zaux = c1*early_rate;
f_probconv=(cpay1-cpay2)/zaux;
if (f_probconv > 0.20)
{
if (Prob_annealing <=5*f_probconv)
// Prob_annealing =5*f_probconv;
Prob_annealing =Prob_annealing;
}

if ((Prob_annealing >= 1) | (Prob_annealing <= 0.01))
{
if ((5*f_probconv) >=1)
// Prob_annealing =1;
Prob_annealing =Prob_annealing;
else
// Prob_annealing =5*f_probconv;
Prob_annealing =Prob_annealing;
}
Prob_annealing << "  " << f_probconv << endl;
```

```
// fout1 << "And the new Prob_annealing is: "<< Prob_annealing <<  endl;

if ((user_probability<Prob_annealing)|(finalcaspoint[i]->arr_time >pow(10,12)))
{

// EXTREMELLY IMPORTANT FOR THE RANDOM NUMBERS -- CHANGED AT 12/AUG/1999: 4:42 PM
double a;
srand(time(0)*runstotal*(finalcaspoint[i]-> arr_time*finalcaspoint[i]-> des_time)
*cpay2);
a=rand()/(RAND_MAX+1.0);
// fout1 << "The value of a is: " << a << endl;
// n_random = 5*limitdo*a-2.5;
n_random = ((MAX_uniform-1)*a-(MAX_uniform-1))*limitdo;
//EXTREMELLY IMPORTANT FOR THE RANDOM NUMBERS-CHANGED(END) AT 12/AUG/1999:4:42PM
change_commuter_2=1;
auxarr_min=auxarr_min+n_random;
finalcaspoint[i] -> com_paying=aux1pay_per_time_min;
finalcaspoint[i] -> arr_time=auxarr_min;
changetime++;
if (changecom2==0)
{
changecom1++;
changecom2=1;
}
}
}
}
}
else
{

if (((finalcaspoint[i] -> willingness_to_pay) > aux1free_per_time_min) |
(demand_elastic != 1))
{
if (((aux1_max/(finalcaspoint[i]->com_free+0.000001)) > epsilon)|
(finalcaspoint[i]-> arr_time > pow(10,12)))
{

double cfree1,cfree2;
double user_probability, Prob_annealing;
cfree1 = finalcaspoint[i]->com_free;
cfree2=  aux1free_per_time_min;

converge=0;
user_probability = F_probability(ctype,runstotal,cfree1,cfree2);
Prob_annealing=F_prob(cfree1,cfree2,simulations,runstotal,p_f1,c1*early_rate);
zaux = c1*early_rate;
f_probconv=(cfree1-cfree2)/zaux;
if (f_probconv > 0.20)
```

```
{
if (Prob_annealing <=5*f_probconv)
// Prob_annealing =5*f_probconv;
Prob_annealing =Prob_annealing;
}

if ((Prob_annealing >= 1) | (Prob_annealing <= 0.01))
{
if ((5*f_probconv) >=1)
// Prob_annealing =1;
Prob_annealing =Prob_annealing;
else
// Prob_annealing =5*f_probconv;
Prob_annealing =Prob_annealing;
}


if ((user_probability<Prob_annealing)|(finalcaspoint[i]-> arr_time>pow(10,12)))
{
double n_seed;

// EXTREMELLY IMPORTANT FOR THE RANDOM NUMBERS -- CHANGED AT 12/AUG/1999: 4:42 PM
double a;
srand(time(0)*runstotal*(finalcaspoint[i]-> arr_time*
finalcaspoint[i]->des_time)*cfree2);
a=rand()/(RAND_MAX+1.0);
// fout1 << "The value of a is: " << a << endl;
n_random = ((MAX_uniform-1)*a-(MAX_uniform-1))*limitdo;
// EXTREMELLY IMPORTANT FOR THE RANDOM NUMBERS-CHANGED(END)AT 12/AUG/1999:4:42 PM
change_commuter_2=1;
auxarr_min=auxarr_min+n_random;
finalcaspoint[i] -> com_free=aux1free_per_time_min;
// finalcaspoint[i] -> arrival_timef=auxarr;
finalcaspoint[i] -> arr_time=auxarr_min;
changetime++;
if (changecom2==0)
{
changecom1++;
changecom2=1;
}
int testaux=1;
if ((changecom1 > Np) && (runs > 25))
testaux=1;
}
}
}
}

if (finalcaspoint[i] -> pos_cases==1)
```

```
{
if ((change_commuter_2 == 0) && (demand_elastic == 1) &&
((finalcaspoint[i] -> willingness_to_pay) < aux1free_per_time_min))
{
finalcaspoint[i] -> use_bottleneck=0;
     finalcaspoint[i] -> arr_time = pow(10,13);
finalcaspoint[i] -> departure_time = pow(10,13);
finalcaspoint[i] -> dep_time = pow(10,13);
finalcaspoint[i] -> com_paying = (finalcaspoint[i] -> willingness_to_pay);
finalcaspoint[i] -> com_free = (finalcaspoint[i] -> willingness_to_pay);
if (finalcaspoint[i] -> total_cost > finalcaspoint[i] -> initial_cost)
finalcaspoint[i] -> better_off=0;

}
}
if (finalcaspoint[i] -> arriv_position==4)
finalcaspoint[i] -> arriv_position=4;
if (finalcaspoint[i] -> pos_cases==0)
{
if ((change_commuter_2 == 0) && (demand_elastic == 1) &&
((finalcaspoint[i] -> willingness_to_pay) < aux1pay_per_time_min))
{
finalcaspoint[i] -> use_bottleneck=0;
     finalcaspoint[i] -> arr_time = pow(10,13);
finalcaspoint[i] -> departure_time = pow(10,13);
finalcaspoint[i] -> dep_time = pow(10,13);
finalcaspoint[i] -> com_paying = (finalcaspoint[i] -> willingness_to_pay);
finalcaspoint[i] -> com_free = (finalcaspoint[i] -> willingness_to_pay);
if (finalcaspoint[i] -> total_cost > finalcaspoint[i] -> initial_cost)
finalcaspoint[i] -> better_off=0;

}
}
finalcaspoint[i]=finalcaspoint[i] -> next;
}while (finalcasprior[i] != NULL);

} // END OF THE FOR LOOP
double auxlastdep;
for (i=0; i<= limcomb; i++)
{
count = 0;
cumuarrival = 0;
cumudeparture = 0;
finalcaspoint[i]=finalcasstart[i];

finalcaspoint[i]=finalcasstart[i];
do
{
finalcasprior[i]=finalcaspoint[i] -> next;
```

```
if (finalcaspoint[i] -> arr_time < pow(10,12))
{
arrvector[count]=(finalcaspoint[i] -> arr_time);
}
finalcaspoint[i]=finalcaspoint[i] -> next;
count++;
}while(finalcasprior[i] != NULL);

  int sj, si;
  double as;

for (sj=1; sj <=Np; sj++)
{
as=arrvector[sj];
si=sj-1;
while ((si>=0) && (arrvector[si] > as))
{
arrvector[si+1]=arrvector[si];
si--;
}
arrvector[si+1]=as;
}
for (int j=0; j <= Np; j++)
{
if (j==0) depvector[j]=arrvector[j];
else
{
if ((arrvector[j]-depvector[j-1]) >= 1)
depvector[j] =arrvector[j];
else depvector[j]=depvector[j-1] + 1;
}
}
for (int jji=0; jji <= limcomb; jji++)
{
finalauxpoint[jji]=finalcasstart[jji];
do
{
finalauxprior[jji] = finalauxpoint[jji] -> next;
finalauxpoint[jji] -> updateddeparture=0;
finalauxpoint[jji]=finalauxpoint[jji] -> next;
}while(finalauxprior[jji] != NULL);
}
count=0;
double auxlastdep, auxd;
for (j=0;j<=Np;j++)
{
finalcaspoint[i]=finalcasstart[i];
do
{
```

```
finalcasprior[i]=finalcaspoint[i] -> next;

/*********************************************************************
EXPLANATION FOR THE EVALUATION OF THE DEPARTURE TIME
(1) SHE/HE DEPARTS ""AS SOON AS SHE/HE ARRIVES"":
(a) IF she/he is the first to arrive (count=0);
(b) IF she/he arrived more than one second ""after""
the last departure;
(2) OTHERWISE:
 - she/he will depart one second ""after"" the last departure;
*********************************************************************/

/*********************************************************************/
/* FIRST EVALUATING THE DEPARTURE TIME WHEN THE
/* COMMUTERS ARE FREE
/*********************************************************************/

int auxaux;
if (runs >= 1)
auxaux=0;
if (finalcaspoint[i]->arr_time < pow(10,12))   // ADDED AT 10/AUG/1999
{ // ADDED AT 10/AUG/1999 (FROM THE "IF" ABOVE)
if (((finalcaspoint[i]-> arr_time)==arrvector[j]) &&
(finalcaspoint[i] -> pos_cases ==1) && (
finalcaspoint[i] -> updateddeparture != 1) && (auxlastdep != depvector[j]))
{
if (count==0)
{
//finalcaspoint[i]->departure_timef=(finalcaspoint[i]->arrival_timef);
finalcaspoint[i]->dep_time=(finalcaspoint[i]->arr_time);
finalcaspoint[i] -> updateddeparture = 1;
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
else
{
if (((finalcaspoint[i]-> arr_time)-auxlastdep) > 1)
{
// finalcaspoint[i]->departure_timef=(finalcaspoint[i]->arrival_timef);
finalcaspoint[i]->dep_time=(finalcaspoint[i]->arr_time);
finalcaspoint[i] -> updateddeparture = 1;
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
else
{
// finalcaspoint[i]->departure_timef=auxlastdep+1;
finalcaspoint[i]->dep_time=auxlastdep+1;
finalcaspoint[i] -> updateddeparture = 1;
```

```
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
}
count++;
}

/*********************************************************************/
/* SECOND EVALUATING THE DEPARTURE TIME WHEN THE
/* COMMUTERS ARE PAYING
/*********************************************************************/

if ((finalcaspoint[i]->arr_time==arrvector[j]&&finalcaspoint[i]->pos_cases==0)
&&(finalcaspoint[i] -> updateddeparture != 1) && (auxlastdep != depvector[j]))
{
if (count==0)
{
// finalcaspoint[i]->departure_timep=(finalcaspoint[i]->arrival_timep);
finalcaspoint[i]->dep_time=(finalcaspoint[i]->arr_time);
finalcaspoint[i] -> updateddeparture = 1;
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
else
{
if (((finalcaspoint[i]-> arr_time)-auxlastdep)>1)
{
// finalcaspoint[i]->departure_timep=(finalcaspoint[i]->arrival_timep);
finalcaspoint[i]->dep_time=(finalcaspoint[i]->arr_time);
finalcaspoint[i] -> updateddeparture = 1;
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
else
{
// finalcaspoint[i]->departure_timep=auxlastdep+1;
finalcaspoint[i]->dep_time=auxlastdep+1;
finalcaspoint[i] -> updateddeparture = 1;
auxd=(finalcaspoint[i] -> dep_time);
auxlastdep=auxd;
}
}
count++;
}

} // ADDED AT 10/AUG/1999

finalcaspoint[i] = finalcaspoint[i] -> next;
}while (finalcasprior[i]!=NULL);
```

```
}
auxlastdep=0;
}
for (i=0; i<=limcomb; i++)
{
finalcaspoint[i]=finalcasstart[i];
do
{
finalcasprior[i]= finalcaspoint[i] -> next;

desired_time=finalcaspoint[i] -> des_time;
if (finalcaspoint[i] -> arriv_position==4)
finalcaspoint[i] -> arriv_position=4;

if ((finalcaspoint[i] -> pos_cases)==0)
{
// auxdep=(finalcaspoint[i]->departure_timep);
auxdep=(finalcaspoint[i]->dep_time);
alfa_rate=finalcaspoint[i] ->difer_value_of_time;
nauxtollcost=C_toll(f,dif_value_of_time,
auxdep,alfa_rate,toll_rate1,toll_rate2,WB1,WB2,WE1,
WE2,vector_toll_start, vector_toll_point, vector_toll_prior, Np);
auxarr=(finalcaspoint[i] -> arr_time);
newauxcost=Cost(auxarr,auxdep,desired_time,early_rate,late_rate);
nauxcostpay=newauxcost+nauxtollcost;
if (auxdep < pow(10,12))
{
finalcaspoint[i] -> com_paying=nauxcostpay;
}
else finalcaspoint[i] -> com_paying = finalcaspoint[i] -> willingness_to_pay;

}

else
{
auxdep=(finalcaspoint[i]->dep_time);
auxarr=(finalcaspoint[i] -> arr_time);
newauxcost=Cost(auxarr,auxdep,desired_time,early_rate,late_rate);
nauxcostfree=newauxcost;
if (auxdep < pow(10,12))
{
finalcaspoint[i] -> com_free=nauxcostfree;
}
else finalcaspoint[i] -> com_free = finalcaspoint[i] -> willingness_to_pay;
}
finalcaspoint[i]=finalcaspoint[i] -> next;
} while(finalcasprior[i] != NULL);
}
```

```
        for (int jji=0; jji <= limcomb; jji++)
{
finalauxpoint[jji]=finalcasstart[jji];
do
{
finalauxprior[jji] = finalauxpoint[jji] -> next;
finalauxpoint[jji]=finalauxpoint[jji] -> next;
}while(finalauxprior[jji] != NULL);
}
double freecost;
double paycost;
double totalcost;
for (j=0; j <= Np; j++)
{
freecost=0;
paycost=0;


if (j==0)
j=0;
for (int ji=0; ji <= limcomb; ji++)
{
finalauxpoint[ji]=new commuters;
finalauxpoint[ji]=finalcasstart[ji];
do
{
finalauxprior[ji]=finalauxpoint[ji] -> next;
if ((finalauxpoint[ji] -> arriv_position)==j)
{
if ((finalauxpoint[ji] -> pos_cases)==1)
{
if (finalauxpoint[ji] -> com_free != -1)
    freecost += (finalauxpoint[ji] -> com_free);
else freecost += (finalauxpoint[ji] -> willingness_to_pay);
}

else
{
   if (finalauxpoint[ji] -> com_paying != -1)
    paycost += (finalauxpoint[ji] -> com_paying);
else paycost += (finalauxpoint[ji] -> willingness_to_pay);

}
}
finalauxpoint[ji] = finalauxpoint[ji] -> next;
}while(finalauxprior[ji] != NULL);
}
if (finalcaspoint[ji] -> arriv_position==1)
```

```
finalcaspoint[ji] -> arriv_position=1;
if (f==0) totalcost=paycost;///double(n_denominator);
else
{
if (f==1) totalcost=freecost;///double(n_denominator);
else
totalcost=((f*freecost/n_numerator)+((1-f)*paycost)/(n_denominator-n_numerator));
}

for (ji=0; ji<= limcomb; ji++)
{
finalauxpoint[ji]=finalcasstart[ji];
do
{

finalauxprior[ji]=finalauxpoint[ji] -> next;
if (finalauxpoint[ji] -> arriv_position==j)
{
                            finalauxpoint[ji] ->total_cost=totalcost;
}
finalauxpoint[ji]=finalauxpoint[ji] ->next;
}while (finalauxprior[ji] != NULL);
// cout << " I passed here 2" << endl;
} // END OF THE for (ji=0...)
} // END OF THE for (j=0; j <= (Np-1); j++)*/

/*************************************************************************
FREEING THE DYNAMIC ALLOCATED MEMORY FOR THE time_vector_point VARIABLES
*************************************************************************/
time_vector_point[runs]=time_vector_start[runs];
count=0;
if (runstotal>=1)
{
do
{
time_vector_prior[runs]=time_vector_point[runs] -> next;
a_time[count]=time_vector_point[runs] ->arr_time;
d_time[count]=time_vector_point[runs] ->dep_time;
time_vector_point[runs] =time_vector_point[runs]  -> next;
count++;
}while(time_vector_prior[runs] != NULL);
}
int MAX_sj;
MAX_sj=count-1;
fout1<<endl<<"The values of a_time and d_time are:AFTER THE FIRST RUN "
<< endl<<endl;
count=0;
if (runstotal>=1)
{
```

```
for (count=0; count <= MAX_sj; count++)
{

fout1 << a_time[count] << "   " <<  d_time[count] << endl;

count++;
}
}
time_vector_point[runs] = time_vector_start[runs];
do
{
// cout << endl << "intruns = " << runs << endl;
if (time_vector_point[runs] == NULL)
cout << "Problem already 2 in the intruns." << endl;
time_vector_prior[runs]=time_vector_point[runs] -> next;
delete (time_vector_point[runs]);
time_vector_point[runs] = time_vector_prior[runs];
} while (time_vector_prior[runs] != NULL);
runs=0;


/*************************************************************************
ENDING THE DYNAMIC ALLOCATED MEMORY FOR THE time_vector_point VARIABLES
*************************************************************************/
             for (int ji=0; ji<= limcomb; ji++)
{
finalcaspoint[ji]=finalcasstart[ji];
do
{

finalcasprior[ji]=finalcaspoint[ji] -> next;
if ((finalcaspoint[ji] -> total_cost) <= (finalcaspoint[ji] -> initial_cost))
{
                              finalcaspoint[ji] ->better_off=1;
}
else
{
// if (finalcaspoint[ji] -> use_bottleneck != 0)
finalcaspoint[ji] ->better_off=0;
}
finalcaspoint[ji]=finalcaspoint[ji] ->next;
}while (finalcasprior[ji] != NULL);
// cout << " I passed here 2" << endl;
}

if ((runstotal ==0) |)|((runstotal >2050) &&(runstotal <2067))|
((runstotal > 9100) && (runstotal <=9127))| (( runstotal > 25550) &&
(runstotal < 25577))|((runstotal > 46100)&&(runstotal <=46150))|
((runstotal > 60100)&&(runstotal <=60200)))
```

```
{
fout1 << " THE VALUE OF THIS RUN IS: " << runs << " AND RUNSTOTAL IS: " <<
runstotal << endl << endl;

for (int ii=0; ii<= limcomb; ii++)
{
count=0;
fout1 << "Combination: " << ii << " The commuter and toll costs are " << endl;
finalcaspoint[ii]=finalcasstart[ii];
fout1 << "Commuter - Costs:       Willingness to Pay      Initial Cost
Free    " << "      Paying    " << "      Total    " << "Arr. Time" <<
"     Dep. Time" << "    Poss.Cases " <<  "   Desired Time" <<
"     Use Bottleneck      Better off " << endl;
do
{
fout1 << " " << count << "   ";
finalcasprior[ii]=finalcaspoint[ii] -> next;
if((finalcaspoint[ii]->arr_time < 0.000001)&&(finalcaspoint[ii]->arr_time>
(-0.000001)))
finalcaspoint[ii] -> arr_time = 0;
if((finalcaspoint[ii]->dep_time < 0.000001)&&(finalcaspoint[ii]->dep_time>
(-0.000001)))
finalcaspoint[ii] -> dep_time = 0;
fout1 << "                " << setw(13) <<
finalcaspoint[ii] -> willingness_to_pay  << setw(20) <<
finalcaspoint[ii] -> initial_cost << setw(20) << finalcaspoint[ii] -> com_free
<< setw(13) << finalcaspoint[ii] -> com_paying
<< setw(13) << finalcaspoint[ii] -> total_cost << setw(16) <<
finalcaspoint[ii] -> arr_time << setw(13)
   <<finalcaspoint[ii] -> dep_time<<setw(13)<<finalcaspoint[ii] -> pos_cases<<
setw(15) << finalcaspoint[ii] -> des_time << setw(15) <<
finalcaspoint[ii]->use_bottleneck<<setw(15)<<finalcaspoint[ii]->better_off<<endl;
finalcaspoint[ii] =  finalcaspoint[ii] ->next;
count++;
}while(finalcasprior[ii] != NULL);
cout << "  "; //endl;
}
cout << " The percent of commuters who changed their times in this run was " <<
changecom1*100/(Np+1) << endl;
fout1 <<"  The value of changecom1 is " << changecom1 << endl;
fout1 << " The percent of commuters who changed their times in this run was " <<
changecom1*100/(Np+1) << endl;
fout1 << "The value of start is: " << start << endl;
fout1 <<endl << "END OF THE RESULTS FOR THE RUN " << runs << "." <<
" AND RUNSTOTAL IS: " << runstotal << "." << endl << endl << endl <<endl;

/*****************************************************************************
   THE NEXT PRINT STATEMENTS WERE ADDEDS AT 10/AUG/1999
*****************************************************************************/
```

```
if ((runstotal ==0) ||((runstotal >2050) && (runstotal <2067))||
((runstotal > 9100) && (runstotal <=9127))||(( runstotal > 25550) &&
(runstotal < 25577)) ||((runstotal > 46100)&& (runstotal <=46150))||
((runstotal > 60100)&&(runstotal <=60200)))
{
fout1 << "The value of WB1 and WB2 are: " << WB1 << " and " << WB2 << endl;
fout1 << "The value of WE1 and WE2 are: " << WE1 << " and " << WE2 << endl;
fout1 << " The value of limitdo, w and MAX_uniform are " << limitdo <<
" " << w << " " << MAX_uniform << endl;
for (int ii=0; ii<= limcomb; ii++)
{
// if (ii==0)
// {
count=0;
// fout1 << "The desired Time is:  " << desired_time << endl;
fout1 << "Combination: " << ii << " The commuter and toll costs are " << endl;
finalcaspoint[ii]=finalcasstart[ii];
do
{
// fout1 << count << "   ";
finalcasprior[ii]=finalcaspoint[ii] -> next;
if ((finalcaspoint[ii] -> arr_time < 0.000001) &&
(finalcaspoint[ii] -> arr_time > (-0.000001)))
finalcaspoint[ii] -> arr_time = 0;
if ((finalcaspoint[ii] -> dep_time < 0.000001) && (
finalcaspoint[ii] -> dep_time > (-0.000001)))
finalcaspoint[ii] -> dep_time = 0;
if ((finalcaspoint[ii] -> arr_time < pow(10,9)))
{
fout1 << count << "   ";
fout1 << "  " << setw(16) << finalcaspoint[ii] -> arr_time << setw(13)
<<finalcaspoint[ii]->dep_time<<setw(15)<<finalcaspoint[ii]->des_time<<endl;
// finalcaspoint[ii] =  finalcaspoint[ii] ->next;
count++;
}
finalcaspoint[ii] =  finalcaspoint[ii] ->next;
}while(finalcasprior[ii] != NULL);
cout << "  "; //endl;
}
}

}

runstotal++;
runs++;
}while(runs<simulations);
fout1 << "The value of the runs were " << (runs-1) << "and runstotal " <<
(runstotal-1) << endl;
```

```
fout1 << "The value of WB1 and WB2 are: " << WB1 << " and " << WB2 << endl;
fout1 << "The value of WE1 and WE2 are: " << WE1 << " and " << WE2 << endl;

count=0;
finalcaspoint[0]=finalcasstart[0];
do
{
fout1 << count << "   ";
finalcasprior[0]=finalcaspoint[0] -> next;
if ((finalcaspoint[0] -> arr_time < 0.000001) &&
(finalcaspoint[0] -> arr_time > (-0.000001)))
finalcaspoint[0] -> arr_time = 0;
if ((finalcaspoint[0] -> dep_time < 0.000001) &&
(finalcaspoint[0] -> dep_time > (-0.000001)))
finalcaspoint[0] -> dep_time = 0;
fout1 << "  " << setw(16) << finalcaspoint[0] -> arr_time << setw(13)
<<finalcaspoint[0] -> dep_time << setw(15)<<finalcaspoint[0] -> des_time<<endl;
finalcaspoint[0] =  finalcaspoint[0] ->next;
count++;
}while(finalcasprior[0] != NULL);
cout << "  "; //endl;
fout1.close();


}

/***   Function to calculate the Cost for a Commuter  ***/

double Cost(double A_t, double D_t, double Des_t, double e_rate, double l_rate)
{
double C_schedule, C_queuing;

if (D_t <= Des_t)
C_schedule = e_rate * (Des_t - D_t);

else
C_schedule = l_rate * (D_t - Des_t);

C_queuing = D_t - A_t;

return (C_schedule + C_queuing);
}

double C_toll(double f, double val_of_time, double D_t, double a_rate,
double t_rate1, double t_rate2, double W_B1, double W_B2, double W_E1,
double W_E2, function_toll *v_toll_start, function_toll *v_toll_point,
function_toll *v_toll_prior, int np)
{
double Ctoll,x0,y0, y1, y;
```

```
int f_true;

if ((D_t <= W_B2) || (D_t >= W_E2) || (f ==1))
{
 Ctoll=0;
}
else
{
    if (toll_funct==2)
{
        if ((D_t > W_B2) && (D_t < W_E2))
{
       if ((a_rate*t_rate1*(D_t-W_B2)) < (a_rate*t_rate1*(W_E2 - D_t)))
     Ctoll = a_rate*t_rate1 * (D_t - W_B2);
       else
     Ctoll = a_rate*t_rate1 * (W_E2 - D_t);
}
    else
  Ctoll=0;

}
else  // ELSE 14A1
{
if (toll_funct==1)
{
if ((D_t <= 0) || (D_t >= np) )
{
Ctoll=0;
}
else
{
v_toll_point = v_toll_start;
v_toll_prior = v_toll_point -> next;
x0 = (v_toll_point -> first_value);
y0 = (v_toll_point -> value);
if (D_t <= (x0+1))
{
y1= v_toll_prior -> value;
y=(y1-y0)*(D_t-x0)+y0;
Ctoll=y;
}
else
{
v_toll_point = (v_toll_point -> next);
f_true=0;
do
{
v_toll_prior = (v_toll_point -> next);
x0 = (v_toll_point -> first_value);
```

```
y0 = (v_toll_point -> value);

if (D_t <=(x0+1))
{
y1= v_toll_prior -> value;
y=(y1-y0)*(D_t-x0)+y0;
Ctoll=y;
f_true=1;
}
else
{
v_toll_point = (v_toll_point -> next);
}

}while ((f_true =0) || (v_toll_point != NULL));
}
}
}
else // ELSE 13A1
{
if ((f !=0) && (f!=1) && (val_of_time !=1))
{
if ((D_t > W_B1) && (D_t < W_E1))
{
if ((a_rate*t_rate1*(D_t-W_B1)) < (a_rate*t_rate1*(W_E1 - D_t)))
  Ctoll = a_rate*t_rate1 * (D_t - W_B1);
else
 Ctoll = a_rate*t_rate1 * (W_E1 - D_t);
}
}
else
{
 if ((f==0) && (val_of_time ==1))
{

 if((W_B2 < -pow(10,9))&&(W_E2 < pow(10,9))&&((D_t <= W_B1)||(D_t >= W_E1)))
 {
 Ctoll=0;
 }

 else // ELSE 13C
 {

if ((((D_t > W_B2) && (D_t < W_B1)) || ((D_t > W_E1) && (D_t < W_E2))) &&
(W_B2 > -pow(10,9)) && (W_E2 < pow(10,9)))
{
if ((a_rate*t_rate2*(D_t-W_B2)) < (a_rate*t_rate2*(W_E2 - D_t)))
Ctoll = a_rate*t_rate2 * (D_t - W_B2);
else
```

```
Ctoll = a_rate*t_rate2 * (W_E2 - D_t);
}
else
{
if ((D_t > W_B1) && (D_t < W_E1) && (W_B2 > -pow(10,9)) && (W_E2 < pow(10,9)))
{
if ((a_rate*t_rate1*(D_t-W_B1)+a_rate*t_rate2*(W_B1-W_B2)) < (a_rate*t_rate1*
(W_E1 - D_t)+a_rate*t_rate2*(W_E2-W_E1)))
Ctoll = a_rate*t_rate1*(D_t-W_B1)+a_rate*t_rate2*(W_B1-W_B2);
else
Ctoll = a_rate*t_rate1*(W_E1 - D_t)+a_rate*t_rate2*(W_E2-W_E1);
}

}

 } // ELSE 13C
}
 else
 {
 if ((f!=0) && (val_of_time ==1))
{

 if ((((D_t > W_B2) && (D_t < W_B1)) || ((D_t > W_E1) && (D_t < W_E2))) && (
 W_B2 > -pow(10,9)) && (W_E2 < pow(10,9)))
{
if ((a_rate*t_rate2*(D_t-W_B2)) < (a_rate*t_rate2*(W_E2 - D_t)))
 Ctoll = a_rate*t_rate2 * (D_t - W_B2);
else
Ctoll = a_rate*t_rate2 * (W_E2 - D_t);
}
else
 {
if ((D_t > W_B1) && (D_t < W_E1) && (W_B2 > -pow(10,9)) && (W_E2 < pow(10,9)))
{
if ((a_rate*t_rate1*(D_t-W_B1)+a_rate*t_rate2*(W_B1-W_B2)) < (a_rate*t_rate1*
(W_E1 - D_t)+a_rate*t_rate2*(W_E2-W_E1)))
Ctoll = a_rate*t_rate1*(D_t-W_B1)+a_rate*t_rate2*(W_B1-W_B2);
else
Ctoll = a_rate*t_rate1*(W_E1 - D_t)+a_rate*t_rate2*(W_E2-W_E1);
}
else
{
if ((W_B2 < -pow(10,9)) || (W_E2 > pow(10,9)))
{
if ((D_t > W_B1) && (D_t < W_E1))
{
if (a_rate*t_rate1*(D_t-W_B1) < a_rate*t_rate1*(W_E2 - D_t))
Ctoll = a_rate*t_rate1*(D_t-W_B1);
else
```

```
Ctoll = a_rate*t_rate1*(W_E2 - D_t);
}

}
}
}
 }
 else // ELSE 13A
 {
 if ((f==0) && (val_of_time !=1))
 {

if ((D_t <= W_B1) || (D_t >= W_E1))
{
Ctoll = 0;
}

else
{
if ((D_t > W_B1) && (D_t < W_E1))
{
if ((a_rate*t_rate1*(D_t-W_B1)) < (a_rate*t_rate1*(W_E1 - D_t)))
Ctoll = a_rate*t_rate1*(D_t-W_B1);
else
Ctoll = a_rate*t_rate1*(W_E1 - D_t);
}

}
 }
 } // ELSE 13A
 } // ELSE BEFORE IF ((F!=0) && (VAL_OF_TIME ==1))
}
} //ELSE 13A1
} //ELSE 14A1
}
  return(Ctoll);
}

double F_probability (int type, int runsf, double cost1,double cost2 )
{
double x_book_simul, u_prob;

if (type == 1)
{
srand((runsf+1)*cost1*cost2);
u_prob = double (rand() % 1000 *100/1000) /100;
}
else
{
```

```
if (type == 2)
{
srand((runsf+1)*cost1);
x_book_simul= - log(1- (double (rand() % 1000 *100/1000) /100));
u_prob=1-exp(-x_book_simul);
//srand(time(0));
}
}
return (u_prob);
}


double F_prob(double c1, double c2, int sim, int runs_f, double pf1, double z)
{
double fprob;
double aux_f, delta1;

if (c1 > c2)
delta1 = c1-c2;
else
delta1=c2-c1;
aux_f=alfa*pow(runs_f+alfai,alfa_power)*(1-exp(-(c1-c2)/(z*pow((runs_f+p_f2),2))
*pf1));
fprob = aux_f;
return (fprob);
}
```