August 1995

UILU-ENG-95-2232
DC-168

Decision and Control

Lane Assignment on Automated Highway Systems

Deepa Ramaswamy

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-95-2232 DC- 168	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (City, State, and ZIP Code) 1308 West Main Street Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Washington, DC 20050
8a. NAME OF FUNDING / SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Washington, DC 20050	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

LANE ASSIGNMENT ON AUTOMATED HIGHWAY SYSTEMS

12. PERSONAL AUTHOR(S)

RAMASWAMY, DEEPA

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	August 1995	270

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Automated Highway Systems, IVHS, Lane Assignments, Simulator

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

See attached

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.*

LANE ASSIGNMENT ON AUTOMATED HIGHWAY SYSTEMS

Abstract

The automation of highways as part of the IVHS program is seen as a way to alleviate congestion on urban highways. This thesis discusses the concept of lane assignment in the context of automated highway systems (AHS). Lane assignments represent the scheduling of the path taken by the vehicle once it enters an automated multi-lane corridor. In this thesis, we formulate the static lane assignment problem as an optimization problem. We establish a classification of lane assignment strategies, beginning from totally unconstrained strategies to general strategies, to constant lane strategies, to destination and origin monotone strategies and ending with highly ordered monotone strategies (the latter three are called partitioned strategies).

Lane assignment is performed and analyzed under two distinct performance measures. First, we consider the minimization of total travel time (i.e., the time spent traveling at steady state plus the time spent maneuvering) of all of the vehicles on the AHS. Two different models of maneuvering cost have been investigated, corresponding to different levels of congestion on the AHS. Under low levels of congestion, the lane assignment problem reduces to a linear programming (LP) problem. Under high levels of congestion, the lane assignment reduces to a quadratic programming (QP) problem. We show that under high levels of congestion, the optimal lane assignment tends to be partitioned or nearly partitioned. We also present algorithms that find a partitioned strategy in the neighborhood of a nonpartitioned strategy. Second, we study the balanced use of the lanes of the AHS. The lane assignment is formulated as a constrained optimization problem with the performance criterion being the balancing of the excess capacities in the sections of the AHS. This problem reduces to a worst case maxmin problem. We have developed algorithms for obtaining the optimal or suboptimal strategy. We present and analyze the results of a case study of a hypothetical automated highway under the two performance measures stated above. The results show that at high levels of congestion, the optimal lane assignment strategies are close to being partitioned and that the developed algorithms perform efficiently in calculating the optimal or suboptimal strategies.

The static lane assignment problem was studied to grasp the essential features of lane scheduling under simplifying assumptions that decouple the system-wide traffic flow problem from the dynamic effects of the motion of individual vehicles. To study the effect of various control, communication and scheduling schemes on the dynamic capacity of the highway, a C++ class library and a Tcl front end have been developed. These can be used to create an AHS simulator that may be used to study the speed, concentration and flow characteristics of traffic flow on an AHS. The AHS simulator is modular and expandable to enable the insertion of new options and tasks. A number of test simulation runs have been performed using the front end and the AHS classes in order to perform a face validation of the algorithms that have been coded in. Although no large-scale AHS simulations have been performed, the C++ classes and the front end provide the tools required to set one up, to run it and to obtain statistical information from it.

Keywords: Automated Highway Systems, IVHS, Lane assignment, Simulator

LANE ASSIGNMENT ON AUTOMATED HIGHWAY SYSTEMS

BY

DEEPA RAMASWAMY

B.Tech., Indian Institute of Technology, Madras, 1989
M.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

LANE ASSIGNMENT ON AUTOMATED HIGHWAY SYSTEMS

Deepa Ramaswamy, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1995
J. Medanic, W.R. Perkins, Advisors

The automation of highways as part of the IVHS program is seen as a way to alleviate congestion on urban highways. This thesis discusses the concept of lane assignment in the context of automated highway systems (AHS). Lane assignments represent the scheduling of the path taken by the vehicle once it enters an automated multi-lane corridor. In this thesis, we formulate the static lane assignment problem as an optimization problem. We establish a classification of lane assignment strategies, beginning from totally unconstrained strategies to general strategies, to constant lane strategies, to destination and origin monotone strategies and ending with highly ordered monotone strategies (the latter three are called partitioned strategies).

Lane assignment is performed and analyzed under two distinct performance measures. First, we consider the minimization of total travel time (i.e., the time spent traveling at steady state plus the time spent maneuvering) of all of the vehicles on the AHS. Two different models of maneuvering cost have been investigated, corresponding to different levels of congestion on the AHS. Under low levels of congestion, the lane assignment problem reduces to a linear programming (LP) problem. Under high levels of congestion, the lane assignment reduces to a quadratic programming (QP) problem. We show that under high levels of congestion, the optimal lane assignment tends to be partitioned or nearly partitioned. We also present algorithms that find a partitioned strategy in the neighborhood of a nonpartitioned strategy. Second, we study the balanced use of the lanes of the AHS. The lane assignment is formulated as a constrained optimization problem with the performance criterion being the balancing of the excess capacities in the sections of the AHS. This problem reduces to a worst case maxmin problem. We have developed algorithms for obtaining the optimal or suboptimal strategy. We present and analyze the results of a case study of a hypothetical automated highway under the two performance measures stated above. The results show that at high

levels of congestion, the optimal lane assignment strategies are close to being partitioned and that the developed algorithms perform efficiently in calculating the optimal or suboptimal strategies.

The static lane assignment problem was studied to grasp the essential features of lane scheduling under simplifying assumptions that decouple the system-wide traffic flow problem from the dynamic effects of the motion of individual vehicles. To study the effect of various control, communication and scheduling schemes on the dynamic capacity of the highway, a C++ class library and a Tcl front end have been developed. These can be used to create an AHS simulator that may be used to study the speed, concentration and flow characteristics of traffic flow on an AHS. The AHS simulator is modular and expandable to enable the insertion of new options and tasks. A number of test simulation runs have been performed using the front end and the AHS classes in order to perform a face validation of the algorithms that have been coded in. Although no large-scale AHS simulations have been performed, the C++ classes and the front end provide the tools required to set one up, to run it and to obtain statistical information from it.

ACKNOWLEDGMENTS

The first people that I would always thank for being there for me are (in no particular order) my parents, my husband, Mike, my brother, Madhusudan, and my sister, Prema. I am what I am largely due to them, and I am grateful to them for it.

I thank my advisors, Professor Medanic and Professor Perkins, for a wonderful four years in which I have learned many things. I also thank Professor Benekohal for his ever-present good nature and willingness to help. I would also like to thank Professors Kumar, Meyn, Van Dooren, Bamieh, Basar, Spong, Sreenivas and Voulgaris, and my friends in the control group Rayadurgam Ravikanth, Sunil Kumar, Song Wu Lu, Zigang Pan, Joonyoul Choi, Mike Stewart, J. Sreedhar, Garry Didinsky, Steve Lu, Dave Hoover, Doug Down, Lyndon Brown ... and especially Becky, Francie and Dixie for all their help.

And thank you, too, Chechu Mama, for having encouraged me to continue with my Ph.D., for if you hadn't, I would not be writing this.

TABLE OF CONTENTS

CHAPTER		PAGE
1	**INTRODUCTION**	1
	1.1 Previous Work	2
	1.2 Contributions of this Thesis	3
	1.2.1 Analytical formulation	4
	1.2.2 AHS simulator	5
	1.2.3 Dynamic capacity and route guidance	7
	1.2.4 Issues for further study	7
	1.3 Organization of the Thesis	8
2	**LANE ASSIGNMENT ON THE AHS**	10
	2.1 The Need for Lane Assignment	10
	2.2 Organizational Structure Assumed for the AHS	12
	2.3 Modeling Traffic on an AHS	15
	2.3.1 Network model	16
	2.4 The Statement of the Problem	19
3	**LANE ASSIGNMENT STRATEGIES**	22
	3.1 Nonpartitioned Strategies	22
	3.2 Partitioned Strategies	23
	3.3 Size of the Classes Without Splitting of Flows	28
	3.3.1 Totally unconstrained strategies	29
	3.3.2 General lane assignment strategies	29
	3.3.3 Constant lane strategies	29
	3.3.4 Destination monotone strategies	30
	3.3.5 Origin monotone strategies	30
	3.3.6 Monotone strategies	30
	3.4 Size of the Classes with Splitting of Flows	32
4	**MINIMIZATION OF TOTAL TRAVEL TIME**	34
	4.1 The Travel Time Optimization Problem	34
	4.1.1 Motivation for minimization of travel time	34
	4.1.2 Motivation for the use of constant lane strategies	34
	4.1.3 Path determination through travel time comparison	35
	4.2 Formulation of the General Problem	36

4.3 Analytical Formulation . . . 37
4.3.1 Linear formulation . . . 37
4.3.2 Nonlinear formulation . . . 44
4.4 Sensitivity of the Optimal Assignment . . . 48
4.5 Extension of the Framework to Other Situations . . . 49

5 BALANCING EXCESS CAPACITIES . . . 50
5.1 Problem Formulation . . . 50
5.1.1 Motivation for balancing of excess capacities . . . 50
5.1.2 Motivation for the use of destination monotone strategies . . . 51
5.1.3 Balancing excess capacities . . . 51
5.2 Computing the Optimal Assignment . . . 54
5.2.1 Greedy-type algorithm iterations . . . 55
5.2.2 Gradient algorithm iteration . . . 56
5.2.3 Number of iterations required to reach the optimum . . . 57

6 FURTHER STUDIES ON PARTITIONED STRATEGIES . . . 59
6.1 Some Properties of Partitioned Strategies . . . 59
6.2 Parametric Study of the Optimal Strategies . . . 64
6.2.1 Suboptimal partitioned strategies . . . 67

7 CASE STUDY . . . 71
7.1 Problem Description . . . 71
7.2 Minimization of Travel Time . . . 73
7.2.1 Illustrative example . . . 73
7.2.2 Effect of maneuver costs on partitioning costs . . . 74
7.2.3 Effect of maneuver costs on partitions . . . 76
7.2.4 Performance loss with suboptimal partitioned assignments . . . 76
7.3 Balancing Excess Capacities . . . 78
7.3.1 Implementation of the lane assignment algorithm . . . 78
7.3.2 Discussion of the results . . . 79
7.3.3 Reducing the time boundary effect . . . 82
7.3.4 Computational requirements . . . 83

8 DESIGN OF AN AHS SIMULATOR . . . 85
8.1 Need for an AHS Simulator . . . 85
8.2 Survey of Existing Simulators . . . 86
8.3 Development of a C++ Class Library . . . 89
8.3.1 Abstraction . . . 90
8.3.2 Encapsulation . . . 91
8.3.3 Modularity . . . 92
8.3.4 Inheritance . . . 92
8.3.5 Creating new applications . . . 92
8.4 Description of the Various Classes . . . 93
8.4.1 Helper classes . . . 94

8.4.2 Network classes . . . 96
8.4.3 Driver/vehicle classes . . . 97
8.4.4 Generator classes . . . 104
8.4.5 Simulation classes . . . 107
8.4.6 Data collection classes . . . 111
8.4.7 Miscellaneous classes and structures . . . 113

9 VEHICLE CONTROL ALGORITHMS . . . 114
9.1 Longitudinal Control Logic . . . 114
9.1.1 Side control . . . 116
9.2 Car Following Algorithms . . . 116
9.2.1 Calculation of the noncollision jerk . . . 118
9.2.2 Calculation of the realistic jerk . . . 122
9.2.3 Follow a prescribed trajectory . . . 122
9.2.4 CSCarFollower's human car follower . . . 124
9.2.5 Vehicle model for the remaining car followers . . . 126
9.2.6 Automated intelligent cruise control . . . 127
9.2.7 Continuous platooning . . . 128
9.2.8 Discrete platooning . . . 129
9.2.9 Cruise control . . . 130
9.3 Lateral Control Logic . . . 130
9.3.1 Lane changing algorithms . . . 133
9.4 Path Control Logic . . . 137
9.5 Segment Control Logic . . . 138

10 FRONT END FOR THE SIMULATOR . . . 140
10.1 Why a Front End is Necessary . . . 140
10.1.1 Compiled versus interpreted front ends . . . 141
10.2 Creating a Front End in Tcl . . . 141
10.2.1 Classes used in the front end . . . 142
10.3 List of Commands in the Tcl Front End . . . 148
10.4 Post-processing . . . 148

11 SIMULATION TEST RUNS . . . 149
11.1 Test Run 1 . . . 149
11.2 Test Run 2 . . . 152
11.3 Test Run 3 . . . 153

12 DYNAMIC CAPACITY AND ROUTE GUIDANCE . . . 157
12.1 Estimating the Dynamic Capacity . . . 158
12.2 Route Guidance . . . 160
12.2.1 User optimality versus system optimality on AHS . . . 160
12.2.2 Background material on route assignment . . . 161
12.2.3 Route guidance on urban corridors . . . 164
12.2.4 Implementation of lane assignment on the urban corridor . . . 167

13 CONCLUSION 169

REFERENCES 172

APPENDIX A DATA FOR EXAMPLE IN CHAPTER 7 177

APPENDIX B PROGRAMS FOR THE TEST RUNS 179
B.1 Code for Test Run 1 179
B.2 Code Used for Test Run 2 181
B.3 Code for Test Run 3 182

APPENDIX C THE .h FILES IN THE C++ CLASS LIBRARY 186

APPENDIX D THE .h FILES IN THE FRONT END 252

VITA 261

CHAPTER 1

INTRODUCTION

The Automated Highway System (AHS) is considered an important component in the development of Intelligent Vehicle-Highway Systems (IVHS) in the United States [1], [2]. The AHS development was initiated through precursor studies [3], [4], and an AHS Consortium is currently spearheading future developments. The AHS is emerging as the first significant IVHS project in which critical operational issues will have to be resolved in order to demonstrate the capability to significantly increase highway capacity in addition to improving safety, trip planning, trip quality and beneficial effects on the environment. In addition to demonstrating the ability of available technology to control individual vehicles, the AHS must simultaneously demonstrate the ability to coordinate and manage traffic. This should be done for a realistic scenario of an AHS in the urban environment, with huge numbers of commuters traveling daily to and from their workplace to their residences, because this is where the acute need to reduce congestion exists.

The technological issues associated with the long-range development of AHS were addressed in a series of precursor studies [3], [4]. A number of Representative System Configurations (RSCs) have been defined that represent potential implementation of the AHS in different stages of development. These identify various possibilities for highway infrastructure, traffic synchronization, instrumentation distribution, operating speeds, vehicle classes, vehicle-road interaction, vehicle power, headway strategies, lateral and longitudinal control strategies, traffic control locations and check-in and checkout strategies on the AHS. How-

ever, operational plans for each RSC have not yet been developed for the realistic scenario of a multiple lane, multiple entry/exit AHS in peak flow conditions when control of traffic on the AHS will be important to achieve increased capacity with an acceptable level of safety. Not coincidentally, in none of the RSCs is an authority identified, with its activities defined, that will provide each vehicle with information to aid it, in its travel on the AHS. We believe that the lack of this important component of the AHS architecture is a major deficiency of the concepts proposed thus far.

Additionally, with three- to four-fold increases in capacity as predicted by studies such as in [5], a large number of lane change maneuvers on a multi-lane AHS is anticipated. We believe that it is crucial that a realistic assessment be made of the dynamic capacity of the AHS, i.e., of the achievable throughput in terms of the number of maneuvers that must be completed in each section of the AHS.

This thesis addresses the problem of assigning traffic to the lanes of an AHS, referred to as lane assignment. It describes the analytical formulation of the lane assignment problem. It also describes a C++ class library that was developed to enable software simulation of an AHS. The class library has been designed to be flexible and expandable through the use of object-oriented design. The class library is a tool that may be used in the future to study the effects of various control, communication and lane assignment algorithms on the dynamic capacity of the AHS.

1.1 Previous Work

Much of the research and development work to date has concentrated on the control of individual vehicles to achieve a specified velocity/distance profile, to execute specific maneuvers, to enact intelligent cruise control, or to participate in the maneuvers required for a vehicle to join, participate in, or depart from a platoon of vehicles. The implication is that the technologically difficult problem associated with driving on the AHS will be the control of individual vehicles. While there may exist novel issues encountered with the problem of driving vehicles under close distances, the problems now being considered, such as forming a platoon of vehicles on an otherwise empty road of "infinite" length, are

by no means new and have been solved in similar control applications. New challenges in scheduling, communications and control are posed by the problem of coordinating the motion of all vehicles on a multi-lane AHS so as to increase highway capacity.

Systematic studies of automated highway system capacity have been carried out by Rumsey and Powner [6] and by Shladover of the PATH program at Berkeley. Shladover's studies include capacity estimates when a *discrete platoon* car following scheme is implemented on the highway. They estimate capacity as a function of various safety criteria, but do not calculate the effect of maneuvering on highway capacity. Rao and Varaiya [7] have studied the effect of a specific car following algorithm known as *AICC* (Automated Intelligent Cruise Control) on AHS capacity. The effects of lane changing on highway capacity have been investigated by Rao, Varaiya and Eskafi [8] and by Tsao et al. [9] through the use of a simulator called SmartPath. More recently, Hall [10] uses deterministic approximations to model highway throughput analytically, accounting for both longitudinal requirements (i.e., lane flow) and lateral requirements (i.e., lane changing).

The question of traffic management and control on the AHS is still an open one and has been treated in few references [11], [12]. Rao and Varaiya [11] address the real-time control problem and have proposed a set of laws to be implemented by link layer controllers to achieve real-time control of traffic on the AHS. These link layer controllers operate primarily by using information available to the link from its own sensors and from some information communicated to each link from its neighboring upstream and from up to five downstream links.

1.2 Contributions of this Thesis

The central theme of this thesis is the lane assignment problem. We define the lane assignment problem as that of assigning to each vehicle that uses the AHS a lane of steady-state travel with the goal of increasing highway capacity as much as possible. In this thesis, we study the scheduling of traffic on the AHS by a central scheduler that assigns paths to vehicles that use the AHS based on their origin and destination and on conditions on the highway.

We believe that lane assignment (as opposed to unsupervised paths of vehicles on the AHS based on fully distributed decision-making within the vehicles) will be required to achieve the expected benefits in an urban environment in which the AHS will be characterized by many relatively closely spaced entry and exit points, with possibly 15,000 to 20,000 vehicles per hour in bottleneck sections at peak times (for an AHS with three lanes) and, therefore, with many thousands of maneuvers resulting in a comparable number of path intercepts. We see maneuvers leading to path intercepts on congested highway conditions as the main reason for delays and capacity reductions. Reducing the path intercepts will reduce turmoil on the AHS and approach the potential highway capacity implied by ideal steady-state flow conditions with specified intervehicle distances and velocities. With this in mind, lane assignment strategies that attempt to reduce path intercepts have been investigated.

1.2.1 Analytical formulation

In this thesis, the static lane assignment problem is formulated as an optimization problem. A classification of lane assignment strategies has been established, beginning from totally unconstrained strategies to general strategies, to constant lane strategies, to destination and origin monotone strategies, and ending with highly ordered monotone strategies. For each strategy we discuss the communication requirements, the effect on maneuvers and the size of that class for a particular size of highway. Lane assignment is performed and analyzed under two distinct performance measures. First, the minimization of total travel time (i.e., the time spent traveling at steady-state, plus the time spent maneuvering) of all of the vehicles on the AHS is considered.

Two different models of maneuvering cost have been investigated, corresponding to different levels of congestion the AHS. Under low levels of congestion, the lane assignment problem reduces to a linear programming (LP) problem. Under high levels of congestion, the lane assignment reduces to a quadratic programming (QP) problem. We show that under high levels of congestion, the optimal lane assignment tends to be partitioned (see Chapter 3) or nearly partitioned. Algorithms that find a partitioned strategy in the neighborhood of a nonpartitioned strategy are presented. Second, we study the balanced use of the lanes

of the AHS. The lane assignment is formulated as a constrained optimization problem with the performance criterion being the balancing of the excess capacities in the sections of the AHS. This problem reduces to a worst case maxmin problem. We have developed algorithms for obtaining the optimal or suboptimal strategy. We present and analyze the results of a case study of a hypothetical automated highway under the two performance measures stated above. The results show that at high levels of congestion, the optimal lane assignment strategies are close to being partitioned and that the developed algorithms perform efficiently in calculating the optimal or suboptimal strategies.

1.2.2 AHS simulator

The static lane assignment problem [12], [13] was studied to grasp the essential features of lane scheduling under simplifying assumptions that decouple the system-wide traffic flow problem from the dynamic effects of the motion of individual vehicles. These assumptions include coarse time discretization of the daily operating cycle (and neglect of the coupling of flows between different time instants), fixed capacities of lanes and sections of the AHS, approximate (implicit) representation of maneuvering delays in determining the optimal and suboptimal lane assignments and nonconsideration of the actual control and communication strategies used. However, the dynamic capacity of the AHS will be dependent on the actual control, communication and scheduling strategy used. Software simulation of traffic flow on the AHS is the only possible way of assessing the effect that a proposed lane assignment strategy will have on the speed, concentration and capacity of the sections of the AHS under different origin/destination distributions. Furthermore, simulation of traffic flow for a given lane assignment strategy will provide information on how to perturb the lane assignment strategy iteratively in order to better alleviate bottlenecks and increase the capacity.

For these purposes, we have developed a C++ class library that can be used to create an AHS simulator. The AHS simulator may be used to study the speed, concentration and flow characteristics of traffic flow on an AHS under various communication, control and scheduling schemes. The AHS simulator is modular and expandable to enable the insertion of new options and tasks. The requirements of flexibility and expandability have been

met through the use of object-oriented design. The library is composed of building blocks that may be combined to create a specific AHS simulator characterized by such entities as number of lanes, number of entry and exit points, distribution of vehicles at each entry point with respect to destination, lane speeds, intervehicle distances, maneuvering protocols, control system characteristics and scheduling algorithms. The various classes that have been developed may be classified as follows.

1. **Helper** classes such as *ValArray*, which is used to create dynamically resizable objects of arbitrary type; *DLinkedList*, which is used to create a doubly linked list; and *RandGen*, which is used to create a random number generator.

2. **Network** classes such as *Highway, Section* and *Lane* that are used to create, respectively, the highways, sections and lanes of an AHS.

3. **Driver/Vehicle** classes such as *CarUnit, Vehicle, Vision, CarFollower, LongControl, Platoon, LaneChanger, LatControl, PathControl* and *SegControl* that are used to create the various components of a driver/vehicle unit.

4. **Generator** classes that are used to generate the objects listed in (3) above.

5. **Simulation** classes such as *EventCal*, which is used to create the event calendar required in discrete event simulation, and *Event* classes such as *CarGEvent* and *PathCEvent* that are used to create car generation, path control and other types of events.

6. **Data Collection** classes such as *TripWire* and *StatCollector* that are used to collect statistics on the speed, flow and concentration of AHS traffic.

7. **Miscellaneous** classes such as *CarSimData*, which contains the physical vehicle and driver data.

A front end to the simulator has also been developed. It provides a safe and user friendly way of using the AHS simulator classes. In addition, it provides the functionality of an interpreter that allows the simulation to be run in an interactive fashion. The front end has been written in a language called Tcl [14], which is a simple scripting language that provides generic programming facilities and is embeddable in applications such as the AHS simulator.

A number of test simulation runs have been performed using the front end and the AHS classes in order to perform a face validation of the algorithms that have been coded in. Although no large-scale AHS simulations have been performed, the C++ classes and the front end provide the tools required to set one up, to run it and to obtain statistical information from it.

1.2.3 Dynamic capacity and route guidance

The AHS simulator opens up possibilities for the study of a host of open problems. An algorithm has been proposed to estimate the dynamic capacity of the AHS and to simultaneously obtain the optimal lane assignments for this capacity. Another algorithm has been proposed to perform route guidance on urban corridor systems. These algorithms make use of the AHS simulator to obtain realistic estimates of the lane capacity and travel times. The route guidance algorithm uses lane assignment along some or all portions of the routes to maximize capacity on those portions of the routes.

1.2.4 Issues for further study

The formulation and analysis of the lane assignment problem under the two performance criteria mentioned in Section 1.2.1 have been completed. Classification and analysis of lane assignment strategies have been done. The C++ classes for the AHS simulator and the Tcl front end have been developed. These tools will enable the future study of the dynamic capacity and the route guidance problem.

A number of issues associated with the operation on the AHS are also recognized, but not treated here. These issues include maneuvering around long-term and short-term obstacles in particular lanes, and operating under capacity reductions in certain sections of the highway, as well as the interaction of lane assignment strategy with other components of the overall control structure. In particular, this includes (1) the role of ramp metering [15], [16], and (2) the effect of reference paths implied by the lane assignment strategy on the AVCS (Automated Vehicle Control System) in the vehicles. The first topic addresses the interface between the AHS and the surrounding traffic system, and the second addresses the

interface between lane assignment and the functions of the AVCS in the vehicles [17], [18]. While these are all challenging issues in their own right, in our view, they can be properly considered subsequent to the assignment of vehicles to lanes in sections of the AHS.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 discusses the need for lane assignment, describes the system architecture required to implement lane assignment and presents the network model used for describing traffic flow on the AHS. Chapter 3 describes a classification of lane assignment strategies and calculates the cardinality of each set of strategies. Chapter 4 describes lane assignment done on the AHS so as to minimize the total travel time of all the vehicles on the AHS. Chapter 5 describes lane assignment done on the AHS that balances the traffic load across the lanes of the AHS. It presents an algorithm that efficiently calculates the optimal or suboptimal lane assignment. Chapter 6 describes additional properties of a particularly interesting lane assignment strategy, known as a partitioned strategy. It presents algorithms to obtain suboptimal partitioned strategies that are "close" to optimal nonpartitioned ones. Chapter 7 describes a case study of an example lane assignment problem. Results are presented for both the travel time and the load balancing criteria. Chapter 8 describes the need for and design of an AHS simulator. A survey of existing simulators is presented. The object-oriented design of the simulator through the use of C++ classes is presented. Chapter 9 describes the various vehicle control algorithms that have been implemented in the AHS simulator. These include car following, longitudinal, lane changing, lateral, path and segment control algorithms. Chapter 10 describes the front end to the simulator. It discusses the C++ classes that were linked with the Tcl language to create the front end. Chapter 11 presents some test runs of the simulator and the front end. Chapter 12 discusses the use of the simulator in the estimation of dynamic capacity and in route guidance on urban corridors.

Finally, the thesis concludes with appendices that augment understanding of some of the above chapters. Appendix A contains the data used for the case study in Chapter 7. Appendix B contains the programs used for the test runs in Chapter 11. Appendix C

contains the .h files used in the design of the simulator C++ classes. Appendix D contains the .h files used in the design of the front end to the simulator C++ classes.

CHAPTER 2

LANE ASSIGNMENT ON THE AHS

2.1 The Need for Lane Assignment

Most of the current studies of control issues associated with the operation of the AHS stem from the tenet that the control of individual vehicles is the deciding issue in controlling the flow of traffic on the AHS. Accordingly, the completed precursor studies have focused on many diverse issues related to the operation of AHS. Some of these studies are related directly to issues of control and communication requirements, but hardly any encompass scheduling issues. Our view is that the problem of routing vehicles and coordinating the massive number of maneuvers that accompany the traffic flow on the urban AHS, where congestion problems exist and must be resolved, will be as critical to successful operation as the above studies, if not more so [12], [19], [20]. Indeed, the coordination of scheduling and control warrants early study because the information collected on the AHS can help increase its capacity by proper scheduling, even in the manual or low-automation phase of development. Moreover, operating the AHS without having some authority route traffic by scheduling paths of vehicles would mean that the on-board controls would operate on purely local information. This would defeat the purpose of coordinating the maneuvers by using system-wide information. And if paths will have to be scheduled, the scheduling should try to maximize throughput by taking into account the origins and destinations of all the

vehicles as well as the dynamic effect of maneuvers implied by the proposed path-scheduling strategy on time of travel and thus on highway capacity.

To put the issue in perspective, consider the relationship [21] developed to assess the effects of platooning on capacity that relates the capacity C in veh/lane/hour to the steady-state speed v in km/hour, the interplatoon spacing D in m, the intraplatoon spacing d in m, the vehicle length s in m and the number of vehicles in a platoon n.

$$C_{ideal} = v\frac{n}{ns + (n-1)d + D}1000 \text{ veh/lane/hour}$$

As noted in [21] for $s = 5$ m, $v = 72$ km/hour, $n = 1$ and $D = 30$ m, one obtains approximately the current traffic conditions (no platoons) and a capacity of $C = 2100$ veh/lane/hour which is in the range of observed maximal capacities C_{max} on freeways. However, the observed distance D between individual vehicles under high capacity conditions is much smaller than the 30 m used above. Using a more typical value of D in the above equation of 7 m, $C_{ideal} = 6000$ veh/lane/hour is obtained. Nothing close to this value has ever been observed in existing freeways. Therefore, we must conclude that the above relationship defines the ideal capacity in conditions of unobstructed longitudinal flow.

The observed maximum capacities are reduced because of the direct and indirect effects of obstructions in the longitudinal flow due to essential maneuvers (taken by vehicles maneuvering towards their exits) and unessential maneuvers (taken by drivers in response to local observation and desire to overtake slower vehicles). The ratio $\eta = C_{max}/C_{ideal} \approx 1/3$ indicates that maneuvering drastically reduces the achievable capacity and also restricts the achievable capacity on the AHS in conditions characterized by a large number of maneuvers. However, the negative effect of maneuvering will be mitigated on the AHS by the positive effect of automated driving, with improved reactions of automated vehicles over human drivers, and more importantly, by the positive effects of lane scheduling that will reduce the number of maneuvers on the AHS.

We believe that lane assignment (as opposed to unsupervised paths of vehicles on the AHS based on fully distributed decision-making within the vehicles) will be required to achieve the expected benefits in an urban environment in which the AHS will be characterized by many relatively closely spaced entry and exit points, with possibly 15,000 to 20,000 vehicles

per hour in bottleneck sections at peak times (for an AHS with three lanes) and, therefore, with many thousands of maneuvers, resulting in a comparable number of path intercepts. We see maneuvers leading to path intercepts on congested highway conditions as the main reason for delays and capacity reductions. Reducing the path intercepts will reduce turmoil on the AHS and approach the potential highway capacity implied by ideal steady-state flow conditions with specified intervehicle distances and speeds.

2.2 Organizational Structure Assumed for the AHS

The system architecture required to support lane assignment on the AHS would consist of an ATMS (Advanced Traffic Management System), which is a system wide traffic flow monitoring and scheduling system; an Automated Highway Control Center (AHCC), which determines the lane assignment strategy and assigns a lane to every vehicle using the automated highway; a supervisory discrete event controller that manages vehicle maneuvers; and vehicle on-board longitudinal and lateral control systems. We assume the presence of vehicle-to-roadside beacon communication (or some technically feasible alternative, such as antenna towers) in intermediate phases of development, as well as vehicle-to-vehicle communication in the high-end state of development. This architecture is fully consistent with overall system architectures envisaged for IVHS [1], [21], [22], [23]. The lane assignment problem is not restricted to this system architecture, but it will serve conveniently for the stated purpose.

The AHCC is envisaged as a data processing node. It controls the scheduling of operations on the controlled access highway and is not necessarily an independent physical facility. The two-way information link between the ATMS and the AHCC governs the coordinated use of the automated highway corridors within the traffic system. Based on the demand levels transmitted to it, the AHCC will determine the appropriate lane assignments. These are sent to the roadside beacons, which in turn will broadcast the relevant information to the vehicles. The roadside beacons will then collect the origin/destination data of the current system users and transmit these to the AHCC so that modifications in the assignments can be made based on real-time system data. In addition, the AHCC will update estimates on

the current distribution of the load on the system and travel times based on this information and will transmit these to the ATMS to aid the assessment of system-wide traffic conditions. A typical vehicle would enter the automated lanes from an entry point, which may be from the feeder roads or from another highway, receive a reference trajectory based on the lane assignment strategy, proceed gradually to the lane assigned to it and possibly change lanes during the trip as the lane assignment dictates. Then, as it approaches its destination, it gradually maneuvers back to the slowest lane and ultimately exits the automated corridor.

It is envisaged that lane assignment can be introduced in an evolutionary fashion at various levels of automation at which it would provide information to drivers based on system-wide conditions. In the existing system, it would first fulfill an **advisory** role. This means that information would be broadcast to the drivers indicating the route that the traffic management system advises them to take. However, the drivers would be under no obligation to follow this advice. Then, data could be collected to determine the average level of compliance by the drivers, as well as to determine the effect of lane assignment on traffic conditions. Collected origin/destination data would also update the best estimates of origin/destination distributions during different time periods of the daily cycle.

As the phase of semi-automated driving is entered, a **compliant** phase of lane assignments would be introduced, in which the vehicle operator would be asked to comply with received instructions. Each vehicle is still assumed to remain under the control of the human operator who will decide when to activate and deactivate intelligent cruise control (automatic longitudinal control). However, since maneuvering time would remain driver initiated and thus stochastic, and in view of diverse driver instincts and habits, data would be collected both to assess the actual level of compliance and the traffic conditions resulting from the lane assignment. To increase the level of compliance, data collected by the AVCS would be periodically examined to determine the level of congruence between the actual trip trajectories and the prescribed trajectories. Enforcement plans would be used to induce drivers to adhere to the lane assignment strategies. These would involve the assessment of fines or the revocation of the right to use the highway in case of significant noncompliance.

Finally, as technological advances enable the transition to fully automated highways, lane assignment would become **mandatory**, with the AVCS taking over both the longitudinal

and lateral control of the vehicle trajectory. Here, two-way communications with requests processed by mobile local area communication networks (much as messages are processed in present computer networks) would schedule and initiate maneuvers to be carried out by the cooperative action of the AVCS in all of the vehicles affected by each maneuver. While the evolutionary introduction of automated driving appears to be the most realistic scenario, the system architecture will also support a controlled access AHS in which fully automated driving will be the sole mode of vehicle control from the beginning of its operation.

The challenging problem is to achieve this on an AHS serving an urban community in which the AHS will be characterized by many entry and exit points separated by relatively short distances, requiring many maneuvers per unit time and unit length of the AHS. This, however, is where the severest congestion problems exist, and it is within this environment that the congestion problem must be resolved if improved mobility is to be achieved together with other important goals including safety, trip planning and trip quality.

There are two main functions expected of lane assignment within the system architecture on the AHS. First, it will employ the system-wide information available to the AHCC and thereby improve system operation by assigning lanes to vehicles depending on system conditions and vehicle origin/destination patterns. Second, lane assignment will provide the individual vehicle controllers with guidance as to the path and maneuvers that they must execute during the portion of a trip on the AHS. The benefits of lane assignment may be separated into two categories. The first is simply the benefits attained by using system-wide information available on the AHS for which origin/destination data can be easily collected and compliance with assigned lanes can be monitored and enforced much more readily than on other highways. The second is the additional benefit that can be attained when looking for the strategy that optimizes highway capacity by minimizing some judiciously selected measure of system performance.

Three operating regimes should be distinguished on the AHS, i.e., normal operation, scheduled maintenance operation and emergency operation. Normal operations and scheduled maintenance operations differ only in the number of available lanes, section capacity and lane speeds, and thus may be considered within the framework developed here for the lane assignment problem. Emergency operation due to an incident or accident differs sig-

nificantly and is not considered here. Emphasis in this mode of operation is much more on real-time control than on the advance scheduling that is tackled by lane assignment. It will however provide data based on which rescheduling will be performed.

2.3 Modeling Traffic on an AHS

Macroscopic speed/flow/concentration relationships have been used in freeway traffic studies for over half a century to predict highway capacity and travel times. Speed/flow/concentration relationships have been developed based on observations taken on specific freeways, quantifying to some degree the relationship between traffic flow and speed in free-flow and jam conditions. These are in general agreement with observed speed/flow distributions and the maximum observed highway throughput (i.e., highway capacity) of about 2200 veh/lane/hour on existing freeways. However, a number of recent efforts have highlighted the inadequacy of simple analytical descriptions of this complex relationship [24], [25], [26]. The difficulties inherent in the modeling of these relationships are due to two distinct factors.

The first is that while the underlying system being modeled [27] contains two physical principles (flow = speed*concentration, continuity of the number of vehicles), these are complemented by the multitude of inherently subjective relationships between speed and concentration that individual drivers establish based on individual driving styles. Transition to automated driving, and even to driving on an AHS with mixed traffic (containing both non-automated and automated vehicles) with prescribed lane assignments, lane velocities and intervehicle distances will significantly affect the macroscopic flow parameters and, therefore, warrants a study of the speed/flow/concentration characteristics of driving on the AHS. The second factor is that the relationships in question take on quite different forms in conditions of free-flow downstream of a bottleneck, in conditions of stop-and-go traffic, and in conditions in which vehicles weave between lanes [26], [28], [29]. The diverse operating conditions have not been, and most probably cannot be, captured by a simple two-dimensional or three-dimensional relationship. For the above reasons, rather than macro-modeling the traffic on the AHS by a speed/flow/concentration relationship that is currently unknown, we will use a micro-model of the traffic in which each vehicle is modeled.

2.3.1 Network model

We will consider a network model of an AHS that is static in time and is, therefore, defined by time invariant flows from each entry to each exit of the AHS. We now outline the assumptions made in this network model. We model an AHS with three lanes, but the model can easily be extended to more lanes. The need for lane assignment arises because an operational AHS must have more than one lane. A single lane will not characterize an operational AHS (but may be used on demonstration problems) because the AHS would have to close when maintenance or incidents occur. We have chosen to consider three lanes, because the lane assignment problem will still be required even when one lane is closed.

The following lists summarize the notation that will be used throughout the remainder of this thesis.

(K_1, K_2) system	A segment of automated highway with K_1 entrances and K_2 exits
$N_{i,j}$	Number of vehicles per unit time that enter node i and will be exiting at j
N_i	Number of vehicles per unit time that enter node i
$\rho_{i,j}^m$	Number of vehicles per unit time traveling from node i to j in lane m
$l_{i,j}$	Distance between nodes i and j
L_i	Lane i
s_i	Section of highway between nodes i and $i+1$.
v_i	Reference speed for L_i
z_i	Capacity of L_i (vehicles/time)
$\gamma_{i,j}^m$	Maneuver cost for moving to lane m while traveling from node i to j

The following are the maneuver cost weighting constants for the costs incurred.

C_1^a	On entering the highway (at lane 1)
C_2^a	While maneuvering from L_1 to L_2 after entering
C_3^a	While maneuvering from L_2 to L_3 after entering
C_3^b	While maneuvering from L_3 to L_2 before exiting
C_2^b	While maneuvering from L_2 to L_1 before exiting
C_1^b	While exiting the highway (from L_1)

This is the remainder of the notation.

q	Greatest common factor of C_m^a, C_m^b
Q_1 and Q_2	Matrices used in determining total travel time
P_1 and P_2	Destination monotone partitions
R_1 and R_2	Origin monotone partitions
$N_b(K)$	Number of general assignment strategies for K sections
$N_c(K)$	Number of constant lane assignment strategies for K sections
$N_d(K)$	Number of destination monotone assignment strategies for K sections
$N_f(K)$	Number of monotone assignment strategies for K sections
A_r	Matrix used in determining destination monotone costs
A_c	Matrix used in determining origin monotone costs
A_1	Matrix in the LHS of the conservation constraints
b_1	Vector in the RHS of the conservation constraints
A_2	Matrix in the LHS of the capacity constraints
b_2	Vector in the RHS of the capacity constraints
$e_m(k)$	Excess capacity in section k and lane m
$J(P_1, P_2)$	Minimal excess capacity obtained with partitions P_1 and P_2
J_{ideal}	Largest possible minimal excess capacity

Consider the following origin/destination (OD) matrix.

$$N = \begin{bmatrix} 0 & N_{0,1} & N_{0,2} & N_{0,3} & \dots & N_{0,K} \\ 0 & 0 & N_{1,2} & N_{1,3} & \dots & N_{1,K} \\ 0 & 0 & 0 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & N_{K-1,K} \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{2.1}$$

where $N_{i,j}$ are the best estimates obtained from historical data. The traffic flow on the AHS is composed of vehicles $N_{i,j}$ that enter the AHS at various entrances i and leave it at various exits j (see Figure 2.1). This traffic will be assigned to various lanes m as $\rho_{i,j}^m$. This means that, for an AHS with K_1 entrances and K_2 exits, the following conservation constraint must

hold.

$$\sum_{m=1}^{3} \rho_{i,j}^{m} = N_{i,j} \qquad 1 \leq i \leq K_1,\ 1 \leq j \leq K_2 \tag{2.2}$$

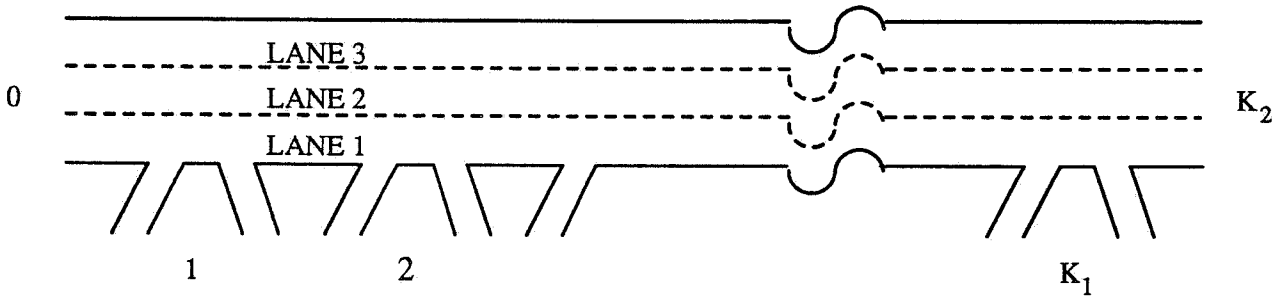


Figure 2.1: AHS with K_1 entrances and K_2 exits

Each section s_i of the highway is characterized by a length l_i. The lanes L_m of the AHS have a capacity z_m, which means that the largest total flow that can be assigned to L_m is z_m. This means that the following capacity constraint must hold.

$$\sum_{l=0}^{i} \sum_{s=i+1}^{K_2} \rho_{l,s}^{m} \leq z_m \qquad m = 1,2,3,\ 1 \leq i \leq K_1 \tag{2.3}$$

The relationship between speed and intervehicle distances realizable on the AHS will directly influence highway capacity. Its effect on lane assignment would be through the reference lane speeds that may be dependent on flow. It is crucial that a more realistic assessment be made of the **dynamic capacity** of the AHS and to model accurately the effects of vehicle dynamics, maneuvering protocols and communication delays on the traffic flow. Some work has been done in this area (see [8] for the effect of various entry and egress maneuvering strategies on traffic flow and [7] for the effect of partial traffic automation on flow), but in the absence of a sufficiently broad-based model, we have chosen not to take it into account at the present time.

A reference speed v_m is assigned to each lane L_m that vehicles traveling on it attempt to maintain. Vehicles traveling from i to j that travel in L_m will spend $\gamma_{i,j}^{m}$ time maneuvering into and out of L_m. The total capacity utilized in a lane in a section is taken to be the sum of the flows assigned to that lane in that section. The outermost lane L_1 provides a buffer zone between fast-moving traffic in the inner two lanes and slower-moving traffic entering

the AHS and leaving the AHS via the on and off ramps. Vehicles exit the AHS and enter lower speed zones via off ramps from the outermost lane. For this reason, the reference speed v_1 of the outermost lane L_1 is less than that of the inner two lanes so that $v_1 < v_2, v_3$. For generality, we assume that the reference speed increases as we move inwards in terms of lanes so that $v_1 < v_2 < v_3$.

The on and off ramps connected to the outermost lane L_1 provide access into and out of the AHS. The length of highway between an on ramp and the immediately downstream off ramp forms a **section** of the AHS. The flow of traffic onto and off the highway is described by an origin destination matrix N. It is limited by the total dynamic capacity of the the three lanes. This capacity will be a function of the origin destination matrix N, the number of maneuvers that take place and of the AHS operating strategies. For the purposes of calculating the lane assignment strategy, however, this capacity is assumed to be fixed and is taken to be a number that is determined either through simulation or from historical data.

Additionally, we assume that no incidents occur on the highway and that all traffic entering the AHS does eventually leave the AHS.

2.4 The Statement of the Problem

Given this scenario of an AHS, we are now in a position to state the problem of lane assignment. The lane assignment problem will be viewed as a constrained optimization problem over a strategy space and is fully defined by:

(1) An origin/destination matrix $N = [N_{ij}]$, where N_{ij} is the flow of vehicles from entry point i and traveling to the exit j, in the considered time interval. (Depending on the length of sections and the time discretization used in particular instances, it may be necessary to apply time shifting of forecast origin/destination data to take into account travel along sections of the highway at the nominal speeds prescribed for each lane, and this will be assumed).

(2) Physical constraints imposed by the system structure, which include:

(a) capacity constraints for each lane in each section of the highway; it will be assumed here that capacity constraints are uniform throughout a section, although capacity reductions due to vehicles remaining on the highway from a previous time period, due to maintenance, or due to accidents, represent obvious reasons for the subsequent study of nonuniform capacities,

(b) conservation constraints, which guarantee that all of the N_{ij} vehicles traveling from entrance i to exit j will be assigned one of the three lanes, and

(c) non-negativity constraints, to obtain realistic solutions.

(3) Admissible class of strategies A classification of lane assignment strategies has been provided in Chapter 3. The admissible class of strategies is simply the constraint that defines the lane assignment strategy required, namely, constant lane assignment strategies or partitioned strategies (destination monotone, origin monotone or monotone), which are the two classes of strategies that we are interested in.

(4) Performance criterion used to extract optimal, or suboptimal strategy from the admissible class. A number of candidate performance criteria may be considered, such as the balanced use of lanes, minimization of the number of path intersects and, in particular, the minimization of the total travel time of all vehicles. For constant lane strategies, the criterion will have the general form $J = J(L(i,j), N)$. For partitioned strategies, the value of the criterion will depend on the partitions $\{P_1, P_2\}$ and reduces to the form $J = J(P_1, P_2, N)$.

The lane assignment problem can then be formulated as follows: Given a system defined by the origin/destination matrix in (1) subject to physical constraints defined in (2) for a selected performance criterion defined in (4), determine the optimal or suboptimal lane assignment strategy from the admissible class defined by (3).

In later chapters, we will formulate the lane assignment problem in a static fashion. This is done with the understanding that when the parameters affecting lane assignment (primarily the origin/destination matrix) change, the assignment must be recalculated. If we assume that the flows as provided by the origin/destination matrix provide a reasonably

accurate estimate of the actual flows into and out of the AHS for, say, the next hour or half an hour, then the assignment must be redone every hour with updated OD matrices. This has been done in the case study discussed in Chapter 7.

Performing the lane assignment for each time period (say, 1 hour) will be accompanied by time boundary effects. Assume that lanes are assigned to incoming traffic at the start of every hour. The lane assignments could change significantly from one time period to the next. The assumption made in this thesis is that the new assignments would affect only vehicles entering the highway and not vehicles already on the highway. The boundary issue may be treated separately within sensitivity studies.

CHAPTER 3

LANE ASSIGNMENT STRATEGIES

In this chapter, we present a classification of lane assignment strategies and calculate the cardinality of each set of strategies for an AHS with K entrances and exits.

3.1 Nonpartitioned Strategies

(1) **Unrestricted policy, with totally unconstrained behavior:** This typifies the present state of operation, in which there is practically no specific law or rule that requires that vehicles remain in one lane, even within a section of the highway between two entry/exit points. While the weaving of certain vehicles through traffic may have some redeeming features on existing highways due to the wide diversity of driver reactions and driving habits, it cannot be easily justified when vehicles are under automatic control.

(Present operational policies recognize the utility of lane assignment strategies by designing separate local and express lanes, and designating the express lane for vehicles traveling to distant exit points by not providing access from express lanes to intermediate exits.)

(2) **General lane assignment policy:** Here the lane assignment strategy takes the form

$$L = f(i,j,k) \quad i,j,k = 1,\ldots,K,\ L = L_1, L_2, L_3$$

where the lane assigned to the vehicle is a function of the origin i, the destination j and the section k of the highway that the vehicle is currently traversing.

This is the least constrained type of strategy but its implementation would demand either that vehicles obtain the entire path from the AHCC at entry, or maintain contact with the AHCC at every entry/exit point via roadside beacons to receive new instructions. Moreover, a vehicle would have to carry out many maneuvers to change lanes from section to section.

(3) **Constant lane policy:** Here the lane assignment strategy is independent of the section that the vehicle is traversing and takes the form

$$L = f(i,j) \quad i,j = 1,\ldots,K,\ L = L_1, L_2, L_3$$

Thus, each vehicle travels in the same lane save for maneuvers to get to the lane, and to exit at its destination. This type of lane assignment strategy minimizes the information exchange between the roadside beacon and the vehicle. The point of origin is known to the beacon, the destination is transmitted from the vehicle to the beacon and, assuming a lane assignment strategy has been determined, the beacon simply transmits to the vehicles the lane it should use while on the automated corridor. Furthermore, constant lane strategies clearly reduce the number of maneuvers on the AHS and thus increase highway capacity. Work done by Hall [10] also underscores the importance of reducing lane change maneuvers in order to maximize highway capacity.

3.2 Partitioned Strategies

Restricting attention to constant lane strategies, we may wish to determine subclasses of strategies that further reduce path intercepts, if not the required maneuvers per vehicle. Partitioned lane assignment strategies are subclasses of constant lane strategies that satisfy certain additional restrictions. The rationale for partitioned lane assignment strategies is that they are conducive to establishing a quasi steady-state flow by reducing as much as possible the number of maneuvers and path intercepts, while utilizing all three lanes and satisfying capacity constraints. In the self-organizing architecture on the AHS, in which the enactment of each maneuver is preceded by intervehicle communications to coordinate the motion of all affected vehicles, partitioned strategies should reduce the communication and

control overhead associated with coordination of vehicle motion, and reduce the number of maneuvers required for vehicles to enter the traffic flow on the AHS and exit at their destinations. Three categories of partitioned lane assignment strategies have been defined that exploit the origin/destination characteristics of the traffic on the AHS.

(4) **Destination monotone policy:** Here the lane assignment strategy assigns constant lanes to vehicles with the additional constraint that given two vehicles entering at the same entry point, the vehicle traveling to a farther destination will be assigned the same lane or a faster lane than a vehicle traveling to a nearer destination. That is, given

$$L_a = f(i_a, j_a)$$

$$L_b = f(i_a, j_b)$$

and $j_a \leq j_b$, then $L_a \leq L_b$ (where L can take on values $\{1, 2, 3\}$ with lane numbers as defined previously).

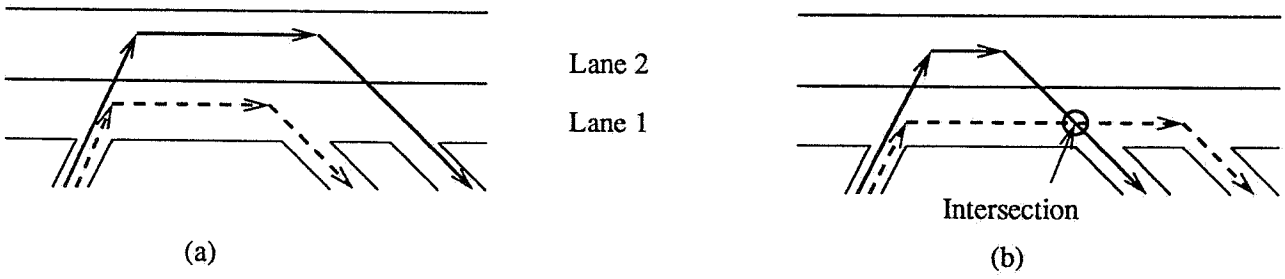


Figure 3.1: Demonstration of path intercepts

As illustrated in Figure 3.1(a), this lane assignment strategy will eliminate unnecessary communications, negotiations and excess maneuvers that would otherwise occur due to path crossings if a vehicle traveling to a closer destination were assigned a faster lane (although due to many entry points, all path crossings cannot be eliminated).

(5) **Origin monotone policy:** Here the lane assignment strategy assigns constant lanes to vehicles with the constraint that given two vehicles exiting at the same exit point, the vehicle coming from a farther upstream origin (entry point) will be assigned the

same lane or a faster lane than one assigned to a vehicle coming from a nearer point of origin. That is, given

$$L_a = f(i_a, j_a)$$

$$L_b = f(i_b, j_a)$$

if $i_a \leq i_b$, then $L_a \geq L_b$.

This lane assignment strategy will eliminate unnecessary communications, negotiations and maneuvering that would occur due to path crossings if a vehicle traveling from a closer point of origin were assigned to a faster lane.

(6) **Monotone policies:** These are partitioned lane assignment strategies that are both destination monotone and origin-monotone. That is, given

$$L_a = f(i_a, j_a)$$

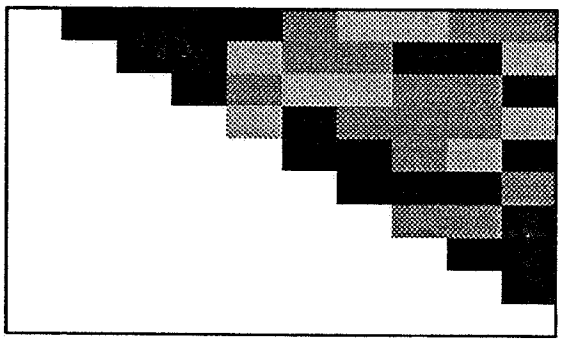
$$L_b = f(i_b, j_b)$$

then (a) If $i_a = i_b$ and $j_a \leq j_b$, then $L_a \leq L_b$ and (b) if $j_a = j_b$ and $i_a \leq i_b$, then $L_a \geq L_b$.

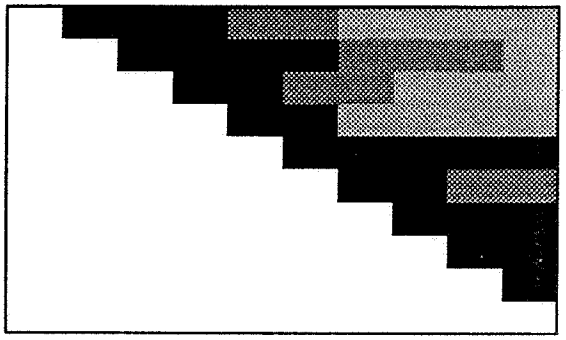
This appears to be the most desired class of strategies in terms of the promise of reducing path intercepts. It is also the most constrained and, hence, may result in a more significant reduction in attainable performance as measured by the selected performance criterion, which could be a function of additional variables, such as travel time and flow.

Utilizing a 20x20 entry/exit origin/destination matrix N, and shading progressively lighter origin/destination pairs allocated the use of lanes L_1, L_2 and L_3, respectively, a non-partitioned constant lane assignment strategy is illustrated in Figure 3.2(a), a destination monotone strategy is illustrated in Figure 3.2(b), an origin monotone strategy is illustrated in Figure 3.2(c) and a monotone strategy is illustrated in Figure 3.2(d).

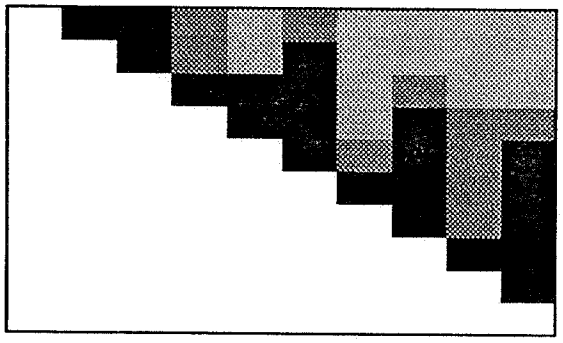
The vertical edges separating the different shades in Figure 3.2(b) represent, for each entrance i, destination monotone partitions $P_1(i)$ and $P_2(i)$. The horizontal edges in Fig 3.2(c) represent origin monotone partitions $R_1(j)$ and $R_2(j)$. Thus, a destination monotone strategy is characterized by a pair of partitions $\{P_1(i), P_2(i) : i = 1, 2, \ldots, K\}$, mapping



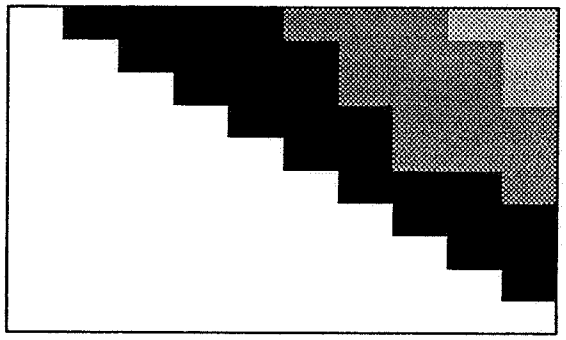
(a) Nonpartitioned strategy



(b) Destination monotone strategy



(c) Origin monotone strategy



(d) Monotone strategy

Figure 3.2: Lane assignment strategies

entry points i into lane usage boundaries $P_1(i)$ and $P_2(i)$. Origin monotone strategies must be characterized via mappings of exit points j into positions of the the horizontal portions of the partitions, and will be not discussed in this thesis in detail. (The fundamental reason for this difference in the treatment of various partitioned strategies stems from the nature of the involved mappings. For monotone strategies, the mapping of the entry points i into $P_1(i)$, $P_2(i)$ is a monotone function, with a unique inverse map that maps the exit points j into a monotone inverse mapping $R_1(j)$ and $R_2(j)$. For destination monotone strategies, the mapping of i into $P_1(i)$, $P_2(i)$ remains a well-defined function, but it has no inverse. For origin monotone strategies, the mapping of j into $R_1(j)$, $R_2(j)$ is a well-defined mapping, but it has no inverse. Consequently, P_1, P_2 cannot be used to uniquely characterize origin monotone partitions.)

Thus, all destination monotone strategies can simply be characterized by the two partitions P_1 and P_2 that separate the origin/destination matrix into three regions characterized by lane usage: P_1 separates exits using L_1 from those using L_2, while P_2 separates exit pairs using L_2 from those using L_3. If $P_1(i) = P_2(i)$ for some i, then no vehicle entering at i is assigned L_2, and if $R_1 1(j) = R_2(j)$ for some j, no vehicle exiting at j is assigned L_2. Destination monotone strategies that allow vehicles going to P_1 or P_2 to be split into two lanes, or origin monotone strategies that allow coming vehicles from R_1 and R_2 to be split into two lanes, will be referred to as *partitioned strategies with splitting.* Those that constrain these vehicles to travel in one lane will be referred to as *partitioned strategies without splitting.* For an m lane highway, $[P_1, P_2, \ldots, P_m]$ and $[R_1, R_2, \ldots, R_m]$ define destination and origin monotone strategies as follows. For entry i and exit j, Equation (3.1) defines destination monotone strategies without splitting, Equation (3.2) defines destination monotone strategies with splitting, Equation (3.3) defines origin monotone strategies without splitting and Equation (3.4) defines origin monotone strategies with splitting.

$$j \leq P_1(i) \Rightarrow L_1$$

$$P_1(i) < j \leq P_2(i) \Rightarrow L_2$$

$$P_2(i) < j \leq P_3(i) \Rightarrow L_3$$

$$\ldots$$

$$P_{m-1}(i) < j \Rightarrow L_m \tag{3.1}$$

$$\begin{aligned} j \leq P_1(i) &\Rightarrow L_1 \\ P_1(i) \leq j \leq P_2(i) &\Rightarrow L_2 \\ P_2(i) \leq j \leq P_3(i) &\Rightarrow L_3 \\ &\cdots \\ P_{m-1}(i) \leq j &\Rightarrow L_m \end{aligned} \tag{3.2}$$

$$\begin{aligned} i \geq R_1(j) &\Rightarrow L_1 \\ R_1(j) > i \geq R_2(j) &\Rightarrow L_2 \\ R_2(j) > i \geq R_3(j) &\Rightarrow L_3 \\ &\cdots \\ R_{m-1}(j) > i &\Rightarrow L_m \end{aligned} \tag{3.3}$$

$$\begin{aligned} i \geq R_1(j) &\Rightarrow L_1 \\ R_1(j) \geq i \geq R_2(j) &\Rightarrow L_2 \\ R_2(j) \geq i \geq R_3(j) &\Rightarrow L_3 \\ &\cdots \\ R_{m-1}(j) \geq i &\Rightarrow L_m \end{aligned} \tag{3.4}$$

3.3 Size of the Classes Without Splitting of Flows

The computational burden involved in solving the lane assignment problem within the defined classes of lane assignment strategies may be grasped by computing the total number of strategies of each defined category for a (K, K) system.

3.3.1 Totally unconstrained strategies

The assumption here is that a vehicle can change lanes as many times as desired in a section of an AHS, as well as many times within any section. The number of possible strategies is therefore virtually infinite, even for one vehicle.

3.3.2 General lane assignment strategies

The assumption here is that every vehicle may be asked to change lanes in every section of the highway to improve highway conditions, but that all vehicles traveling from a given entry point to a given exit will be assigned the same lane in every section of the freeway. (If the last assumption is not true, then splitting of traffic flows occurs.) To compute the number N_b of possible lane assignment strategies, note that there are exactly $K-1$ pairs for which the vehicles will travel only one section, $K-2$ pairs for which vehicles will travel two sections, and so on, to the single pair that will travel through all $K-1$ sections. To each pair, it is possible to assign independently in every section of the associated trips any of the three lanes L_1, L_2 or L_3. Thus, the total number of different general strategies is

$$N_b(K) = (3^1)^{K-1}(3^2)^{K_2}(3^3)^{K-3}\ldots(3^{K-2})^2(3^{K_1})^1 = 3^{\sum_{i=1}^{K-1} i(K-i)} = 3^{K(K^2-1)/6} \qquad (3.5)$$

3.3.3 Constant lane strategies

With constant lane assignment strategies, a lane is assigned to each entry/exit pair. Since, for each entry/exit pair there are three possible lanes, and since there are $K(K-1)/2$ different entry pairs, the total number of constant lane strategies is

$$N_c(K) = 3^{K(K-1)/2}$$

It is pointed out that $N_b(K) = N_c(K)N_b(K-1)$.

3.3.4 Destination monotone strategies

The only constraint here is that for any i

$$i \leq P_1(i) \leq P_2(i) \leq K+1$$

The total number of strategies is the total number of ways that we can choose the partitions $P_1(i)$ and $P_2(i)$. This is equal to the number of ways to pick two distinct numbers from the set $\{i, i+1, \ldots, K+1\}$. For any entrance i, the total number of ways to pick two distinct numbers is

$$\bar{S}(i) = \begin{pmatrix} (((K+1)-i)+1) \\ 2 \end{pmatrix} = \frac{(K-i+2)(K-i+1)}{2} \tag{3.6}$$

Therefore, the total number of combinations for all the exits is

$$N_d(K) = \prod_{i=1}^{K} S(i) \quad = 2^{-K}(K+1)!K! \tag{3.7}$$

3.3.5 Origin monotone strategies

The number of these strategies is the same as $N_d(K)$.

$$N_e(K) = 2^{-K}(K+1)!K!$$

3.3.6 Monotone strategies

In this case, we have not been able to find a closed-form value to the number of strategies. However, a recursive relationship can be provided that can be used to compute the actual number of strategies for every K.

To develop the recursive relationship, consider a possible pair of partitions $P_1(i), P_2(i), i = 1, \ldots, K$. Clearly, it is possible for simplicity of exposition to reverse the order of counting and consider the row and column indices i and j to increase in the upward direction and

towards the left, respectively. Consider the ith row and view it as a row through which partitions P_1 and P_2 must pass as they are extended towards row K. Furthermore, in row i, the partition $P_1(i)$ can end at any one of the locations $P_1(i) = p1, 1 \leq p1 \leq i$, while $P_2(i)$ can end at any of the locations $P_2(i) = p2, 1 \leq p2 \leq p1$, where $p1$ and $p2$ must satisfy the monotonicity constraint. First consider the number of ways to arrive at some pair $\{p1, p2\}$ in row i. Let $S(i, p1, p2)$ be this number (i.e., the number of partitions extended from row 1 to row i that satisfy $P_1(i) = p1, P_2(i) = p2$). Similarly, let $S(i-1, s1, s2)$ be the number of ways that the partitions P_1 and P_2 can pass through the point $P_1(i-1) = s1$, $P_2(i-1) = s2$ in row $i-1$.

Now consider the determination of $S(i, p1, p2)$ from the collections of all different partitions $S(i-1, s1, s2)$, $1 \leq s1 \leq s2 \leq i-1$. Because the strategies are monotone and must satisfy the additional monotonicity constraints, for a given $p1$ (where $p1$ can have any value from 1 to $i-1$), and given $p2$ (where $p2$ can have any value from 1 to $p1$), there are exactly

$$S(i, p1, p2) = \sum_{s1=1}^{p1} \sum_{s2=1}^{min\{s1,p2\}} S(i-1, s1, s2) \quad 1 \leq p1 \leq i-1 \quad 1 \leq p2 \leq p1 \tag{3.8}$$

ways to pass through the point $\{p1, p2\}$ in row i, because we can come to the position $\{p1, p2\}$ only from positions $s1$ such that $s1 \leq p1$, and from positions $s2$ such that $s2 \leq min\{s1, p2\}$.

To compute the total number of possible end points of trajectories at row i, it is necessary to add to the above all pairs ending at positions $\{i, p2\}$, $p2 = 1, \ldots, i$. It is easy to see that

$$S(i, i, p2) = S(i, i-1, p2) \quad 1 \leq p2 \leq i-1 \tag{3.9}$$

with $S(i, i-1, p2)$ computed from (3.8). Finally, it can also be seen that

$$S(i, i, i) = S(i, i-1, i-1) \tag{3.10}$$

This implies from Equation (3.8) that

$$S(i, i, i) = \sum_{p1=1}^{i-1} \sum_{p2=1}^{p1} S(i-1, p1, p2) = N_f(i-1) \tag{3.11}$$

and so $S(i, i, i)$ is equal to the total number of strategies for a system with $i-1$ rows and columns. Thus, starting with $i = 1$ and $S(1, 1, 1) = 1$, recursively compute $S(i, p1, p2)$ using

first (3.8) and then (3.9) and (3.10) for each i, incrementing i and repeating the procedure until $i = K$ is reached. At this point, the total number of partitions can be computed using

$$N_f(K) = \sum_{p1=1}^{K} \sum_{p2=1}^{p1} S(K, p1, p2) \tag{3.12}$$

The recursive operations required to compute $N_f(i), i = 1, 2, \ldots, K$ can be efficiently accomplished by defining the so-called lexicographical sum. Given a matrix A, the lexicographical sum of A is the matrix B of the same dimensions as A with elements

$$b_{ij} = \sum_{p=1}^{i} \sum_{q=1}^{j} a_{pq}, \quad \forall i, j$$

If A has a particular structure in which certain elements are to remain zero, then the above definition is modified in that the lexicographical sum is performed only for the nonzero elements of A.

Applying the definition to the number of monotone strategies, let S_k denote the matrix with elements $S_k(i, j)$ as defined by (3.8) through (3.10), with the element $S_k(i, j)$ indicating the number of strategies with k rows that end at positions $P_1(k) = i$, $P_2(k) = j$. Furthermore, let LS_k denote the lexicographic sum of S_k. Defining the vector J_m to be a vector with m leading ones and the rest trailing zeros, i.e., $J_m^T = [\ 1\ 1\ \ldots\ 1\ 0\ 0\ \ldots\ 0]$, the elements of LS_k can be computed from

$$LS_k(i, j) = J_i^T S_k J_j$$

Defining $e_n^T = [0\ 0\ \ldots\ 0\ 1]$, the recursive computation of $N_f(k)$ then reduces to the following:

$$S_{k+1} = \begin{bmatrix} LS_k & 0_{k\times 1} \\ e_n^T LS_k & e_k^T LS_k e_k \end{bmatrix}$$

$$N_f(k) = J_k^T S_k J_k \quad k = 1, 2, \ldots, K$$

3.4 Size of the Classes with Splitting of Flows

In general, strategies that allow splitting allow the flow $N(i, j)$ to be split into one or more lanes. (For partitioned strategies, this definition is more restrictive and allows the flow to

be split into a maximum of two lanes.) All classes of strategies that allow splitting have many more members than those that do not allow splitting. The number of such strategies is not only a function of the number of entry/exit pairs K, but is also a function of each element $N(i,j)$ of the origin destination matrix, because each flow $N(i,j)$ can potentially be split into two or more lanes. To compare the relative merits (in terms of admissible strategy space) of the various strategies, independent of the origin/destination matrix, we do not calculate the cardinality of the sets of strategies with splitting.

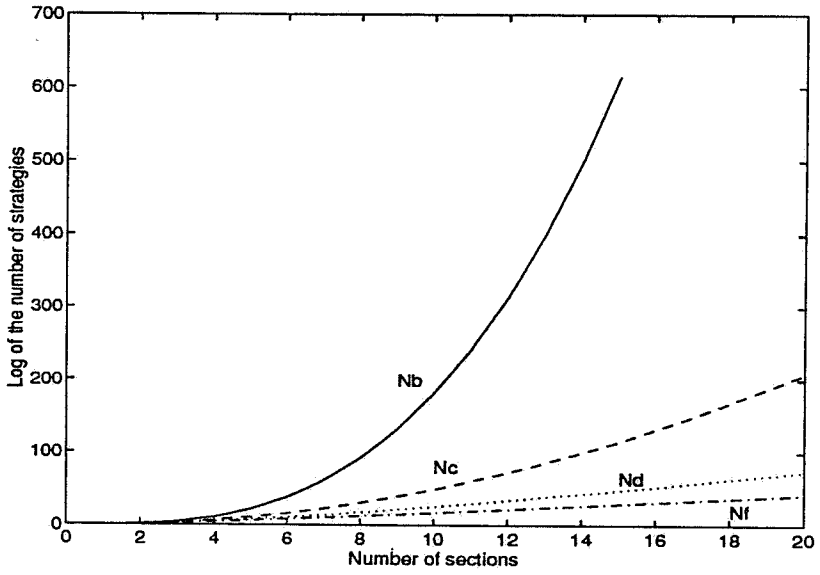


Figure 3.3: Logarithmic growth of the number of strategies

For a $(K-1, K)$ system the logarithmic growth of the number of admissible strategies of the type (2), (3), (4) and (6) is shown on Figure 3.3. The admissible strategy space increases much more sharply with problem size K for general lane assignment strategies as opposed to the constant lane and partitioned strategies.

CHAPTER 4

MINIMIZATION OF TOTAL TRAVEL TIME

4.1 The Travel Time Optimization Problem

In this chapter, we will discuss the travel time performance criterion, the reasons for having chosen that performance criterion, the lane assignment strategy space that was chosen and the reasons for having chosen that strategy space.

4.1.1 Motivation for minimization of travel time

The minimization of total travel time of all vehicles on a segment of an AHS is a viable performance criterion when system-wide performance is considered. While this is not the only criterion that is important in the problem of lane assignment, and we have studied other criteria (see Chapter 5), the minimization of travel time would be of greatest interest to the mass of commuters on an AHS. This is especially true under congested conditions.

4.1.2 Motivation for the use of constant lane strategies

The minimization of travel time has been carried out in the class of constant lane strategies with splitting. The reasons for this are as follows. Constant lane strategies minimize information exchange between the roadside beacon and the vehicle. They also call for fewer lane change maneuvers than general assignment strategies and, therefore, come closer to achieving a quasi steady-state flow. A lane change maneuver by a vehicle is associated with

the formation of a LAN with its neighbors in order to execute a coordinated maneuver. It may also be accompanied by a slowing down of vehicles in either the lane that they maneuvers to or the lane that they come from. This in turn increases the total travel time for all the vehicles. While the number of path intercepts is further reduced by the use of partitioned strategies, we chose not to restrict ourselves to partitioned strategies at the onset. Furthermore, because splitting can improve on the travel time achievable without splitting, we will work from now onwards in a class of strategies that allows splitting. (The reason for having calculated the number of strategies without splitting was to compare the cardinalities of the various strategy classes. The number of strategies for any class with splitting could not be calculated without knowing the origin/destination matrix.) Later in this thesis, we will compare the attainable performance of a constant lane nonpartitioned strategy to those of partitioned strategies. We will also study conditions under which the optimal constant lane strategy is a partitioned strategy.

4.1.3 Path determination through travel time comparison

An important characteristic of congested flow conditions is that the time lost in maneuvering can significantly affect highway throughput. To properly account for the effect of maneuvers, the total travel time should be composed of the time spent maneuvering as well as the time spent traveling at the reference lane speed. To introduce the approach used to develop the performance criterion, consider vehicles on an automated highway with two lanes with flows N_1 and N_2 (see Figure 4.1). A car entering the highway and restricted to a constant lane strategy has a choice between paths MM' and NN'. To take path MM', it enters the highway in L_1, then merges onto L_2. Before exiting, it maneuvers into L_1 and then exits the highway. To take path NN', it merges into L_1. Let γ^2 be the maneuver costs associated with MM' and γ^1 be the maneuver cost associated with NN'. Note that $\gamma^2 > \gamma^1$ because MM' involves an extra maneuver into and out of L_2. The total travel times when such a vehicle takes either of the two paths are

$$T_{MM'} = x(N_2 + 1)/v_2 + xN_1/v_1 + \gamma^2$$

and

$$T_{NN'} = xN_2/v_2 + x(N_1 + 1)/v_1 + \gamma^1$$

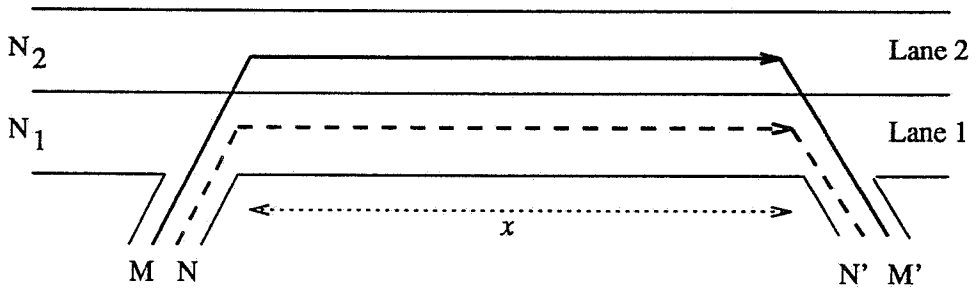


Figure 4.1: An example of travel time comparison

Path MM' is preferable to NN' if $T_{MM'} < T_{NN'}$. Similar constructs for all constant lane path options and the associated travel times form the basis for the development of the performance criterion for the three-lane case considered in the next section.

4.2 Formulation of the General Problem

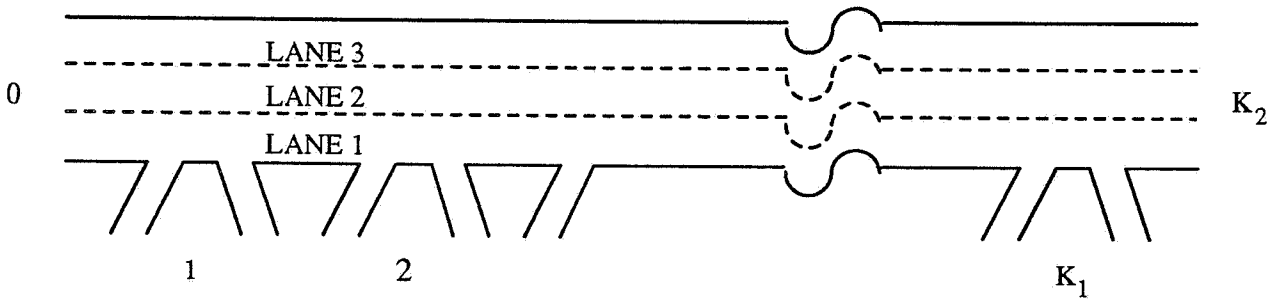


Figure 4.2: K_1, K_2 system

To develop the performance criterion involving travel time, consider a segment of a general three-lane highway shown in Figure 4.2. Vehicles entering at node 0 represent all vehicles entering this segment of the highway from entrances ahead of the studied segment. The vehicles leaving at K_2 are all of the vehicles that entered at node 0 plus those that entered at nodes 1 through K_1 less those that exited prior to K_2. The problem is to place the N_i

vehicles arriving at the K_i entrances into the three lanes, so as to minimize the total travel time of $\sum_{i=0}^{K_1} N_i$ vehicles. We cannot simply fill the fastest lane L_3 to capacity because in congested conditions, which are of the greatest interest, the capacity of L_3 may be less than $\sum_{i=0}^{K_1} N_i$. Furthermore, given some initial guess for the lane assignment, adding more vehicles to L_3 may not be desired because the cost of maneuvering to L_3 might offset the gain in time obtained by traveling at a higher speed v_3. The lane assignment problem considered here can be stated as follows: Determine, for a given origin/destination matrix, the flow of vehicles in each lane of the highway in order to minimize a performance index that is a function of total travel time.

We will now develop an analytical formulation of the problem assuming that the performance index represents the sum of the time spent traveling at the reference speed and the time spend maneuvering, and that the lane assignment strategies are restricted to constant lane strategies with splitting, including an analytical description of the relevant conservation of flow, capacity and non-negativity constraints.

4.3 Analytical Formulation

4.3.1 Linear formulation

Cost function

Let us restrict the set of allowable assignments to constant lane assignments. Now we are in a position to calculate the total travel time T_{total} using the relation

T_{total} = *Time spent traveling at the reference speed + Time spent maneuvering.*

$$T_{total} = T_{ss} + T_{maneuver} \tag{4.1}$$

The time spent traveling at reference speed is

$$= \sum_{m=1}^{3} \sum_{i=1}^{K_1} \sum_{j=i+1}^{K_2} (l_{i,j}/v_m)\rho_{i,j}^m$$

and, therefore, is linear with respect to the lane assignments $\rho_{i,j}^m$. For the present, assume that the cost of maneuvering per vehicle is independent of highway congestion and is assessed

at a value dependent only on the lane maneuvered to, and is equal to γ^m. This assumption has been made for the following two reasons. First, it represents a situation in which traffic on the AHS is light and maneuvers by a vehicle do not significantly affect the traffic flow around it. Second, the study of this simple case is done as a first step towards understanding the more complex case in which the maneuver cost is taken to be dependent on congestion. The total time for maneuvering for the congestion independent case is then

$$T_{manuever} = \sum_{m=1}^{3} \sum_{i=1}^{K_1} \sum_{j=i+1}^{K_2} (\gamma_{i,j}^m) \rho_{i,j}^m$$

and is also linear with respect to $\rho_{i,j}^m$. The total travel time, therefore, is

$$T_{total} = \sum_{m=1}^{3} \sum_{i=1}^{K_1} \sum_{j=i+1}^{K_2} (l_{i,j}/v_m + \gamma_{i,j}^m) \rho_{i,j}^m \quad = \mathbf{c}^{\mathbf{T}} \rho \tag{4.2}$$

Constraints

All of the vehicles entering the highway must be assigned to one of the three lanes. This results in conservation constraints. Furthermore, the traffic flow must satisfy capacity constraints. Finally, because each $\rho_{i,j}^m$ represents a vehicle flow, it cannot be negative. Therefore, assuming constant lane capacities through all the sections, the minimization problem must be solved under the following constraints.

Conservation constraints

$$\sum_{m=1}^{3} \rho_{i,j}^m = N_{i,j} \qquad 1 \leq i \leq K_1,\ 1 \leq j \leq K_2 \tag{4.3}$$

Capacity constraints

$$\sum_{l=0}^{i} \sum_{s=i+1}^{K_2} \rho_{l,s}^m \leq z_m \qquad m = 1,2,3,\ 1 \leq i \leq K_1 \tag{4.4}$$

Non-negativity constraints

$$\rho_{i,j}^m \geq 0 \qquad m = 1,2,3,\ 1 \leq i \leq K_1,\ 1 \leq j \leq K_2 \tag{4.5}$$

It is assumed that the vehicles entering from node 0 have already been assigned to their respective lanes and, hence, in all the above equations, the terms, $\rho_{0,l}^m$, $l = 1,2\ldots,K_2$, $m = 1,2,3$, are constants.

We will now vectorize $\rho_{i,j}^m$ as follows. The first element in ρ corresponds to the flow from entrance 1 to exit 2 in L_1. The second element corresponds to the flow from 1 to 2 in L_2, etc. Thus iterating first over lanes, then over exits and finally over entrances, we form the vector ρ where $\rho_{i,j}^m$, corresponds to the kth element in the vector ρ, where k is given by

$$k = \sum_{entr=1}^{i-1} 3(K_2 - entr) + 3(j - (i+1)) + m$$

Because the maneuver costs $\gamma_{i,j}^m$ are independent of ρ, we see that the cost function is linear in ρ. The constrained minimization problem is of the form

$$\min \mathbf{c}^{\mathbf{T}} \rho \tag{4.6}$$

subject to

$$\mathbf{A_1}\rho = \mathbf{b_1}, \quad \mathbf{A_2}\rho \leq \mathbf{b_2}, \quad \rho \geq 0 \tag{4.7}$$

where ρ = vector of lane assignments. The above is a linear programming problem and can be solved using, for example, the Simplex method.

This problem has a solution only if each section of the highway has the capacity to accept all of the vehicles entering it. Therefore, we assume that the following basic feasibility condition holds throughout the rest of this thesis.

Assumption 4.3.1

$$\sum_{l=1}^{i} \sum_{j=i+1}^{K_2} N_{l,j} < z_1 + z_2 + z_3 \qquad 1 \leq i \leq K_1$$

4.3.1.1 Analysis of a $(1, K)$ system with linear costs

To gain an understanding of the lane assignment problem, let us start with a simple case. Consider a portion of the highway with one entrance and K exits. The following lemma provides an optimal assignment policy under certain capacity constraints.

Lemma 4.3.1 *For a $(1, K)$ system with $\gamma_{i,j}^m = \gamma^m$ and $\gamma^1 < \gamma^2 < \gamma^3$ in the LP problem, let p_1 and p_2 be the smallest integers such that*

$$l_{1,p_1+1} \geq \frac{\gamma^2 - \gamma^1}{(1/v_1 - 1/v_2)}, \; l_{1,p_2+1} \geq \frac{\gamma^3 - \gamma^1}{(1/v_1 - 1/v_3)} \tag{4.8}$$

Define two numbers α and β as follows

$$\alpha = p_1 \quad \beta = p_2 + 1 \quad \textit{if } p_1 \leq p_2 \tag{4.9a}$$

$$\alpha = p_2 \quad \beta = p_2 + 1 \quad \textit{otherwise} \tag{4.9b}$$

Then, if the lane capacities satisfy

$$\sum_{i=1}^{\alpha} N_{1,i} \leq z_1, \quad \sum_{i=\alpha+1}^{\beta-1} N_{1,i} \leq z_2, \quad \sum_{i=\beta}^{K} N_{1,i} \leq z_3$$

the optimal lane assignments are such that

$$\{\textit{Vehicles } N_{1,i}: \quad 2 \leq i \leq \alpha\} \qquad \textit{Go via } L_1 \tag{4.10}$$

$$\{\textit{Vehicles } N_{1,i}: \quad \alpha < i < \beta\} \qquad \textit{Go via } L_2 \tag{4.11}$$

$$\{\textit{Vehicles } N_{1,i}: \quad \beta \leq i \leq K-1\} \qquad \textit{Go via } L_3 \tag{4.12}$$

Proof: The integer p_1 represents the first exit such that the maneuver cost for switching to L_2 is less than the time gained by traveling to that exit at a higher speed v_2. Similarly, p_2 represents the first exit such that the maneuver cost for going to L_3 from L_1 is less than the time gained by traveling to that exit at a higher speed v_3. We first prove the lemma for the case $p_1 \leq p_2$. Now $\alpha = p_1$ and $\beta = p_2 + 1$. First, let us assume that there are no capacity constraints, i.e.,

$$\sum_{i=1}^{K} N_{1,i} \leq min\{z_1, z_2, z_3\}$$

The total travel time under the assignment outlined in the lemma is

$$T_{tot} = \sum_{i=1}^{p_1} N_{1,i}(l_{1,i}/v_1 + \gamma^1) + \sum_{i=p_1+1}^{p_2} N_{1,i}(l_{1,i}/v_2 + \gamma^2) + \sum_{i=p_2+1}^{K} N_{1,i}(l_{1,i}/v_3 + \gamma^3)$$

Consider a different policy where $\alpha = \bar{p}_1$, where $\bar{p}_1 < p_1$. Denoting the new travel time by $\bar{T}_{tot}$, and using (4.8) we have

$$T_{tot} - \bar{T}_{tot} = \sum_{i=\bar{p}_1+1}^{p_1} ((l_{1,i}/v_1 + \gamma^1) - (l_{1,i}/v_2 + \gamma^2))N_{1,i} \quad \leq 0$$

The same is true for $\bar{p}_1 > p_1$. Along the same lines, we can prove the same for all $\bar{p}_2 \neq p_2$. Thus, for all integers α and β, under no capacity constraints, this policy is optimal. Now, we also need to consider other policies for which we one can split up the flow of vehicles going to

the same exit into the three lanes. Because p_1 is the first exit at which it profitable to change to L_2 and because there are no capacity constraints, T_{tot} is minimized when all vehicles with exits up to and including stay p_1 in L_1. Similarly, all vehicles with exits between p_1 and p_2 stay in L_2. Hence, no splitting up of any group of $N_{1,i}$ vehicles occurs for any i. The same argument can be used for the case when $p_1 > p_2$. Therefore, the stated policy is indeed optimal when we have no capacity constraints. The capacity required with this, however, is only such that the flow of vehicles in $L_m < z_m$, i.e.,

$$\sum_{i=1}^{\alpha} N_{1,i} \leq z_1, \quad \sum_{i=\alpha+1}^{\beta-1} N_{1,i} \leq z_2, \quad \sum_{i=\beta}^{K} N_{1,i} \leq z_3$$

Therefore, the stated policy is optimal even under the above capacity constraint. □

The idea behind Lemma 4.3.1 is that it becomes profitable to switch to another lane if the savings in travel time on that lane is greater than the time to maneuver (maneuver cost) to that lane. α represents the first exit for which vehicles going to exits beyond it incur lower travel times by not traveling in L_1. Depending on γ^m and on v_m, α might actually be equal to p_2, which means that it is profitable to switch to L_3 directly from L_1.

The following lemma provides a condition that an assignment for a $(1, K)$ system must satisfy to be optimal. First, we have the following definition.

Definition 4.3.1 *A lane L_i is said to be* **slower** *than a lane L_j if the reference speed v_i for L_i is smaller than the reference speed v_j for L_j.*

Lemma 4.3.2 *For a $(1, K)$ system with $\gamma_{i,j}^m = \gamma^m$, a minimizing assignment for the LP problem cannot assign vehicles traveling longer distances to a lane slower than one in which vehicles travel shorter distances.*

Proof: We can consider a $(1, 3)$ system with two lanes L_1 and L_2 w.l.o.g. Let

x_a = Portion of $N_{1,2}$ flow in L_1

x_b = Portion of $N_{1,2}$ flow in L_2

y_a = Portion of $N_{1,3}$ flow in L_1

y_b = Portion of $N_{1,3}$ flow in L_2.

We must prove that

$$x_b > 0 \Rightarrow y_a = 0 \text{ and } y_a > 0 \Rightarrow x_b = 0 \tag{4.13}$$

The total travel time for any arbitrary assignment x_a, x_b, y_a and y_b is

$$T_{tot} = x_a(l_{1,2}/v_1 + \gamma^1) + x_b(l_{1,2}/v_2 + \gamma^2) + y_a(l_{1,3}/v_1 + \gamma^1) + y_b(l_{1,3}/v_2 + \gamma^2)$$

Let $x_b > 0$ and $y_b > 0$. We will show that by exchanging vehicles between the two lanes, we will achieve lower travel times.

Step 1. Move one car from y_a to L_2 and one car from x_b to L_1. The difference between this travel time $\bar{T}_{tot}$ and the old T_{tot} is

$$T_{tot} - \bar{T}_{tot} = \frac{(l_{1,3} - l_{1,2})(v_2 - v_1)}{v_2.v_1} > 0$$

Step 2. Repeat Step 1 a number of times equal to $min(x_b, y_a)$. Each step will reduce the travel time. Finally, we will be left with

$$x_b > 0 \text{ and } y_a = 0 \text{ or } y_a > 0 \text{ and } x_b = 0$$

If either of them were not equal to 0, we could perform the operation in Step 1. This establishes Equation (4.13) and the lemma. □.

Theorem 4.3.1 *Consider a $(1, K)$ system with linear costs, i.e., with $\gamma_{i,j}^m = \gamma^m$ and with arbitrary lane capacities z_1, z_2 and z_3. The optimal lane assignment that minimizes the linear cost is as follows. There exist two integers π and δ such that*

$$\begin{array}{llll} \rho_{1,j}^1 = N_{1,j} & \rho_{1,j}^2 = 0 & \rho_{1,j}^3 = 0 & 2 < j < \pi \\ \rho_{1,j}^1 = 0 & \rho_{1,j}^2 = N_{1,j} & \rho_{1,j}^3 = 0 & \pi < j < \delta \\ \rho_{1,j}^1 = 0 & \rho_{1,j}^2 = 0 & \rho_{1,j}^3 = N_{1,j} & \delta < j < K \end{array} \tag{4.14}$$

and

$$\rho_{1,j} = d_{1,j}N_1 + c_{1,j}Z \qquad For j = \pi, \delta \tag{4.15}$$

where $\rho_{1,j} = [\rho_{1,j}^1\ \rho_{1,j}^2\ \rho_{1,j}^3]^T$ and $N_1 = [N_{1,1}\ N_{1,2} \ldots N_{1,K}]^T$ and $Z = [z_1\ z_2\ z_3]^T$, and $d_{1,j}$ and $c_{1,j}$ are constant matrices.

Proof: We prove the theorem for the two cases when the capacity assumptions of 4.3.1 are satisfied and when they are not.

Case 1. If the capacities satisfy the capacity assumption of Lemma 4.3.1, then the partitioning is such that $\pi = \alpha$ and $\delta = \beta$ with α and β defined as in Lemma 4.3.1.

Case 2. Let us form a set of vehicles in increasing order of destination. i.e.,

$$C = \{1, 2, \ldots, N_{1,2}, N_{1,2} + 1, \ldots, N_{1,2} + N_{1,3}, \ldots, N_{1,2} + N_{1,3} + \ldots + N_{1,K}\}$$

Let C(i) be the ith member of set C. Define L = {L(i) : L(i) = the lane assigned to C(i)} From Lemma 4.3.2 we have

$$i < j \Rightarrow L(i) \text{ is only as fast as } L(j)$$

The only assignment compatible with Lemma 4.3.2 is of the form

$$L = \{L_3, L_3, \ldots, L_2, L_2, \ldots, L_1, L_1\} \tag{4.16}$$

i.e., $\exists i$ and j such that

$$\begin{aligned} L(k) &= L_1 \qquad && 1 \leq k \leq i \\ L(k) &= L_2 \qquad && i < k < j \\ L(k) &= L_3 \qquad && j \leq k \leq N_{1,2} + N_{1,3} + \ldots + N_{1,K} \end{aligned}$$

Now, set π = destination of car C(i) and δ = destination of car C(j). This establishes Equation (4.14) of the theorem.

Now we establish Equation (4.15). Note that the objective function of the LP problem attains its optimum at a basic feasible solution. The basic feasible solutions are the extreme points of the convex set formed by the constraints. The extreme points are linear combinations of the RHS of the constraints. Therefore, any optimal solution must be of the form

$$\rho_{1,j} = d_{1,j} N_1 + c_{1,j} Z$$

and this establishes Equation (4.15). □

The optimal assignment for the $(1, K)$ system discussed above corresponds to a destination monotone (with splitting) assignment. Equation (4.14) implies that no splitting into two or more lanes occurs among groups of vehicles with destinations other than π and δ.

Equation (4.15) implies that the groups going to π and δ, however, may be split into different lanes.

We can show by means of a simple example (see Chapter 7) that Theorem 4.3.1 does not hold for a (K_1, K_2) system with a linear cost function for $K_1 > 1$. This is because Lemma 4.3.2 does not hold for such a system.

The optimal solution is, therefore, not guaranteed to belong to any of the partitioned strategy classes defined in Section 3.2, nor is it guaranteed to belong to the class of constant lane strategies without splitting. Imposing either of these requirements would result in additional constraints of algebraic and Boolean form and would significantly complicate the optimization problem. Recognizing the utility of partitioned strategies without splitting from an implementation point of view, we will later discuss the performance degradation when these additional requirements are imposed.

The $\gamma_{i,j}^m$'s reflect highway conditions. We estimate their value and use them in decision-making. The sensitivity of the optimal strategy to the $\gamma_{i,j}^m$'s is an important issue to study if we cannot otherwise estimate them. In the next subsection, we will consider a different model of maneuvering costs and study the resulting optimal strategies.

4.3.2 Nonlinear formulation

The above formulation considered the maneuver cost to be fixed, which represents the situation when the highway is lightly loaded and the overhead involved in maneuvering due to communication, is constant. A case of interest is when maneuver cost is proportional to the congestion of the highway. As the density of vehicles increases with congestion, the time required to complete the maneuver can be considered to depend on the densities $\rho_{i,j}^m$ of vehicles in both lanes influenced by the maneuver. This is because of the effect of many requests for maneuvers on the communication times accompanying maneuvers and the increased time required for the affected vehicles to generate enough space for the maneuvering vehicle to move from one lane to the other. This assumption leads to a quadratic programming problem with equality and inequality constraints and is formulated for a (K_1, K_2) system. Here, the simplest case will be considered in which maneuver costs are taken to be proportional to

the number of vehicles per unit time that wish to maneuver and the flow of vehicles in the lane to which they are maneuvering. (Microscopic simulation using the AHS simulator will provide a better model for maneuvering costs in the future.) Maneuver costs are taken into account at the time of entering the highway and at the time of exiting from the highway. Therefore, the maneuvering costs $\gamma^m_{i,j}$ for vehicles with assigned lane L_m are represented by

$$\gamma^1_{i,j} = C^a_1(\underbrace{\sum_{l=0}^{i-1}\sum_{s=i+1}^{K_2}\rho^1_{l,s}}_{\text{Vehicles(A,1,i)}} + \underbrace{\sum_{l=i+1}^{K_2}(\rho^3_{i,l}+\rho^2_{i,l}+\rho^1_{i,l})}_{\text{Vehicles(E,1,i)}}) + C^b_1(\underbrace{\sum_{l=0}^{j-1}(\rho^3_{l,j}+\rho^2_{l,j}+\rho^1_{l,j})}_{\text{Vehicles(X,j)}})$$

$$\gamma^2_{i,j} = \gamma^1_{i,j} + C^a_2(\underbrace{\sum_{l=0}^{i-1}\sum_{s=i+1}^{K_2}\rho^2_{l,s}}_{\text{Vehicles(A,2,i)}} + \underbrace{\sum_{l=i+1}^{K}(\rho^2_{i,l}+\rho^3_{i,l})}_{\text{Vehicles(E,2,i)}}) + C^b_2(\underbrace{\sum_{l=0}^{j-1}\sum_{j}^{K_2}\rho^1_{l,s}}_{\text{Vehicles(A,1,j-1)}} + \underbrace{\sum_{l=0}^{j-1}(\rho^2_{l,j}+\rho^3_{l,j})}_{\text{Vehicles(X,j)}})$$

$$\gamma^3_{i,j} = \gamma^2_{i,j} + C^a_3(\underbrace{\sum_{l=0}^{i-1}\sum_{s=i+1}^{K_2}\rho^3_{l,s}}_{\text{Vehicles(A,3,i)}} + \underbrace{\sum_{l=i+1}^{K_2}\rho^3_{i,l}}_{\text{Vehicles(E,3,i)}}) + C^b_3(\underbrace{\sum_{l=0}^{j-1}\sum_{j}^{K_2}\rho^2_{l,s}}_{\text{Vehicles(A,2,j-1)}} \underbrace{\sum_{l=0}^{j-1}\rho^3_{l,j}}_{\text{Vehicles(X,j)}})$$

where

Vehicles(A,n,i) = No. of vehicles already in L_n in section $s_{i,i+1}$.

Vehicles(E,n,i) = No. of vehicles entering L_n at node i.

Vehicles(X,i) = No. of vehicles exiting the lane at node i.

Because vehicles maneuvering to L_2 will necessarily have to maneuver to L_1 first, they will incur $\gamma^1_{i,j}$ as part of their maneuver cost. Similarly, vehicles maneuvering to L_3 will necessarily incur both $\gamma^1_{i,j}$ and $\gamma^2_{i,j}$ as part of their maneuver cost. The total travel time of all the vehicles is given now given by

$$\begin{aligned} T_{tot}(\rho) &= \sum_{i=1}^{K_1}\sum_{j=i+1}^{K_2}\sum_{m=1}^{3}\rho^m_{i,j}(\frac{l_{i,j}}{v_m}+\gamma^m_{i,j}) \\ &= q\rho'\mathbf{Q_1}\rho + \mathbf{Q_2}\rho \end{aligned} \tag{4.17}$$

ρ is the vector of lane assignments $\rho^m_{i,j}$ $1 \le i \le K_1$, $2 \le j \le K_2$ and $1 \le m \le 3$ of length $M = 3K_1(2K_2 - K_1 - 1)/2$. Q_1 is a matrix of size $(M\text{x}M)$, Q_1 is a column vector of length (M) and q is the largest common multiple of the maneuver cost proportionality constants C^a_m and C^b_m for m = 1, 2, 3. We will use q to model the uncertainty in the dependence of travel time on congestion. In particular, we will study how the nature of the optimal strategies changes with changes in q.

This problem of minimization of the criterion (4.17) is solved under the same constraints as the linear problem (Equations (4.3), (4.4) and (4.5)) and is, therefore, a quadratic program of the form

$$\min_{\rho} q\rho^T \mathbf{Q}_1 \rho + \mathbf{Q}_2 \rho$$

subject to

$$\mathbf{A}_1 \rho = \mathbf{b}_1, \quad \mathbf{A}_2 \rho \leq \mathbf{b}_2, \quad \rho \geq 0$$

Let us examine the question of existence and uniqueness of solutions to Equation (4.17). The linear constraints defined by Equations (4.3), (4.4) and (4.5) form a closed convex polyhedron. The cost function $T_{tot}(\rho)$ is a continuous function of ρ and is defined over a closed and bounded set. By the Weierstrass theorem [30], $T_{tot}(\rho)$ achieves its minimum over Ω, and so at least one solution exists to (4.17). The matrix Q_1 is symmetric, but not positive definite. Therefore, $T_{tot}(\rho)$ is not strictly convex (in fact, not even convex), and we cannot guarantee existence of a unique solution. The reason behind Q_1 not being positive definite can be understood by realizing that it is inherently not diagonally dominant. This is because maneuver costs arise when the path of two cars cross. For Q_1 to be diagonally dominant, the maneuver cost of a car crossing its own path would have to be greater than that of it crossing other cars' paths, which is not reasonable.

There exist a number of efficient computer codes such as BQPD [31], LINDO [32], PC-PROG, MATLAB [33] and QPOPT [34] that are designed for solving quadratic programs that are not necessarily convex. The examples presented in this thesis were, however, solved using the gradient projection algorithm (admittedly not the most efficient one). It is subject to the possibility of jamming, but has been used successfully for numerous nonlinear programming problems without jamming. The initial feasible point was found by solving the LP problem with the same constraints as the QP problem. The optimal solution of the LP problem was used as the initial feasible point for the QP problem. One point to note is that these algorithms will not in general give rise to integer solutions, and rounding off or truncation would be necessary to obtain an approximate integer solution. The steps in the gradient projection algorithm as given in Luenberger [35] are summarized below.

In brief, we start at an initial feasible point and move in the direction of the projection of the negative gradient in the feasible region. A line search we move along this direction in order to minimize the cost function; we move along this direction either to this minimum if it is feasible, or to the edge of the feasible region if not. The process is repeated until a point is reached where the Kuhn-Tucker conditions are satisfied.

1. Obtain an initial feasible point.

2. Determine which of the inequality constraints are active. (All of the equality constraints **must** be active.) Form the matrix A_q, which is the left-hand side of the active constraints $A_q\rho = b_q$. q represents the number of active constraints.

3. Calculate the projection

$$P = I - A_q'(A_qA_q')^{-1}A_q$$

and the feasible direction

$$d = -P\nabla f(x)'$$

4. If $d \neq 0$, find α_1 and α_2 achieving

$$\begin{aligned} \alpha_1 &= max\{\alpha : x + \alpha d \text{ is feasible}\} \\ \alpha_2 &= min\{f(x + \alpha d) : \ 0 \leq \alpha \leq \alpha 1\} \end{aligned}$$

$$x_{k+1} = x_k + \alpha_2 d$$

Go to Step 2.

5. If $d = 0$, find

$$\beta_k = -(A_qA_q')^{-1}A_q\nabla f(x)'$$

(a) If $\beta_j > 0 \ \forall j \ 1 \leq j \leq q$, stop.

(b) Else, delete the row from A_q corresponding to the inequality with the most negative β_j. Go to step 3.

An optimizer has been written in C++ for the purpose of solving the quadratic problem. It uses the gradient projection method to find a local minimum. (In retrospect, a QP

optimizer should have been used.) The optimizer has been successfully applied to a number of test problems. The LP problem was solved using a public domain package called lp_solve [36].

4.4 Sensitivity of the Optimal Assignment

We briefly present the calculation of the sensitivity of the optimal QP solution. In the constrained minimization problem defined above, at the optimal point ρ^*, the following two conditions must be satisfied.

$$g(\rho) = b_q \tag{4.18}$$

$$\nabla f(\rho) + \beta^T \nabla g(\rho) = 0 \tag{4.19}$$

The first equation above defines the active constraints. The second equation represents the Kuhn-Tucker conditions at the optimal point with $f(\rho)$ the cost function, β the Lagrange multipliers and $g(\rho)$ the matrix of active constraints. The active constraints consist of the equality constraints and the active capacity and non-negativity (inequality) constraints. Together, these two equations can be used to solve for the ρ and β. Each active constraint (q in all) has a β associated with it. Let n be the number of variables ρ. Thus, we have $n+q$ unknowns and $q+n$ equations. For the QP problem, the system of equations that can be used to solve for ρ and β are

$$\begin{bmatrix} A_q & 0 \\ 2qQ_1 & A_q' \end{bmatrix} \begin{bmatrix} \rho \\ \beta \end{bmatrix} = \begin{bmatrix} b_q \\ -Q_2' \end{bmatrix} \tag{4.20}$$

In order to calculate sensitivities, we must make the following assumption.

Assumption 4.4.1 *The set of active constraints A_q at the optimal point is invariant to an infinitesimal change in a parameter.*

If even a minute change $\Delta\phi$ in a parameter ϕ causes A_q to change, the perturbed optimal point will be significantly different from the unperturbed one. This indicates that the sensitivity of ρ to ϕ may be discontinuous at that point.

Under the above assumption, we have

$$\nabla g(\rho)\frac{d\rho}{d\phi} = \frac{db_q}{d\phi} \quad (4.21)$$

$$\frac{d}{d\phi}(\nabla f(\rho)) + \nabla^2 f(\rho)\frac{d\rho}{d\phi} + \frac{d\beta}{d\phi}\nabla g(\rho) + \beta\frac{d}{d\phi}(\nabla g(\rho)) + \beta\nabla^2 g(\rho)\frac{d\rho}{d\phi} = 0 \quad (4.22)$$

For the QP problem, this is equivalent to

$$\begin{bmatrix} A_q & 0 \\ 2qQ_1 & A_q' \end{bmatrix} \begin{bmatrix} \frac{d\rho}{d\phi} \\ \frac{d\beta}{d\phi} \end{bmatrix} = \begin{bmatrix} \frac{db_q}{d\phi} \\ -\frac{d}{d\phi}Q_2' - 2\frac{d}{d\phi}(2qQ_1)\rho \end{bmatrix} \quad (4.23)$$

Equations (4.20) and (4.23) are of the form

$$Fy = E$$

Note that the F matrix is the same for calculation of ρ and β as well as for their sensitivities. The vector E varies from parameter to parameter. Therefore, we solve the same system of equations with different right-hand sides for different parameters.

4.5 Extension of the Framework to Other Situations

The optimization problem framework described above can be extended to other situations. Consider the case with nonconstant lane capacities. Assume that the capacities can vary section by section as well as lane by lane. In that case, the ρ vector can be redefined to be

$$\rho = \{\rho(i,j,m,s) : \ 1 \le i \le K_1, \ 2 \le j \le K_2, \ 1 \le m \le 3, \ i \le s \le j-1\}$$

and the optimization problem would be

$$\min_{\rho} \sum_{m=1}^{3} \sum_{i=1}^{K_1} \sum_{j=i+1}^{K_2} \sum_{s=i}^{j-1} (l_{s,s+1}/v_m + \gamma(i,j,m,s))\rho(i,j,m,s) \quad (4.24)$$

subject to

$$\sum_{m=1}^{3} \rho(i,j,m,s) = N_{i,j} \qquad 1 \le i \le K_1, \ 1 \le j \le K_2, \ i \le s \le j-1$$

$$\sum_{l=0}^{s} \sum_{r=s+1}^{K_2} \rho(l,r,m,s) \le z(m,s) \qquad m = 1,2,3, \ 1 \le s \le K_1$$

$$\rho(i,j,m,s) \ge 0 \qquad m = 1,2,3, \ 1 \le i \le K_1, \ 1 \le j \le K_2, \ i \le s \le j-1$$

CHAPTER 5

BALANCING EXCESS CAPACITIES

5.1 Problem Formulation

In this chapter, we will discuss the balancing excess capacity performance criterion. We describe the reasons that justify this performance criterion, the admissible space of lane assignment strategies, and the reasons for having chosen that admissible strategy space.

5.1.1 Motivation for balancing of excess capacities

The performance criterion considered is the balancing of excess capacities (load balancing) over the lanes of the AHS. We believe that this criterion is of particular interest when moving from periods of low congestion to periods of high congestion, so that excess capacity exists in all lanes and sections of the highway at the beginning of the high-congestion time interval. The flow of traffic through a segment of a highway would be limited by the flow of traffic through a bottleneck section (if any) in that segment. The lane assignment is carried out to distribute the traffic as equally as possible within the three lanes of the AHS, in the bottleneck section, to utilize it to the maximum. Work has also been done in this area by Hall [10], who considers an objective that equalizes the workload across lanes in order to equalize the congestion across lanes.

5.1.2 Motivation for the use of destination monotone strategies

In Chapter 4, we considered a travel time minimization problem in which the travel time was calculated as

$$T_{total} = T_{steadystate} + T_{maneuver}$$

$T_{maneuver}$ is calculated by placing an explicit penalty on lane changes. A lane change maneuver by a vehicle is associated with the formation of a LAN with its neighbors in order to execute a coordinated maneuver. It may also be accompanied by a slowing down of vehicles in either the lane that it maneuvers to or the lane that it comes from. This in turn increases the total travel time for all the vehicles. In this chapter, while we do not explicitly take maneuver costs into account as in Chapter 4, we strive to use lane assignment strategies that minimize lane changes. For this reason, we restrict ourselves to constant lane strategies. For reducing computational complexity as evidenced by Figure 3.3, we further restrict ourselves to destination monotone strategies without splitting. Monotone strategies can be treated in the same manner with additional constraints.

5.1.3 Balancing excess capacities

The goal in the balanced excess lane capacity problem is to determine optimal partitioned lane assignment strategies that maximize the least excess capacity over all lanes and all sections of the highway.

The excess capacity is defined as the difference between the available lane capacity and the traffic volume in the lane resulting from the selected lane assignment strategy. The adjective balanced is used because in the section where the least excess lane capacity ultimately occurs, the best strategy will exhibit a tendency for balancing the excess capacity between all the lanes. It is noted that with the selected criterion here (see equation (5.6) later on in this section), no such balancing effect occurs in sections far from the section where the least excess capacity occurs. In sections with a large excess capacity, congestion is not as severe and bottlenecks are less likely to appear.

Consider the maximization of the minimal excess lane capacity in the class of destination-monotone lane assignment strategies without splitting. Let the vectors $\{P_1, P_2\}$ define a

candidate partitioned strategy. This strategy implies that vehicles entering at i will use lane L_1 if traveling to exits $i+1$ through $P_1(i)$, will use lane L_2 if traveling to exits $P_1(i)+1$ through $P_2(i)$, and will use lane L_3 if traveling to exits $P_2(i)+1$ through K. Furthermore, the pair $\{P_1(i), P_2(i) : i = 1, \ldots, K\}$ defining a destination monotone strategy must satisfy the condition

$$i \leq P_1(i) \leq P_2(i) \leq K \tag{5.1}$$

A simple way of characterizing traffic flow in each lane is to partition the N matrix into three origin destination matrices N_1, N_2 and N_3 characterizing the usage of each lane. This is accomplished by defining the partitioning matrices A_1, A_2 and A_3, defined as follows

$$A_1(i,j) = \begin{cases} 1, & i < j \leq P_1(i) \\ 0, & else \end{cases} \qquad A_2(i,j) = \begin{cases} 1, & P_1(i) < j \leq P_2(i) \\ 0, & else \end{cases}$$

$$A_3(i,j) = \begin{cases} 1, & P_2(i) < j \leq K \\ 0, & else \end{cases} \tag{5.2}$$

The number of vehicles traveling in lanes L_1, L_2 and L_3, respectively, is

$$N_m = N.A_m \quad m = 1,2,3 \tag{5.3}$$

where the operation $C = A.B$ implies element by element product of matrices A and B (i.e., if $A = [a_{ij}]$ and $B = [b_{ij}]$, then $C = [c_{ij}] = [a_{ij}b_{ij}]$.) Define now the matrix J of dimension KxK and the K-dimensional vector u as

$$J = \begin{bmatrix} 1 & 0 & 0 & \ldots & \ldots & 0 & 0 \\ 1 & 1 & 0 & \ldots & \ldots & 0 & 0 \\ 1 & 1 & 1 & \ldots & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & 1 & 1 & \ldots & \ldots & 1 & 0 \\ 1 & 1 & 1 & \ldots & \ldots & 1 & 1 \end{bmatrix} \qquad u = \begin{bmatrix} 1 \\ 1 \\ 1 \\ \ldots \\ \ldots \\ \ldots \\ 1 \\ 1 \end{bmatrix} \tag{5.4}$$

and notice that $N_m u, m = 1,2,3$ are vectors whose components represent the total number of vehicles entering at various entry points and traveling in lane $L_m, m = 1,2,3$, respectively.

Moreover, $N_m^T u$, $m = 1,2,3$ are vectors whose components represent the total number of vehicles exiting at various exits and traveling in lane $L_m, m = 1,2,3$, respectively, while $sv_m = J(N_m - N_m^T)u$, $m = 1,2,3$ are vectors whose components represent the traffic volume in lane L_m, $m = 1,2,3$.

The excess lane capacity in each section k and lane m is a function of the selected partitioned lane assignment strategy $\{P_1, P_2\}$, and is given by

$$e_m(P_1, P_2, k) = z_m - sv_m(P_1, P_2, k) \quad m = 1,2,3 \tag{5.5}$$

(For brevity, the dependence of sv_m on P_1, P_2 is omitted in the sequel.) The performance criterion for the balanced excess lane capacity problem is

$$J(P_1, P_2) := e_{min}(P_1, P_2) = \min_{i\in\{1,2,3\}}\{\min_{k\in\{1,2,\ldots,K\}}\{e_i(k)\}\}. \tag{5.6}$$

The optimal partitioned strategies are, then, the strategies maximizing $J(P_1, P_2)$. The problem of maximizing excess lane capacity thus reduces to a maxmin type of problem consisting of the maximization with respect to $\{P_1, P_2\}$ of the min function in Equation (5.6).

Note that the total flow through the sections of the highway is given by the vector

$$sv = sv_1 + sv_2 + sv_3 \quad = J(N - N^T)u$$

and is independent of the lane assignment strategy. This enables us to calculate an upper bound for J. Let

$$sv_{max} = \max_k\{sv(k)\}$$

Then,

$$J_{opt} \leq J_{ideal} = \frac{1}{3}(z_1 + z_2 + z_3 - sv_{max}) \tag{5.7}$$

This upper bound may not be achievable with partitioned strategies that do not allow splitting, but is achievable by those that allow splitting of the flow from a particular entry point to a particular exit point into more than one lane.

The use of this performance criterion will result in the choice of a lane assignment strategy (i.e., partitions P_1, P_2) that balances the excess lane capacities to the utmost possible extent (within the class of admissible strategies) in the section with the heaviest traffic flows. This

is because if such balancing of lane capacities were not achieved to the fullest possible extent by a candidate optimal strategy, a contradiction occurs, because a better strategy could be found by further increasing the minimal excess capacity and providing a better value of the performance criterion.

The computational problem reduces to the problem of how to determine the optimal strategy, or a suboptimal strategy. An optimal strategy will always exist because the class of admissible strategies is finite. However, if a complete search is too costly, it is often acceptable to determine a suboptimal strategy. A suboptimal strategy is any strategy $\{P_1, P_2\}$ such that $[J_{opt} - J(P_1, P_2)] < \Delta$, where Δ is a suitably selected bound, so that a satisfactory solution close enough to the optimum is obtained. It is possible in this problem to define a reasonable $\Delta > 0$, and thus a viable suboptimal strategy, because an upper bound on the value of J_{opt} is known. Accordingly, a stopping criterion may be easily adopted to be

$$J_{ideal} - J(P_1, P_2) < \Delta \tag{5.8}$$

for some selected Δ. This guarantees $J_{opt} - J(P_1, P_2) < \Delta$ because $J_{opt} \leq J_{ideal}$.

5.2 Computing the Optimal Assignment

The algorithm described below was developed to solve the restricted balanced excess lane capacity problem in the class of destination monotone strategies. It is referred to as a "greedy-type" algorithm because in the process of modifying the strategy pair $\{P_1, P_2\}$, the algorithm concentrates first on the effect that the modification will have on the section where the least excess capacity occurred under the current strategy. Furthermore, in every iteration only one element of P_1 and/or P_2 is modified, as opposed to the entire modification of P_1 and/or P_2. (An analogous algorithm solves the problem in the class of monotone strategies with the additional restrictions on the strategies defined by Equation (5.2) taken into account; however, it will not be discussed here in detail.)

The overall concept is best described as consisting of two basic principles. The first principle is an application of a "greedy-algorithm"-type philosophy. This principle is applied far away from the optimum, and "far" from the boundary (i.e., not on the boundary) of

the admissible domain defined by Equation (5.1) for destination monotone strategies. The second principle is the application of a gradient-based strategy modification adjusted to take into account the quantized nature of flows that are switched from lane to lane when P_1 and/or P_2 are modified, and the small number (three) of functions (e_1, e_2 and e_3) involved in computing the min function in Equation (5.6). The gradient-type step is applied at the boundaries of the admissible domain defined by Equations (5.1) and (5.2) and close to the optimum.

5.2.1 Greedy-type algorithm iterations

The algorithm may be described as follows. Consider the lane flows in iteration m with the lane assignment strategy $P_1(i)$, $P_2(i)$, $i = 1, \ldots, K$. Compute $\min_{k\in\{1,\ldots,K\}}\{e_1(k), e_2(k), e_3(k)\}$ and determine the section $\bar{K}$ and the lane $\bar{L}$ for which the minimum is attained. If $J(P_1, P_2)$ is far from the optimum, and $\{P_1, P_2\}$ is not on the boundary, iteratively modify P_1 or P_2 into $\hat{P}_1$, $\hat{P}_2$ as follows:

$$\hat{P}_1(\bar{K}) = \begin{cases} P_1(\bar{K}) - 1 & if\ \bar{L} = 1 \\ P_1(\bar{K}) + 1 & if\ \bar{L} = 2\ and\ e_1(\bar{K}) > e_3(\bar{K}) \\ P_1(\bar{K}) & else \end{cases} \tag{5.9}$$

$$\hat{P}_2(\bar{K}) = \begin{cases} P_2(\bar{K}) + 1 & if\ \bar{L} = 3 \\ P_2(\bar{K}) - 1 & if\ \bar{L} = 2\ and\ e_1(\bar{K}) < e_3(\bar{K}) \\ P_2(\bar{K}) & else \end{cases} \tag{5.10}$$

where "else" refers to other cases involving other values of $\bar{L}$ and all other $k \neq \bar{K}$.

The strategy is applied if it will not violate the admissible domain as defined by Equation (5.1). On its own, the greedy-type algorithm has been found efficient in converging to the vicinity of the optimal solution from various initializing partitions. However, it sometimes jams down in local maximums, or cycles, mainly because it approaches a boundary of the admissible domain. The greedy-type algorithm works only on the partitions for entrance $\bar{K}$, and so its effects are very localized.

5.2.2 Gradient algorithm iteration

When $J(P_1, P_2)$ is near the optimum, or $\{P_1, P_2\}$ is on the boundary of the admissible domain, the algorithm switches to a gradient-type iteration. When this will occur is determined in advance by defining a switchover point utilizing the known value of J_{ideal}. That is, a bound δ is defined and the greedy-type iterations are used if $J_{ideal} - J(P_1, P_2) > \delta$, while a gradient-type iteration is used if $J_{ideal} - J(P_1, P_2) < \delta$. The algorithm terminates when the condition $J_{ideal} - J(P_1, P_2) \leq \Delta$ is satisfied with $\Delta < \delta$.

In general, when carrying out the maximization of so-called min functions, the approach is to determine the next iterate not only based on the current subset of minimizers, referred to as the "answering set," but also based on an enlarged "answering set" that includes functions close to the minimizer. This essentially implies considering not only the lane with the least excess lane capacity for the given lane partitioning strategy, but also the lane with the next least excess lane capacity. The main idea in the gradient-type iteration is that if the current minimal excess capacity cannot be improved, then the chances for doing so in successive iterations can be improved by decreasing the excess capacity in the lane with the largest excess capacity, because the constant sum of excess lane capacities results in an increase in the excess capacities of the other two lanes. (The method extends easily to other multi-lane situations).

To perform the gradient-type iterations, we consider all components of the strategy pair $\{P_1, P_2\}$ upstream of the section $\bar{K}$ as candidates for finding an improved $\{\hat{P}_1, \hat{P}_2\}$. The gradient-type algorithm modifies the partitions of entrances before $\bar{K}$ in the hope of improving the excess capacity in section $\bar{K}$. Its effects are not localized and may be felt from entrances 1 through $\bar{K}$. The following is the overall gradient-type strategy:

1. If $\bar{L} = 1$, first calculate $e_2(\bar{K})$ and $e_3(\bar{K})$.

   (a) If $e_2(\bar{K}) < e_3(\bar{K})$ (lane L_2 is the lane with the next least excess capacity), first modify an element $P_2(i)$, $i = 1, \ldots, \bar{K}$ of the partition P_2 to maximize $min\{e_2(\bar{K}), e_3(\bar{K})\}$. Once P_2 is modified into $\hat{P}_2$ resulting in the excess capacities $\{e_{2m}(\bar{K}), e_{3m}(\bar{K})\}$, modify an element $P_1(i)$, $i = 1, \ldots, \bar{K}$ of the partition P_1 in order to maximize $min\{e_1(\bar{K}), e_{2m}(\bar{K}), e_{3m}(\bar{K})\}$.

(b) If not (lane L_3 is the lane with the next least excess capacity), modify only P_1 as described.

2. If $\bar{L} = 3$, first calculate $e_2(\bar{K})$ and $e_1(\bar{K})$.

   (a) If $e_2(\bar{K}) < e_1(\bar{K})$ (lane L_2 is the lane with next least excess capacity), first modify an element $P_1(i), i = 1, \ldots, \bar{K}$ of the partition P_1 to maximize $min\{e_1(\bar{K}), e_2(\bar{K})\}$. Once P_1 is modified into $\hat{P}_1$ resulting in the excess lane capacities $\{e_{1m}(\bar{K}), e_{2m}(\bar{K})\}$, modify an element $P_2(i)$, $i = 1, \ldots, \bar{K}$ of the partition P_2 in order to maximize $min\{e_{1m}(\bar{K}), e_{2m}(\bar{K}), e_3(\bar{K})\}$.

   (b) If not (lane L_1 is the lane with the next least excess capacity), modify only P_2 as described.

3. If $\bar{L} = 2$, first calculate $e_1(\bar{K})$ and $e_3(\bar{K})$.

   (a) If $e_1(\bar{K}) < e_3(\bar{K})$ (lane L_1 is the lane with the next least excess capacity), modify partition P_2 as described.

   (b) If not, (lane L_3 is the lane with the next least excess capacity), modify partition P_1 as described.

If the strategy pair is not at the boundary of the admissible domain, or if the solution is far from the optimum, then the greedy-type step is applied. The strategy modifications defined above are carried out if the admissible domain is not violated; otherwise, the algorithm terminates.

5.2.3 Number of iterations required to reach the optimum

Because one, or at most two, elements of P_1 or P_2 are modified in each iteration, it is possible to provide an estimate for the order of magnitude of the average number of iterations required to reach a suboptimal solution when starting the algorithm from certain initial partitions. For extreme partitions in which all vehicles are initially allocated to one lane (for example, the middle lane), we can compute an average number of iterations as follows. Let $T = u^T N u$ be the total number of vehicles in the origin/destination matrix N,

and let $EC_0 = z_2 - T$ be the initial value of the minimal excess capacity (this represents the worst-case situation when all of the T cars are in L_2 in some section). The maximal value of the minimal excess capacity that can be achieved is J_{ideal}. Because the number of nonzero elements of N is $R = K(K-1)/2$, the average value of nonzero elements of N is T/R. This quantity can be considered to be the average number of vehicles moving from an entrance to an exit, and so it is indicative of the average improvement in the performance criterion per iteration. (We neglect the fact that there may exist iterations in which the value of the criterion is not increased, but on the positive side we also neglect the fact that there are iterations in which an element of P_1 and P_2 is simultaneously modified in the gradient-type iterations.)

The order of magnitude for the average number of iterations required to converge to the vicinity of the optimum can now be assessed from the ratio of the upper bound on the total required change in performance value $J_{ideal} - EC_0$ to the average increase in the value of performance per iteration T/R, resulting in

$$\begin{aligned} M_A &= \frac{(J_{ideal} - EC_0)R}{T} \\ &= R - R\frac{sv_{max}}{3T} + \frac{((z_1 + z_2 + z_3)/3 - z_2)R}{T} \end{aligned} \tag{5.11}$$

Now, in general, the last term is negligible with respect to the other two terms because $(z_1 + z_2 + z_3)/3 \approx z_2$ and T is large. Hence, this term can be neglected, producing

$$M_A = R - Rsv_{max}/3T \tag{5.12}$$

Thus, the number is, on average, smaller than $R = K(K-1)/2$. For the case study considered below, with $K = 20$, $R = 190$, and $sv_{max}/3T = .2$, it follows that $M_A = 152$, which is many orders of magnitude smaller than $N_f(20)$ (see Figure 3.3). [Note that M_A should be roughly equal to the number of elements in the upper triangular portion of the origin destination matrix (not counting the 0 valued diagonal elements). For $K = 20$, this figure is equal to $19 * 20/2 = 190$, which is of the same order as the M_A calculated above.]

CHAPTER 6

FURTHER STUDIES ON PARTITIONED STRATEGIES

We have seen that for a $(1, K)$ system, with constant maneuver costs, the optimal solution of the resultant LP problem is a partitioned solution. Partitioned strategies are desirable because they lead to laminar traffic flow, cause less communication overhead and cause fewer path intersections. Partitioned strategies are also easy to implement compared to a more general lane assignment because for each entrance (exit) on a three-lane highway, only two numbers corresponding to the partitions must be kept track of, unlike a general assignment, in which more information must be stored. We will show by means of a simple example (in Chapter 7) that for a (K_1, K_2) system with linear cost, the optimal solution is not always a partitioned one. There is also no guarantee that with nonlinear costs the solution will be partitioned as will be shown by means of examples. This motivates the study of the existence, computability and optimality of partitioned strategies and the approximation of optimal nonpartitioned strategies by suboptimal partitioned strategies.

6.1 Some Properties of Partitioned Strategies

Definition 6.1.1 *Partitioning algorithms are lane assignment algorithms that assign two exits π and δ to each entrance i, such that cars traveling to exits $j < \pi$ travel in L_1, cars traveling to exits $\pi < j < \delta$ travel in L_2 and cars traveling to exits $j > \delta$ travel in L_3. Cars traveling to π may travel in L_1 or L_2, and those traveling to δ may travel in L_2 or L_3. The strategy obtained by a partitioning algorithm is called a partitioned strategy.*

Theorem 6.1.1 *There exists a feasible destination monotone (with splitting) or origin monotone lane assignment (with splitting.)*

Proof: As a first step, list the vehicles entering at node 1 as members of the following set

$$C_1 = \{1, 2, 3, \ldots, N_{1,2}, N_{1,2}+1, \ldots, N_{1,2}+N_{1,3}, \ldots, N_{1,2}+N_{1,3}+\ldots+N_{1,K_2}\}$$

Choose two members $C_1(n_1)$ and $C_1(n_2)$ from the above set with $\pi = C_1(n_1)$'s exit and $\delta = C_1(n_2)$'s exit, such that the lane capacity constraints are not violated. This is possible due to the feasibility assumption (4.3.1) with $i = 1$, i.e.,

$$\sum_{j=2}^{K_2} N_{1,j} \leq z_1 + z_2 + z_3$$

At node 2, fill as many vehicles as possible into, say, L_3, then fill L_2 and then put the remaining in L_1. This is possible once again because of Assumption 4.3.1 with $i = 2$. Repeat this for the remaining entrances until K_1 by invoking Assumption 4.3.1. This results in a feasible solution □

Note that the destination monotone partitioned solution obtained here will in general require *splitting* of the vehicles going to the partitions $P_1(i)$ and $P_2(i)$. The same holds for origin monotone assignments.

We can also show by means of the following example that a feasible monotone assignment does not always exist. Consider a two-lane AHS with $z_1 = 1, z_2 = 1$ and

$$\bar{N} = \begin{bmatrix} 1 & 1 & 0 \\ & 0 & 1 \\ & & 1 \end{bmatrix}$$

The only two possible assignments are shown in (a) and (b) in Figure 6.1. (a) is destination monotone, but not origin monotone. (b) is origin monotone, but not destination monotone. The reason that this happens is as follows. For an particular entrance (or exit), the available capacities might be such that we have no flexibility in partitioning, i.e., there is only one feasible partition. This will not affect the row (column) partitioning for destination (origin) monotoneness, but the resulting strategy might not be origin (destination) monotone.

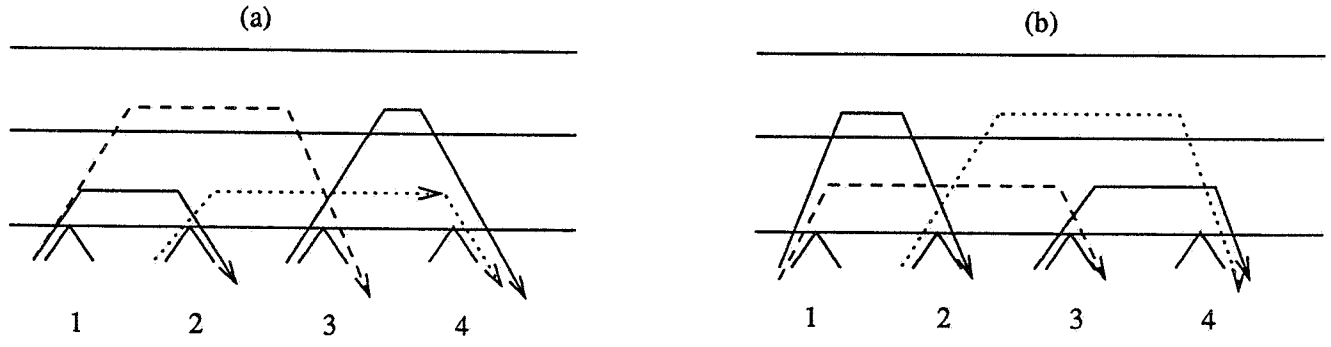


Figure 6.1: The only two feasible assignments

Theorem 6.1.2 *There exist algorithms that find the optimum partitioned lane assignment in polynomial time.*

Proof: First we must carefully define the variable in our problem. For any lane assignment problem, a window of length K_2 in terms of the sections on the highway, is considered. What varies from problem to problem is the flow of vehicles arriving at the entry nodes. For each entry node i define the sets,

$$C_i := \{1, 2, \ldots, N_{i,i+1}, N_{i,i+1} + 1, \ldots, N_{i,i+1} + N_{i,i+2}, \ldots, N_{i,i+1} + \ldots + N_{i,K_2}\}$$

Now, for each set C_i, we wish to find two members $C_i(n_1)$ and $C_i(n_2)$ such that

$$C_i(n_1) \leq z_1,\ C_i(n_2) - C_i(n_1) \leq z_2,\ \sum_{j=i+1}^{K_2} N_{i,j} - C_i(n_2) \leq z_3$$

Therefore, the π_i and δ_i split up the vehicles entering at node i into three subsets of sizes

$$s_i^1 = n_1, s_i^2 = n_2 - n_1, s_i^3 = \sum_{j=i+1}^{K_2} N_{i,j} - n_2$$

The number of possible ways of partitioning $\sum_{j=i+1}^{K_2} N_{i,j}$ into three subsets is

$$\begin{pmatrix} (\sum_{j=i+1}^{K_2} N_{i,j} + 2) \\ 2 \end{pmatrix}$$

The +2 term in the above arises because an assignment that places vehicles into only two lanes or only one lane is also a partitioning solution and must be included. Therefore, we

can enumerate all possible assignments (some infeasible) by listing sets whose elements are the 3-tuples (s_i^1, s_i^2, s_i^3), irrespective of the capacity constraints. Now, let $\max_{i,j} N_{i,j} = N_{max}$. Let us now find the maximum possible number of assignments. The maximum number of assignments for node 1 is

$$\begin{aligned} &\leq \begin{pmatrix} (N_{max}(K_2-1)+2) \\ 2 \end{pmatrix} \\ &= \frac{(N_{max}(K_2-1)+2)!}{(N_{max}(K_2-1))!\ 2!} \\ &\leq N_{max}^2(K_2+1)K_2/2 \end{aligned} \tag{6.1}$$

Therefore, the maximum number of total assignments A_{max} for a (K_1, K_2) system is

$$A_{max} \leq \frac{N_{max}^{2K_1}}{2}[(K_2+1).K_2.K_2.(K_2-1)\ldots 3.2] \tag{6.2}$$

From Equation(6.2) we see that the maximum number of solutions is polynomial in N_{max}. Therefore, even a naïve enumeration algorithm is polynomial in N_{max}. The properties of this search algorithm can be improved by more information, such as the exact nature of the cost function. □

Theorem 6.1.3 *(a) A partitioned solution does not necessarily obey Lemma 4.3.2.*
(b) A solution that obeys Lemma 4.3.2 for a (K, K) system is a partitioned solution.

Proof: (a) The fact that a strategy is partitioned simply dictates its behavior when two exits from the same entrance are compared or vice versa. It does not say anything about comparisons of two exits from two different entrances or vice versa.
(b) Let us prove that Lemma 4.3.2 for a (K, K) system implies destination monotoneness. We will use the following notation:
Let $Max(i,j)$ = the fastest lane that any of the N(i,j) travel in
and $Min(i,j)$ = the slowest lane that any of the N(i,j) travel in.
For a three-lane AHS, the functions Max and Min can take values of 1, 2 or 3. If Lemma 4.3.2 is obeyed,

$$Min(i, j_a) \geq Max(i, j_{a-1}) \quad i+2 \leq j_a \leq K_2, \quad 1 \leq i \leq K_1$$

$$Max(i,j) \geq min(i,j) \; implies$$

$$Min(i,i{+}1) \leq Max(i,i{+}1) \leq Min(i,i{+}2) \leq Max(i,i{+}2) \leq \ldots \leq Min(i,K_2) \leq Max(i,K_2)$$

$$\Rightarrow Max(i,i+1) \leq Max(i.i+2) \leq \ldots \leq Max(i,K_2) \quad 1 \leq i \leq K_1$$

Because $Max(i,j)$ can take values 1, 2 or 3 only (for a three-lane AHS), $\exists j_a$ and j_b such that

$$i \leq j_a \leq j_b \leq K_2$$

so that

$$j < j_a \Rightarrow Max(i,j) = 1, \quad j_a \leq j \leq j_b \Rightarrow Max(i,j) = 2, \quad j > j_b \Rightarrow Max(i,j) = 3$$

Then,

$$1 \leq Min(i,i+1) \leq Max(i,i+1) \leq Min(i,j_a-1) \leq Max(i,j_a-1) \leq 1 \tag{6.3}$$

$\Rightarrow$ Cars going to exits in the interval $[i+1, j_a-1]$ travel in L_1.

$$1 \leq Min(i,j_a) \leq Max(i,j_a) = 2 \tag{6.4}$$

$\Rightarrow N_{i,j_a}$ travel in lanes 1 or 2 or both.

$$2 = Max(i,j_a) \leq Min(i,j_a{+}1) \leq Max(i,j_a{+}1) \leq \ldots \leq Min(i,j_b) \leq Max(i,j_b) = 2 \tag{6.5}$$

$\Rightarrow$ Cars going to the exits in the interval $[j_a+1, j_b]$ travel in L_2.

$$2 = Max(i,j_b) \leq Min(i,j_b+1) \leq Max(i,j_b) = 3 \tag{6.6}$$

$\Rightarrow N_{i,j_b+1}$ travel in lane 2 or 3 or both.

$$3 = Max(i,j_b+1) \leq Min(i,j_b+2) \leq \ldots \leq Max(i,K_2) = 3 \tag{6.7}$$

$\Rightarrow$ Cars going to the exits in the interval $[j_b+2, K_2]$ travel in L_3. Equations (6.3) through (6.7) above prove that the strategy is destination monotone. The fact that the strategy is also origin monotone can be proved similarly.

Therefore, the condition that a strategy obey Lemma 4.3.2 is much stronger than that of it being a partitioned strategy. $\square$.

6.2 Parametric Study of the Optimal Strategies

Theorem 4.3.1 in Section 4.3 showed that the optimal solution for a $(1, K)$ system with constant maneuver costs was destination monotone, but examples show that, in general, the optimal assignment for a (K, K) system is not destination monotone. However, when maneuvering costs are weighted more, indicating increased maneuvering delays, by being dependent on the level of congestion, it may be hypothesized that order-preserving assignments such as destination monotone, origin monotone or monotone will reduce maneuvering cost and may be optimal. To understand the structure of the assignment when the maneuver costs are congestion dependent, the sensitivity of the optimal strategy to the maneuvering costs was studied. At issue was how the partitioned (or nonpartitioned) nature of the optimal strategy is related to the type of maneuvering cost assumed.

This was done by studying the optimal strategies obtained when the cost in (4.1) is replaced by the cost

$$T_{total} = T_{ss} + q.T_{maneuver} \tag{6.8}$$

and the common multiplier q of the maneuver cost constants was varied. The use of first-order sensitivities with respect to q is not sufficient (though computationally feasible) because the effect of *large* changes in q on the optimal assignment must be studied. Calculating sensitivities to a parameter at a point assumes that the set of active constraints at that point is invariant to changes in that parameter, an assumption that would undoubtedly not be satisfied for large changes in q.

To study the effect of increasing maneuver costs, reflected in larger values of q, on the partitioned nature of the solution with respect to the maneuver cost q, we require the notion of partitioning costs, i.e., costs that arise as a result of ρ not being partitioned.

Definition 6.2.1 *Split the travel time cost function $J(\rho)$ into two components $\bar{J}_1(\rho)$ and $\bar{J}_2(\rho)$ such that*

$$J(\rho) = \bar{J}_1(\rho) + \bar{J}_2(\rho)$$

where

$$\bar{J}_2(\rho) = 0 \text{ iff } \rho \text{ is partitioned}$$

Then, $\bar{J}_2(\rho)$ is called the partitioning cost.

We will first relate partitioning cost to path intercepts and then extract the path intercept information from the maneuvering costs. To quantify the partitioning costs arising from path intercepts, consider a simple (1,3) system. If the lane assignment is required to conform to being destination monotone (possibly with splitting), the following Boolean constraint must be satisfied.

$$(\rho^3_{1,2} > 0 \text{ AND} \rho^2_{1,3} > 0)\ OR\ (\rho^3_{1,2} > 0 \text{ AND } \rho^1_{1,3} > 0)\ OR\ (\rho^2_{1,2} > 0 \text{ AND } \rho^1_{1,3} > 0) = \phi \quad (6.9)$$

This implies

$$(\rho^3_{1,2} = 0\ OR\ \rho^2_{1,3} = 0)\ AND\ (\rho^3_{1,2} = 0\ OR\ \rho^1_{1,3} = 0)\ AND\ (\rho^2_{1,2} = 0\ OR\ \rho^1_{1,3} = 0) \neq \phi \quad (6.10)$$

because the complement of $\rho^m_{i,j} > 0$ is $\rho^m_{i,j} = 0$. The above is equivalent to

$$g(\rho) \triangleq \rho^3_{1,2}\rho^2_{1,3} + \rho^3_{1,2}\rho^1_{1,3} + \rho^2_{1,2}\rho^1_{1,3} = 0 \quad (6.11)$$

Therefore,

$$g(\rho) = [\rho^1_{1,2}\ \rho^2_{1,2}\ \rho^3_{1,2}\ \rho^1_{1,3}\ \rho^2_{1,3}\ \rho^3_{1,3}] \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .5 & 0 & 0 \\ 0 & 0 & 0 & .5 & .5 & 0 \\ 0 & .5 & .5 & 0 & 0 & 0 \\ 0 & 0 & .5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \rho^1_{1,2} \\ \rho^2_{1,2} \\ \rho^3_{1,2} \\ \rho^1_{1,3} \\ \rho^2_{1,3} \\ \rho^3_{1,3} \end{bmatrix} = \rho^T A_r \rho$$

The constraint in Equation (6.9) was formed by considering those intersections of paths that are forbidden under the destination monotone requirement. Note that the $g(\rho)$ is equal to zero only if the solution ρ is destination monotone. Note also that $g(\rho)$ is quadratic in ρ. The quadratic nature of the $g(\rho)$ arises out of a intersection of two paths and is thus independent of other characteristics of the highway such as number of lanes or number of sections. Construction of $g(\rho)$ for a (K_1, K_2) highway can be done in a fashion similar to that done above by considering all of the Boolean constraints that must be satisfied in order for the solution to be destination monotone. Because these must be simultaneously satisfied,

$g(\rho)$ will be of the same quadratic form as before. Along similar lines, one can construct a constraint function $h(\rho)$ similar to $g(\rho)$ that arises out of path intercepts forbidden by origin monotone policies. It will also be quadratic in nature. The equivalent constraint function for monotone strategies will simply be a sum of those for destination and origin monotone strategies.

We will now use $g(\rho)$ and $h(\rho)$ to calculate the partitioning cost for the QP problem. Recall that for the QP problem, the cost function for travel and maneuver time minimization was also quadratic in nature. The quadratic nature of that cost arose out of the maneuvering cost placed according to the congestion of the highway. It was, in effect, a cost placed on the intersection of all paths taken by all vehicles. It therefore incorporates within it costs proportional to $g(\rho)$ and $h(\rho)$. The criterion in the nonlinear optimization problem, which was of the form,

$$q\rho^T Q_1 \rho + Q_2 \rho$$

can be decomposed as

$$\{\underbrace{q\rho^T A\rho + Q_2\rho}_{J_1(\rho,q)} + \underbrace{q\rho^T A_r\rho}_{qg(\rho)} + \underbrace{q\rho^T A_c\rho}_{qh(\rho)}\} \tag{6.12}$$

where

$$A + A_r + A_c = Q_1$$

We can view Equation (6.12) as arising out of a multi-objective optimization problem with the individual objectives being a cost $J_1(\rho)$, the destination monotone partitioning cost $\rho^T A_r \rho$ and the origin monotone partitioning cost $\rho^T A_c \rho$, with the latter two costs, denoted by $J_2(\rho)$, being weighted by a parameter q. (The parameter q enters into $J_1(\rho)$ as well, but it cannot be factored out as a weight.) The QP problem may thus be viewed as follows: Determine

$$\min_{\rho}\{J_1(\rho, q) + qJ_2(\rho)\} \tag{6.13}$$

subject to

$$\mathbf{A_1}\rho = \mathbf{b_1}, \quad \mathbf{A_2}\rho \leq \mathbf{b_2}, \quad \rho \geq 0$$

We hypothesize that the optimal strategy will become monotone by increasing the weight q on the cost $J_2(\rho)$. Recall that q was defined as the greatest common factor of the maneuvering

cost constants of proportionality C_1^a, C_2^a, C_3^a, C_1^b, C_2^b and C_3^b. By increasing q, a penalty function type effect will be achieved such that the optimal solution of the QP problem will tend to be monotone. In other words, the developed formulation of the lane assignment problem on a congested highway will tend to produce a monotone strategy as the optimal strategy that minimizes travel and maneuver time. The fact that $J_1(\rho, q)$ will also increase with q simply reflects the fact that travel time will increase with congestion even with zero partitioning costs. Confirmation of this hypothesis was provided in the case study (see Section 7.2.2).

6.2.1 Suboptimal partitioned strategies

The question of "nearness" of an optimal strategy to a monotone strategy becomes important because while monotone strategies are desirable, optimal strategies may not be monotone. The problem of determining a suboptimal monotone strategy (possibly with splitting) was considered using two approaches. The first concentrated on minimizing a criterion reflecting only the cost of violating a monotone assignment; the second considered finding a monotone strategy in the neighborhood of the optimal strategy. The approaches, along with the associated algorithms, are presented below for destination monotone assignments. The performance of these algorithms was measured by calculating the extra cost incurred, in terms of total travel time, by using the suboptimal partitioned assignment instead of the optimal nonpartitioned one. Changes or extensions to these algorithms to obtain origin monotone or nearly monotone solutions can be easily made.

Approach 1

To find a destination monotone assignment that is "near" an optimal assignment ρ^*, consider the minimization problem:

$$\min_{\rho} \rho^T A_r \rho$$

subject to

$$\mathbf{A_1}\rho = \mathbf{b_1}, \quad \mathbf{A_2}\rho \leq \mathbf{b_2}, \quad \rho \geq 0$$

To find solutions in the neighborhood of the optimal solutions of Equation (6.13), solve the above minimization problem with the initial feasible point being ρ^*.

The cost function in the above problem is the destination monotone partitioning cost. It is equal to zero when the assignment ρ is destination monotone. Because there exists at least one destination monotone solution (see Theorem 6.1.1), the optimum cost is equal to 0. Because A_r is not strictly positive definite, a number of local minima exist. The rationale behind choosing the initial feasible point to be ρ^* is that the gradient projection algorithm will converge to a local optimum (a destination monotone solution) that is close to ρ^* (a nonpartitioned solution). The use of this algorithm to find an alternate lane assignment $\bar{\rho}$ indicates a willingness to sacrifice the travel time optimality of ρ^* in order to obtain a destination monotone solution.

The travel time corresponding to $\bar{\rho}$ may be unacceptably large however and in such cases, we seek a compromise between minimizing travel time and getting a partitioned assignment. Approach 2 described below provides a means to achieve this compromise.

Approach 2

Once again, we would like to find a destination monotone assignment that is "near" an optimal assignment ρ^*. This time, however, we are stricter in our requirement of "near" solutions. Let $P(I, \rho)$ represent the partitions for the set of entrances I under assignment ρ. Let I_1 be the set of entrances in the OD matrix N such that the assignments in ρ^* satisfy the destination monotone requirement. Let $P_1(I_1, \rho^*)$ and $P_2(I_1, \rho^*)$ be the set of partitions associated with I_1. Let I_2 be the set of remaining entrances. (To clarify the above, consider the illustrative example in Chapter 7. For the assignment ρ_{ex} in Equation (7.1), $I_1(\rho_{ex}) = \{2, 3, 4\}$, $I_2(\rho_{ex}) = \{1\}$ and $P_1(I_1, \rho_{ex}) = \{4, 4, 5\}$ and $P_2(I_1, \rho_{ex}) = \{5, 5, 5\}$). We would like to find an assignment $\bar{\rho}$ that is destination monotone with the additional requirement that the partitions for I_1 in $\bar{\rho}$ remain close to $P_1(I_1)$ and $P_2(I_1)$. (Note that there does not always exist a destination monotone $\bar{\rho}$ such that the partitions for I_1 remain $P_1(I_1)$ and $P_2(I_1)$.) Then, $\bar{\rho}$ is the solution to the following problem

$$\min_{\rho} \rho^T A_r \rho + w(P(I_1, \rho) - P(I_1, \rho^*)$$

subject to

$$\mathbf{A_1}\rho = \mathbf{b_1}, \quad \mathbf{A_2}\rho \leq \mathbf{b_2}, \quad \rho \geq 0$$

with the initial feasible point being ρ^*.

The term $(P(I_1, \rho) - P(I_1, \rho^*)) = \delta P$ is linear in ρ (see Equation (6.14)), and $w\delta P$ represents the requirement that the partitions for I_1 remain nearly the same. It is given by the following equations.

$$\begin{aligned}
\sum_{i+1}^{P_1(i)-1} (\rho^2_{i,j} + \rho^3_{i,j}) &= 0 \\
\sum_{P_1(i)+1}^{P_2(i)-1} (\rho^1_{i,j} + \rho^3_{i,j}) &= 0 \\
\sum_{P_2(i)+1}^{K_2} (\rho^1_{i,j} + \rho^2_{i,j}) &= 0 \\
\rho^3_{i,P_1(i)} &= 0 \\
\rho^1_{i,P_2(i)} &= 0
\end{aligned} \tag{6.14}$$

This requirement has been introduced as a soft constraint by placing it in the objective function weighted by a weight w, rather than as a hard constraint. The rationale behind this is that placing it as a hard constraint would guarantee $P(I_1, \rho^*) = P(I_1, \bar{\rho})$, but then $\bar{\rho}$ might not be destination monotone. While placing it as a soft constraint does not guarantee that we will get a destination monotone assignment, it indicates a willingness to sacrifice some closeness of partitions in the hope of getting a destination monotone solution.

Note that both the above algorithms require the solution of a quadratic programming problem. As before, because we do not have convexity, we cannot guarantee the existence of a unique global minimum. The optimizer that had been developed for solution of the nonlinear assignment problem can be (and was), therefore, used here as well.

A number of sample problems for a $(10, 10)$ system were tested using the above algorithms. In every case the suboptimal destination monotone solution cost less than 2% more, in terms of travel time, than the optimal solution.

In addition to the above approaches, the following algorithm supplied by an anonymous reviewer for [37] may also be considered. It includes the requirement that the suboptimal strategy be "close" to the optimal strategy ρ^* in the cost function as follows.

$$\min_{\rho}(\rho^T A_r \rho + w(\rho - \rho^*)^T(\rho - \rho^*))$$

subject to

$$\mathbf{A}_1\rho = \mathbf{b}_1, \quad \mathbf{A}_2\rho \leq \mathbf{b}_2, \quad \rho \geq 0$$

For sufficiently low weight w, this approach will provide a monotone strategy $\bar{\rho}$, and will simultaneously keep the distance between $\bar{\rho}$ and ρ^* small. This may be more reliable than using a initial feasible point ρ^*, which does not guarantee that $\bar{\rho}$ will be close to ρ^*.

CHAPTER 7

CASE STUDY

7.1 Problem Description

A hypothetical automated highway with three lanes for automated driving and with 20 entry and 20 exit points was used to study the properties of lane assignment strategies and the algorithm described in the previous section. The origin/destination matrix N was assumed to change during the day, and so the system is defined by $N(t)$, where $t = 1, \ldots, T$. For each t, $N(t)$ is a KxK matrix with zero elements on and below the main diagonal. The rows of N denote entry points, and the columns of N denote the exit points.

The origin/destination data was defined in hourly increments, and was defined by the relationship

$$N(t) = N0 + F(t)\Delta N$$

where the KxK matrix $N0$ defined a base loading independent of time. The second term represents the time-dependent component and was modeled as a product of a diagonal KxK matrix whose diagonal elements were less than or equal to one and represented the time variation of origin destination flows. The KxK matrix ΔN represented the magnitude of the variation. The study simulated hourly variations of N from 6 a.m. ($t = 1$) to 9 p.m. ($t = 16$), and the diagonal element $F_{ii}(t)$ represented the time variation of the traffic flow onto the AHS at entry i at time t. While a different $F_{ii}(t)$ function for every section would be used in reality, here only three different functions were used. The first six diagonal elements of F

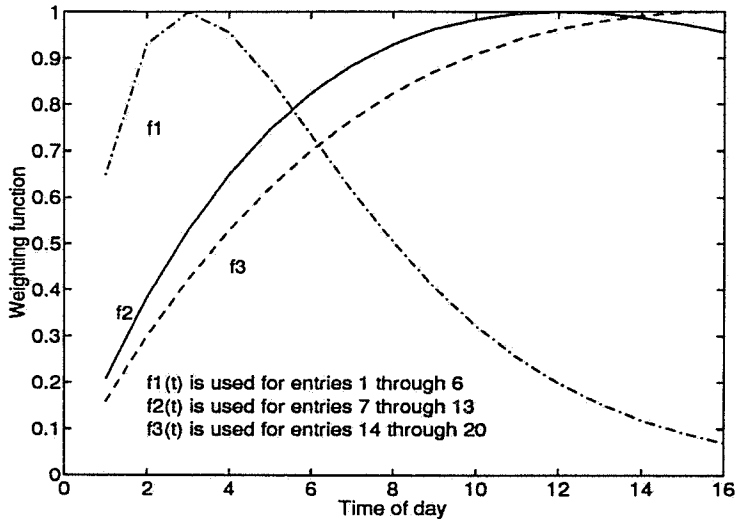


Figure 7.1: Weighting factors used in the case study

were defined using a selected function $f_1(t)$, the next seven using a function $f_2(t)$, and the last seven elements were defined using $f_3(t)$. The weighting functions $f_j(t)$, $j = 1, 2, 3$ are shown in Figure 7.1. The implication is that the traffic volume at the first six entry points exhibits a marked morning peak, with the maximum occurring at 9 a.m. ($t = 4$), that the next seven entry points have a less pronounced afternoon peak, with the maximum occurring at 5 p.m. ($t = 12$), while the last 7 entry points exhibit a gradual increase in volume towards the late afternoon. It is clear that in a more realistic study, each element of F could be considered a separate time-weighting function, with the matrix multiplication replaced by the element-wise multiplication of elements of F with the corresponding elements of ΔN.

As indicated earlier, the total traffic flow in sections is independent of the lane assignment strategy and depends only on the origin/destination data. We have studied various traffic distributions and present the results from one of them. In this example, the morning peak is most pronounced, and the afternoon peak occurs at different sections of the highway than does the morning peak. Furthermore, in the example the total highway capacity is assumed to be constant throughout the sections. The presented results are for $C = 20000$ veh/hour, which roughly implies 111 veh/min/lane. However, in view of the different lane speeds ($v_1 =$

90 km/hour, $v_2 = 100$ km/hour and $v_3 = 110$ km/hour), two different sets of lane capacities were assumed, namely, $z_1 = 5850$, $z_2 = 6900$ and $z_3 = 7250$, and $z_1 = 5000$, $z_2 = 6500$ and $z_3 = 8500$. This is about four times higher than the currently observed maximal capacities of 2200 veh/lane/hour on the busiest existing highways. This reflects the increase in lane capacities expected from the AHS component of IVHS [4].

A smaller section of the highway with ten sections was also studied in analyzing the performance of algorithms. These sections were basically the first ten sections of the larger portion of highway described above. An example data file for the 20-section highway is given in Appendix A.

7.2 Minimization of Travel Time

7.2.1 Illustrative example

To illustrate the nature of the results, consider first a completely fictitious (and unrealistic) example. Let us suppose that the following data represent a (4,5) highway system.

$$\bar{N} = \begin{bmatrix} 0 & 5 & 8 & 7 & 8 \\ 0 & 0 & 7 & 5 & 7 \\ 0 & 0 & 0 & 6 & 6 \\ 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Lane capacities $z_1 = 20$, $z_2 = 20$, $z_3 = 13$

Lane Speeds $v_1 = 70$ km/hour, $v_2 = 85$ km/hour, $v_3 = 100$ km/hour

Distance of exits 1, 2, 3, 4, 5 and 6, respectively, from the entrance = 0 2 5 7 10 14 km

With constant maneuver costs, the linear cost problem resulted in the following lane assignments

$$S = \begin{bmatrix} 0 & (0,5,0) & (0,2,6) & (0,7,0) & (0,1,7) \\ 0 & 0 & (7,0,0) & (2,3,0) & (0,7,0) \\ 0 & 0 & 0 & (4,2,0) & (0,0,6) \\ 0 & 0 & 0 & 0 & (0,6,0) \end{bmatrix} \tag{7.1}$$

The first member of the triplet for the (i,j)th element of S is the flow of vehicles going from i to j assigned to the slowest lane L_1, the second member is that assigned to L_2 and the third is that assigned to L_3. Note that this is a constant lane assignment with splitting, but it is not destination monotone or origin monotone. For example, the assignment from entrance 1 is not destination partitioned. This is because some vehicles from 1 to 3 travel in L_3, but the slowest lane used by vehicles traveling from 1 to 4 is only L_2. However, the assignments from entrances 2, 3 and 4 are destination partitioned. For example, vehicles going from entrance 2 to exits closer than 4 travel in L_1, and those going to exits beyond 4 travel in L_2, those going to exit 4 travel in either L_1 or L_2.

The assignment resulting from the nonlinear cost criterion with maneuver cost constants equaling

$$C_1^a = 0.0001,\ C_1^b = 0 \qquad C_2^a = 0.0001,\ C_2^b = 0 \qquad C_3^a = 0.0001,\ C_3^b = 0$$

is

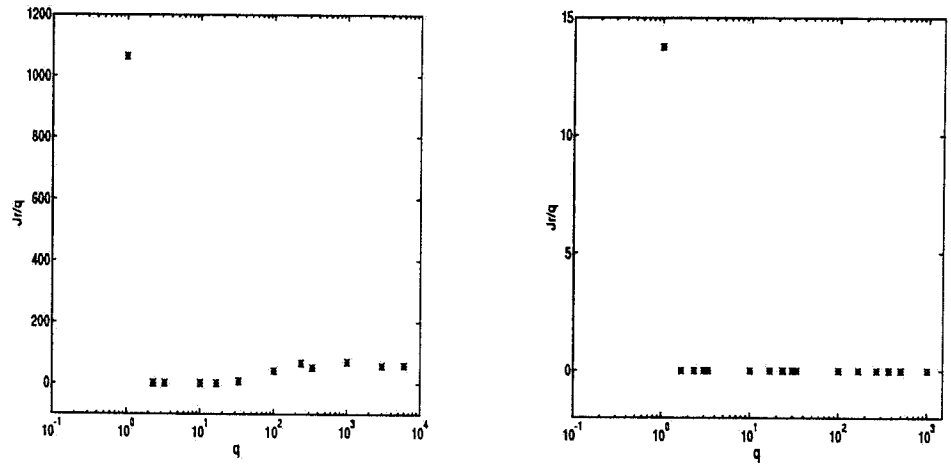
$$\rho = \begin{bmatrix} 0 & (5,0,0) & (8,0,0) & (0,6.2,0.8) & (0,0,8) \\ 0 & 0 & (7,0,0) & (0,5,0) & (0,2.8,4.2) \\ 0 & 0 & 0 & (6,0,0) & (0,6,0) \\ 0 & 0 & 0 & 0 & (5.3,0.7,0) \end{bmatrix}$$

An examination of this assignment reveals that it is destination monotone.

7.2.2 Effect of maneuver costs on partitioning costs

In this section, we present selected results from our studies. These results correspond to two origin destination matrices for a (20,20) system representing traffic conditions at 9 a.m. and 3 p.m. [20]. For each time, the optimal assignment was determined for a sequence of increasing values of q. As can be seen in the simple example given at the beginning of this section, the optimal strategies resulting from constant maneuver costs will not, in general, be partitioned. For each optimal assignment, we determined, using Equation (6.12) the destination monotone and the origin monotone partitioning costs that arose out of path intercepts forbidden by destination monotone and origin monotone strategies. If this cost was zero, it indicated that the optimal strategy was a destination (origin) monotone strategy. The

destination monotone partitioning cost was calculated as $q\rho^T A_r \rho$, and the origin monotone cost was calculated as $q\rho^T A_c \rho$, as defined earlier in Equation (6.12).



Time = 9 a.m. Time = 3 p.m.

Figure 7.2: Change in destination monotone partitioning cost with q

Figure 7.2 shows the variation of the destination monotone partitioning cost $\rho^T A_r \rho$ (shown as '*'s) with q. It indicates the general trend in destination-monotone and origin-monotone partitioning costs with respect to q . The costs are high for small values of q and form a significant proportion of the total cost, signifying that for optimal assignments obtained by not weighting maneuvering costs, the actual maneuvering costs will be large. We observed in most cases a value of $q = qmin$ such that for q greater than $qmin$, the optimal solution is either destination (origin) monotone or close to it. This can also be seen in the results shown in Figure 7.2. The "closeness" to being partitioned was measured by how close the quadratic partitioning cost was to zero. This trend was also observed in the (11,11) system for various lane capacities. This behavior seems reasonable and indeed was expected. Our rationale in formulating the QP was that when maneuver costs are taken into account, as they should be for a congested highway, the optimal solution would tend to reduce the number of path intersections that give rise to the maneuver costs. Furthermore,

accounting for maneuvering costs in the criterion eliminates the need for additional logical constraints when solving the problem in the class of partitioned assignments.

7.2.3 Effect of maneuver costs on partitions

As noted above, once q was above a $qmin$, the optimal solution tended to remain partitioned or close to it. This enabled us to study the movement of the partitions with an increase in q. Figures 7.3(a) through (d) show the destination monotone partitions for four different q for the (20,20) system. The figures are to be interpreted as follows. Consider any entrance i. Locate it on the Y axis of one of the figures. Now determine where a horizontal line extending from i in the +X direction would intersect the $P1$ and the $P2$ curves. Call these intersections $P1(i)$ and $P2(i)$. This means that vehicles traveling from entrance i to exits closer than $P1(i)$ would travel in lane 1, those traveling to exits between $P1(i)$ and $P2(i)$ would travel in lane 2 and those going beyond $P2(i)$ would travel in lane 3. Those vehicles traveling from entrance i to $P1(i)$ would travel in either lane 1 or lane 2, while those to $P2(i)$ would travel in lane 2 or lane 3. The value of q increases as one goes from Figure 7.3 (a) to (d). The figures indicate a movement of the partitions to the right as q is increased. A movement of the partitions to the right implies that more and more vehicles tend to stay in the slower lanes. This is reasonable because this means that the vehicles avoid incurring the cost of maneuvering to the faster lanes as q increases. It is to be noted, however, that while this trend was observed in a number of the experiments, it was not observed in all, because optimal partitions will also depend on the distribution of vehicles in the origin/destination matrix.

7.2.4 Performance loss with suboptimal partitioned assignments

As mentioned earlier, the ease of implementation of partitioned assignments and reduction of path intercepts implied by such assignments makes them an attractive alternative to non-partitioned optimal solutions. Approaches 1 and 2 described in Chapter 6 were applied to the (20,20) and to the (11,11) systems for weightings of maneuver costs that produced non-

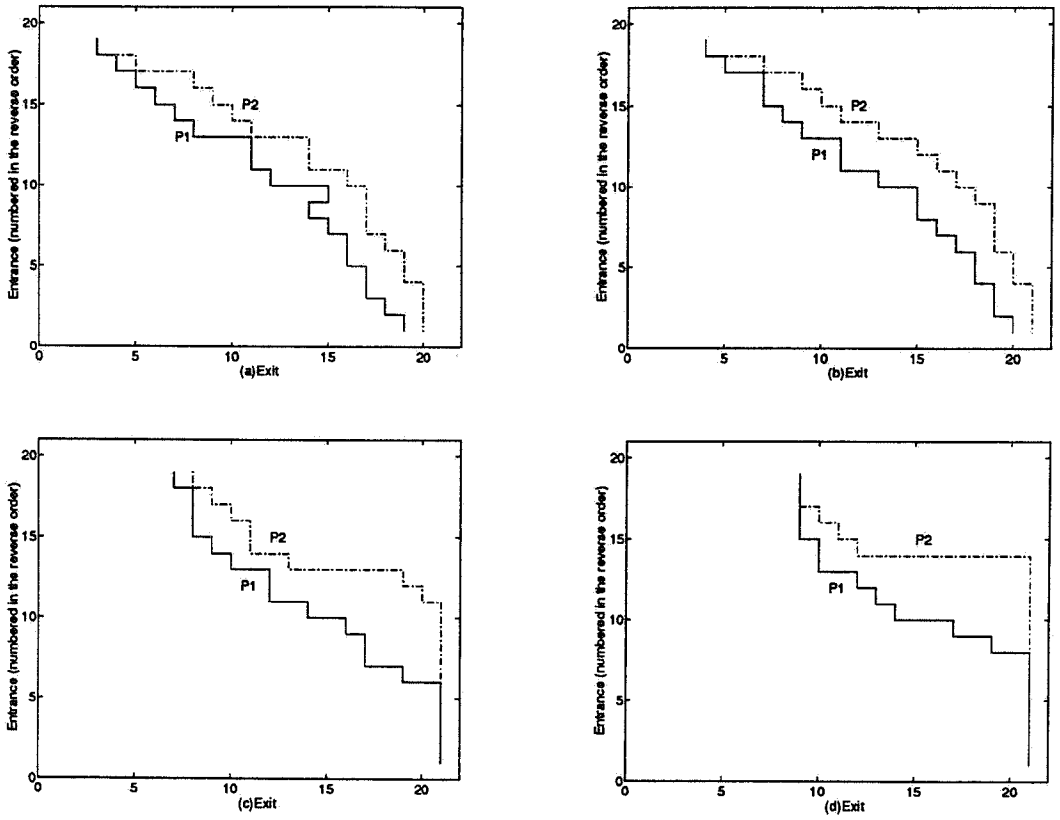


Figure 7.3: Change in partitions with increase in q

partitioned optimal assignments. The largest percentage increases in cost of the partitioned assignment over the nonpartitioned assignment are given in Table 7.1.

As Table 7.1 shows, the increase in travel time cost for all of the various partitioning cases was always less than 2%. This implies that for the highway segments considered, we can find partitioned assignments that are not very far from optimal. This supports the view that for systems that are representative of actual highway systems, partitioned assignments could be optimal or near optimal.

We also calculated the difference in partitions between the corresponding partitioned destinations (origins) of the nonpartitioned and of the partitioned assignments. The average percentage changes are tabulated in Table 7.2. Table 7.2 shows that, in general, the difference in partitions with Approach 2 is smaller or nearly equal to that with Approach 1. This is to

Table 7.1: Loss of performance

Largest % increases in cost with					
Origin Monotone		Destination Monotone		Monotone	
Algorithm1	Algorithm2	Algorithm1	Algorithm2	Algorithm1	Algorithm2
0.5854	0.5622	0.3296	0.4338	1.0819	1.0164

Table 7.2: Change in partitions

Average % differences in partitions with					
Origin Monotone		Destination Monotone		Monotone	
Algorithm1	Algorithm2	Algorithm1	Algorithm2	Algorithm1	Algorithm2
6.2	3.3	5.2	5.2	11.2	11.3

be expected because Approach 2 has this difference in partitions as one of the terms in its objective function.

7.3 Balancing Excess Capacities

7.3.1 Implementation of the lane assignment algorithm

The lane assignment algorithm was coded as a Matlab program and was run for the different OD matrices corresponding to the 16 time periods on a Sparc2 workstation to compute the nearly optimal partitions P_1 and P_2. Recall that for a given $N(t)$, the two partition vectors P_1, P_2 completely define lane usage and excess lane capacities as described earlier. Vehicles traveling to exits 1 through $P_1(i)$ use lane L_1, vehicles traveling to exits $P_1(i)+1$ through $P_2(i)$ use lane L_2 and vehicles traveling to exits $P_2(i)+1$ through K use lane L_3. Various types of partitions were considered for the initialization of the algorithm. They are defined below.

1. **A nominal partition:** This is a partition of the OD matrix that utilizes all three lanes for the initial placement of the vehicles. This partition divides the OD matrix into three nonempty bands, one traveling in L_1, one in L_2 and one in L_3.

2. **An extreme partition:** This is a partition of the OD matrix that utilizes only one lane for the initial placement of the vehicles. This is an extreme case of the nominal partition in which two of the bands are empty.

3. **A previously optimal partition:** The initial partition of the OD matrix used here in the placement of the vehicles is the nearly optimal partition obtained by the assignment of vehicles in the immediately preceding time period. The initial partition used for the first time period is a nominal or an extreme partition. The rationale for this is that if the OD matrix changed little from one period to the next, then convergence could be achieved quickly if the algorithm used the previously nearly optimal partition as the initial partition.

For each of the above initial partitions, nearly optimal partitions were computed for the various time periods using the developed algorithm.

7.3.2 Discussion of the results

Selected results are presented here to illustrate the convergence of the algorithm, the deviation of the resultant excess capacities from the ideal and the change in partitions over time, ensuing from the maximization of the selected criterion.

7.3.2.1 Convergence

In every studied case, the processing took no more than two minutes (real-time) on the Sparc2 workstation, indicating that the algorithm is not computationally intensive and can be used in real time even on smaller time intervals than the one-hour interval used in this case study. Moreover, convergence occurred in less than 150 iterations confirming the relevance of the developed estimate in Equation (5.12) for the average number of iterations to convergence.

Figure 7.4 shows the convergence of the excess capacities for the 16 time periods. It highlights the uniform nature of the convergence process for diverse origin destination data. The initial negative value of the excess capacity is, of course, the consequence of the arbitrary

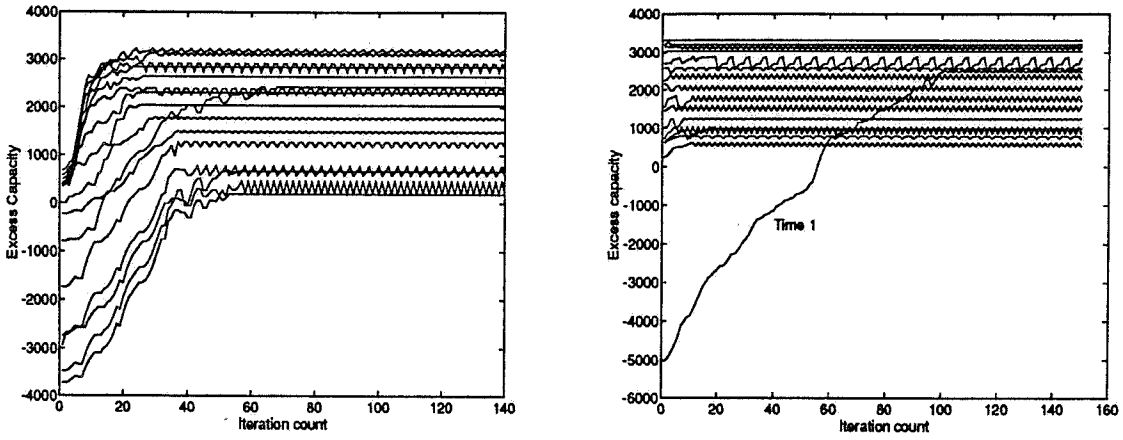


Figure 7.4: Algorithm convergence

selection of the initial lane partitioning strategy. A negative value, or value close to zero after convergence is reached, is an indication of bottlenecks occurring within the section where such a small excess capacity occurs. Figure 7.4(a) shows the convergence results for the case in which for each time period, the initial partition used was the nominal partition. Note that in every case convergence is reached in less than 80 iterations. Figure 7.4(b) shows the convergence for the case in which the initial partition used for the very first time period (Time 1) was an extreme partition with all vehicles in L_3. Convergence took longer here than in Figure 7.4(a), about 110 iterations. The initial partition for the succeeding time periods was the nearly optimal partition obtained in the previous time period. Convergence is achieved very rapidly in these cases, indicating that the previously optimal partition is a good choice for the initial partition.

7.3.2.2 Percentage deviation from ideal

For every studied case, we calculated the difference in the obtained excess capacity from the ideal excess capacity defined by Equation (5.7). Figure 7.5 shows the percentage deviation of the excess capacity from ideal for four initial partitions. For the first type of initial partition, all vehicles in L_1; for the second, all vehicles in L_3; the third type of initial partition was a

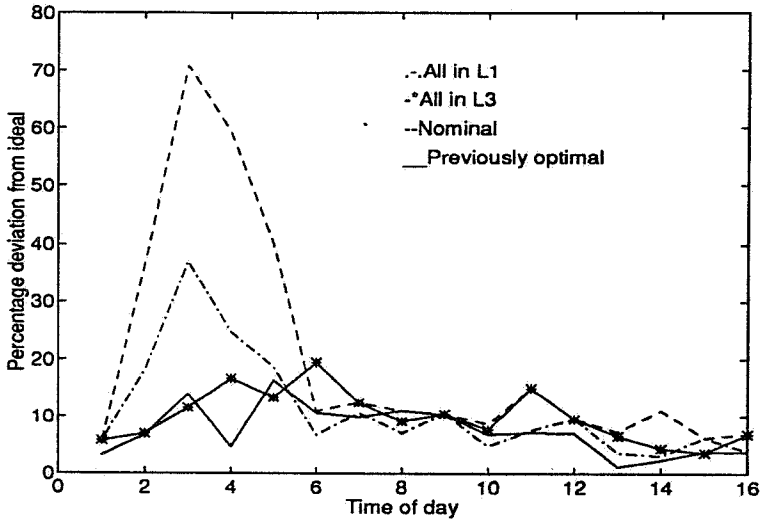


Figure 7.5: Percentage deviation from ideal

nominal partition and the fourth type of initial partition was the previously optimal partition which started with all vehicles in L_3. These are plotted for the 16 time periods. The following inferences can be drawn from the figure.

1. Different initial conditions result in different final excess capacities and partitions. This indicates that the local nature of the algorithm will stop at points that are at local optima, and that the problem has (in fact many) local maximizing points. (Local here is taken to mean in the neighborhood of a nominal partition, and a neighborhood is defined as consisting of partitions that differ from the nominal partition by one element of P_1 or P_2).

2. The percentage deviation from ideal obtained for the initial partition type being the optimal partition from the previous time period, is as low at 1.2%, which means that the obtained excess capacity is only 1.2% off from the ideal value. This implies that constraining strategies to partitioning strategies need not be conservative.

3. The smallest deviation from ideal was observed when the initial partition type was the optimal partition from the previous time period. With such an initialization, the

largest deviation from ideal was about 15%, compared to the 20%, 35% and 70% obtained when the initial partition was such that (a) all the vehicles were in L_3, (b) all vehicles were in L_1 and (c) a nominal partition, respectively. Once again, this indicates that the previously optimal partition is a good choice for an initial partition, if the percentage deviation from ideal is small for the first time period and if the OD matrix does not change drastically from one time period to the next.

In cases in which the origin destination matrix changes drastically from one time period to the next, it might be better to initiate the algorithm using an extreme partition. This is because the optimal strategy for the previous time interval may get jammed in a local extremum, while the origin/destination distribution has changed sufficiently so that a better partition may exist far from the previous optimal partition.

7.3.2.3 Change of partitions with time

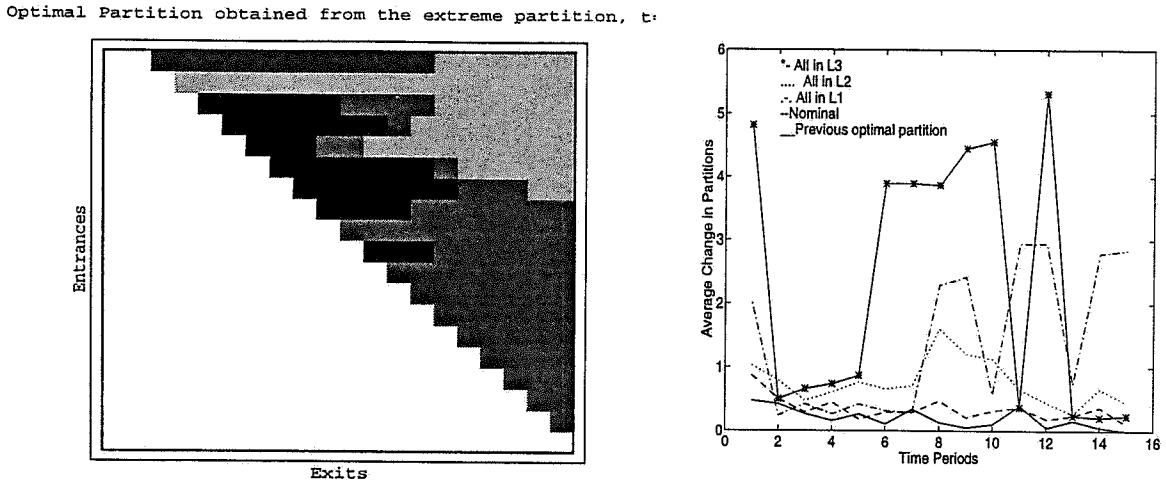
A sample resultant partition is shown in Figure 7.6(a). The average change in the nearly (optimal) partitions with time was studied. The average change in the partitions from one time period to the next was calculated using the formula

$$\text{Average change(t)} = (mean(|P1(t+1) - P1(t)|) + mean(|P2(t+1) - P2(t)|))/2$$

The results obtained for initializing the algorithm from five different initial partitions are shown in Figure 7.6. The largest changes in partition observed from one time period to the next were observed when the algorithm was initialized for all vehicles in L_3. The smallest changes were observed when the algorithm was initialized with the previously optimal partition, which itself was initialized with all vehicles in L_3. This indicates that in situations in which the rates of change of traffic volumes are small, the strategy may not have to be modified even if there is a better strategy, if the improvement, as measured by the performance criterion is small. This would alleviate the time boundary effects mentioned in Chapter 2.

7.3.3 Reducing the time boundary effect

The time boundary effect described in Chapter 2 results in additional lane changes that interrupt the smooth flow of traffic. Our choice of using partitioned strategies is beneficial in



(a) Sample resultant partition

(b) Change in partitions

Figure 7.6: Average change in partitions with time

mitigating the effects of the time boundary effect. Loosely speaking, if the OD matrix does not change significantly from one time period to the next, neither will the optimal partitions obtained from one period to the next. As mentioned in Section 7.3.2.3, this was seen in the case study when the initial partition used was the previously optimal partition. If the shift of the partitions to the left or right from one period to the next is small, so is the shift in traffic flow from one lane to the next. Therefore, the number of lane changes required will also be small.

7.3.4 Computational requirements

The algorithms discussed here were coded in Matlab and were used to find the lane assignment for various OD matrices. The nonoptimized Matlab program running on a Sparc2 workstation took no more than two minutes for finding the lane assignments for various test OD matrices. In all of the test cases, the solutions converged in less than 150 iterations. This computational speed is more than sufficient to handle OD matrix updates done every hour, or even every quarter of an hour.

From these and other examples, it appears that the developed "greedy" lane partitioning algorithm is an effective means of converging towards the optimal lane assignment strategies.

That there is excess capacity that cannot be allocated in the optimal manner is a consequence of partitioning, because no traffic volume from any entry point to any exit point is allowed to be split between two or more lanes. This leads to an allocation process that necessarily operates with quantized numbers and optimal solutions below those that are achievable with arbitrary splitting of traffic flows. However, the rationale is that partitioned lane assignment strategies are easy to implement and demand the least information exchange between the AHCC via the roadside beacons and the individual vehicles. Of course, if traffic volumes are so large that the excess capacity is considered too small, or is even negative, then splitting particular flows (from a given entrance to a given exit) may have to be resorted to. This can be achieved by using a "randomized" lane assignment strategy at the relevant entry point, whereby a vehicle traveling to the relevant exit would be assigned a lane by some probabilistic mechanism. This mechanism would decide the lane to be used by each vehicle, with the probability distribution ensuring that for large numbers of vehicles a split close to the desired is achieved.

Another point to note is that the presented algorithm is also applicable to the problem of lane assignment on AHSs with nonuniform capacity along the sections and is capable of handling capacity reductions from vehicles remaining in the AHS from a previous time period, due to prescheduled maintenance or from accidents.

CHAPTER 8

DESIGN OF AN AHS SIMULATOR

8.1 Need for an AHS Simulator

The opportunity offered by the AHS is that automated driving can be organized to conform to stringent rules of operation. Combined with the use of system-wide information to determine lane assignments, known vehicle characteristics, AVCS characteristics and communication protocols, AHS should enable more efficient and better organized traffic flows. The crucial advantage expected of the AHS is its ability to provide a significant increase in highway capacity with improved safety, while servicing from six to eight thousand vehicles per lane per hour in a multi-lane AHS with many entry and exit points.

Three factors will have a crucial effect on the throughput of the AHS: (1) The scheduling of vehicles to lanes on the AHS, (2) the architecture to provide communication protocols required for scheduling and control of vehicles on the AHS, and (3) the actual traffic dynamics established on the AHS and its effect on throughput. The traffic dynamics, and in particular the macroscopic speed-flow characteristics and throughput, will be significantly affected by the lane assignments obtained from the scheduling algorithm, by the characteristics of the control systems in the vehicles and by the characteristics of the communication protocols. (These protocols define the actual time needed to communicate and carry out a continuous stream of maneuvers as vehicles attempt to reach their assigned lanes of travel or maneuver towards their exits.) Thus, the capacity of the AHS and, consequently, the benefits of the

AHS will depend greatly on these factors. However, these expected benefits cannot be tested presently or in the foreseeable future on any physical environment.

This limitation has motivated the development of a framework to create a simulator for studying traffic flow on the AHS under various scenarios. The simulator will enable the study of flow on the AHS with diverse (1) origin/destination loadings of the system, (2) lane scheduling algorithms, (3) communication protocols and (4) AVCS characteristics. The simulator can be used to simulate the actual flow of traffic on a multi-lane AHS, as well as traffic flow (assuming mixed traffic, i.e., automated and nonautomated) in the transition phase of development. It can be used to determine how the macroscopic flow characteristics are affected by the characteristics of the AVCS in the vehicles, by the communication protocols employed in seeking permission to maneuver, by rules of operation of vehicles on the AHS and by the lane assignment strategies used to schedule paths of vehicles.

The AHS simulator has been given a modular and expandable structure to enable the insertion of new options and tasks. The requirements of flexibility and expandability have been met through the use of object-oriented design. A class library written in the C++ language has been developed and can be gradually expanded to meet the above requirements. The library is composed of building blocks that may be combined to create a specific AHS simulator characterized by such entities as number of lanes, number of entry and exit points, distribution of vehicles at each entry point with respect to destination, lane speeds, intervehicle distances, maneuvering protocols, control system characteristics and scheduling algorithms.

8.2 Survey of Existing Simulators

Many diverse simulators exist for simulating traffic in various stages of planning, scheduling and operational control of freeways, arterials and networks. The macroscopic models essentially operate with macroscopic traffic flows and have been widely used in planning and scheduling (NETSIM [38], FRESIM [39], RAMBO [40], TRANSYT [41], HCS [42]). They provide information on macroscopic variables in the broad sense, but cannot be used to de-

termine the effect of lane scheduling, communications and maneuvering on the macroscopic variables important for operational scheduling and real-time control.

A wide variety of microscopic models have been developed to simulate the effect of individual vehicles on traffic flow. These include VTAC [43] for the simulation of flows on arterial networks, INTEGRATION [44] for the simulation of flows on arterial networks and freeways, INTRAS [45] for the simulation of longitudinal traffic flow in a lane, CARSIM [46] for the simulation of longitudinal traffic flow with more realistic modeling of individual vehicle characteristics and WEAVSIM [47] for the simulation of maneuvering on sections of freeway where vehicle flows intersect. These models simulate individual vehicles by associating with each vehicle specific attributes such as vehicle type, driver reaction time, braking time, maximum accelerations and decelerations. They include lane change algorithms and contain provisions for defining the geometry of the roadway. In addition to presenting detailed information on the motion of individual vehicles, the models can also calculate the summary effects of an individual vehicle on macroscopic flow parameters.

However, as far as freeway traffic is concerned, the operational rules of the road are constrained to behavior typical of human drivers on existing freeways. In particular, this implies that there is no paradigm for when and how vehicles are assigned to lanes or when a vehicle should change lanes. Typical are statements to the effect that lane changing algorithms move vehicles by first establishing "a desire or a need" to change lanes, the usual motivation being to exit the freeway or to pass slower vehicles. Maneuvers are initiated when gaps appear in traffic in the lane to which the vehicle desires to move. There is no provision in these models for any degree of automated driving for assessing the effect of communication protocols, for assessing the effect of AVCS on longitudinal motion, for lane changing and for macroscopic flow parameters under these conditions. These models have been used to schedule traffic operations, but in the absence of clear and mandatory lane assignment strategies, have been used mainly to assess the characteristics of traffic flow under different assumptions and road conditions.

Simulation models have also been developed for simulation of real-time operations under a given set of communication protocols and control strategies. Typical are FASTCARS [48], developed to help assess and select en route vehicle behavior in response to real-time traffic

information and IGHLC [49], an expert systems based program for guidance of autonomous vehicles on a freeway. As far as we are aware, SmartPath2.0 [50] is the only simulator developed for simulation of real-time operations on the AHS. SmartPath2.0 models the identity of each vehicle. It models the system architecture as layers consisting of the physical layer, the regulation layer, the platoon layer and the link layer. The physical layer, the regulation layer and the platoon layer are modeled in detail. A simple link layer controller is considered.

While SmartPath2.0 is suitable for certain studies about an AHS, it lacks the capability to study the scheduling issues associated with the AHS. In particular, it exhibits the following shortcomings. (1) Only one link layer controller is provided. Modifying the code to study other link layer controllers would not be a trivial task. (2) SmartPath2.0 has a specified control systems architecture hard-coded into the program. For example, the car following strategy assumed is that of discrete platoon [23] formation. The study of other strategies such as continuous platooning is not possible. (3) The origin destination flows cannot be specified as time varying. This precludes the study of the flow of traffic on an AHS for an extended period of time, for example, a day, because the origin destination flows will change during the course of a day. (4) All of the vehicles are assumed to be "smart," i.e., equipped with sensors, controls and communication equipment to allow automated lateral and longitudinal control. A study of AHS operation with "mixed" traffic, i.e., with smart and nonsmart cars is not possible. (5) There is no provision for scheduling vehicles to lanes of travel based on their destination or based on the traffic conditions in the entire AHS. We see the inclusion of this provision as crucial on the AHS where the availability of global information on the state of demand for lanes on the AHS will be known with much greater certainty than on existing highways, and where paths of vehicles through the system can be prescribed.

Although it might be possible to modify an existing simulator and add the features required for AHS simulation, it would be difficult and would be tantamount to using a tool designed for one task for a different one. It is more appropriate, and timely, to develop a simulator dedicated to the AHS environment that concentrates on its unique operational features. Such a simulator can be used to study various controller architectures, various

maneuvering protocols, various car following strategies and mixed traffic operation, i.e., various representative system configurations. The development of a C++ class library that will enable the creation of such a simulator is described below.

8.3 Development of a C++ Class Library

The AHS is a very complex system that is described by the behavior of a multitude of vehicles interacting with each other, with the highway and with the traffic management system. To represent this real-world complexity within a software system, it is essential to decompose it into smaller and smaller parts, each of which may be refined independently (Djikstra). An object-oriented decomposition [51] greatly reduces the risk of building complex software systems, and is designed to evolve incrementally from smaller systems. Object-oriented decomposition creates software that is resilient to change and is written with economy of expression.

Software building blocks that represent components of all entities comprising the AHS have been designed. The programming language that has been used for this purpose is C++. The building blocks are referred to as *classes* and the set of classes is referred to as a C++ class library. We have made use of the techniques of abstraction, encapsulation, modularity and inheritance provided by object-oriented design in the creation of a framework for the development of an AHS simulator. Depending on the desired application, the user can combine the building blocks to create different applications, within the common framework.

Examples of each of the above techniques in the context of a software description of an AHS are provided below. At first are listed some formal definitions as provided in [51].

Definition 8.3.1 *A class is a set of objects that share a common structure and a common behavior.*

Definition 8.3.2 *An object is a single instance of a class.*

Definition 8.3.3 *A member is a repository for part of the state of an object; collectively, the members of an object constitute its structure.*

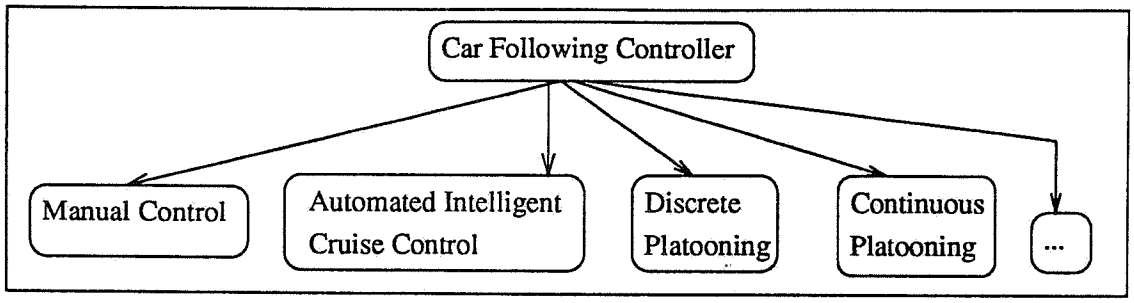


Figure 8.1: The *CarFollower* class and its subclasses

Definition 8.3.4 *A method is an operation upon an object, defined as part of the declaration of a class.*

Definition 8.3.5 *A subclass is a class that inherits structure and behavior from one or more classes (called its immediate superclass); a subclass typically augments or redefines the existing structure and behavior of its superclasses.*

Definition 8.3.6 *The interface of a class is the outside view of a class, which emphasizes its abstraction, while hiding its structure and the secrets of its behavior.*

8.3.1 Abstraction

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects. It represents a useful model of an entity and provides a generalized set of operations of the entity. It represents the *interface* of a class and defines its essential behavior but does *not* go into the details of the implementation of that behavior.

To give a concrete example, consider the longitudinal motion of a vehicle on a highway. The motion of the vehicle is determined by its current position, speed and acceleration, by throttle or brake and external disturbance inputs. The vehicle may be under automated control and using some specified car following law, or it may be driven by a human driver. While it is necessary to model both types of car following algorithms, once the throttle or brake input is provided to the vehicle, its motion is controlled by the laws of physics. A *CarFollower* class (see Figure 8.1) could therefore be made an abstract class, one of whose

essential characteristics is the providing of the throttle or brake input to the vehicle. C++ provides the mechanism for such abstractions through abstract *classes*. A *CarFollower* class would therefore be one logical class required for AHS simulation. It would have subclasses (*ManualControl, AICC, DiscretePlatooning*, etc.) which represented human car following and various types of automated car following. The *CarFollower* class would have an abstract "method" called *Control()*, whose function would be to calculate the throttle or brake input to the vehicle.

8.3.2 Encapsulation

Encapsulation allows program changes to be reliably made with little effort. It is the process of hiding all of the details of an object that do not contribute to its essential characteristics. This is achieved by hiding the *implementation* of the mechanisms that achieve the desired behavior of the class. A drawback of programming large systems is that once a system reaches a certain size it becomes difficult to keep track of all of the variables and function names, which leads to slow progress and error-prone code. In most cases, the use of an object in computation does not require the knowledge of the implementation of its behavior. In such cases, the internal workings of the object are best kept hidden through encapsulation.

To continue with the example presented in Section 8.3.1, consider the *Control()* method of the *CarFollower* class. This method is invoked to determine the jerk input to the vehicle. For an automated vehicle proceeding in a discrete platoon, this jerk would be a function of the states of the current vehicle, the vehicle immediately in front of it and the leader of the platoon. For a vehicle proceeding under manual control, the jerk may only be a function of the states of the current vehicle and the vehicle in front of it. Thus, the implementation of the *Control()* method would be different for the subclasses *ManualControl, AICC, DiscretePlatooning*, etc. of *CarFollower*. This is the principle of encapsulation and can be achieved in C++ through the use of a virtual (abstract) method *Control()* in the abstract class *CarFollower*.

8.3.3 Modularity

While the act of partitioning a program into individual components can reduce its complexity, this can result in a very large number of classes. Combining classes that are related in some way into a module is essential to help manage complexity. Consider, for example, a *matrix* class and a *vector* class that provide the functionality of matrix and vector operations. Other classes may make use of the matrix and vector classes, for example, in defining origin and destination matrices, but may be distinct from them. The matrix and vector classes may be combined into a module. This module may be used by other classes without the details of the implementation of the module being available to those classes.

Another example of the usefulness of modularity can be brought out by the use of a *simulator* module. This module can be used for creating events, for scheduling them in the event calendar and for executing them. It consists of a set of classes that provide the functionality mentioned above and provides facilities for interacting with other classes outside the module.

8.3.4 Inheritance

Inheritance is a relationship among classes wherein one class shares the structure or behavior defined in one or more classes. For example, two subclasses *AICC* (automated intelligent cruise control) and *CP* (continuous platooning) derived from the base class *CarFollower* share all of the essential behavior defined for the class *CarFollower*. (The implementation of this essential behavior is made different, via encapsulation.) With inheritance, the two subclasses inherit the structure, i.e., the *interface* of their superclass *CarFollower*. However, they might also have additional structure of their own which is required by their implementation of the superclass's behavior.

8.3.5 Creating new applications

Libraries may be built out of the kinds of classes described above. They support design and reuse of code by supplying building blocks and providing ways of combining the building blocks; the application builder designs a framework into which these common blocks are

fitted. Consider, for example, the application created by the use of the *Highway*, *AICC*, *Vehicle* and other supporting classes. This may be used for the study of automated driving on an AHS. By creating instances of a *Highway*, an *AICC*, a *CSCarFollower* (a manual car follower) and a *Vehicle* and other supporting classes, one has created an application for the study of mixed driving on an AHS.

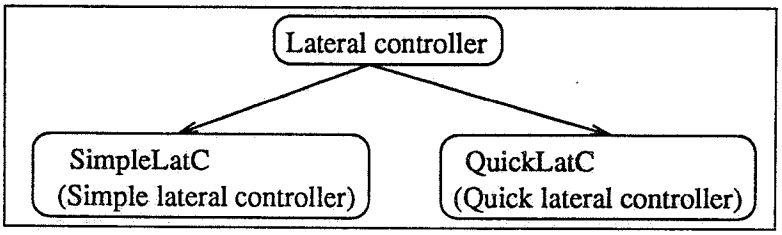


Figure 8.2: The lateral controller class and its subclasses

The use of abstract classes is critical in providing flexibility in an efficient manner. Suppose the abstract class *CarFollower* has four subclasses (it actually has more) as shown in Figure 8.1. The abstract class *LatControl* (the lateral controller) has two subclasses as shown in Figure 8.2. The number of possible combinations of car following and lateral controllers, therefore, is 4*2 = 8. Instead of implementing each of the eight combinations as an object, we can use the concepts of abstraction, encapsulation and inheritance to implement 4+2 = 6 objects. In general, the savings would be $n * m - (n + m)$, which, when looked at in the context of a large system such as an AHS simulator, could be large indeed.

8.4 Description of the Various Classes

This section will provide a succinct description of the important classes required by the AHS simulator. The complete "interface" of all of the classes used is provided in the .h files listed in Appendix C. The description provided here of a class consists of a description of its interface and of its implementation.

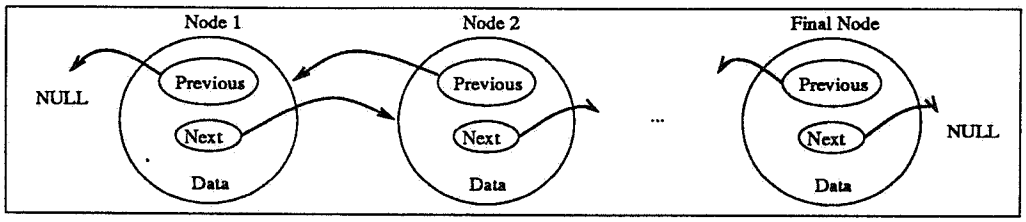


Figure 8.3: A doubly linked list

8.4.1 Helper classes

These are tools that are not specific to the problem of simulation or to automated highway systems. They make use of the "template" feature of C++ that allows class definitions to be parameterized by one or more generic types.

8.4.1.1 ValArray

A *ValArray* class provides the functionality of a dynamically resizable array. It has been written using "templates," so that it may be used to create an array of objects of arbitrary type, such as *Vehicles, Lanes* or *Sections* as follows.

$ValArray < Vehicle* > array\_of\_vehicles$ is a *ValArray* of *Vehicles.*

$ValArray < Lane* > array\_of\_lanes$ is a *ValArray* of *Lanes.*

8.4.1.2 DLinkedList and LinkedList

A *DLinkedList* class provides the functionality for a doubly linked list. A *LinkedList* class provides the functionality for a singly linked list. Each *DLinkedList* and *LinkedList* object consists of a set of *Nodes.* Each *Node* consists of some data and a pointer to the previous and the next nodes in the list (see Figure 8.3) for a doubly linked list, and a pointer to the next node only for a singly linked list. These two classes have also been written using templates, so that they may be used to create a doubly or singly linked list of objects of arbitrary type. Both arrays and linked lists can be used to store a collection of objects. The advantage of a linked list over an array is that insertion of a node into a linked list is more efficient than insertion of a node into an array. The disadvantage is the additional overhead

in maintaining pointers to the previous and next nodes. When a need for insertion into a list is anticipated, a linked list is preferable to an array.

The event calendar (see Section 8.4.5) used to drive the discrete event simulation is modeled as a *LinkedList.* Each node in the event calendar consists of an event and the pointer to the next event. (A pointer to the previous event is not required). A *DLinkedList* object has been used to store the information about car units in a lane. Each lane on a highway contains a doubly linked list of the car units (called *cul*) in that lane. The advantages provided by using a *DlinkedList* in this case are as follows.

1. By using the next and the previous pointers, each car unit can easily locate the car unit in front of it and the car unit behind it.

2. When a car unit changes lanes within a section, or when it moves from one section to another, it must be plucked from the *cul* of one lane and inserted into the *cul* of another lane. Such removals and insertions are efficiently handled by *DLinkedList.*

8.4.1.3 RandGen

This class is used to create objects that generate random numbers. While a single stream of random numbers can be generated by repeated calls to the C function *random()*, multiple streams **must** be available in order to have independent random variables for various aspects of the simulation. For example, the distribution governing arrivals of cars at some entrance to the highway must be independent of the distribution governing the arrivals of cars at some other entrance on a different highway. To generate multiple streams of random numbers that are independent of each other, we can use the fact that a sequence of, say 100,000 numbers, generated by repeated calls to *random()* is reasonably independent of the next sequence generated by the next 100,000 calls to *random()* (See [52]).

random() uses a nonlinear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1) [53]. Different streams of random numbers can be obtained by initializing the random number generator with a different *seed.* Using an arbitrary first seed and then using

the number resulting from the 100,000th call with this starting seed as the next starting seed, we obtain two sequences of random numbers that are reasonably independent of each other. The directory *Seeds* contains 50 seed sets in the files *Seeds1* through *Seeds50*. Each file contains a list of 1500 seeds that are 100,000 numbers apart in the sequence of numbers generated by *random()*.

A *RandGen* object is used to generate random numbers between 0 and 1. Each *RandGen* object reads a different seed from one of the files in the *Seeds* directory. One file is used per simulation, but if the number of *RandGen* objects created is greater than 1500, a new file is used. In all, there are 1500*50 = 75,000 possible seeds, and the probability that more than 75,000 independent streams are required in any one simulation is negligible and, therefore, these seeds should be sufficient to run most simulations. Repeated calls to the *GetRand()* method of *RandGen* will produce random numbers from the unique starting seed.

8.4.2 Network classes

This set of classes describes the physical structure of the highway such as the number of sections and lanes and their connectivity.

8.4.2.1 Domain

A *Domain* class represents the entire highway network. There is only one *Domain* object per simulation. It consists of an array of highways and thus contains the problem as a whole. In addition, it has knowledge of, and provides access to, the event calendar.

8.4.2.2 Highway

The *Highway* class represents a highway. It contains an array of sections. It has knowledge of the *Domain* object.

8.4.2.3 Section

The *Section* class is used to model a section of a highway. A *Section* object has knowledge of the *Highway* object that it is contained in. It is assigned an ID that is unique within

that *Highway* and is assigned a length. The *Section* class contains two subclasses, *StdSection* and *TransitionSection*. Both subclasses contain an array of lanes used by vehicles traveling on the highway. In addition to these lanes, a *TransitionSection* contains an entry/exit lane (with ID = 0) used by vehicles entering or exiting the highway.

8.4.2.4 Lane

The *Lane* class models the portion of lane contained within a section of a highway. The *Lane* class is a subclass of an abstract class called *CarAcceptor*, which as its name indicates, has the capability of accepting cars. Like *Domain*, *Highway* and *Section* objects that are time-invariant, a *Lane* object contains time-invariant information describing its physical characteristics such as lane width and connectivity. Each *Lane* object contains a pointer to the *Section* that it is in, to the next *Lane* and to the previous *Lane*. This information is used to set up connectivity in the highway. Each *Lane* is also assigned an ID or lane number that is unique within the *Section* that it is in. Thus, a *Lane* can be accessed by the $< Highwayname, SectionID, LaneID >$ information. The width assigned to a *Lane* is used to compute the lateral position of the center of the lane. To enable statistics collection, each *Lane* has an array of *TripWire* objects (described in Section 8.4.6.2) and a *CarUnitDisposal* object (described in Section 8.4.6.1). Unlike the *Domain*, *Highway* and *Section* objects, however, the *Lane* object contains information that changes as the simulation proceeds. This information is a doubly linked list of the car units (called *culs*) that are traversing it. This list changes as the simulation proceeds, as car units leave to go to another *Lane* in another *Section*, or as car units enter the *Lane* from a different *Lane* in the same *Section*. Each *Lane* is assigned a reference speed, which may be changed as the simulation proceeds, that the car units traveling on it attempt to maintain.

8.4.3 Driver/vehicle classes

This group of classes describes the driver/vehicle interaction. The physical characteristics and states of the car unit are incorporated in the *Vehicle* class. The driver/vehicle interaction and controls are modeled with *Vision*, *LongControl*, *LatControl* and *PathControl* classes.

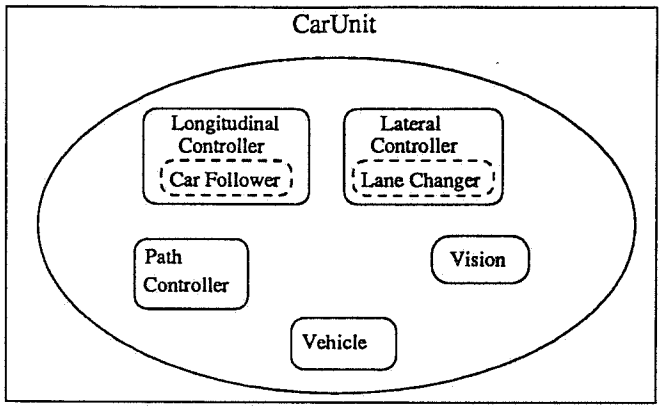


Figure 8.4: The components of a *CarUnit*

8.4.3.1 CarUnit

The *CarUnit* class models the driver/vehicle unit. A *CarUnit* object (see Figure 8.4) contains one each of a *Vehicle*, *Vision*, *LongControl*, *LatControl* and a *PathControl* object. A *CarUnit* object is assigned an ID, and it contains the information about its entry point onto the highway and its desired exit point. It has knowledge of the *Lane* that it is currently traversing.

8.4.3.2 Vehicle

A *Vehicle* class is used to model a vehicle. A *Vehicle* object contains information of the physical characteristics and states of the *CarUnit*. It contains a pointer to its parent *CarUnit*. It stores state and parameter information in *VehicleState*, *ActuatorState* and *VehicleParams* structures. The *VehicleState* contains the vehicle states such as longitudinal and lateral position, speed and acceleration. The *ActuatorState* contains the jerk and the steer inputs to the vehicle. The *VehicleParams* contain information on the mass, length, starting acceleration, maximum deceleration and maximum speeds. A vehicle has an *UpdateState()* method that uses the actuator inputs applied to the vehicle to update its lateral and longitudinal states. Only one subclass of *Vehicle*, called *PMVehicle*, has been implemented. It represents a point mass vehicle.

8.4.3.3 Vision

A *Vision* class is used to model the driver's vision function. A *Vision* object is used by its parent *CarUnit* to identify its neighboring *CarUnits*, both ahead of it and in the side *Lanes*. Under *AICC* car following, a car unit needs to know only the car unit immediately in front is, whereas under *DP* control, a car unit needs to know who is in front of it and who the platoon leader is. This information is provided by the *FrontNbor()* method of the *Vision* object and its subclasses. In addition, prior to and while changing lanes, a car unit needs to know who its neighbors are in the immediately adjacent lane and who its neighbors are in the far lane. The *Vision* object can provide this information. It has a range of view in meters (both front and side) within which it looks for neighbors. It has two front ranges, a smaller one used to find a car unit to follow and a larger one used to calculate the noncollision jerk. It has two side ranges, a larger one used for the lane immediately adjacent to the lane that the car unit is currently in, and a smaller one for the far lane. The *Vision* class has three subclasses, namely, *HumanVision*, *CPVision* and *DPVision*. The latter two are used with *CP* and *DP* car followers, respectively; the former is used with the remaining car followers.

8.4.3.4 CarFollower

The *CarFollower* class is used to model the car following algorithm being used. At each integration time step, the longitudinal states of the *Vehicle* are updated by taking into account the jerk acting on the *Vehicle*. The jerk at each time step depends on the car following algorithm being used. The *Vision* object provides information on the longitudinal neighbors of the car unit. This information is used by the *CarFollower* to calculate the jerk. The algorithms used are different when the longitudinal controller of the car unit is in the *LONGC* mode (a longitudinal control mode for pure car following) and when it is in one of *LATC*, *LATC_DEC* or *LATC_MONITOR* modes (lateral control modes, used when the car unit or its neighbors are changing lanes.) The final jerk applied to the *Vehicle* depends not only on the car following algorithm, but also on the noncollision jerk and on the limits on jerk, acceleration and speed. The *NonCollisionJerk()* and *FindActJerk()* (see Sections 9.2.1

and 9.2.2) methods are used to calculate the final actual jerk applied to the *Vehicle*. The *CarFollower* class has the following subclasses.

- *FollowTraj* calculates the required jerk for the vehicle to follow a prescribed trajectory.
- *CSCarFollower* models the car following algorithm used by humans. This is based on work done by Benekohal and Treiterer [46].
- *CC* is simple cruise control, in which the vehicle's speed is regulated about a reference speed.
- *AICC* is automated intelligent cruise control based on the control law developed by Chien and Ioannou [54]. Here the vehicle attempts to stay a certain distance behind the vehicle in front of it. This distance is a function of the longitudinal positions and speeds of the two vehicles.
- *CP* is continuous platooning developed by Ren and Green [55]. Continuous platooning is an extension of *AICC*, in which the control law makes use of the states of not only one vehicle in front of it, but of n vehicles in front of it. This number n is called the *lookahead*. Ren and Green have designed about nine controllers with lookaheads of one, two and three vehicles, with velocity dependent and independent steady state spacing requirements.
- *DP* is discrete platooning [23]. This is based upon a control law developed by Sheikholeslam [56] of the PATH project at Berkeley.

8.4.3.5 Platoon

The *Platoon* class models a grouping of *CarUnits*. It provides the functionality for the formation of these groupings. It also provides the functionality for splitting away from the grouping and merging into the grouping. During the course of a simulation, many platoons may be formed and destroyed. The *Platoon* class provides an efficient means of construction and destruction of *Platoon* objects. This is done through the use of a *Pool* of *Platoons*. When a new *Platoon* object is required, it is taken from the *Pool* (as opposed to allocating

new memory for the *Platoon* object). When a *Platoon* object is no longer required, it is not destroyed, but is simply returned to the *Pool*. Only when the *Pool* runs dry is new memory actually allocated for the creation of additional *Platoon* objects.

8.4.3.6 LongControl

This class provides the driver functionality of longitudinal control. It makes use of a *CarFollower* to calculate the jerk according to some car following control law. The *LongControl* decides what car following control law needs to be applied. For example, it decides whether a car unit should merge with the platoon ahead of it, or whether it should maintain a safe distance from the car unit in the adjacent lane that is changing lanes. *LongControl* has three subclasses.

- *FollowTrajLC* directs the vehicle to follow a prescribed trajectory. It makes use of the *FollowTraj* car follower.
- *CarSimLC* models human longitudinal control. It makes use of the *CSCarFollower* car following algorithm. The desired vehicle speed is drawn from a truncated normal distribution with mean 55 mph and a standard deviation of 5 mph.
- *PlatoonLC* represents some form of automated control. It makes use of the car following laws of *AICC*, *CP* or *DP* to direct vehicles to travel in platoons. The size of the platoons is a parameter that can be set. The *PlatoonLC* decides when and how the vehicle should merge with another platoon, split from a platoon, be a follower in a platoon, track some desired speed (which is the reference lane speed), become a leader, become a free agent and perform other longitudinal control tasks (see Section 9.1).

The classes *CarSimLC* and *PlatoonLC* have four modes of operation denoted by *LONGC*, *LATC*, *LATC_DEC* and *LATC_MONITOR*. In the *LONGC* mode of operation, the longitudinal controller directs the vehicle to travel in its lane according to its state and car following law. In the other modes, the longitudinal controller either (1) maintains a safe distance from a vehicle in the adjacent lane that is changing lanes or (2) maintains a safe distance from vehicles in the adjacent lanes, while the vehicle itself changes lanes.

8.4.3.7 LaneChanger

This class provides the functionality for the actual mechanics of lane changing. It has only one subclass, *SimpleLaneC*. The *SimpleLaneC* object is used by the lateral controller to direct a vehicle to change lanes in the space of three seconds. The trajectory followed by the vehicle in these three seconds is an inverse tangent curve. The *LaneChanger* object modifies only the lateral position of the vehicle. The longitudinal controller and car follower modify the longitudinal position, speed and acceleration.

8.4.3.8 LatControl

This class provides the functionality for lateral control of the vehicle. It makes use of a *LaneChanger* object to execute the actual lane change. The *LatControl* object receives lateral control commands from the path controller. The *LatControl* object decides if it is safe to change lanes. If not, it commands either its car unit or neighboring car units to decelerate by placing one or more longitudinal controllers in the *LATC*, *LATC_DEC* or *LATC_MONITOR* control mode. The logic used in the lateral control algorithm has been described in Section 9.3. The *LatControl* class has two subclasses.

- The *SimpleLatC* controller makes sure that the car unit will not collide with cars in the neighboring lane or with cars that are maneuvering from the far lane to the near lane. If the car unit has a *PlatoonLC* longitudinal controller, then it must be placed in the *FREE AGENT* general state before it is allowed to change lanes.

- The *QuickLatC* controller uses the same logic as the *SimpleLatC* controller, except that if a car unit has a *PlatoonLC* longitudinal controller, then it must only be placed in the *PRE_LANE_CHANGE* general state before changing lanes. In this general state, the car unit does not have to physically split away from the rest of the platoon.

8.4.3.9 PathControl

The *PathControl* class provides the path control logic, i.e., it determines or obtains (not necessarily all in advance) the path, in terms of lanes, that a vehicle must follow. When the

car unit must change lanes, the *PathControl* object calls upon the *LatControl* object to do so. It has four subclasses.

- *HumanPC.* This represents human path control. Essentially, the logic used is that the vehicle changes lanes in order to overtake another vehicle or to move into the exit lane.
- *SemiCompliantPC.* Under this type of path control, a vehicle is prescribed a path that it must follow. This type of path controller may or may not comply with the prescribed path.
- *CompliantPC.* The difference between this and *SemiCompliantPC* is this type of path control always complies with the prescribed lanes. Note, however, that the actual path followed by the vehicle may not correspond to the prescribed path. This could happen because the lateral controller is unable to execute a lane change maneuver (due to heavy congestion, for example) when told to do so.
- *AutomatedPC.* This type of path control works with constant lane assignments. The assignment is obtained from the segment controller.

8.4.3.10 SegControl

This class provides the functionality for providing constant lane assignments. The *SegControl* object does not compute the lane assignments. It obtains them from the optimization routine that calculates the optimal or suboptimal assignments. When called by the *PathControl* object, it uses these assignments to assign lanes to the car units. It has the following subclasses.

- *ConstLaneSC* represents constant lane assignments. For a given OD matrix $N(i,j), 0 \leq i \leq K_1, 1 \leq j \leq K_2$, the lane assignments are specified as $\rho(i,j,m)$, which is the flow traveling from i to j assigned to lane m.
- *DestMonoSC* represents destination monotone (without splitting) lane assignments.
- *OrigMonoSC* represents origin monotone (without splitting) lane assignments.

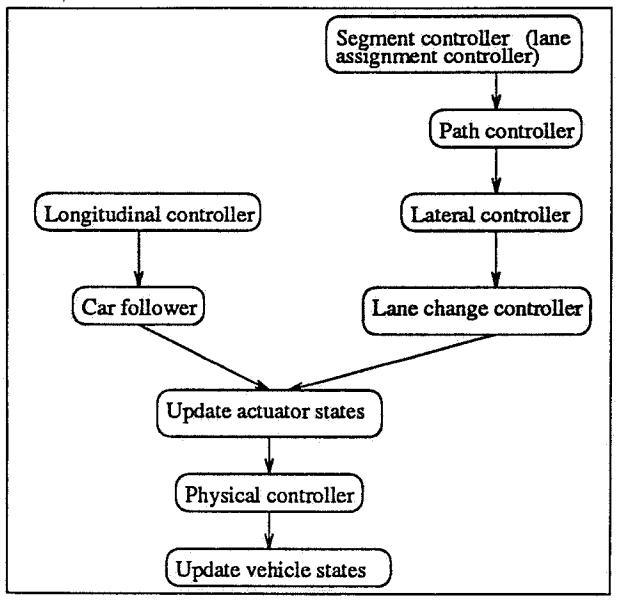


Figure 8.5: The vehicle controllers

- *DestSplitSC* represents destination monotone assignments. The flow going from each entrance to the partitions for that entrance may be split into two lanes.
- *OrigSplitSC* represents origin monotone assignments. The flow going to each exit from the partitions for that exit may be split into two lanes.

Figure 8.5 shows the controller hierarchy. Figure 8.6 illustrates the interfacing of the vehicle controllers with the discrete event simulator. The vehicle controllers are activated through control events (described in Section 8.4.5).

8.4.4 Generator classes

This section describes classes that are used to generate the various components of a car unit and also to generate the car unit itself. Certain restrictions must be kept in mind while generating a car unit. For example, a *CarUnit* with an *AICC* car follower must have a vision object of type *HumanVision*. It cannot have a vision object of different type, such as *DPVision*. Restrictions of this type have been coded into the component generators, so that when a *CarUnit* object is generated, it does not contain incompatible components.

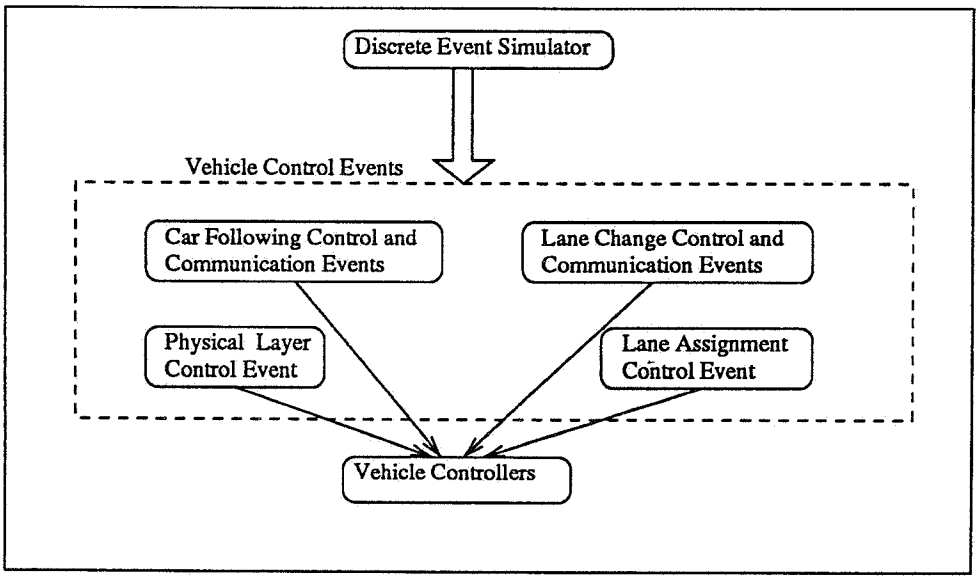


Figure 8.6: Interfacing the simulator with the car unit

8.4.4.1 VehicleGen

The *VehicleGen* class provides the functionality for generating *Vehicle* objects.

8.4.4.2 CarFollowerGen

The *CarFollowerGen* class provides the functionality for generating *CarFollowerGen* objects. It is an abstract class and has the following subclasses.

- The *FollowTrajGen* class provides the functionality for generating a *FollowTraj* car follower. In addition, it creates a *HumanVision* vision component for use in its car unit.

- The *CSCarFollowerGen* class provides the functionality for generating a *CSCarFollower* car follower. In addition, it creates a *HumanVision* vision component for use with its car unit.

- The *AICCGen* class provides the functionality for generating a *AICC* car follower. In addition, it creates a *HumanVision* vision component for use with its car unit.

- The *CP_*Gen* class provides the functionality for generating a *CP_** car follower. In addition, it creates a *CPVision* vision component for use with its car unit.
- The *DPGen* class provides the functionality for generating a *DP* car follower. In addition, it creates a *DPVision* vision component for use with its car unit.

8.4.4.3 LongCGen

The *LongCGen* class provides the functionality for generating *LongControl* objects. It requires knowledge of a compatible *CarFollowerGen* object. When a *LongCGen* object is called to generate a *LongControl* object, it asks the *CarFollowerGen* object to generate a compatible *CarFollower* and *Vision* object. The *LongCGen* class is an abstract class and has three subclasses. The *FollowTrajLCGen, CarSimLCGen* and *PlatoonLCGen* subclasses provide the functionality for generating *FollowTrajLC, CarSimLC* and *PlatoonLC* longitudinal controllers, respectively.

8.4.4.4 LaneChGen

The *LaneChGen* class provides the functionality for generating *LaneChanger* objects.

8.4.4.5 LatCGen

The *LatCGen* class provides the functionality for generating *LatCGen* objects. It requires knowledge of a *LaneChGen* object. When a *LatCGen* object is called to generate a *LatControl* object, it asks the *LaneChGen* object to generate a *LaneChanger* object. The *LatCGen* class is an abstract class and has two subclasses. The *SimpleLatCGen* and *QuickLatCGen* subclasses provide the functionality for generating *SimpleLatC* and *QuickLatC* objects, respectively.

8.4.4.6 PathCGen

The *PathCGen* class provides the functionality for generating *PathControl* objects. It is an abstract class and has four subclasses. The *HumanPCGen, SemiCompliantPCGen,*

CompliantPCGen and *AutomatedPCGen* subclasses provide the functionality for generating *HumanPC, SemiCompliantPC, CompliantPC* and *AutomatedPC* objects, respectively.

8.4.4.7 CarUnitGen

The *CarUnitGen* class models a *CarUnit* generator. A *CarUnitGen* object has knowledge of a *CarAcceptor* object, typically a *Lane* object, and it places a generated *CarUnit* object in that *CarAcceptor* object. It also constructs, if necessary, the various components of a *CarUnit* object, namely, a *Vehicle*, a *LongControl*, a *LatControl*, a *Vision* and a *PathControl* object. The *CarUnitGen* also assigns to each *CarUnit* an ID and a desired destination based upon the flow rate from the entrance where it is located, to various destinations. The *GenerateCarUnit()* method of the *CarUnitGen* object is called by the *CarGEvent* event to generate a *CarUnit*. The *CarUnitGen* class is an abstract class with two subclasses.

- The *StdCarUnitGen* class provides functionality for generation of a *CarUnit* that contains a *PMVehicle*, a *HumanVision*, a *CSCarFollower*, a *CarSimLC*, a *LaneChanger*, a *SimpleLatC* and a *HumanPC*. This *CarUnit* object represents a vehicle under human control.

- The *SpecialCarUnitGen* class provides functionality for generation of a *CarUnit* with arbitrary (but not incompatible) components. It calls upon a *VehicleGen*, a *LongCGen*, a *LatCGen* and a *PathCGen* to generate the various car unit components.

8.4.5 Simulation classes

This section describes classes that deal with the discrete event simulator. The basic idea behind discrete event simulation is as follows. An object called an event calendar operates on objects called events that are contained within it. Each event is associated with a time of execution and an action that should be executed at that time. The event calendar keeps track of a global clock variable. It sorts the events that it contains in increasing order of their times of execution. The simulation is begun when the event at the head of the queue is popped from the queue and its action executed. One by one, the event calendar pops

the event on top of its stack of events and executes the event's action. It then updates the global clock to be equal to the event's execution time. When events are executed, they may insert additional events into the event calendar, and may even reschedule themselves. When the event calendar has no more events left in its queue, or if a specified stopping criterion is met, the simulation is considered to have ended. The two basic objects required for discrete event simulation are, therefore, the event calendar and the event.

8.4.5.1 EventCal

The *EventCal* class models the event calendar. There is only one *EventCal* object per simulation. An *EventCal* object is a singly linked list of nodes. A node is composed of an event and a time of execution of the event. The *EventCal* object maintains a global clock that is updated each time an event is executed. It has some rudimentary monitoring features that can be activated by means of a *trace* flag. When this flag is set, the name and the times of the events being executed are printed out. The *EventCal* class has a *Run()* method that is responsible for removing events from the event calendar and for executing them.

8.4.5.2 Event

The *Event* class is an abstract class that models an event. All *Event* objects must provide an *Action()* method that is called in order to execute the event. *Event* objects can be prioritized. This feature is not being currently used though. The *Event* class has various subclasses that model the various events required for AHS simulation. They are described below.

8.4.5.3 CarGEvent

The *CarGEvent* class models a car generation event. Its *Action()* method calls upon a specified car unit generator and asks it to create a car unit. The various subclasses of the *CarGEvent* class are described below.

- *OneShotCarGEvent* models an event that generates a *CarUnit* object only once.

- *PerCarGEvent* models an event that periodically (with period p) generates *CarUnit* objects. It needs to be scheduled only once. When its *Action()* method is called, it generates a *CarUnit* object and then reschedules itself at a time p units from the current time.

- *UnifCarGEvent* models an event that generates *CarUnit* objects. The time between generations is uniformly distributed between two specified limits. Like *PerCarGEvent*, it is scheduled only once. It will reschedule itself for a time that is uniformly distributed between two specified limits.

- *ExpCarGEvent* models an event that generates *CarUnit* objects. The time between generations is exponentially distributed with mean λ. It will reschedule itself for a time that is exponentially distributed with mean λ.

- *NormCarGEvent* models an event that generates *CarUnit* objects. The time between generations is a truncated normally distributed random variable with mean λ and standard deviation ρ. It will reschedule itself for a time that is a truncated normally distributed random variable with mean λ and standard deviation ρ.

8.4.5.4 LatCEvent

The *LatCEvent* class models a lateral control event. It calls upon the *LatControl* of a specified *CarUnit* object to execute one lane change in the specified direction. Generally, it does not reschedule itself. It is scheduled by the *PathControl* object when the path controller wishes the car unit to change lanes. It is also sometimes scheduled by the lateral or longitudinal controller of either the car unit trying to change lanes, or the neighbors of the car unit trying to change lanes. For details, refer to the lateral control logic in Section 9.3.

8.4.5.5 PltLCStateEv

The *PltLCStateEv* class models an event that updates the *control_state* of a *PlatoonLC* longitudinal controller. It acts upon the *LongControl* of a specified *CarUnit* object and

decides what *control_state*, namely, *FOLLOWER, TRACKER, MERGER* or *SPLITTER* it should be in. Once scheduled, it reschedules itself at regular intervals.

8.4.5.6 PathCEvent

The *PathCEvent* class models a path control event. It calls the *Control()* method of the *PathControl* of a specified *CarUnit* object. Once scheduled, it reschedules itself at regular intervals.

8.4.5.7 Physical

The *Physical* class models an event that updates the lateral and longitudinal states of every *Vehicle* in the *Domain*. Once scheduled, it reschedules itself at regular intervals. It is also used to periodically print out these states. It calls upon an *Integrator* object to calculate the new jerk for every *Vehicle* object and then to update the states for every *Vehicle* object.

8.4.5.8 StatCEvent

The *StatCEvent* class models an event that is used to regularly collect traffic statistics. When a *StatCEvent* event is executed, it calls upon a *StatCollector* object (described in Section 8.4.6.3) to collect statistics on traffic speed, concentration and flow.

8.4.5.9 EndSim

The *EndSim* class models an event that is used to terminate the simulation at a particular time. To terminate the simulation at a time T_F, the *EndSim* event should be scheduled for time T_F.

It is interesting to note the frequency of scheduling of the events listed above. A *Physical* event is scheduled most often. The time between two *Physical* events is the smallest time step in the simulation. A *PltLCStateEv* event is scheduled less often because changes in the *control_state* of the longitudinal controller occur less often than changes in the jerk. A *PathCEvent* event is scheduled much less often; this is because path control decisions need to be made less frequently. A *CarGEvent* event may be randomly scheduled. It will, however,

be much less frequent than a *Physical* event. The *EndSim* event will be scheduled only when a stopping criterion is met. Typically, the simulation is scheduled to end after a fixed amount of time, and so only one *EndSim* event will be scheduled.

8.4.6 Data collection classes

These are classes that enable data collection for future processing.

8.4.6.1 CarUnitDisposal

The *CarUnitDisposal* class is used to stop control of *CarUnit* objects. A *CarUnitDisposal* object is attached to a *Lane* object. It stops control of car units once they have reached their destination or if they have gone off the road. It provides information on the number of car units that have reached their destination and that have gone off the road. However, the memory allocated to these car units is not immediately freed up because, although the car units will themselves no longer affect the simulation, other objects such as *LatEvents* and *PathCEvents* that are waiting in the *EventCalendar* for processing may contain references to these car units. A system of *reference counting* keeps track of how long a car unit must be kept "alive." Each car unit contains a *reference count* that is equal to the number of objects that have references to it. When all of the objects that have references to a car unit drop those references, the *reference count* of the car unit drops to zero and it is deleted from the simulation.

8.4.6.2 TripWire

The *TripWire* class enables a change of state to be recorded. This might be the state of a car unit or of another object. A *TripWire* object can be "laid" across the end of a lane or at an exit. When a car unit crosses the *TripWire*, the car unit is flagged and appropriate action is taken. A *TripWire* class is an abstract class, with three subclasses.

- *EOLWire* (end of lane wire) is a *TripWire* that is crossed when a car unit reaches the end of a lane. At this point, the car unit is placed in the next lane, if one exists, or is considered to be off-road, if none exists. In the latter case, the *CarUnitDisposal* stops

control of the car unit. If the car unit is in the middle of a lane change when it reaches the end of a lane, and the lane that it is trying to change to does not extend beyond the current section, the lane change maneuver is aborted.

- *AEWire* (at exit wire) is a *TripWire* that is crossed when a car unit reaches its exit. When this happens, the car unit is removed from the lane and is handed to the *CarUnitDisposal* to be disposed of. An *AEWire* object is typically laid across the entry/exit lane of a *TransitionSection* object.

- *FlowWire* is a *TripWire* that measures flow past a point. If the point of interest is the middle of a lane, then a *FlowWire* object is placed across the middle of the lane. It records the number of car units that cross it in a specified time interval and uses this information to calculate the flow. *FlowWire* objects can be placed anywhere along a lane, and a lane may contain more than one *FlowWire*. The *StatCEvent* event causes periodic flow calculations to be made by a *FlowWire* object. Additionally, at the end of the simulation, the *FlowWire* calculates the net flow across it during the entire simulation.

8.4.6.3 StatCollector

The *StatCollector* class is used for collection of statistics on speed, concentration and flow. It has one subclass, *TrafStatCollector*. The *StatCEvent* periodically calls upon a *StatCollector* object to collect statistics. The *TrafStatCollector* class calculates the spot speed of each lane. For a lane containing N cars, the spot speed $\bar{u} = \frac{1}{N}\sum_{i=1}^{N} u_i$, where u_i is the speed of the ith car. The concentration measurement is made as follows.

$$\begin{aligned} \text{Let } k_i &= \frac{1}{\text{Position of car } i - \text{Position of car } i-1} \\ k &= \frac{1}{1/N\sum_{i=2}^{N} 1/k_i} \end{aligned}$$

The flow measurement is made by calls to the *FlowWire* objects that the *StatCollector* has knowledge of. The flow across *FlowWire* i in time ΔT = Number of cars that crossed i in $\Delta T/\Delta T$.

8.4.7 Miscellaneous classes and structures

This section describes other classes and structures that do not fall into the categories listed above.

8.4.7.1 Integrator

The *Integrator* class is used to update the states of every *Vehicle* in the *Domain.* An *Integrator* object can obtain knowledge of every *CarUnit* in the *Domain* by looking at the *culs* of every *Lane* object in each *Section* and *Highway* of the *Domain.* The *Integrate()* method of the *Integrator* object is called by the *Physical* event at regular time intervals. For each car unit in each highway, section and lane, the jerk to be applied at that instant is calculated by calling upon its longitudinal controller. Once all of the jerks have been calculated, the *UpdatePos()* method of each *Vehicle* object is called to update the vehicle states according to the jerk that has been applied.

8.4.7.2 CarSimData

The *CarSimData* class is a storehouse for vehicle and driver data used by the *CarFollower* class and used in the CarSim simulator [46]. It contains information such as the maximum allowable accelerations and decelerations at various speeds and the probabilistic reaction time of human drivers. It also defines the maximum and minimum jerk allowed for a typical vehicle [57].

8.4.7.3 Trajectory and PathArray

A *Trajectory* is simply a $ValArray < double >$. It is used to specify the trajectory of a *FollowTraj* car follower. A *PathArray* is an array of the $< Highway, Section, Lane >$ triple, and is used by *CompliantPC* and *SemiCompliantPC* path control objects.

CHAPTER 9

VEHICLE CONTROL ALGORITHMS

This chapter presents the control logic that has been coded in the AHS simulator for the various car unit controllers. These are longitudinal control, car following, lateral control, lane changing, path control and segment control. One or more control algorithms have been considered for each type of control and a description has been provided for each. It must be noted, however, that a critical analysis of the proposed schemes has not been presented. With regard to the car following algorithms, most are well known in literature and their relative merits already known. A detailed analysis needs to be done for the other control algorithms in order to study their relative merits and their effect on AHS traffic characteristics. This will be a topic of future research.

9.1 Longitudinal Control Logic

The *LongControl* object controls the longitudinal motion of a car unit. It calls upon its car follower to calculate the jerk input to the car. Three kinds of longitudinal controllers have been implemented, namely, the *FollowTrajLC*, *CarSimLC* and *PlatoonLC* longitudinal controllers. The first two require the *FollowTraj* and *CSCarFollower* car followers, respectively. The *PlatoonLC* longitudinal controller can use any one of *AICC*, *CP* or *DP* car followers.

A longitudinal controller operates in four control modes that are described below.

- The default mode is the *LONGC* mode. In this mode, based upon the *control_state* that the longitudinal controller is in, it calls upon different car following algorithms. The *FollowTraj* and *CarSimLC* controllers are always in the TRACKER *control_state*, whereas the *PlatoonLC* controller can be in one of FOLLOWER, TRACKER, SPLITTER and MERGER *control_states*. A *LongControl* object is in the *LONGC* mode when the vehicle is traveling in its own lane, is not preparing to make a lane change and is not affected by the lane changes of car units in neighboring lanes.

- When a longitudinal controller is required to stay a safe distance behind one or more car units in neighboring lanes, called *side_cars*, it is put in *LATC* mode. When necessary, it schedules lane change events for its *side_cars*.

- Before beginning a lane change and while executing a lane change, a car unit continuously monitors the car units in the neighboring lane and keeps track of the two car units flanking it, referred to as *side_car_rear* and *side_car_front*, in the neighboring lane. The longitudinal controller must be in the *LATC_MONITOR* mode for this. It puts the *side_car_rear* in *LATC* mode so that the vehicle stays a safe distance behind it. It stays a safe distance behind *side_car_front*.

- The *LATC_DEC* mode is similar to *LATC_MONITOR* in that the car unit maintains a safe distance behind *side_car_front* and asks *side_car_rear* to stay a safe distance behind it. It does not, however, continuously update *side_car_rear* and *side_car_front*.

The *PlatoonLC* controller is also responsible for performing merging and splitting maneuvers. It has a *busy_state* that is set to BUSY when it is performing such maneuvers. At other times it is set to NOT_BUSY. Only the *PlatoonLC* longitudinal controller has a defined protocol for performing such maneuvers. It has additional features that are discussed below. A car unit that has a *PlatoonLC* longitudinal controller is part of a platoon of car units. Each *PlatoonLC* longitudinal controller has a *control_state* and a *general_state* that it can be in. The four *control_states* are FOLLOWER, TRACKER, SPLITTER and MERGER. A *PlatoonLC* is a TRACKER when it is the leader of a platoon (possibly with only one car in it). It is a FOLLOWER if it is following another car unit in its platoon. It is a MERGER if it

is part of a platoon that is merging with a platoon ahead of it. It is a SPLITTER if it is part of a platoon that is splitting from a platoon ahead of it. A *PlatoonLC* can have up to four different car followers to execute different control laws depending on the *control_state* that it is in. The *FindConState()* method of the *PlatoonLC* object is periodically called by the *PltStateEv* event to determine the appropriate *control_state* for the longitudinal controller. Figure 9.1 shows the logic used to determine the *control_state*.

Each *PlatoonLC* can be in one of four *general_states*. It is a LEADER if it is the leader of a platoon. It is a FREE_AGENT if it is a one car platoon. It is in the PRE_LANE_CHANGE state just prior to changing lanes. This *general_state* is used only by the *QuickLatC* lateral controller. Finally, if it is in none of the above *general_states*, it is said to be an OTHER.

9.1.1 Side control

When the longitudinal controller is in one of its lateral control modes, the car follower executes a side control algorithm. At all times, in the *LATC_DEC, LATC_MONITOR* and *LATC* modes, the car follower uses the noncollision constraint (described in section 9.2.1) to make sure that it stays a safe distance behind the car unit in front of it, in its own lane. When the longitudinal controller is in the *LATC_DEC* or *LATC_MONITOR* mode, the *OneSideControl()* method of the car follower is invoked. This causes the car unit to decelerate to get a safe distance behind *side_car_front* in the near lane before the lane change begins. When safe to do so, it schedules a lane change for itself and then attempts to satisfy the noncollision constraint with *side_car_front* as well. When the longitudinal controller is in the *LATC* mode, the car unit keeps track of a list of *side_cars* in its near lanes. The car follower invokes its *ManySideControl()* method to stay a safe distance behind its *side_cars* and schedules lane change events for them, if necessary.

9.2 Car Following Algorithms

The principal function of all car following algorithms is to calculate the jerk input to a vehicle. The jerk is determined by the car follower as described below.

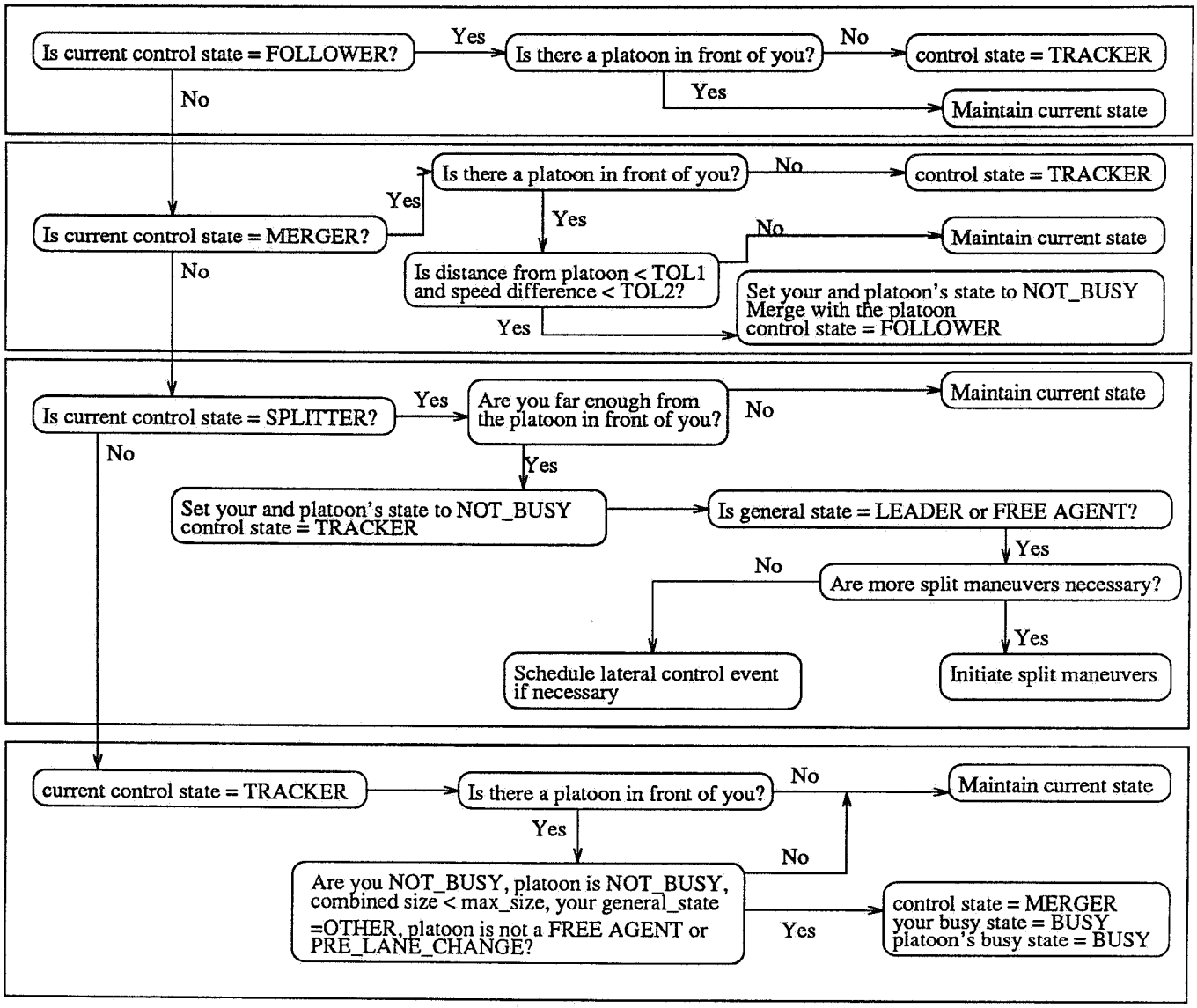


Figure 9.1: Determining the control state

- Determine the jerk $jerk_c$ according to the car following control law.
- Determine the noncollision jerk $jerk_{ncc}$, which is the largest jerk that can be applied without the car unit crashing into the car in front of it.
- Find $jerk_{tentative} = MIN(jerk_c, jerk_{ncc})$.
- Using $jerk_{tentative}$, find the realistic jerk $jerk_r$ by applying realistic limits on speed, acceleration and jerk. This is the jerk that is applied to the vehicle.

The following sections will present the calculation of the noncollision jerk and the realistic jerk. They will be followed by the calculation of the jerk by the car following algorithms used in *FollowTraj, CSCarFollower, AICC, CP* and *DP.*

9.2.1 Calculation of the noncollision jerk

Definition 9.2.1 *Given an integration time-step Δt, two vehicles L and F, with vehicle L leading vehicle F, the* **noncollision jerk** *$jerk_{ncc}$ is the largest jerk that can be applied to a vehicle F at time t, such that if vehicle L were to brake at its minimum (negative) jerk and deceleration at time $t + \Delta t$, and vehicle F were also to brake at its minimum (negative) jerk and deceleration at time $t + \Delta t + t_{br}$, where t_{br} is F's brake reaction time, then vehicle F would not collide with vehicle L.*

We assume the following limits on jerk, acceleration and velocity.

$$minimum\ jerk = J_{min}, \quad minimum\ acceleration = D_{min}, \quad minimum\ velocity = 0$$

Additionally, we make the assumption that there are no limits on the rate of change of jerk. This assumption is not strictly valid, but the calculation of the noncollision jerk is an approximate one, as seen below. We will make use of the following familiar equations of motion in our calculations.

$$s = ut + at^2/2 + jt^3/6 \tag{9.1}$$

$$v = u + at + jt^2/2 \tag{9.2}$$

$$v^2 = u^2 + 2as \ when\ jerk = 0 \tag{9.3}$$

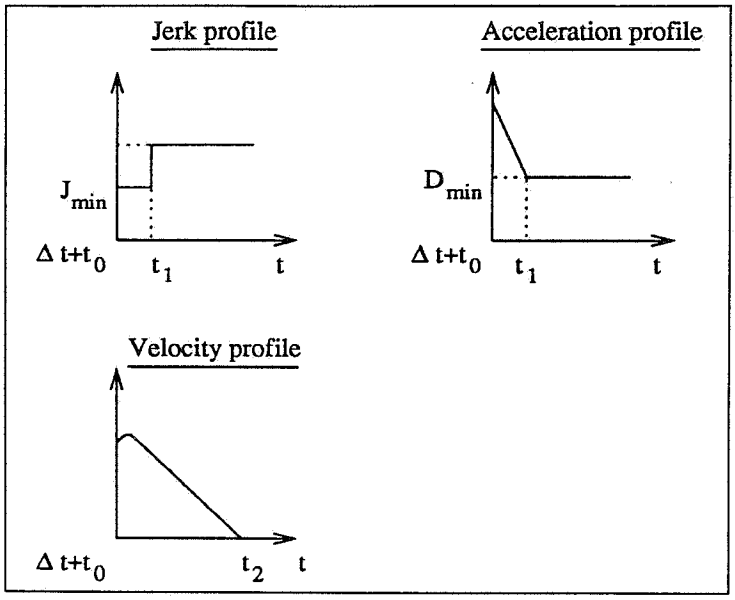


Figure 9.2: Vehicle state profiles after sudden braking

where s is the distance, u is the velocity, a is the acceleration and j is the jerk. We further assume for the calculations that follow that the brake reaction time t_{br} of the following vehicle is 0. This assumption is a reasonable one for the automated car followers, since the time required for the leader to communicate to the follower that it is braking, is negligible. A human car following algorithm, on the other hand, must take into account the t_{br} of the human driver. The effects of t_{br} on the noncollision jerk are considered later in this section. Assume that at time $t_0 + \Delta t$, the vehicle has velocity $v(t_0 + \Delta t)$ and acceleration $a(t_0 + \Delta t)$. At this time, a jerk of J_{min} is applied, until time t_1, when the acceleration falls to D_{min}. At time t_1, the jerk becomes 0, because the acceleration cannot drop any further. The velocity starts falling from time $t_0 + \Delta t$ and becomes 0 at time t_2 (see Figure 9.2). Let

s_1 = distance traveled in time $(t_1 - (t_0 + \Delta t))$ and

s_2 = distance traveled in time $(t_2 - t_1)$.

Then, the noncollision constraint can be stated as

$$s_{1L} + s_{2L} + x_L(t_0 + \Delta t) - (s_{1F} + s_{2F} + x_F(t_0 + \Delta t)) \geq len_F + \epsilon \tag{9.4}$$

where L refers to the leading vehicle, F refers to the following vehicle, $x(t)$ is the longitudinal position of a vehicle at time t, len_F is the length of the following vehicle and ϵ is a safety

distance. Let us now calculate s_1 and s_2.

$$\dot{a} = J_{min} \; and \; a(t_1) = D_{min} \Rightarrow t_1 = \frac{D_{min} - a(t_0 + \Delta t)}{J_{min}} + (t_0 + \Delta t) \tag{9.5}$$

Let $dt = t_1 - (t_0 + \Delta t)$. Then

$$dt = \frac{D_{min} - a(t_0 + \Delta t)}{J_{min}}$$

Using Equation (9.2)

$$v(t_1) = v(t_0 + \Delta t) + a(t_0 + \Delta t)dt + .5J_{min}dt^2 \tag{9.6}$$

Using Equation (9.1)

$$s_1 = v(t_0 + \Delta t)dt + a(t_0 + \Delta t)dt^2/2 + J_{min}dt^3/6 \tag{9.7}$$

Using Equation (9.3)

$$s_2 = \frac{-v(t_1)^2}{2D_{min}}$$

If $v(t_1) < 0$, however, we must calculate the time $\bar{\Delta}t_1$ at which $v(t + \bar{\Delta}t_1) = 0$ and find the distance $s_1 + s_2$ traveled in that time.

For the leader

$$\Delta t_{1,L} = \frac{D_{min,L} - a_L(t_0 + \Delta t)}{J_{min,L}} \approx \frac{D_{min,L} - a_L(t_0)}{J_{min,L}} \tag{9.8}$$

$$v_L(t_1) = v_L(t_0 + \Delta t) + a_L(t_0 + \Delta t)\Delta t_{1,L} + .5J_{min}\Delta t_{1,L}^2 \approx v_L(t_0) + a_L(t_0)\Delta t_{1,L} + .5J_{min}\Delta t_{1,L}^2 \tag{9.9}$$

If $v_L(t_1) > 0$, then

$$s_{1,L} \approx v_L(t_0)\Delta t_{1,L} + a_L(t_0)\Delta t_{1,L}^2/2 + J_{min}\Delta t_{1,L}^3/6 \; and \tag{9.10}$$

$$s_{2,L} = \frac{-v_L(t_1)^2}{2D_{min,L}} \tag{9.11}$$

else

$$\bar{\Delta}t_{1,L} \approx \frac{-2a_L(t_0) - \sqrt{4a_L(t_0)^2 - 8v_L(t_0)J_{min,L}}}{2J_{min,L}} \tag{9.12}$$

$$s_{1,L} + s_{2,L} \approx v_L(t_0)\bar{\Delta}t_{1,L} + a_L(t_0)\bar{\Delta}t_{1,L}^2/2 + J_{min,L}\bar{\Delta}t_{1,L}^3/6 \tag{9.13}$$

For the follower

$$\Delta t_{1,F} \approx \frac{D_{min,F} - a_F(t_0)}{J_{min,F}} \tag{9.14}$$

$$v_F(t_1) \approx v_F(t_0) + a_F(t_0)\Delta t_{1,F} + .5J_{min}\Delta t_{1,F}^2 \tag{9.15}$$

If $v_F(t_1) > 0$, then find $\bar{s}_{2,F}$, which is the maximum distance that the follower can travel in time $t_2 - t_1$ and still not collide with the leading car. Find the velocity $\bar{v}_F$ that would produce this $\bar{s}_{2,F}$. Use this to find the maximum allowable jerk. Otherwise if $v_F(t_1) < 0$, find the time $\bar{\Delta}t_{1,F}$ at which $v(t + \bar{\Delta}t_{1,F}) = 0$ and the distance $s_{1,F} + s_{2,F}$ traveled in this time. Then find the jerk from the non collision constraint.

If $v_F(t_1) >= 0$

$$\begin{aligned}
s_{1,F} &\approx v_F(t_0)\Delta t_{1,F} + a_F(t_0)\Delta t_{1,F}^2/2 + J_{min}\Delta t_{1,F}^3/6 && (9.16)\\
\bar{s}_{2,F} &\approx s_{1,L} + s_{2,L} + x_L(t_0) - s_{1,F} - x_F(t_0) - len_F - \epsilon && (9.17)\\
\Rightarrow \bar{v}_F &= \sqrt{-2\bar{s}_{2,F}D_{min,F}} && (9.18)\\
But\ \bar{v}_F = v_F(t_1) &= v_F(t_0 + \Delta t) + a_F(t_0 + \Delta t)\Delta t_{1,F} + .5J_{min}\Delta t_{1,F}^2 && (9.19)
\end{aligned}$$

Substituting for $a_F(t_0 + \Delta t)$ and $v_F(t_0 + \Delta t)$ in terms of $a_F(t_0)$, $v_F(t_0)$ and $j(t_0)$ gives

$$\begin{aligned}
\bar{v}_F &= v_F(t_0) + a_F(t_0)(\Delta t_{1,F} + \Delta t) + .5J_{min,F}\Delta t_{1,F}^2 + j(t_0)(.5\Delta t^2 + \Delta t\Delta t_{1,F})\\
\Rightarrow j(t_0) &= \frac{\bar{v}_F - v_F(t_0) - a_F(t_0)(\Delta t_{1,F} + \Delta t) - .5J_{min,F}\Delta t_{1,F}^2}{.5\Delta t^2 + \Delta t\Delta t_{1,F}} && (9.20)
\end{aligned}$$

else if $v_F(t_1) < 0$

$$\begin{aligned}
\bar{\Delta}t_{1,F} &\approx \frac{-2a_F(t_0) - \sqrt{4a_F(t_0)^2 - 8v_F(t_0)J_{min,F}}}{2J_{min,F}} && (9.21)\\
s_{1,F} + s_{2,F} &= v_F(t_0 + \Delta t)\bar{\Delta}t_{1,F} + a_F(t_0 + \Delta t)\bar{\Delta}t_{1,F}^2/2 + J_{min,F}\bar{\Delta}t_{1,F}^3/6 && (9.22)
\end{aligned}$$

Substituting for $a_F(t_0 + \Delta t)$ and $v_F(t_0 + \Delta t)$ in terms of $a_F(t_0)$, $v_F(t_0)$ and $j(t_0)$ gives

$$\begin{aligned}
s_{1,F} + s_{2,F} &= v(t_0)\bar{\Delta}t_{1,F} + a_F(t_0)\Delta t\bar{\Delta}t_{1,F} + a_F(t_0)\bar{\Delta}t_{1,F}^2/2 + J_{min,F}\bar{\Delta}t_{1,F}^3/6 +\\
&\quad j(t_0)(.5\Delta t^2\bar{\Delta}t_{1,F} + .5\Delta t\bar{\Delta}t_{1,F}^2) && (9.23)\\
&= A + j(t_0)B && (9.24)
\end{aligned}$$

Application of the noncollision constraint gives

$$j(t_0) = \frac{s_{1,L} + s_{2,L} + x_L(t_0) - x_F(t_0) - A - len_F - \epsilon}{B}$$

If the brake reaction time is taken into account, Equations (9.20) and (9.23) must be replaced by Equations (9.25) and (9.26), respectively.

$$\Rightarrow j(t_0) = \frac{\bar{v}_F - v_F(t_0) - a_F(t_0)(\Delta t_{1,F} + \Delta t + t_{br}) - .5J_{min,F}\Delta t_{1,F}^2}{.5(\Delta t + t_{br})^2 + (\Delta t + t_{br})\Delta t_{1,F}} \tag{9.25}$$

$$\begin{aligned} s_{1,F} + s_{2,F} &= v(t_0)\bar{\Delta}t_{1,F} + a_F(t_0)(\Delta t + t_{br})\bar{\Delta}t_{1,F} + a_F(t_0)\bar{\Delta}t_{1,F}^2/2 + J_{min,F}\bar{\Delta}t_{1,F}^3/6 + \\ & \quad j(t_0)(.5(\Delta t + t_{br})^2\bar{\Delta}t_{1,F} + .5(\Delta t + t_{br})\bar{\Delta}t_{1,F}^2) \end{aligned} \tag{9.26}$$

9.2.2 Calculation of the realistic jerk

Figure 9.3 shows the sequence of steps involved in the calculation of the realistic jerk. The limits on acceleration $acc2$ and deceleration $acc4$ are given in Table 9.1. The acceleration that will cause the maximum speed to be attained is simply

$$acc3 = (max\_speed - current\_speed)/\Delta t$$

To find $acc5$, first find the jerk that makes the velocity go to zero.

$$v(t + \Delta t) = v(t) + a(t)\Delta t + .5j(t)\Delta t^2 \tag{9.27}$$

$$v(t + \Delta t) = 0 \Rightarrow j(t) = \frac{-v(t) - a(t)\Delta t}{.5\Delta t^2} \tag{9.28}$$

$$\Rightarrow a(t + \Delta t) = acc5 = -2v(t)/\Delta t - a(t) \tag{9.29}$$

9.2.3 Follow a prescribed trajectory

In this case, the path that the vehicle must follow, in terms of longitudinal acceleration versus time, is prescribed. *FollowTraj* calculates the $jerk_c$ at each time step, so that the vehicle's trajectory will be as close as possible to the prescribed one.

Table 9.1: Typical acceleration and deceleration rates on a level road

Speed in mph	Speed in m/s	Acceleration in m/s^2	Deceleration in m/s^2
0-15	0-6.71	2.684	-2.3699
15-30	6.71-13.41	1.6775	-2.0557
30-40	13.41-17.88	1.5768	-1.4762
40-50	17.88-22.35	1.2718	-1.4762
50-60	22.35-26.82	0.9394	-1.4762
>60	>26.82	0.6374	-1.4762

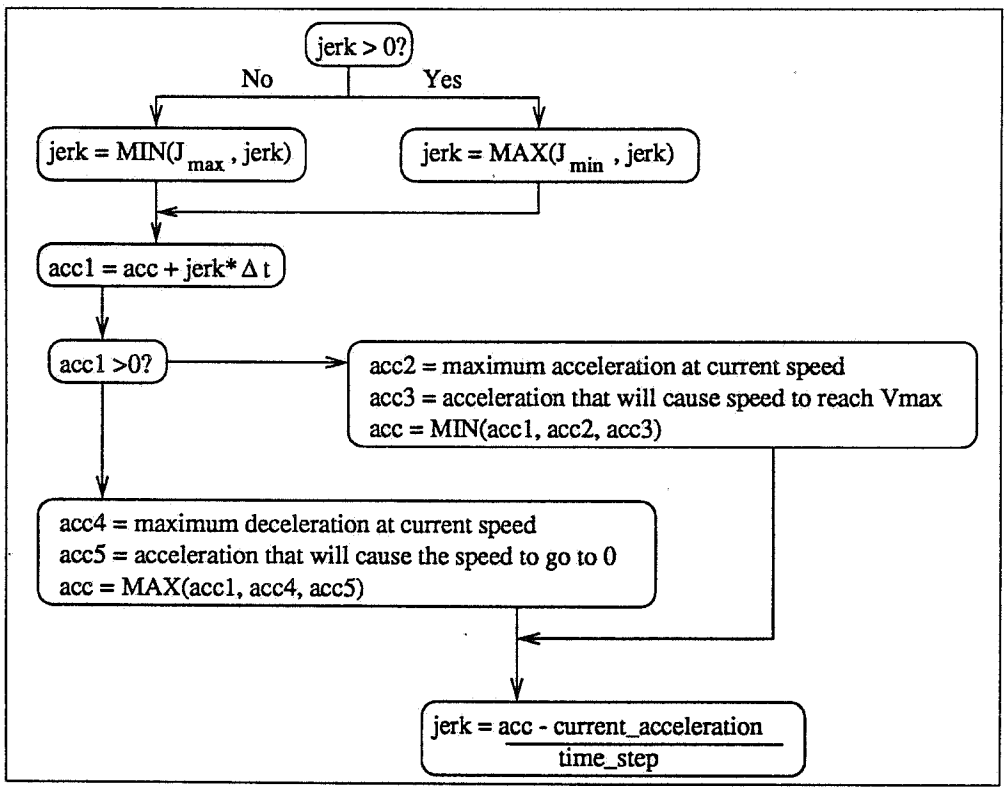


Figure 9.3: Calculation of the realistic jerk

9.2.4 CSCarFollower's human car follower

The car following algorithm used in the *CarSim* simulator [46] models human car following behavior. The car following algorithm used in the class *CSCarFollower* is based on *CarSim's* algorithm. While *CarSim* calculates the acceleration input to the vehicle, *CSCarFollower* calculates the jerk input. The notation used is the following.

- j_1 jerk to track desired speed
- j_2 jerk after a stop
- j_3 jerk to maintain the desired distance headway
- j_4 jerk from the non collision constraint
- a_5 acceleration at start-up time = .61 m/s^2
- j_6 realistic jerk

The algorithm used by *CSCarFollower* is shown in Figure 9.4. The jerks j_1 and j_3 are calculated as follows.

$$\begin{aligned} j_1(t) &= (u_{desired} - u(t) - a(t)\Delta t)\frac{2}{\Delta t^2} \\ j_3(t) &= (x_L(t) + u_L(t)\Delta t + .5a_L\Delta t^2 + \frac{j_L\Delta t^3}{6} - x_F(t) - u_F(t)\Delta t \\ &\quad -.5a_F\Delta t^2 - len_F - K)\frac{6}{\Delta t^3} \end{aligned}$$

where len_F is the length of the following car and K is the desired headway. The desired speed $u_{desired}$ is randomly assigned to each *CarSimLC* longitudinal controller and is drawn from a truncated normal distribution with a mean of 55 mph and a standard deviation of 5 mph. The allocated speed is truncated between the limits of 45 mph and 75 mph. Each *CSCarFollower* is given a brake reaction time t_{br} that is taken from a distribution given in [46] and is used in the calculation of the noncollision jerk j_4. j_2 is calculated via the stop-and-go routine shown in Figure 9.5.

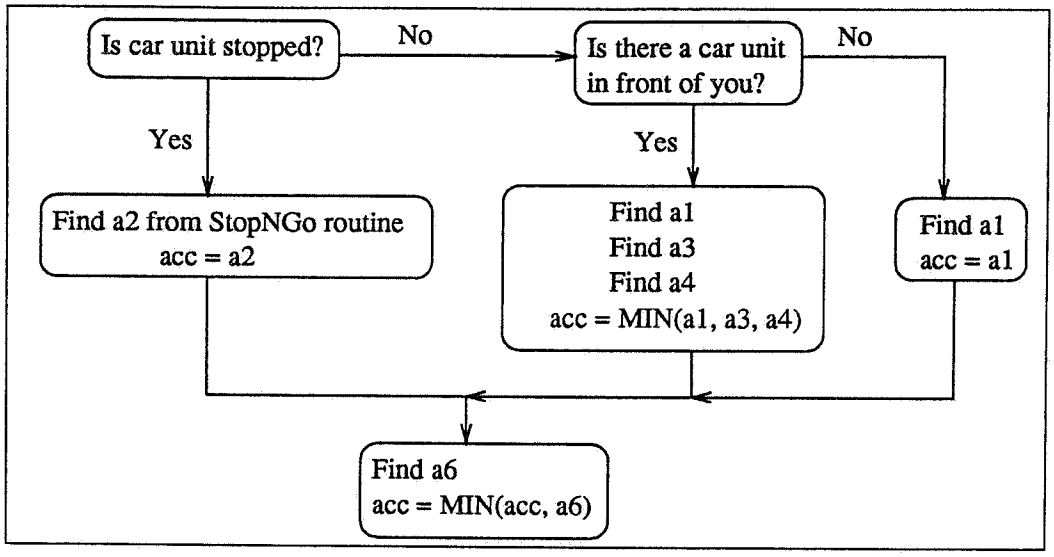


Figure 9.4: The *CSCarFollower* car following algorithm

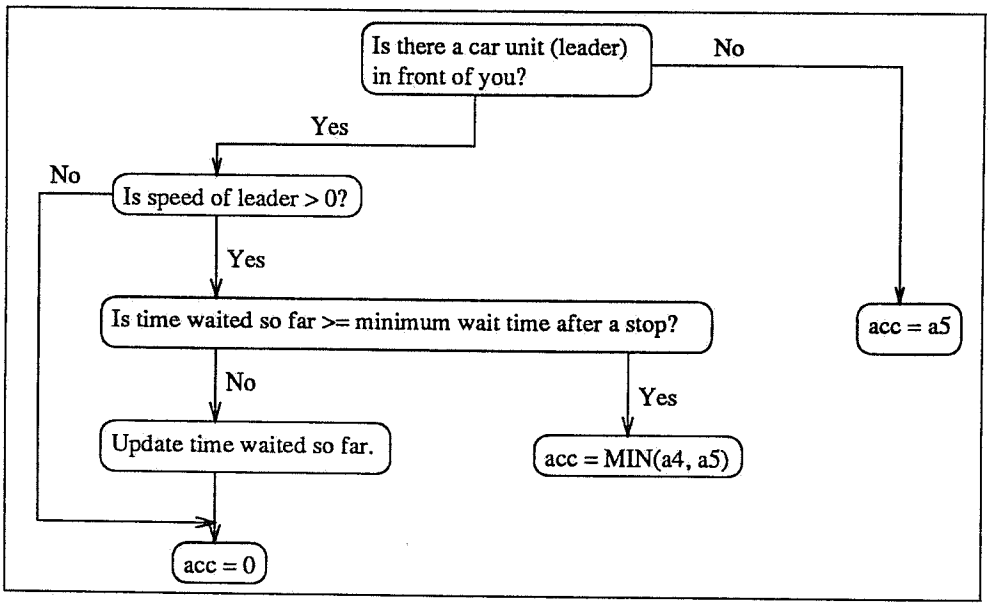


Figure 9.5: The *CSCarFollower* stop-and-go routine

9.2.5 Vehicle model for the remaining car followers

The longitudinal motion of the vehicle is modeled using the following model proposed by Sheikholeslam [56] and subsequently adopted by Chien and Ioannou [54] and Ren and Green [55]. The notation is as follows

x	longitudinal position of the vehicle in m
v	velocity of the vehicle in m/s
a	acceleration of the vehicle in m/s^2
m	mass of the vehicle in kg
F_e	engine traction force
$F_{wind} = -K_w v^2$	force due to wind resistance
K_w	aerodynamic drag coefficient
$F_{drag} = f(v)$	constant force due to mechanical drag, but 0 for 0 velocity
τ	engine time constant
u	throttle/brake input

The engine dynamics and the vehicle model are as follows.

$$F_e = ma + F_{wind} + F_{drag} \tag{9.30}$$

$$\dot{F}_e = -\frac{F_e}{\tau} + \frac{u}{\tau} \tag{9.31}$$

From (9.30) and (9.31),

$$\dot{a} = -\frac{1}{m}(-2K_w v\dot{v} + \frac{F_e}{\tau} - \frac{u}{\tau})$$

Substituting for F_e from (9.30) in the above gives

$$\dot{a} = \frac{-1}{m\tau}[-2\tau K_w v\dot{v} + ma + K_w v^2 + F_{drag} - u]$$

If we assume that we can measure v, a, m and can calculate F_{wind} and F_{drag}, we can perform feedback linearization of the above model by choosing the brake/throttle input u to be

$$u = -m\tau(c + 2\tau v a K_w - ma - K_w v^2 - F_{drag})$$

Now the dynamics of the linearized vehicle are simply given by the following equations with the control input being the jerk c.

$$\dot{x} = v$$

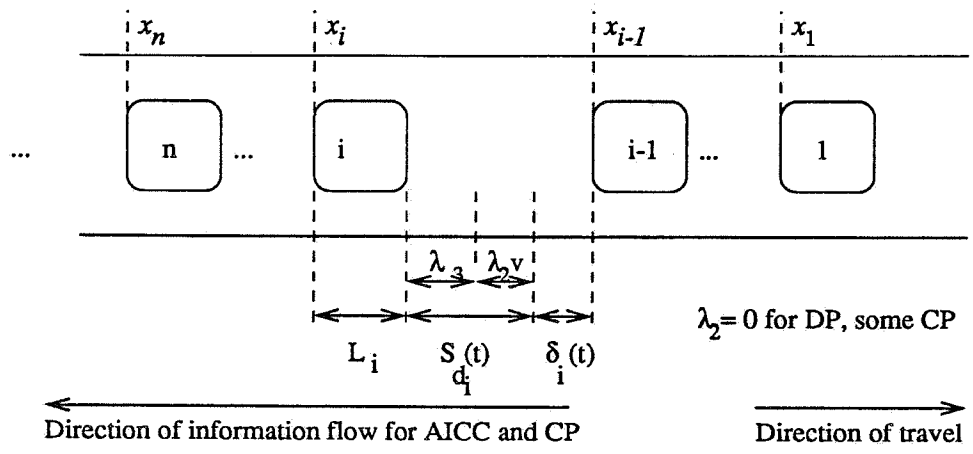


Figure 9.6: A platoon of n vehicles

$$\begin{aligned} \dot{v} &= a \\ \dot{a} &= c \end{aligned} \tag{9.32}$$

The *AICC*, *CP* and *DP* car followers assume that vehicles travel in a platoon. They attempt to regulate the distance headway about some desired headway. The platoon size is a parameter that can be chosen. Figure 9.6 shows the platoon intervehicle spacing $\delta_i(t)$ and the desired intervehicle spacing S_{d_i}.

9.2.6 Automated intelligent cruise control

In the *AICC* system, each vehicle is assumed to be able to measure the relative distance, relative velocity and estimate or measure the relative acceleration between itself and the vehicle immediately in front of it (see Figure 9.6). A vehicle attempts to maintain the following safe distance between it and the vehicle in front of it.

$$S_{d_i} = \lambda_2 v_i + \lambda_3$$

This rule combines the constant time headway rule ($S_{d_i} = \lambda_2 v_i$) with the constant separation rule ($S_{d_i} = \lambda_3$). The control law proposed in [54] is

$$c_i(t) = \frac{1}{1 + \lambda_2 C_a}(C_p \delta_i(t) + C_v \dot{\delta}(t) + C_a(a_{i-1}(t) - a_i(t)) + K_v v_i(t) + K_a a_i(t))$$

$$\begin{aligned}\delta_i(t) &= x_{i-1}(t) - x_i(t) - (L_i + \lambda_3 + \lambda_2 v_i(t)) \\ \dot{\delta}_i(t) &= v_{i-1}(t) - v_i(t) - \lambda_2 a_i(t) \quad (for\ i = 2, 3, \ldots, n)\end{aligned} \tag{9.33}$$

where C_p, C_v, C_a, K_v and K_a are design constants chosen to satisfy stability and performance requirements. The following values proposed in [54] are used.

$$K_v = 0,\ C_p = 224,\ C_v = 127.2,\ C_a = 5,\ K_a = -3.56,\ \lambda_2 = .2seconds,\ \lambda_3 = 2m$$

For *AICC*, $jerk_c = c_i(t)$. The jerk that is finally applied to the vehicle is calculated according to the steps presented at the beginning of Section 9.2.

9.2.7 Continuous platooning

At the heart of continuous platooning [55] is a "relay" system for conveying information through a chain of vehicles, so that each vehicle has state information (relative position, velocity and acceleration) from a number of vehicles in front of it. The ith vehicle in a chain of vehicles receives information about the L preceding vehicles from the vehicle immediately in front of it. It then strips off the information from the Lth car in front of it, appends its own information and transmits this new block of information to the vehicle behind it. When the preview parameter $L = 1$, the control scheme is similar to *AICC*. Unlike *DP*, which is discussed next, there is no leader classification and, thus, the direction of information flow is unidirectional and opposite to the direction of travel. As in *AICC*, the vehicle controller attempts to regulate the distance between it and the vehicle immediately in front of it, about a headway that is a combination of constant time and constant distance headways, i.e.

$$\delta_i(t) = x_{i-1}(t) - x_i(t) - L_i - (H + \lambda v_i(t)) \tag{9.34}$$

The control law is

$$c_i(t) = \sum_{j=1}^{L} K_{p_j}\delta_{i-(j-1)}(t) + K_{v_j}\dot{\delta}_{i-(j-1)}(t) + K_{a_j}\ddot{\delta}_{i-(j-1)}(t) \tag{9.35}$$

From Equation (9.34),

$$\begin{aligned}\dot{\delta}_i &= v_{i-1} - v_i - \lambda a_i \\ \ddot{\delta}_i &= a_{i-1} - a_i - \lambda \dot{a}_i \\ &= a_{i-1} - a_i - \lambda c_i\end{aligned} \tag{9.36}$$

Table 9.2: CP controller parameters

Controller	K_{p_1}	K_{v_1}	K_{a_1}	K_{p_2}	K_{v_2}	K_{a_2}	K_{p_3}	K_{v_3}	K_{a_3}	λ
c)	205.1	250.0	21.5	-	-	-	-	-	-	.1
e)	250.0	250.0	18.2	212.6	208.5	-9.43	-	-	-	.1
g)	208.6	250.0	20.9	204.3	264.2	1.57	97.4	119.4	.34	.1
h)	250.0	250.0	94.9	-	-	-	-	-	-	0
j)	250.0	250.0	94.9	248.6	244.2	94.0	-	-	-	0
l)	249.8	249.8	99.9	247.6	250.0	99.9	249.8	247.3	98.7	0

Substituting Equations (9.34) and (9.36) in (9.35) and solving for c_i gives

$$c_i(t) = \frac{1}{1+\lambda K_a}(K_{p_1}\delta_i + K_{v_1}\dot{\delta}_i + K_{a_1}(a_{i-1} - a_i) + (\sum_{j=2}^{L} K_{p_j}\delta_{i-(j-1)} + K_{v_j}\dot{\delta}_{i-(j-1)} + K_{a_j}\ddot{\delta}_{i-(j-1)})) \tag{9.37}$$

The control law is the same for each vehicle, and when a vehicle has fewer than L vehicles within its visual range, a value of 0 is assumed in place of any missing information. Table 9.2 taken from [55] shows the controller parameters for $L = 1, 2$ and 3. For each value of L, two controllers were designed; one corresponds to regulation about a constant distance headway ($\lambda = 0$), the other includes a time headway as well ($\lambda \neq 0$).

9.2.8 Discrete platooning

The discrete platooning control law proposed by Sheikholeslam [56] differs from *AICC* and *CP* by making use of a platoon leader, thus introducing discrete, as opposed to continuous, platoons. The vehicle controller attempts to regulate the distance between it and the vehicle in front of it about a fixed distance headway, i.e.,

$$\delta_i(t) = x_{i-1}(t) - x_i(t) - L_i - H \tag{9.38}$$

From Equation (9.38) we get

$$\begin{aligned} \dot{\delta}_i &= v_{i-1} - v_i \\ \ddot{\delta}_i &= a_{i-1} - a_i \end{aligned} \tag{9.39}$$

The control law includes lead vehicle information, in addition to the state information from the vehicle immediately in front.

$$c_i(t) = K_{p_1}\delta_i(t) + K_{v_1}\dot{\delta}_i(t) + K_{a_1}\ddot{\delta}_i(t) + K_{v_{lead}}(v_{lead}(t) - v_i(t)) + K_{a_{lead}}(a_{lead}(t) - a_i(t)) \quad (9.40)$$

The controller parameters used are

$$K_{p_1} = 120,\ K_{v_1} = 49,\ K_{a_1} = 5,\ K_{v_{lead}} = 25,\ K_{a_{lead}} = 10$$

9.2.9 Cruise control

The cruise control car following law attempts to regulate the velocity of a vehicle around a specified velocity. This is the car following law used by the longitudinal controller when it is in the *TRACKER control_state*. The control law used is

$$c = K_1\delta + K_2\dot{\delta} \quad (9.41)$$

where

$$\begin{aligned} \delta(t) &= v_{desired} - v(t) \\ \dot{\delta}(t) &= -a \end{aligned} \quad (9.42)$$

The control law parameters chosen were

$$K_1 = 102,\ K_2 = 20.2$$

9.3 Lateral Control Logic

Two types of lateral controllers have been implemented. They are called Simple lateral control and Quick lateral control. The primary difference between the two is that when a vehicle that is part of a platoon wishes to change lanes, in the first control it becomes a free

agent, but in the second control, it does not physically break away from the platoon, as a free agent does, before changing lanes. In what follows, the near lane is the lane into which the car unit wishes to maneuver. The far lane is the lane adjacent to the near lane (it is not the lane that the vehicle is currently in). A lateral controller has the following states that decide its behavior.

- com_state. This takes values TO_CHANGE when the controller needs to change and NOT_TO_CHANGE when it does not.

- change_state. This takes values ALLOWED_TO_CHANGE when the car unit is allowed to change and NOT_ALLOWED when it is not. The car unit is not allowed to change at certain times, for example, when a car unit in the far lane has already begun a lane change into the near lane.

- busy_state. This takes a value of BUSY when the car unit is in the process of executing a lane change and a value of NOT_BUSY when it is not.

The algorithm is best expressed by means of the flow chart in Figure 9.7. If the lateral controller is NOT_BUSY and is ALLOWED_TO_CHANGE, it first checks to see if its longitudinal state permits lane changing. When the longitudinal controller is a *PlatoonLC*, for *SimpleLatC*, the longitudinal state permits lane changing when the the car unit is a FREE_AGENT. For *QuickLatC* control, the longitudinal state permits lane changing when the car unit is not part of a platoon, but not necessarily physically distant from neighboring platoons.

For other types of longitudinal controllers, the longitudinal state always permits lane changing. The lateral controller then checks to see if the near and far lanes are safe. The logic used to check if the near and far lanes are safe is shown in Figures 9.8 and 9.9. If there are no near neighbors in the near and far lane that are a threat to lane changing, the lateral control executes *FarLaneControl()*, shown in Figure 9.10. Then, the longitudinal controller is put in LATC_MONITOR mode. The *LaneChanger* is then commanded to begin the lane change. If the far lane is safe, but the near lane is not, *NearLaneControl()*, as shown in Figure 9.11, is executed. Four different scenarios (not numbered sequentially) are considered

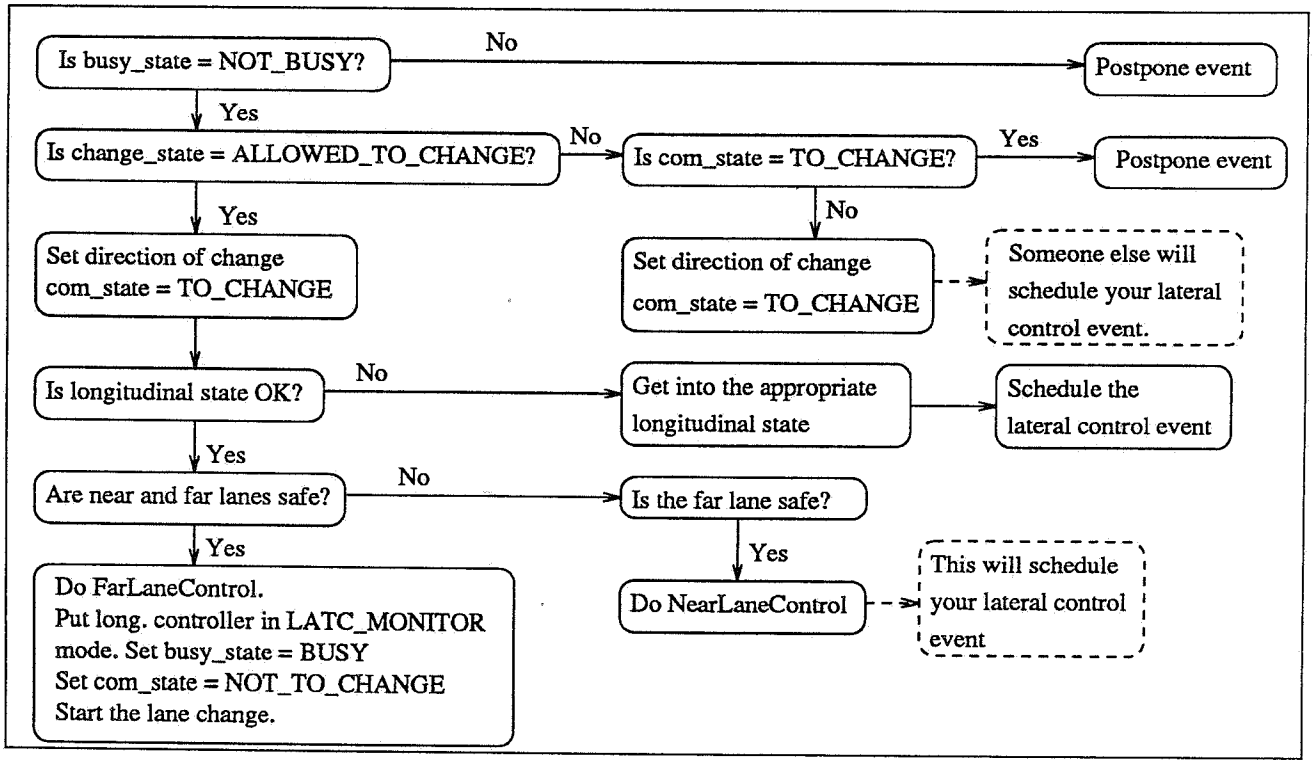


Figure 9.7: The lateral control algorithm

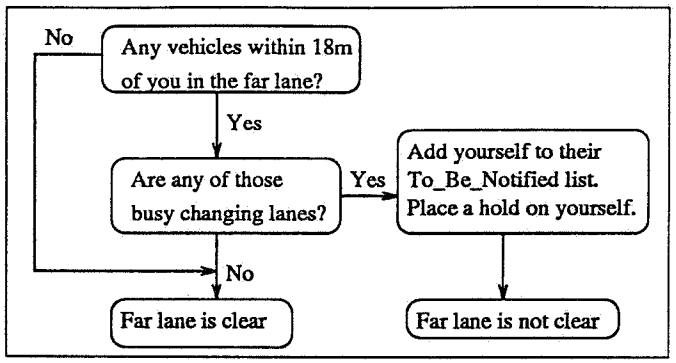


Figure 9.8: Checking out the far lane.

in *NearLaneControl()*. The steps carried out in scenarios 2 and 3 are shown in Figures 9.12 and 9.13, respectively. Scenario 8 is a combination of scenarios 2 and 3. The steps followed for scenario 7 are shown in Figure 9.14. The control actions carried out in these scenarios prepare the car unit and the neighboring units for a safe lane change. When it is safe, as defined below, a lane change is scheduled.

Definition 9.3.1 *Two car units $CU1$ and $CU2$, with $CU1$ ahead of $CU2$ but in a different lane, are a safe distance apart for lane changing purposes if the following holds.*

$$x_1(t_0) - x_2(t_0) >= L_2 + safe\_dist \quad AND \quad x_1(t_0 + T_1) - x_2(t_0 + T_2) >= L_2 + Tol$$

where $x_i(t)$ is the longitudinal position of car unit i at time t, L_2 is the length of $CU2$, t_0 is the time at which the lane change is begun, $safe\_dist$ and Tol are specified distances and T_i is the time required for car unit i to come to a stop if it were to brake at its maximum capacity after the current time step.

The calculation of $x_i(t_0 + T_i)$ is similar to that done in the calculation of the noncollision constraint. Once the lane change is completed, post-processing as shown in Figure 9.15 must be carried out. This completes the lateral control logic.

9.3.1 Lane changing algorithms

Only one kind of lane changer has been implemented. A lane change takes three seconds, and the trajectory followed corresponds to an inverse tangent function. The change in

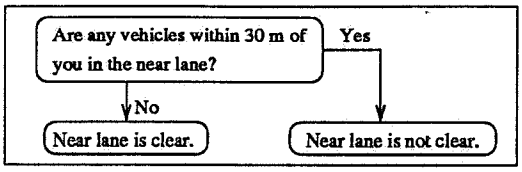


Figure 9.9: Checking out the near lane

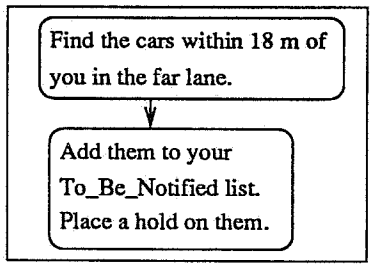


Figure 9.10: Far lane control

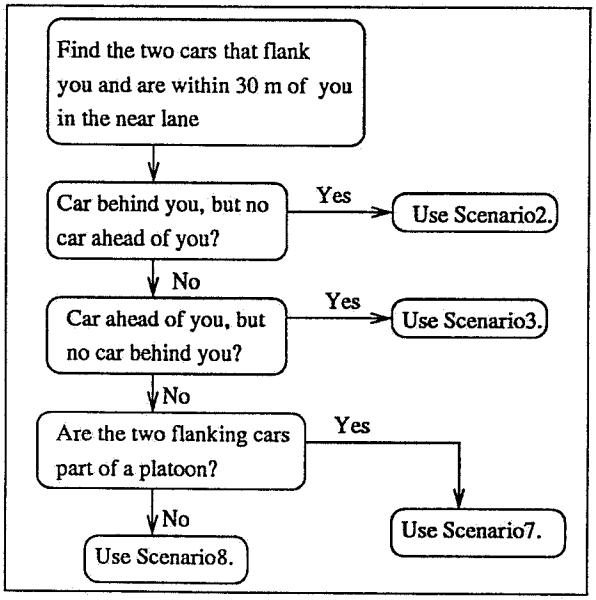


Figure 9.11: Near lane control

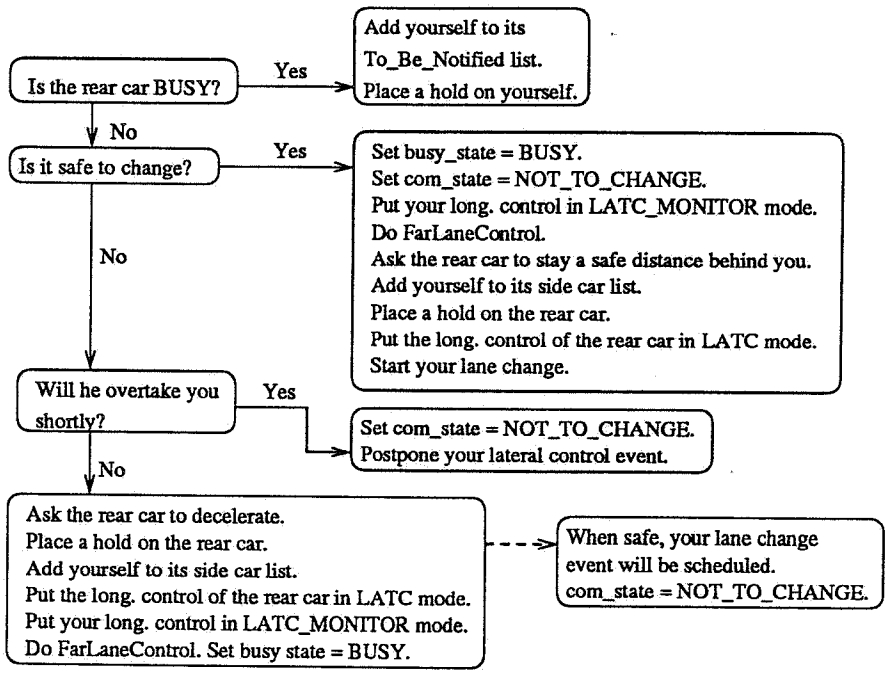


Figure 9.12: Scenario 2 of near lane control

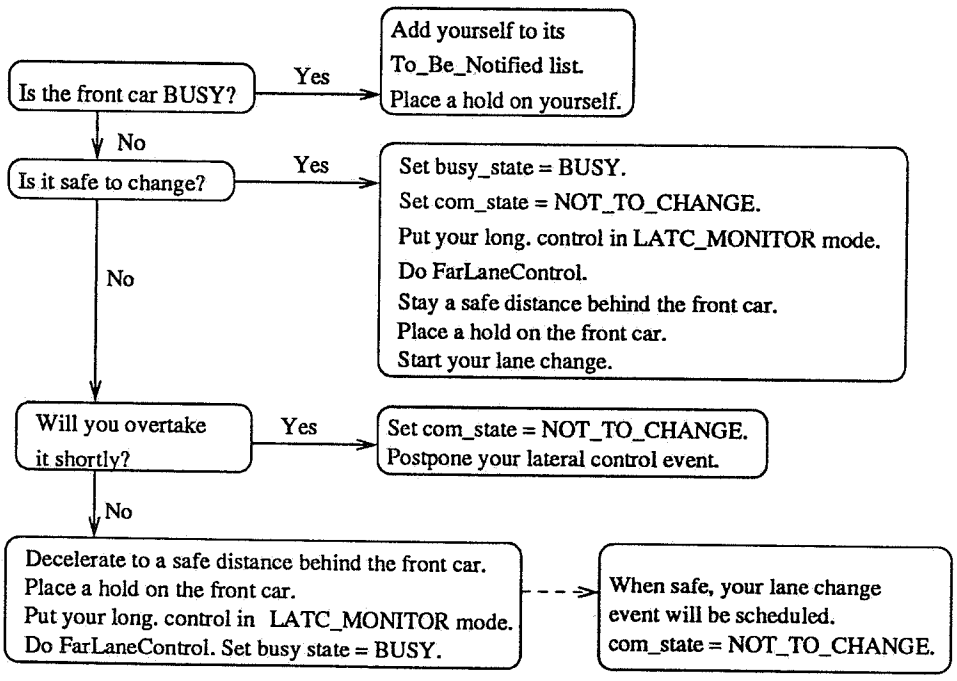


Figure 9.13: Scenario 3 of near lane control

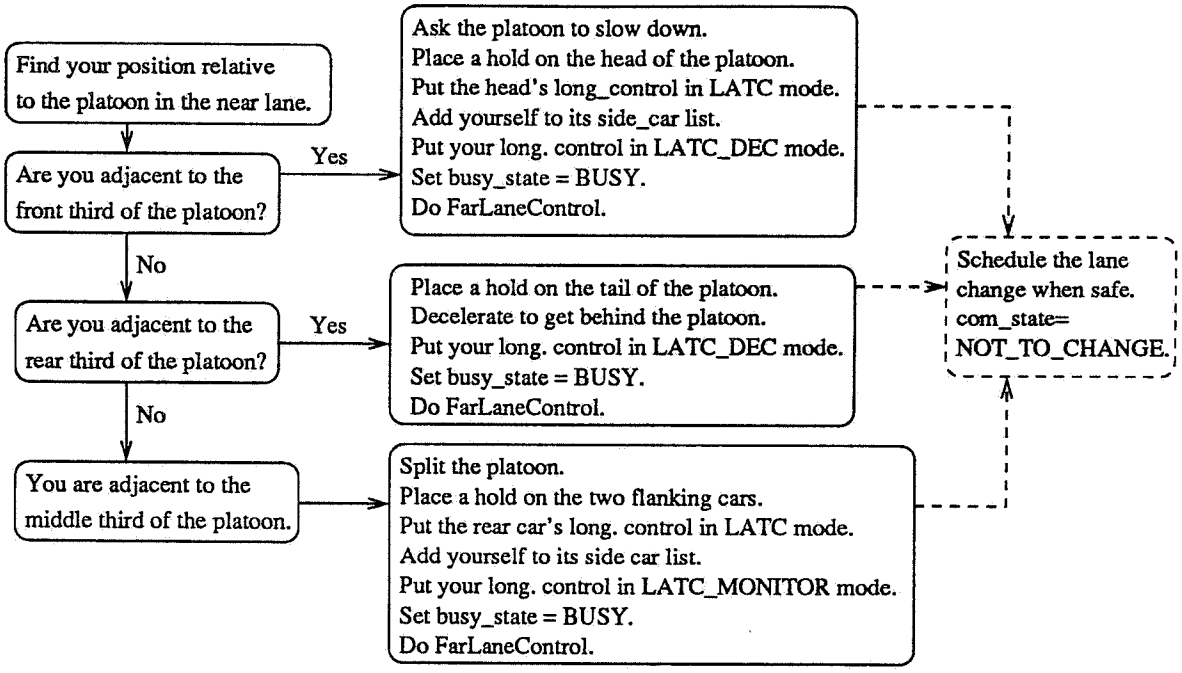


Figure 9.14: Scenario 7 of near lane control

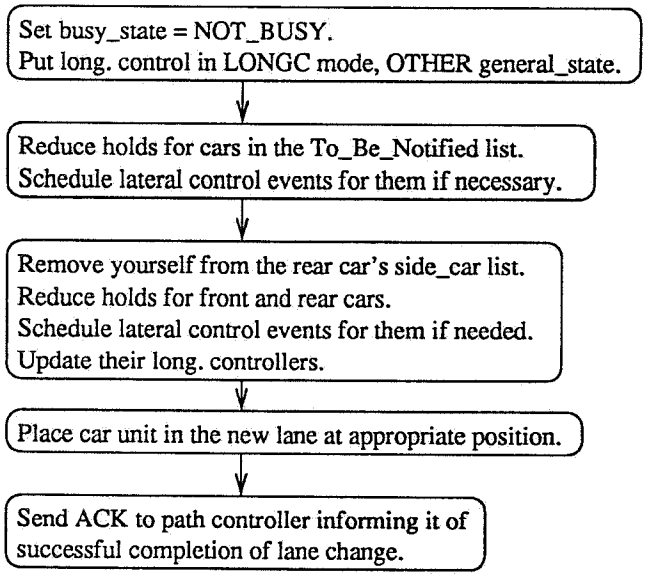


Figure 9.15: Post-processing after lateral control

position $\Delta y(t)$ is computed as follows.

$$\begin{aligned} W(t) &= 10^{4*t/T_c-2} \quad 0 \leq t \leq T_c \\ \Delta y(t) &= \frac{lane\_width}{\pi/2} tan^{-1}(W(t)) \quad 0 \leq t \leq T_c \end{aligned}$$

where T_c is the time required to perform a lane change.

9.4 Path Control Logic

The path control object's *Control()* method is called upon periodically by the path control event. It decides whether a lane change is required and if so, it schedules a lateral control event. Each path controller has an *ack_state* that takes two values ACK and NACK. When it sends a request to the lateral controller, this state is set to NACK. When the lateral controller completes the lane change, it sends an ACK to the path controller. While the path controller is in the NACK state, it does not schedule any more lateral control events. If a car unit misses its exit, the information is stored in a file. It then attempts to exit at the next exit. Four kinds of path controllers have been implemented.

- The human path control mimics the actions of an "average" human. Upon entry into the highway, it maneuvers the car unit from the transition lane to the inner lanes. When the car unit is far from its exit, it perceives a *need to overtake* if its speed is less than the desired speed due to cars in front of it. It then chooses the lane to the right or left of the current lane depending on the existence of the lane and the number of car units close by, in each lane. When it gets close to its exit, the path control maneuvers the car unit to the right to get to the exit lane.

- A compliant path controller follows a prescribed path. This path is given in terms of a sequence of $<$ *Highway number, section number, lane number* $>$ triples and conforms to a general lane assignment. When the *Control()* method is invoked, the path controller finds the current section and lane, compares them to the prescribed section and lane and then commands the lateral controller to move to the right, the left or not move at all, in order to get the car unit to the prescribed lane.

- A semicompliant controller is also given a prescribed path similar to the Compliant controller, but it adheres to its path only with a certain probability.

- An automated path controller follows a constant lane path. It obtains this lane assignment from a segment controller. On entering the highway, it maneuvers the car unit to the assigned lane. When the car unit gets close to its exit, this path controller maneuvers the car unit to the exit lane.

9.5 Segment Control Logic

A segment controller is an object that contains the path assignments for a segment of highway. These path assignments are constant lane and subclasses of constant lane assignments. A segment controller has a *GetAssLaneId()* method that is invoked by the *AutomatedPC* path controllers of a car unit. The lane assignments are obtained from another object (the program that computes the optimal lane assignments, for example). Different types of segment controllers have been defined corresponding to the various subclasses of constant lane strategies.

- A *ConstLaneSC* object is a constant lane segment controller. It contains a list $F = f(i,j.m)$, $0 \leq i \leq K_1, 1 \leq j \leq K_2, m = 1, \ldots, lanes$, where $f(i,j,m)$ is the fraction of flow $N(i,j)$ that is assigned to lane m. When it receives a request from a path controller for an assignment, it probabilistically decides (based on $f(i,j,m)$) which lane to assign to that car unit.

- The *PartStratSC* is a partitioned strategy (without splitting) segment controller. Here, the assignments are either destination monotone, origin monotone or monotone. For an m lane highway, the lane assignments are described using row partitions $[P_1, P_2, \ldots, P_m]$ or column partitions $[R_1, R_2, \ldots, R_m]$. The lanes are assigned as follows. For a destination monotone assignment with entry i and exit j,

$$j \leq P_1(i) \Rightarrow L_1$$
$$P_1(i) < j \leq P_2(i) \Rightarrow L_2$$

$$P_2(i) < j \leq P_3(i) \Rightarrow L_3$$

$$\ldots$$

$$P_{m-1}(i) < j \Rightarrow L_m \tag{9.43}$$

For an origin monotone assignment,

$$i \geq R_1(j) \Rightarrow L_1$$

$$R_1(j) > i \geq R_2(j) \Rightarrow L_2$$

$$R_2(j) > i \geq R_3(j) \Rightarrow L_3$$

$$\ldots$$

$$R_{m-1}(j) > i \Rightarrow L_m \tag{9.44}$$

- The *DestSplitSC* is a partitioned strategy with splitting controller. This is similar to the partitioned strategy segment controller except that the flow to the partitions may be split into two lanes for a destination monotone strategy and flow from the partitions may be split into two lanes for an origin monotone strategy. Thus, in addition to the $[P_1, P_2, \ldots P_{m-1}]$ partitions, a *DestSplitSC* controller contains a matrix $S(j,l), j = 0, 1, \ldots, K_1, l = 1, \ldots, m$ that defines the split. $S(j,l)$ is the fraction of vehicles going from j to $P_l(j)$ that will be assigned lane l, and $(1 - S(j,l))$ is the fraction that will be assigned lane $l+1$. For an *OrigSplitSC* controller, $S(j,l)$ is the fraction of vehicles going to j from $R_l(j)$ that will be assigned lane l and $(1 - S(j,l))$ is the fraction that will be assigned lane $l+1$.

CHAPTER 10

FRONT END FOR THE SIMULATOR

10.1 Why a Front End is Necessary

The *interface* of a class provides all of the information required to use that class. To set up an AHS simulation, a user must read the interface as specified in the class's *.h* file and then call the necessary *methods*. However, this is not necessarily the best way to accomplish this because:

1. This information is often more than what is required for setting up simple simulations.
2. The code design requires a certain sequencing of operations and sometimes (rarely), some knowledge of the *implementation* of the routines is required as well, in order set up a test program.
3. Invalid objects may be inadvertently created. For example, a car unit that uses a *DP* car follower requires *DPVision*. If it is provided a *HumanVision* instead, the simulation will behave unpredictably.

What is required, therefore, is a program called a *front end* that provides a safe and user-friendly way of using the various classes to run a simulation. By definition, its power is limited, because we have sacrificed the power of pure C++ in order to gain simplicity of use. However, to prevent the front end from being too restrictive in its use, it has been designed

to be easily extendible. The Tcl language (see [14] for a complete description) provides the capability for creating such a front end.

10.1.1 Compiled versus interpreted front ends

Yet another reason for using Tcl (an interpreted language) is the advantage that interpreted languages enjoy over compiled languages. (The disadvantage is that they take longer to execute than compiled languages, but if the core functions are written in a compiled language, execution time will not be an issue.) The front end may be designed in C++, just as the base classes were, by creating a set of *commands* in C++ that make use of the classes in the AHS class library. Simulation programs would then be written using these *commands* to simulate a particular configuration of a highway system. When a different configuration is to be simulated, the test program must be modified and first recompiled for the changes to take effect.

With an interpretive test program (called a *script*), however, no recompilation is necessary for the changes to take effect. The test program can be executed immediately after making the changes. This may result in a significant savings in time and allow the rapid prototyping of new applications. For this reason, *scripts* are also easier to debug. The Tcl language is an interpreted language and has been used to create the front end. Another example of an interpreted language is the set of commands used in Matlab.

10.2 Creating a Front End in Tcl

As defined by Ousterhout [14], "Tcl is a simple scripting language for controlling and extending applications; its name stands for 'tool command language'. Tcl provides generic programming facilities, such as variables and loops and procedures, that are useful for a variety of applications. Furthermore, Tcl is embeddable. Its interpreter is a library of C procedures that can easily be incorporated into applications, and each application can extend the core Tcl features with additional commands for that application."

Chapters 28 through 35 of [14] provide an excellent guide to creating Tcl applications; therefore, the material presented will not be repeated here. The basic philosophy, however, is as follows. Tcl itself is written in C. It consists of a set of commands (much the way that Matlab and *csh* do) that form the language Tcl. When a core command is entered in Tcl, it (typically) invokes a C function. To create a new command, we simply need to bind that new command to a C or C++ function. In brief, the following steps must be followed to create the front end (a Tcl application).

1. Create new Tcl commands and bind them to C++ functions. The application-specific portion of the front end is composed entirely of new Tcl commands. These include, among others (see Section 10.3 for a complete listing) *highway*, *section*, *cargen* and *run*. The command consists of a command_name, *highway* for example, and arguments following the command, *I57* for example. The command set should be made comprehensive enough to create test scripts to simulate traffic situations of interest.

2. The variable names in Tcl, *I57* for example, must be linked to C++ objects through the use of *Hash Tables* [14]. This will provide to access to the C++ object through the use of its Tcl name.

3. The new commands must be compiled with the AHS library C++ classes to create a shell (in this case called *simsh*) within which the Tcl commands can be executed.

10.2.1 Classes used in the front end

As mentioned in the previous section, when a new command is created in Tcl, it must be bound to a C++ function. The following classes were designed to create the application-specific commands in the front end and to bind them to appropriate functions in the AHS class library. Only a brief description of the classes is provided below. Appendix D contains all .h files used in the front end. A complete listing of the Tcl commands and their usage is provided in Tables 10.1 through 10.4.

Table 10.1: Tcl commands

Command	Description
	Commands pertaining to the highway
`highway` *hwyname*	Creates a highway with name *hwyname.*
`section` *sec_type hwyname.secnum ?length?*	Creates a section of type *sec_type* in highway *hwyname* with section ID = *secnum* of length *length. sec_type* must be one of *standard* or *transition.*
`lane` *hwyname.secnum.lanenum*	Creates a lane in section *hwyname.secnum* with lane ID = *lanenum*
	Commands that provide information
`sinfo` *var_type ?var_name?*	Provides information on variable *var_name* which is of type *var_type. var_type* must be one of *domain, highway, section, lane, traj, patharray* or *carunit. domain*, however, does not require a *var_name.*
	Core car unit commands
`vehicle` *name*	Creates a *PMVehicle* with name *name.*
`car_follower` *type name ?other_info?*	Creates a *CarFollower* with name *name* of type *type. type* must be one of *follow_traj, cs_car_follower, aicc, cp_c, cp_e, cp_g, cp_h, cp_j, cp_l* or *dp. other_info* is required only for *follow_traj* and is a trajectory named *traj_name.*
`long_control` *type name cf_name*	Creates a *LongControl* with name *name* of type *type* that uses the car follower *cf_name. type* must be one of *follow_trajlc, carsimlc* or *platoonlc.*
`lat_control` *type name*	Creates a *LatControl* with name *name* of type *type. type* must be one of *simple* or *quick.*
`path_control` *type name ?other_info?*	Creates a *PathControl* with name *name* of type *type. type* must be one of *human_pc, semi_compliant_pc, compliant_pc* or *automated_pc.* The *other_info* is respectively *NULL, path_array_name, path_array_name* and *segcontrol_name.*
	Other car unit commands
`seg_control` *type name*	Creates a *SegmentControl* with name *name* of type *type. type* must be one of *const_lane, dest_mono, orig_mono, dest_split* or *orig_split.*

Table 10.2: Tcl commands

Command	Description
	Commands for creating various generators
`veh_generator name`	Creates a *VehicleGen* with name *name*.
`cf_generator` *type name ?other_info?*	Creates a *CarFollowerGen* with name *name* of type *type*. *type* must be one of *follow_traj, cs_car_follower, aicc, cp_c, cp_e, cp_g, cp_h, cp_j, cp_l* or *dp*. *other_info* is required only for *follow_traj* and is a trajectory named *traj_name*.
`longc_generator` *type name cf_gen_name*	Creates a *LongCGen* with name *name* of type *type* that uses the car follower generator *cf_gen_name*. *type* must be one of *follow_trajlc, carsimlc* or *platoonlc*.
`lanec_generator` *name*	Creates a *LaneChGen* with name *name*.
`latc_generator` *type name lanec_gen_name*	Creates a *LatGen* with name *name* of type *type* that uses the lane changer generator *lanec_gen_name*. *type* must be one of *simple* or *quick*.
`pathc_generator` *type name ?other_info?*	Creates a *PathCGen* with name *name* of type *type*. *type* must be one of *human_pc, semi_compliant_pc, compliant_pc* or *automated_pc*. The *other_info* is respectively *NULL, path_array_name, path_array_name* and *segcontrol_name*.
`cargen` *type name lane_name ?other_info?*	Creates a *CarUnitGen* for the lane *lane_name* of type *standard* or *special*. The latter type requires *other_info* to be *veh_gen_name longc_gen_name latc_gen_name pathc_gen_name*.
	Commands for creating miscellaneous objects
`traj` *name*	Creates a *Trajectory* of name *name*.
`patharray` *name*	Creates a *PathArray* of name *name*.
`flow_wire` *pos lane_name*	Creates a *FlowWire* object and lays it across the lane named *lane_name* at position *pos*.

Table 10.3: Tcl commands

Command	Description
	Commands that set or modify parameters
`integrator_step` *time_step*	This sets the *Integrator's* time step to *time_step.*
`connect` *lane1 lane2*	This adds the lane named *lane2* after the lane named *lane1.*
`add_to_traj` *name value*	This adds the value *value* to the trajectory named *name.*
`add_to_pa` *name lane_name*	This adds the *Lane* named *lane_name* to the *PathArray* named *name.*
`trace` *on/off*	This turns the tracing of events in *EventCal* on or off.
`clear`	This clears all the variables that you have defined so far.
`max_to_generate` *cargen_name num_cars*	This sets the maximum number of car units that the car generator named *cargen_name* can generate, to *num_cars.*
`cargen_exits` *name section_name prob*	This assigns section *section_name* as the destination section with probability *prob* for the car generator named *name.*
`states_car` *carunit_name pos_on_lane ?speed acc?*	This sets the position on the lane to be *pos_on_lane*, the speed to be *speed* and the acceleration to be *acc* for the car unit named *carunit_name.*
`print_params` *long_pos long_vel long_acc lat_pos*	This controls the periodic printing of the longitudinal position, the speed and acceleration and the lateral position for all car units in the system. A value of 0 for any of the parameters *long_pos long_vel long_acc lat_pos* indicates that that parameter should not be periodically printed out.
`update_times` *plt_state_time path_control_time*	This sets *plt_state_time* and *path_control_time* to be the time between two successive schedulings of the *PltLCStateEv* and *PathCEvent* respectively, for a particular car unit.

Table 10.4: Tcl commands

Command	Description
	Commands to do with the simulation/schedule events
`start_physical` *event_time*	This inserts a *PhysicalControl* event at time *event_time* in the event calendar. Note that by default a *PhysicalControl* event is **always** inserted in the event calendar at time 0.0, so this command should almost never be used.
`integrate`	This executes one time step of the simulation. This command should also be used with care.
`stopat stop_time`	This inserts an *EndSim* event into the event calendar at time *stop_time*.
`run`	This command starts the simulation.
`generate_car` *cg_name*	This causes the car generator named *cg_name* to generate one car unit.
`cgevent` *cargenname type event_time other_info*	This command inserts a *CarGEvent* of type *type* at time *event_time* in the event calendar. *type* must be one of *periodic, oneshot, uniform, expon* or *normal*. The *other_info* is respectively, *period, NULL, upper and lower bounds, mean* and *mean and standard deviation.*
`latevent` *car_unit_name direction time*	This command schedules a *LatCEvent* with direction *direction* for the car unit named *car_unit_name* for time *time*.
`stat_event` *update_time start_time*	This command schedules a *StatCEvent* at time *start_time*. The *StatCEvent* is periodically every *update_time* time units.

10.2.1.1 SimHashTable

This class is designed around Ousterhout's *Tcl_HashTable* class. A hash table is a collection of entries, in which each entry consists of a *key* and a *value*. SimHashTables are used to efficiently bind Tcl variable names to C++ objects, with the Tcl variable being the *key* and the C++ object being the *value*. They allow manipulation of the C++ object via its Tcl name.

10.2.1.2 SimData

This class is basically a structure that contains handles to the various objects that can be used during a simulation. It contains pointers to objects that there will only be one of during a simulation, such as an *EventCal* or a *Domain*, and contains *SimHashTables* of multiple objects, such as *Vehicles* or *CarGens*.

10.2.1.3 SimCreate

This class contains the majority of the functions that the Tcl commands are bound to. Its methods provide the functionality for creating various objects such as highways, sections, lanes, car generation events, car units, car unit generators and miscellaneous other objects.

10.2.1.4 Util

This class provides a number of utility functions. Most of the methods of this class are not bound to any Tcl commands, but are called by methods of the other classes described in this section. For example, the methods of the *Util* class provide the functionality to check whether or not a string is a valid integer or a double, whether or not a lane or a section number is valid, etc.

10.2.1.5 SimSet

The Tcl commands that are bound to the methods of this class assign values to parameters. For example, the *integrator_step* Tcl command will cause the time step of the *Integrator* C++ object to be set to the specified value.

10.2.1.6 SimDo

This methods of this class correspond to C++ functions in the AHS class library that "do" things. For example, the Tcl commands *generate_car* and *run* are bound to the methods of this class.

10.2.1.7 SimInfo

The Tcl command *sinfo* is bound to a method of this class. This command requires additional arguments such as *highway* or *lane* or *vehicle* (see Table 10.1 for the options available) and provides information about those arguments. For example, *sinfo highway I76* will return information on the number of sections in highway I76. *sinfo section I76.2* will provide information on the number of lanes in the second section of highway I76.

10.3 List of Commands in the Tcl Front End

Tables 10.1 through 10.4 summarize the application specific Tcl commands. The following points must be kept in mind while using the Tcl commands. First, a NULL means that no argument is required. Second, the notation for naming highways, sections, lanes and car units may be seen from the following example. Consider a `highway` named I76. Then a `section` with name I76.1 is the first section in highway I76, a `lane` named I76.1.1 refers to the first lane in section I76.1 and a `carunit` named I76.1.1.1 refers to the first car unit in lane I76.1.1. Examples that illustrate the use of the front end are presented in the next chapter.

10.4 Post-processing

The `print_params` Tcl command controls the printing of the states of primary interest, namely, longitudinal position, velocity and acceleration, and lateral position. In addition, the *stat_event* Tcl command causes statistics collection to be made and data on speed, flow and concentration to be saved to files that can be read in using packages such as Matlab in order to process or print them.

CHAPTER 11

SIMULATION TEST RUNS

This chapter presents some test runs to illustrate the use of the AHS class library and the Tcl front end. Test run 1 illustrates longitudinal control and the car following behavior of the AICC car following law. It was performed using programs written both in C++ and in the Tcl front end. The programs for test runs 2 and 3 were written using the front end only. Test run 2 illustrates some of the lateral control logic described in Chapter 9. Test run 3 shows how statistics on speed, concentration and flow may be collected. These test runs are presented mainly to illustrate the use of the simulation programs and the nature of some of the vehicle control logic. They do not perform a critical evaluation of the control logic, nor do they present a comparison of different control algorithms. However, they do provide a face validation of the control algorithms used. The front end and AHS classes provide the capability for a critical comparison of different control algorithms to be done in the future.

11.1 Test Run 1

Consider a highway with a single lane that contains 20 car units, initially 30 m apart, all at rest (i.e., their velocity is 0). All of the car units have been given *HumanVision* vision and *PMVehicle* vehicles. The vehicles have identical physical parameters such as length (5 m), maximum and minimum jerks, accelerations, etc. Their lateral and path controllers are irrelevant because there is only lane that they can travel on. The leading car unit is given a prescribed trajectory to follow and has a *FollowTrajLC* longitudinal controller. Its velocity

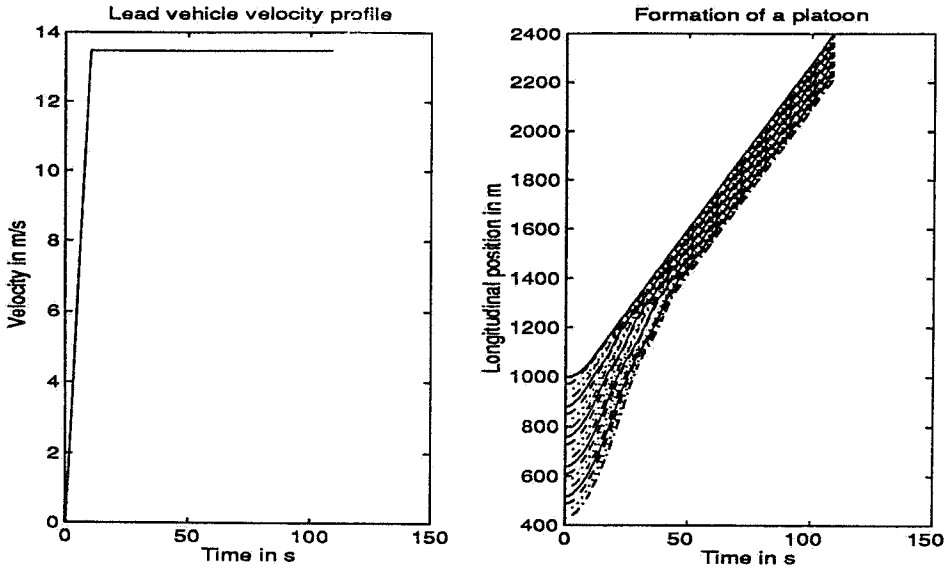


Figure 11.1: The profiles for test run 1

profile is in Figure 11.1. Note that its maximum velocity is 13.5 m/s. The other car units all have *AICC* car followers and *PlatoonLC* longitudinal controllers. They have been given a reference velocity of 25 m/s.

When the simulation begins, the lead car unit starts following its prescribed trajectory. The following car units accelerate to reach their reference velocities of 25 m/s, but are not able to do so because of the slow leading car unit. The following car units merge to form platoons two at a time and eventually merge into a single platoon consisting of the 20 car units, as can be seen from the plot of Figure 11.1. The steady-state interplatoon distance is 9.7 m and the steady state speed is 13.5 m/s. The behavior seen here is as expected according the longitudinal control algorithm and the car following algorithm (*AICC*) used, which predicted a steady-state interplatoon distance of (.2*13.5 + 2) m. This test run thus provides a validation of the code used for these algorithms. The complete C++ code to simulate this system is presented in Appendix B.1. The Tcl code required to simulate this system is given in Figure 11.2. We can tell by inspection that the Tcl code is shorter and simpler than the C++ code. In addition, the Tcl code is interpreted, not compiled.

```
# Create the highway structure
highway I76
section I76.1 5000; section I76.2 5000; section I76.3 5000
lane I76.1.1 25; lane I76.2.1 25
connect I76.1.1 I76.2.1
integrator_step .01
# Create the trajectory for the leading vehicle
traj t1
for { set i 1} { $i <= 45} {incr i} {
    add_to_traj t1 [expr 3*$i*.01]
}
for { set i 46} { $i <= 1000} {incr i} {
    add_to_traj t1 1.35
}
for { set i 1001} { $i <= 1045} {incr i} {
    add_to_traj t1 [expr 1.35 - 3*($i*.01-10)]
}
for { set i 1046} { $i <= 11000} {incr i} {
    add_to_traj t1 0
}
# Leading car unit,  component generators
veh_generator vg
cf_generator follow_traj cfg1 t1
longc_generator follow_trajlc log1 cfg1 t1
lanec_generator lancg
latc_generator simple lag lancg
pathc_generator human_pc pg
# Generate the lead vehicle and set its state
cargen special cgA I76.1.1 vg log1 lag pg
cargen_exits cgA I76.3 1
generate_car cgA; states_car I76.1.1.1 1000 0
# Car unit generators for the following cars
cf_generator aicc cfg2
longc_generator platoonlc log2 cfg2
cargen special cgB I76.1.1 vg log2 lag pg
cargen_exits cgB I76.3 1
# Generate the other cars and set their states
for {set i 2} {$i <= 20} {incr i} {
    set pos [expr 1000 - 30*($i-1)]
    generate_car cgB; states_car I76.1.1.$i $pos 0
}
stopat 109; print_params 1 1 1 1; run
```

Figure 11.2: The Tcl code for test run 1

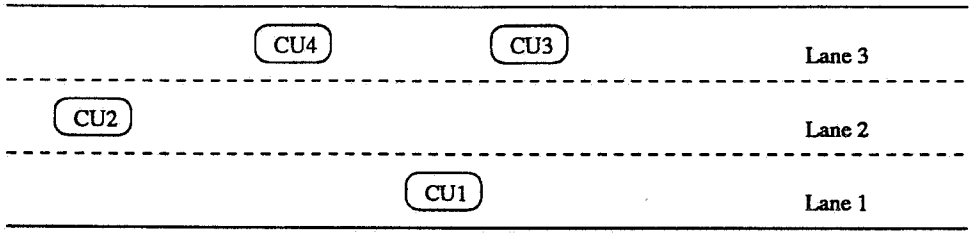


Figure 11.3: Initial placement of cars for test run 2

11.2 Test Run 2

This test run illustrates lane changing and some of the logic used in lateral control. Consider a highway with three lanes and four car units that are placed on the three lanes as shown in Figure 11.3. Each car unit has a *PMVehicle* vehicle, an *AICC* car follower, a *HumanVision* vision, a *PlatoonLC* longitudinal controller, a *SimpleLaneC* lane changer, a *SimpleLatC* lateral controller and a *HumanPC* path controller. However, the longitudinal and path controllers have been effectively deactivated by scheduling their control events only once every 1000 seconds (the simulation is run for only 31 seconds), in order to isolate the effects of the lateral control logic. Car unit 4 is told to change right at time $t = 0$, car unit 3 is told to change right at time $t = 1$, car unit 1 is told to change left at time $t = 1.5$ and car unit 2 is told to change right at time $t = 2$. Each lane change takes three seconds to complete.

When car unit 4 begins its lane change at time $t = 0$, it places a hold on car units 1 and 2 (i.e., they are not allowed to change because they are "close" to car car unit 4). At time $t = 1$, car unit 3 begins its lane change, and it places additional holds on car units 1 and 2. At times $t = 1.5$ and $t = 2$, when car units 1 and 2, respectively, were supposed to change lanes, they cannot do so because of the holds placed on them. At time $t = 3$, car unit 4 completes its lane change and removes the hold placed by it on car units 1 and 2. Shortly after, because car unit 4 is now in between car unit 2 (in lane 2) and car unit 3 (in lane 3), car unit 3 removes its hold on car unit 2 and places one instead on car unit 4. Because car unit 2 no longer has any holds on it, it begins its lane change. Before doing so, it places an additional hold on car unit 1. At time $t = 4$, car unit 3 completes its lane change and

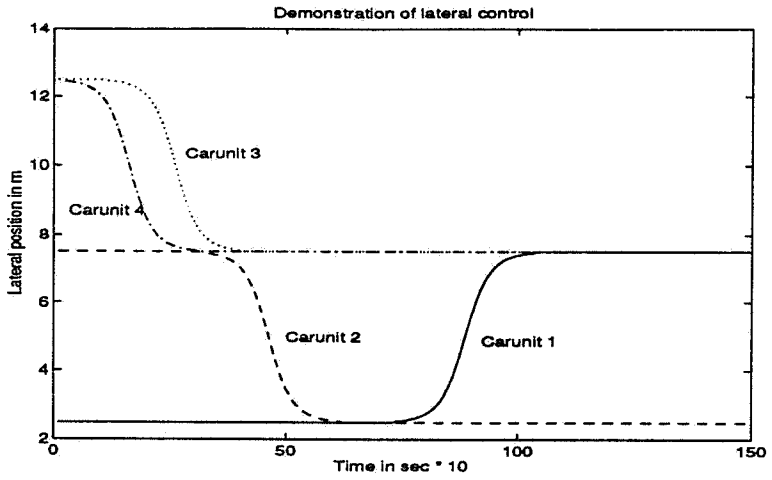


Figure 11.4: The lane change maneuvers of test run 2

removes its hold on car unit 1. Then, at time $t = 6$, car unit 2 completes its lane change and reduces its hold on car unit 1. Because car unit 1 now no longer has any holds on it, it commences its lateral control to the left. However, the presence of car unit 3 in lane 2 prevents it from doing so immediately. It decelerates until it is a safe distance behind car unit 3 and then changes lanes. It completes its lane change at time $t = 10.25$.

Figure 11.4 shows the plot of the lateral positions of the four car units versus time. The lateral control behavior of the car units in this test run is consistent with the lateral control logic in Chapter 9 and thus provides a validation of the coding of the control algorithms. The Tcl code required to execute the simulation is given in Appendix B.2.

11.3 Test Run 3

This next test run illustrates the use of the statistics collector class. The scenario considered is that of a four-section highway segment (see Figure 11.5). Each section has three lanes, one of which serves as a transition lane for entry to and exit from the highway. A car generator has been placed in each of the transition lanes named I75.1.0, I75.2.0, I75.3.0 and I75.4.0. In addition, the lanes I75.1.1 and I75.1.2 also contain car generators. The flow due to the car units generated by these two generators represents the flow entering the highway

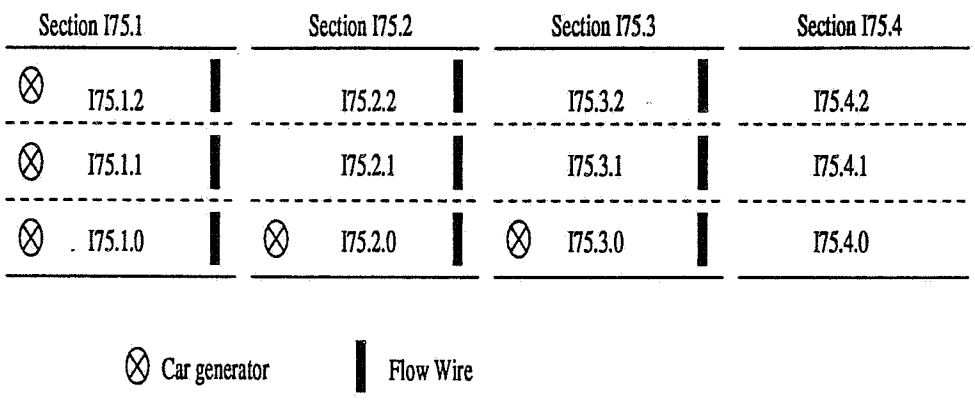


Figure 11.5: The highway segment in test run 3

segment from the previous segment. The time between car generations is set to be normally distributed with a mean of 5 seconds and a standard deviation of .5 seconds. A truncated normal distribution is used, so that the time between generations is never less than four seconds and never more than six seconds. Each car unit has a *PMVehicle* vehicle, a *HumanVision* vision, an *AICC* car follower, a *PlatoonLC* longitudinal controller, a *SimpleLaneC* lane changer, a *SimpleLatC* lateral control and a *HumanPC* path control. Each generated car unit is assigned an exit that is one of the sections I75.1, I75.2, I75.3 or I75.4, and which is randomly generated. We are interested in the speed, concentration and flow characteristics of the traffic in the lanes. *FlowWire* objects have been placed across the lanes as shown in Figure 11.5 in order to measure the flow through them. The speed and concentration statistics will be collected by a *StatCollector* (the front end automatically provides a *StatCollector*). To enable statistics collection, a *StatEvent* is scheduled at time 0.0. This event is periodically rescheduled every five seconds. The simulation is run for 60 seconds and data is collected every five seconds. Sample results are shown in Figures 11.6 through 11.8 for lane I75.2.1, which is the first lane in the second section of the highway segment. Figure 11.6 shows the concentration versus time, Figure 11.7 shows the spot speed versus time and Figure 11.8 shows the flow versus time in that lane.

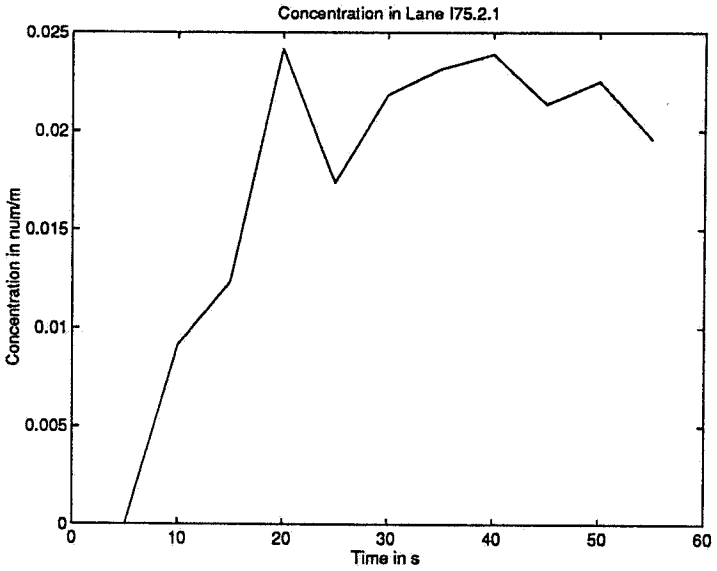


Figure 11.6: Concentration data for test run 3

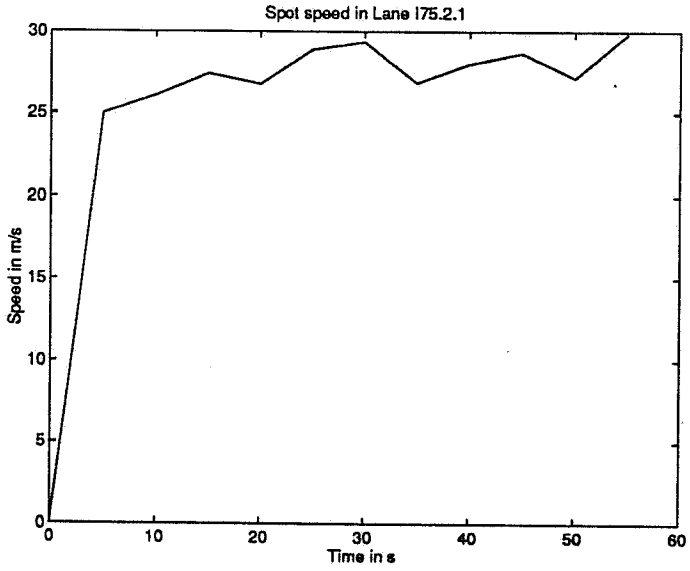


Figure 11.7: Speed data for test run 3

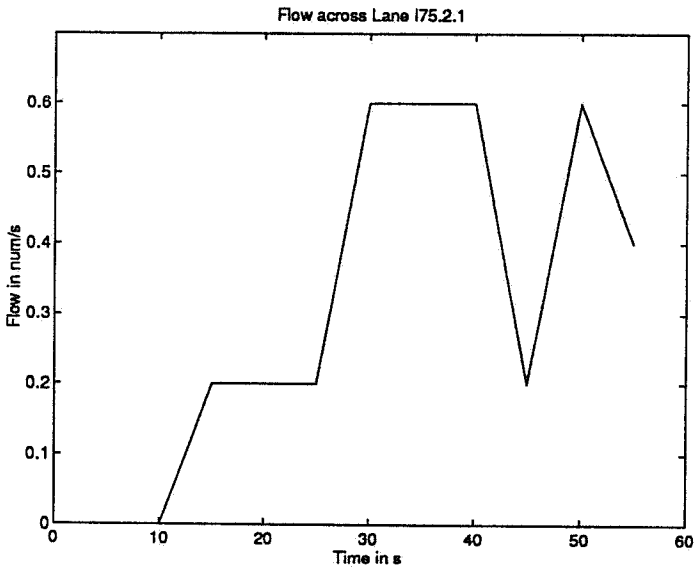


Figure 11.8: Flow data for test run 3

CHAPTER 12

DYNAMIC CAPACITY AND ROUTE GUIDANCE

The AHS simulator opens up possibilities for the study of a host of open problems. The simulator will enable the study of flow on the AHS with diverse (1) origin/destination loadings of the system, (2) lane scheduling algorithms, (3) communication protocols and (4) AVCS characteristics. The simulator can be used for the simulation of the actual flow of traffic on a multi-lane AHS as well as flow assuming mixed (automated and nonautomated) traffic in the transition phase of development. It can be used to determine how the macroscopic flow characteristics are affected by the characteristics of the AVCS in the vehicles, by the communication protocols employed in seeking permission to maneuver, by rules of operation of vehicles on the AHS and by the lane assignment strategies used to schedule paths of vehicles.

In addition to simulating the traffic conditions on the AHS with a selected lane assignment strategy, the simulation package can also be used to optimize lane assignment. Simulation would provide macroscopic parameters for a candidate lane assignment and thus allow the determination of an improved assignment strategy based on the sensitivities of the macroscopic flow parameters to changes in the lane assignment.

In particular, the simulator can be used to assess the **dynamic capacity** of the AHS, namely the speed/concentration/flow relationships that will be established on the AHS with many maneuvering vehicles under full automated control and for situations characterized by various percentages of automated vehicles on AHS with mixed traffic. Additionally, it can

be used to perform a comparative analysis of the effect the large number of maneuvers have on the dynamic capacity of the AHS for (1) the case in which vehicles form platoons with small intraplatoon and large interplatoon distances and (2) the case in which vehicles form continuous streams with large intervehicle distances. The use of large intervehicle distances is supported by major vehicle manufacturers and seems to be the only alternative to operating the AHS during the gradual transition to the fully automated AHS. If the AHS is to be fully automated at the outset, then the analysis of its dynamic capacity is that much more critical and imperative.

12.1 Estimating the Dynamic Capacity

In the formulation of the lane assignment problem, the lane capacities z_1, z_2 and z_3 in sections 4 and 5 were assumed to be fixed. The actual lane capacities z_1, z_2 and z_3 are, however, dependent on the number of maneuvers that occur and, hence, dependent on the lane assignment. The iterative technique described below makes use of the AHS simulator to compute realistic capacities and the optimal assignments for these capacities.

Before the iterative technique can be applied, an estimate of the OD matrix is required. The OD matrix in Equation (2.1) used in lane assignment defines the flow from each entrance to each exit in a freeway segment. These flows arise from three sources:

1. from other highways;
2. from metered on-ramps fed by arterial streets; and
3. from nonmetered ramps fed by arterial streets.

We can, therefore, construct the OD matrices used to calculate the lane assignments by calculating the flow from each of these three sources. The flow from and to other highways is defined by the overall route assignments for the traffic network and can be easily calculated once the route assignments are known. The ramp metering rates provide an estimate of the flow of traffic onto a freeway, but do not provide information on where this flow is headed. The elements in the OD matrix that correspond to flows from metered and nonmetered ramps

must, therefore, be obtained from historical data. The steps in the iterative computation of the dynamic capacity are listed below.

Step 1 For each highway segment, calculate the OD matrix N that results from the three sources listed above using assumed route assignments and ramp metering rates and from the estimated input to the nonmetered ramps. Use estimates for the initial capacities and travel times. Set iteration count = 1.

Step 2 For these capacities and travel times, for each highway segment use the OD matrices found in Step 1 to calculate the resultant optimal lane assignments ρ^*.

Step 3 If required, for each optimal lane assignment ρ^* obtained in Step 2, calculate the "closest" partitioned lane assignment $\bar{\rho}$.

Step 4 Run the simulator for the level of compliance assumed, and for either the constant lane assignment ρ^* or the partitioned assignments $\bar{\rho}$ obtained in Step 3. Obtain estimates of link travel times t_a and capacities C_a.

Step 5 Given the lane assignments ρ and the lane capacities C_a, calculate the excess capacity in each section and lane. If none of the excess capacities is negative, go on to Step 6.

(a) If any of the excess capacities is negative and the iteration count is $< max\_iter$, increment the iteration count and go back to Step 2. This time use the link travel times t_a and capacities C_a found in Step 4.

(b) Else, terminate the algorithm. The dynamic capacity is equal to the highway capacity obtained in the iteration *max_iter-1.*

Step 6 Increase the flow onto the highway by uniformly raising the nonzero elements of the OD matrix. Increase the estimated highway capacity proportionately. Go to Step 2.

12.2 Route Guidance

Methods used to assign routes to vehicles in a traffic system are referred to as route guidance methods.

12.2.1 User optimality versus system optimality on AHS

Route guidance methods are often selected to optimize either the cost to the individual users or the cost to the system (the sum of individual users' costs) as whole. These methods are referred to as user optimal and system optimal strategies, respectively. With user optimal strategies, the goal is to assign the routes in a manner in which, under current traffic conditions, no single user may improve upon his/her cost by taking any route other than the one to which he/she is assigned. This concept is often associated with Wardrop [58], who proposed its use with transportation networks as a means to ensure that users would comply with the selected route assignments because they would have no motivation to deviate from their assigned routes. With system optimal strategies, the objective is to minimize the average cost of the assigned routes. Because the routes are selected to minimize the cost to the system as a whole and not the cost to the individual users, system optimal strategies are not usually used with systems in which the users have decision-making abilities, such as has been the case in transportation applications. The reason for this is that, in general, an individual user may be able to reduce his/her cost by taking a route other than the one to which he/she is assigned and, therefore, the motivation exists for users to deviate from their assigned routes.

It is well known, however, that if all users complied with system optimal decisions, a traffic flow would result that is overall more favorable (as measured in terms of a performance index such as travel time or throughput). Consequently, it is justifiable that in considering traffic flows on automated or semi-automated highways, we seek system optimal solutions as opposed to user optimal solutions. Expanding this idea further, it seems justified to consider system optimal strategies as viable alternatives to user optimal strategies on any part of the highway system where a significant number of users may be induced, or forced, to comply with system optimal strategies. Significant improvement in reducing congestion, and in

reducing adverse effects on the environment, would then be accrued by providing information without necessarily providing full automation. Users might be induced to comply with the suggested strategies by the demonstration of the value of compliance in terms of performance indices such as travel times. On the other hand, compliance could be forced by imposing mechanisms such as fines or the revocation of the right to use the system when noncompliance is ascertained. The determination of compliance may be made by polling equipment or on-board devices that monitor the movement of the vehicle in the traffic system. Such equipment is well within the capabilities of the system architecture being assumed in the IVHS plan for the near future.

12.2.2 Background material on route assignment

Detailed discussions of the algorithms for user and system optimal route assignment are provided in [58], [59] and [60]. A brief discussion on route guidance based on the material in [61] is provide below. The algorithm described in this section applies to applications that may be modeled as a graph having unidirectional links. It is assumed that the cost of a link is differentiable, is an increasing function of its flow and does not depend on the flows of other links. It is also assumed that for the time window for which the route assignment is being performed, a set of origin/destination pairs is given along with their corresponding steady-state flow demands. Table 12.1 contains the notation used in the algorithm.

The notation is similar to that used in [58]. Throughout the following discussion, the route flows are often referred to as route assignments because they specify the flow that is assigned to each route. Any proposed route assignment must satisfy the constraints

$$\begin{aligned} \sum_{k \in K_{ij}} h_{ij}^k &= g_{ij} \quad \forall i,j \in I \\ h_{ij}^k &\geq 0 \quad \forall k \in K_{ij}\ i,j \in I \end{aligned} \tag{12.1}$$

These state that flow must be conserved on the routes used for each O/D pair, and that the flow on each route must be non-negative. The definitions

$$v_a = \sum_{i,j \in I} \sum_{k \in K_{ij}} \delta_{ak} h_{ij}^k \quad \forall a \in A \tag{12.2}$$

Table 12.1: Notation used in route guidance

N	Set of nodes
I	Set of OD pairs, $ij \in I$, origin $i \in N$, destination $j \in N$
g_{ij}	Flow demand for OD pair $ij \in I$
u_{ij}	Cost of minimum cost route for OD pair $ij \in I$
K_{ij}	Set of routes for OD pair $ij \in I$
h_{ij}^k	Flow on route $k \in K_{ij}$, $ij \in I$
s_{ij}^k	Cost of route $k \in K_{ij}$, $ij \in I$
A	Set of links
v_a	Flow on link $a \in A$
$s_a(v_a)$	Cost of link $a \in A$, function of v_a
δ_{ak}	1 if link a is contained in route k, 0 otherwise

$$s_{ij}^k = \sum_{a \in A} \delta_{ak} s_a(v_a) \quad \forall k \in K_{ij}\ i,j \in I \tag{12.3}$$

which define the flow on a link as the sum of the flows on the routes containing the link and the cost of a route as the sum of the costs of the links contained in the route, must also be satisfied.

As stated previously, for a set of route assignments to be user optimal, the routes must be assigned in a manner such that a user is not able to improve upon his/her cost by taking a route other than the one to which he/she is assigned. This implies that the flow demands for each OD pair must be assigned to routes of minimum cost, and if multiple routes are used for a given OD pair, each of these routes must be of minimum, and therefore equal, cost. These conditions for a user optimal route assignment may be expressed in a number of ways, one of which is

$$\begin{aligned} s_{ij}^{k*} &= u_{ij}^* \quad if \quad h_{ij}^{k*} > 0 \\ s_{ij}^{k*} &\geq u_{ij}^* \quad if \quad h_{ij}^{k*} = 0 \end{aligned} \tag{12.4}$$

where the superscript '*' denotes that the solution is optimal. Equation (12.4) states that if a portion of the flow demand for an OD pair is assigned to a route, then the route has minimum cost, and no flow is assigned to routes which are not of minimum cost.

While Equation (12.4) expresses the desired conditions for a user optimal solution, it does not indicate how this solution may be obtained. In [58], it is shown that a solution

satisfying Equation (12.4) along with the constraints of Equation (12.1) may be obtained through the use of variational inequalities. It is then noted that this variational inequality problem may be expressed as an equivalent constrained minimization problem having the objective

$$Z = \sum_{a \in A} \int_0^{v_a} s_a(x)dx \tag{12.5}$$

and the constraints given by Equation (12.1). It is noted that the objective given in Equation (12.4) is convex due to the assumption that link costs are increasing functions of the link flows.

The system and the user optimal problems differ only in the objective used in the constrained optimization problem; the constraints and definitions given in Equation (12.1) hold for both problems. As mentioned earlier, the goal of a system optimal routing strategy is to minimize the average cost of delay for the system. Thus, the objective to be minimized may be stated directly as

$$Z = \sum_{a \in A} v_a s_a(v_a) \tag{12.6}$$

which is once again convex due to the assumption that link costs are increasing functions of the link flows.

Because the objectives given in Equations (12.4) and (12.6) are convex and the constraints given by Equation (12.1) are linear, an iterative linear approximation method, such as the Frank-Wolfe algorithm, may be used to solve this constrained minimization problem. First, an initial set of feasible route assignments is obtained. Then, the iterative portion of the algorithm begins with the calculation of the link costs using the current set of link flows. Next, the solution to the linearized constrained minimization problem is obtained by using a shortest path algorithm to find the minimum cost route for each OD pair and assigning the entire flow demand to this minimum cost route. The route assignments for the next iteration are then calculated as an optimal affine combination of the solution to the linearized constrained minimization problem and the current route assignments. The algorithm is considered to have converged once the shifts in the route assignments from one iteration to the next have become sufficiently small.

12.2.3 Route guidance on urban corridors

The route guidance problem considers the assignment of flows originating at different nodes of the network on the various links of the network in an optimal fashion. Systems for which significant benefits in the reduction of congestion and pollution are potentially possible are the urban highway systems being considered under the Corridors program. In particular, the Gary, Chicago, Milwaukee (GCM) Corridor is a traffic system in which significant through flow of traffic exists in the greater Gary, Chicago and Milwaukee areas, in addition to significant local metropolitan traffic. We believe that a significant improvement in the traffic conditions and the alleviation of congestion on the main interstate highway system in this corridor (see Figure 12.1) may be achieved by combining lane assignment with system optimal route guidance. These may be implemented if a significant portion of the vehicles, particularly commercial vehicles, are equipped to send origin-destination information and receive route guidance and lane assignment information from the traffic control system.

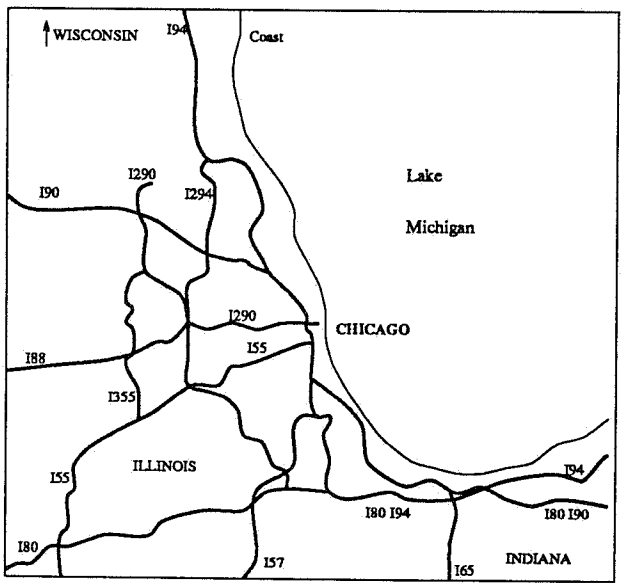


Figure 12.1: Sketch of the important freeways in the Corridor project

The AHS simulator may be combined with traditional route guidance algorithms to obtain optimal route assignments on automated or semi-automated highways such as urban

corridors. In a transportation network, the cost of a link is usually selected to represent the delay that is incurred when traversing the link. This delay is generally modeled as a function of the flow, capacity and other properties of the link. One of the functions used to model the cost/delay of a link is known as the Bureau of Public Roads (BPR) function, which is of the form [62].

$$s_a(v_a) = t_{0,a}\left[1 + \alpha\left(\frac{v_a}{C_a}\right)^{\beta}\right] \tag{12.7}$$

The parameter $t_{0,a}$ denotes the nominal delay of the link, which is the delay incurred when the flow on the link is zero. C_a refers to the link's capacity. The AHS simulator obviates the need to model travel delays on the automated or semi-automated highway by means of formulas such as Equation(12.7) because it will directly provide estimates of the travel times, in addition to providing estimates of link properties such as flow and capacity.

The formulations of route guidance and lane assignment have been done with the assumption that the associated parameters such as the origin destination matrix are constant. This means that assigned routes and lanes are optimal in a time window in which the problem parameters do not vary significantly with time. The understanding is that when the parameters affecting the route guidance and lane assignment optimization problems change, the assigned routes and lanes must be recalculated.

There has been no prior work in the area of combining user optimal or system optimal route guidance with lane assignment. It is, therefore, of interest to formulate the combined route guidance with lane assignment problem, and to develop an environment for simulation, analysis and decision-making and to assess the potential benefits. The specific purposes would be to enable microsimulation of traffic flow on controlled access highways, to compare system optimal with user optimal strategies, and to assess the value of information provided by the system. Additionally we would be able to quantify the benefits of this information as a function of the percentage of penetration of the communication equipment, and of the percentage of compliance by the equipped vehicles with posted routes and lane assignments.

The route guidance algorithm stated in Section 12.2.2 requires information on link travel times and link capacities in order to calculate link costs. The simulator can provide estimates of these quantities. It can also include the effect of partial automation on these quantities. To

incorporate the advantages of system-wide information, lane assignment may be performed in order to minimize travel times on each freeway segment. The optimal lane assignment may then be converted to a partitioned assignment, if necessary. The simulator is used with the obtained assignments (under various levels of compliance) to obtain the actual link travel times and capacities. These quantities are used by the route assignment algorithm to calculate route assignments for the entire network. The steps to be followed in this iterative route guidance and lane assignment can be organized as follows. (Refer to Section 12.2.2 for the notation used below for route guidance.)

Step 1 Use an initial estimate of the travel times t_a and capacities C_a for each link of the network to set up route guidance.

Step 2 Use the Frank-Wolfe algorithm to calculate the optimal route assignments.

Step 3 For each highway segment, calculate the OD matrix N that results from the route assignments in Step 2, from the assumed ramp metering rates and from the estimated input to the nonmetered ramps.

Step 4 For each highway segment, use the OD matrices calculated in Step 3 and calculate the resultant optimal lane assignments ρ^* and the "closest" partitioned lane assignment $\bar{\rho}$.

Step 5 Run the simulator for the level of compliance assumed and for the assignments obtained in Step 4. Obtain estimates of link travel times t_a and capacities C_a.

Step 6 Given the lane assignments ρ and the lane capacities C_a, calculate the excess capacity in each section and lane. If any of the excess capacities is negative, go back to Step 1. This time, use the link travel times t_a and capacities C_a found in Step 5. If none of the excess capacities is negative, terminate the algorithm.

In this algorithm, ramp metering rates will be fixed using one or more decentralized ramp metering strategies now in use (e.g., pretimed control mode or local actuated control mode) or selected system control strategies. Subsequently, however, ramp metering may also be incorporated into the combined route guidance and lane assignment problem. Ramp

metering strategies may be used to modify the OD matrix to reduce bottlenecks in specific sections of the highway and, hence, increase the dynamic capacity of the highway.

12.2.4 Implementation of lane assignment on the urban corridor

The implementation of route and lane assignment would require that the corridor infrastructure contain at least the provisions for a variable message display system on the highway. We also assume that it has the capability to communicate with vehicles on the highway that are equipped with suitable communication equipment. Additionally, we assume that automated control of vehicles is rudimentary and at the level of cruise control or intelligent cruise control (vehicle following). Moreover, we assume that intervehicle distances are similar to those in current traffic conditions and that all of the vehicles will possess suitable communication equipment and among these, not all will obey posted directions.

An urban traffic system such as the GCM Corridor is composed of a network of freeways and arterial feeds. While route guidance considers the network as a whole, lane assignment may be performed on critical segments of this network to reduce the total travel time (or optimize some other performance index) on those segments. The lane assignment described in Chapters 4 and 5 enables us to calculate the optimum $\rho(i,j)^m$ that represents the flow from each entrance i to each j in lane m in the segment of freeway considered. The implementation of this on a nonautomated project such as the Corridors project, would require the use of variable message signs to transmit the assigned lanes to the drivers. Here, partitioned strategies offer additional advantages. Well-defined algorithms exist to determine optimal and near optimal partitioned strategies as defined in Chapters 5 and 6. Partitioned strategies also reduce the amount of information that must be displayed to the participating vehicles. Thus, instead of displaying detailed lane assignment information at every entrance ramp i on lanes assigned to vehicles traveling from i to j, it is sufficient to display information defining the partitions for that entrance. For example, a destination monotone strategy for a three-lane highway is defined by two exits π and δ such that the lane L assigned to vehicles entering at i and exiting at j is as follows:

$$L = L_1 \; for \; i < j \leq \pi, \quad L = L_2 \; for \; \pi < j \leq \delta, \quad L = L_3 \; for \; \delta < j \leq K_2$$

To implement this in the Corridors project, $n-1$ variable message signs could, for example, be placed at each entrance to an n lane freeway. For a three-lane freeway, these signs would display the values of π and δ recommended for use by the vehicles entering the freeway. The number of signs required would be significantly less than that required with constant lane assignments. Apart from the obvious cost-saving benefit, the problem of engulfing drivers with information is also avoided.

CHAPTER 13

CONCLUSION

The automation of highways as part of the IVHS program is seen as a way to alleviate congestion on urban highways. This thesis has discussed the concept of lane assignment in the context of AHS and has formulated the static lane assignment problem as an optimization problem. A classification of lane assignment strategies has been established, beginning from totally unconstrained strategies, to general strategies, to constant lane strategies, to destination and origin monotone strategies, and ending with highly ordered monotone strategies. The lane assignment problem has been analyzed under two distinct performance measures, the first being the minimization of total travel time of the vehicles using the AHS, and the second being the balanced use of lanes. Algorithms have been developed that find the optimal or suboptimal strategy. The structure of the optimal assignments has been studied. The performance of these algorithms and the structure of the optimal and suboptimal strategies have been looked at in a case study of a hypothetical AHS.

Although the capacity was assumed to be fixed in the problem formulations, the dynamic capacity of the AHS will, in fact, depend on many factors, including car following control strategies, lateral control strategies, communication protocols, lane scheduling algorithms and the relative percentages of manual and automated vehicles in the traffic. This complexity of traffic flow on multi-lane AHS required the development of a simulation facility that could model these complex relationships. An AHS simulator that is composed of a C++ class library and a Tcl front end has been developed for this purpose. The classes that have been

designed include simulation classes to enable discrete event simulation, network classes to enable highway description, driver/vehicle classes to model the driver/vehicle unit, statistics collection classes and driver/vehicle generation classes. Some of the important features of the simulator are (1) the incorporation of various control and scheduling strategies, (2) the use of doubly linked lists to model the set of vehicles in a lane, (3) the concept of a *CarUnit* to model the driver/vehicle unit, (4) the concept of a *TripWire* to enable statistics collection, and (5) the Tcl front end to the simulator. The AHS simulator opens up possibilities for the study of a host of open problems. The simulator will enable the study of flow on the AHS with diverse origin/destination loadings of the system, lane scheduling algorithms, communication protocols and AVCS characteristics. In particular, the simulator can be used to assess the dynamic capacity of the AHS, namely, the speed/concentration/flow relationships that will be established on the AHS with many maneuvering vehicles under full automated control and for situations characterized by various percentages of automated vehicles on AHS with mixed traffic.

The developed C++ classes and the Tcl front end provide the tools required to set up, and run, large-scale simulations, and to obtain statistical information from them. This is critical for assessing alternatives, optimizing strategies, and to gain an understanding of technical limitations. Large-scale simulations will enable a study of the relative merits of the various control and scheduling algorithms in terms of their effect on such diverse issues as (1) the macroscopic traffic characteristics of speed, flow and concentration and (2) the effect that various control strategies will have on highway safety. An issue that can be studied with the simulator, and, that would be of particular interest to the commuters on the AHS is the effect that different lane scheduling strategies, communication protocols and lateral control strategies will have on the the probability of missing an exit. The cost of missing an exit (important for an individual commuter) must be weighed against the increase in highway capacity (important for the AHS as a whole), and acceptable solutions may well be a compromise between user optimal and system optimal solutions.

The concepts of lane assignment and dynamic capacity have been used in the proposed route guidance algorithm. The route guidance algorithm seeks a system optimal solution to the problem of assigning routes. The justification is that if all users complied with system

optimal decisions, a traffic flow would result that is overall more favorable (as measured in terms of a performance index such as travel time or throughput) than it would be with a user optimal solution. On an automated or semi-automated highway, good estimates of system-wide information can be obtained, thus increasing user confidence in the system optimal solution. This, along with some use of enforcement mechanisms, will increase user compliance of the assigned routes and lanes, and result in a more favorable traffic flow.

REFERENCES

[1] IVHS Society of America, "Strategic plan for IVHS in the United States, Report No. IVHS-AMER-92-3," Washington, DC, May 1992.

[2] US Department of Transportation, "Department of Transportation's IVHS strategic plan report to Congress," Dec 1992.

[3] Federal Highway Administration, Office of Contracts and Procurements, "Precursor systems analyses of automated highway systems," Washington, DC, Nov. 1992. (BAA) RFP No. DTFH61-93-R-00047.

[4] W. Stevens, "The use of system characteristics to define concepts for automated highway systems (AHS)," *Transportation Research Board Meeting, Paper No. 940990*, Jan. 1994.

[5] U. Karaaslan, P. Varaiya, and J. Walrand, "Two proposals to improve traffic flow," Tech. Rep. UCB-ITS-PRR-91-xx, Institute of Transportation Studies, University of California, Berkeley, 1991.

[6] A. F. Rumsey and E. T. Powner, "The synchronous moving-cell control philosophy for automated transportation systems," *Transportation Planning and Technology*, no. 2, pp. 156–164, 1974.

[7] B. S. Y. Rao and P. Varaiya, "Flow benefits of autonomous intelligent cruise control in mixed manual automated traffic," *Transportation Research Record*, no. 1408, pp. 36–43, 1993.

[8] B. S. Y. Rao, P. Varaiya, and F. Eskafi, "Investigations into achievable capacities and stream stability with coordinated intelligent vehicles," *Transportation Research Record*, no. 1408, pp. 27–35, 1993.

[9] H. S. J. Tsao, R. W. Hall, and B. Hongola, "Capacity of automated highway systems: Effect of platooning and barriers," Tech. Rep., PATH Research Report, 1993.

[10] R. W. Hall, "Longitudinal and lateral throughput on an idealized highway," *Transportation Science*, vol. 29, pp. 118–127, May 1995.

[11] B. S. Y. Rao and P. Varaiya, "Potential benefits of roadside intelligence for flow control in an IVHS," *Transportation Research Board*, 1994. Paper No. 940085.

[12] D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Lane assignment on an automated highway," in *Proceedings of the 1994 American Control Conference*, Baltimore, pp. 413–417, Jun. 1994.

[13] D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Combining lane assignment with route guidance in Corridor systems," in *Proceedings of the 34th IEEE Conference of Decision and Control*, New Orleans, Dec. 1995.

[14] J. Ousterhout, *Tcl and the TK ToolKit.* Reading, MA: Addison Wesley, 1994.

[15] D. P. Masher et al., "Guidelines for design and operation of ramp control systems," Report NCHRP 3-22, SRI Project 3340, Stanford Research Institute, Menlo Park, CA, 1975.

[16] M. Papageorgiou, ed., *Concise Encyclopedia of Traffic and Transportation Systems.* Oxford: Pergamon Press, 1991.

[17] F. Broqua et al., "Cooperative driving: Basic concepts and a first assessment of 'intelligent cruise control strategies' in advanced telematics in road guidance," in *Proceedings of the DRIVE Conference*, Brussels, Amsterdam: Elsevier Science Publishers, Feb. 1991.

[18] T. M. Jefferson, "Longitudinal control design for vehicle platoons on intelligent highways," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1992.

[19] D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Partitioned strategies for lane assignment in multi-lane AHS," in *Proceedings of the 33rd IEEE Conference of Decision and Control*, Lake Buena Vista, pp. 2938–2943, Dec. 1994.

[20] J. Medanic, D. Ramaswamy, W. R. Perkins, and R. Benekohal, "Partitioned lane assignment strategies for balancing excess lane capacity on AHS," in *Proceedings of the 1995 American Control Conference*, Seattle, pp. 3581–3585, Jun. 1995.

[21] P. Varaiya, "Smart cars on smart roads: Problems of control," *IEEE Transactions on Automatic Control*, vol. 38, pp. 195–207, Feb. 1993.

[22] "Solicitation for IVHS architecture development RFP No. DTFHG1-93-R-00065, FHWA," Mar. 1993.

[23] P. Varaiya and S. Shladover, "Sketch of an IVHS system architecture," Tech. Rep., UCB-ITS-PRR-91-3, Institute of Transportation Studies, University of California, Berkeley, CA, Feb. 1991.

[24] J. Banks, "Freeway speed-flow-concentration Relationships: More evidence and interpretations," *Highway Capacity, Flow Measurements and Theory, Transportation Research Record*, no. 1225, TRB, pp. 552–560, 1989.

[25] R. Gilchrist and F. Hall, "Three dimensional relationship among traffic flow theory variables," *Highway Capacity, Flow Measurements and Theory, Transportation Research Record*, no. 1225, TRB, pp. 99–108, 1989.

[26] B. Persaud and V. Hurdle, "Some new data that challenges some old ideas about speed-flow relationships," *Traffic Flow Theory and Highway Capacity, Transportation Research Record*, no. 1194, TRB, pp. 191–198, 1988.

[27] P. Ross, "Some properties of macroscopic traffic models," *Traffic Flow Theory and Highway Capacity, Transportation Research Record*, no. 1194, TRB, pp. 129–134, 1988.

[28] M. Cassidy, A. Skabardonis, and A. May, "Operation of major freeway weaving sections: Recent empirical evidence," *Highway Capacity, Flow Measurements and Theory, Transportation Research Record*, no. 1225, TRB, pp. 60–71, 1989.

[29] N. Duncan, "A further look at speed/flow/concentration," *Traffic Engineering and Control*, vol. 20, pp. 482–483, 1979.

[30] K. A. Ross, *Elementary Analysis:The Theory of Calculus.* New York, NY: Springer-Verlag, 1980.

[31] R. Fletcher, "Resolving degeneracy in quadratic programming," Numerical analysis report NA/135, Department of Mathematics and Computer Science, University of Dundee, Dundee, Scotland, 1991.

[32] "LINDO: An optimization modeling system, text and software." The Scientific Press, San Francisco, CA, 1988.

[33] A. Grace, *Optimization Toolbox User's Guide.* Natick, MA: The MathWorks, Inc., 1990.

[34] "NAG Fortran library manual, Mark 16," Oxford: NAG Ltd., 1993.

[35] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming.* Reading, MA: Addison Wesley, 1973.

[36] M. R. C. M. Berkelaar, *lp_solve: Solve (mixed integer) linear programming problems.* Manual.

[37] D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Lane Assignment on Automated Highway Systems." Accepted for publication by *IEEE Transactions on Vehicular Technology.*

[38] Federal Highway Administration, "TRAN-NETSIM User's Manual," U.S. Department of Transportation, 1989.

[39] A. Halati, J. F. Torres, and S. L. Cohen, "FRESIM-freeway simulation model," *Transportation Research Board*, no. 910202, 1991.

[40] E. C. P. Chang, C. J. Messer, and S. H. Wang, "Development of a freeway ramp control evaluation system," *Transportation Research Board*, no. 940666, 1994.

[41] Federal Highway Administration, "TRANSYT-7F: Traffic network study tool (Version 7F)," U.S. Department of Transportation, 1986.

[42] Federal Highway Administration, "Highway Capacity Software User's Manual," U. S. Department of Transportation, 1987.

[43] M. F. McGurrin, T. Paul, and T. R. Wang, "An object-oriented traffic simulation with IVHS applications," in *Vehicle Navigation and Information Systems Conference Proceedings*, pp. 551–561, 1991.

[44] M. V. Aerde, "INTEGRATION Model User's Guide, Version 2," Transportation Systems Research Group, Queen's University, 1991.

[45] Federal Highway Administration, "Development of INTRAS, A Microscopic Freeway Simulation Model, Vol. 1," U.S. Department of Transportation, 1980.

[46] R. F. Benekohal and J. Treiterer, "CARSIM: Car-following model for simulation of traffic in normal and stop-and-go situations," *Traffic Flow Theory and Highway Capacity, Transportation Research Record*, no. 1194, TRB, pp. 99–111, 1988.

[47] A. Zarean and Z. A. Nemeth, "WEAVSIM: A microscopic simulation model of freeway weaving sections," *Traffic Flow Theory and Highway Capacity, Transportation Research Record*, no. 1194, TRB, pp. 48–54, 1988.

[48] J. Adler, W. Recker, and M. McNally, "A conflict model and interactive simulator(FASTCARS) for predicting en route assessment and adjustment behavior in response to real-time traffic condition information," *Transportation Research Board*, no. 920948, 1992.

[49] A. Niehaus and R. F. Stengel, "Probability-based decision making for automated highway driving," *IEEE Transactions on Vehicular Technology*, vol. 43, pp. 626–634, Aug. 1994.

[50] F. Eskafi, D. Khorrambadi, and P.Varaiya, "SmartPath: An automated highway system simulator," Tech. Rep. 92-0708-1, Institute of Transportation, University of California, Berkeley, 1992.

[51] G. Booch, *Object Oriented Design with Applications.* Redwood City, CA: Benjamin/Cummings Publishing, 1991.

[52] P. Bratley, B. L. Fox, and L. E. Schrage, *A guide to simulation.* New York, NY: Springer-Verlag, 2 ed., 1987.

[53] Sun Microsystems, *SUNOS Manual pages, Version 4.1.*

[54] C. Chien and P. Ioannou, "Automatic vehicle-following," in *Proceedings of the American Control Conference*, Chicago, pp. 1748–1752, Jun. 1992.

[55] W. Ren and D. Green, "Continuous platooning: A new evolutionary and operating concept for automated highway systems," *Memorandum No. UCB/ERL M94/24*, April 1992.

[56] S. Sheikholeslam, "Control of a class of interconnected nonlinear dynamical systems: The platoon problem," Ph.D. thesis, Department of Electrical Engineering, University of California, Berkeley, 1991.

[57] L. L. Hoberock, "A survey of longitudinal acceleration comfort studies in ground transportation vehicles," *Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control*, pp. 76–84, Jun. 1977.

[58] M. Florian, "Nonlinear cost network models in transportation analysis," *Mathematical Programming Study*, vol. 26, pp. 167–196, 1986.

[59] D. P. Bertsekas and R. G. Gallager, *Data Networks*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[60] D. E. Boyce, "A framework for constructing network equilibrium models of urban location," *Transportation Science*, vol. 14, pp. 71–96, 1980.

[61] T. W. Haines, "A neural network shortest path algorithm and its use in a transportation application," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1994.

[62] D. Brandon, "Link capacity functions: A review," *Transportation Research*, pp. 223–236, 1976.

APPENDIX A

DATA FOR EXAMPLE IN CHAPTER 7

```
#System Size
19  20
#Velocities v3,v2,v1(km/hr)
110 100 90
#Congestion constants C3a, C3b, C2a etc.
0.003  0  0.003  0  0.003  0
#Lane capacities z3,z2,z1(vehicles/hour)
 8500 6500 5000
#Road Lengths L(1), L(2) etc. in m
 1600 3200 5000 6000 6600 7200 7800 8300 8900 9400 9900 10400 11000 11800 12600
13600 14600 15600 16600 18000
#rho0s (Flow from entrance 0 in the three lanes)

 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

#The Origin Destination  matrix
```

34	68	103	137	171	205	239	274	308	342	308	274	239	205	171	137	103	68	34
0	68	137	205	274	205	137	103	171	239	308	274	239	205	171	137	103	68	34
0	0	34	68	103	137	171	205	239	239	239	205	171	137	103	68	68	34	34
0	0	0	68	103	137	68	103	137	68	103	137	137	137	103	68	34	34	68
0	0	0	0	137	137	137	171	171	171	103	103	68	68	68	103	137	171	171
0	0	0	0	0	68	103	137	171	205	239	205	171	137	103	68	34	68	103
0	0	0	0	0	0	58	58	43	43	43	29	29	29	43	58	72	86	72
0	0	0	0	0	0	0	72	72	58	58	43	43	43	29	29	29	14	14
0	0	0	0	0	0	0	0	29	43	58	72	58	43	29	14	29	43	58
0	0	0	0	0	0	0	0	0	72	86	101	115	129	115	101	86	72	43
0	0	0	0	0	0	0	0	0	0	43	58	72	58	43	29	14	29	43
0	0	0	0	0	0	0	0	0	0	0	14	29	43	58	43	29	14	14
0	0	0	0	0	0	0	0	0	0	0	0	29	43	58	72	58	43	29
0	0	0	0	0	0	0	0	0	0	0	0	0	49	61	49	37	49	61
0	0	0	0	0	0	0	0	0	0	0	0	0	0	49	61	49	61	37
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	49	37	24	37
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	37	24	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	24	37
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12

APPENDIX B

PROGRAMS FOR THE TEST RUNS

This appendix contains the C++ and Tcl codes used for the test runs in Chapter 11.

B.1 Code for Test Run 1

```
/* This is the C++ code */
#include <stdlib.h>
#include "ahs_lib"

int main()
{
    /*Create an Event Calendar */
    EventCal EC;
    /* Create a car disposer */
    StdCarUnitDisposal SCD;
    /*Create a domain */
    StdDomain SD(&EC);
    /*Create highways */
    StdHighway SH;
    /*Create sections */
    StdSection SS1;
    SS1.SetId(1);
    double length = 5000; // A length of 5000m
    SL1.SetLen(length);
    StdSection SS2;
    SS2.SetId(2);
    /* Create lanes */
    StdLane SL1(&SCD);
    StdLane SL2(&SCD);
    /* Add the lanes to their sections */
```

```
SS1.AddLane(SL1);
SS2.AddLane(SL2);
SL1.SetId(1);
SL2.SetId(2);
SL2.AddAfter(SL1);
/* Add the sections to their highways */
SH.AddSection(SS1);
SH.AddSection(SS2);
/* Add the highways to the domain */
SD.AddHigh(SH);
/* Create an integrator */
StdIntegrator SI(SD, .01);
/* Create the trajectory for the leading vehicle */
ValArray<double>* A = new ValArray<double>;
int i;
for(i = 1; i <= 45; i++)
    A->Append(3 * i *.01);
for(i = 46; i <= 1000; i++)
    A->Append(1.35);
for(i = 1001; i <= 1045; i++)
    A->Append(1.35- 3*(i*.01-10));
for(i = 1046; i <= 11000; i++)
    A->Append(0.);
FollowTrajLC* longc = new FollowTrajLC(*A);
PMVehicle* V1 = new PMVehicle;
/* Create a car generator and set its destination*/
StdCarUnitGen SG1(SL1);
SG1.AddDest(3, 1);
/* Generate the leading car and set its initial state */
SG1.GenerateCarUnit(V1,longc);
VehicleState VS;
V1->GetState(VS);
VS.long_speed = 0;
VS.long_pos = 1000;
VS.pos_on_lane = 1000;
V1->SetState(VS);
/* Generate the remaining vehicles */
PMVehicle **V = new PMVehicle*[19];
PlatoonLC **Pl = new PlatoonLC*[19];
AICC** Co = new AICC*[19];
HumanVision **VIS = new HumanVision*[19];
int num_cars = 19;
for(i = 1; i <= num_cars; i++)
{
```

```
        double dist;
        dist = 30;
        Co[i-1] = new AICC;
        VIS[i-1] = new HumanVision;
        Pl[i-1] = new PlatoonLC(Co[i-1]);
        V[i-1] = new PMVehicle;
        /* Important: The CarUnit should be generated before the vehicle's
        velocity and position are set */
        SG1.GenerateCarUnit(V[i-1], Pl[i-1], VIS[i-1]);
        V[i-1]->GetState(VS);
        VS.long_pos = 1000. - dist*i;
        VS.pos_on_lane = 1000. - dist*i;
        VS.long_speed = 0.;
        V[i-1]->SetState(VS);
    }
    /* Make a end simulation event */
    double endtime;
    endtime = 109.;
    EC.EndSimAt(endtime);
    /* Make a PhysicalControl event and insert it in the event calendar */
    int print_pos = 1;
    int print_vel = 1;
    int print_acc = 1;
    PhysicalControl PC(SI,print_pos, print_vel, print_acc);
    EC.InsertByTime(&PC, 0.0);
    /* Start the simulation */
    EC.Run();
}
```

B.2 Code Used for Test Run 2

```
# Highway Description
highway I75
section I75.1
section I75.2
section I75.3
lane I75.1.1 25 20000
lane I75.1.2 25 20000
lane I75.1.3 25 20000
lane I75.2.1 25 20000
lane I75.2.2 25 20000
lane I75.2.3 25 20000
connect I75.1.1 I75.2.1
connect I75.1.2 I75.2.2
```

```
connect I75.1.3 I75.2.3
integrator_step .01
# Carunit component generators
veh_generator vg
cf_generator aicc cfg
longc_generator platoonlc log cfg
lanec_generator lancg
latc_generator simple lag lancg
pathc_generator human_pc pg
# Create car generators for some lanes
cargen special sg1 I75.1.1 vg log lag pg
cargen special sg11 I75.1.2 vg log lag pg
cargen special sg12 I75.1.3 vg log lag pg
cargen_exits sg1 I75.3 1
cargen_exits sg11 I75.3 1
cargen_exits sg12 I75.3 1
# Generate 4 cars
generate_car sg1 # Car Unit 1
generate_car sg11 # Car Unit 2
generate_car sg12 # Car Unit 3
generate_car sg12 # Car Unit 4
# Set vehicle states
states_car I75.1.1.1 20 25 # Car unit 1
states_car I75.1.3.1 26 25 # Car unit 3
states_car I75.1.3.2 13 25 # Car Unit 4
# Lateral control events
latevent I75.1.1.1 left 1.5
latevent I75.1.3.2 right 0
latevent I75.1.3.1 right 1
latevent I75.1.2.1 right 2
stopat 31
print_params 1 1 1 1
update_times 1000 1000
run
```

B.3 Code for Test Run 3

This program makes use of Tcl procedures. The procedures are listed first, followed by the code for the main program.

```
proc tcl_highway {hwyname num_secs num_lanes} {
    # This procedure creates a highway named hwyname and gives it
    # transition sections and lanes
    global LEN
```

```
    set LEN 300
    highway $hwyname
    for {set i 1} { $i <= $num_secs} {incr i} {
        tcl_section $hwyname $i $num_lanes
    }
    tcl_join_secs $hwyname $num_secs $num_lanes
    # ADD THE HIGHWAY ID HERE
}

proc tcl_section {hwyname sec_id num_lanes} {
    # This procedure creates a transition section with lanes
    global LEN
    section transition [join "$hwyname $sec_id" .] $LEN
    for {set i 0} {$i <= $num_lanes} {incr i} {
        lane [join "$hwyname $sec_id $i" .]
    }
}

proc tcl_join_secs {hwyname num_secs num_lanes} {
    for {set i 1} {$i < $num_secs} {incr i} {
        for {set j 1} {$j <= $num_lanes} {incr j} {
            set lane1 [join "$hwyname $i $j" .]
            set lane2 [join "$hwyname [expr $i +1] $j" .]
            connect $lane1 $lane2
        }
    }
}

proc tcl_wire {} {
    # This routine places flow wires across the lanes of the 1st
    # (n-1) sections
    global LEN
    set highways [sinfo domain]
    foreach i $highways {
        set secs [sinfo highway $i]
        set first_secs [lrange $secs 0 [expr [llength $secs] -2]]
        foreach j $first_secs {
            set lanes [sinfo section $j]
            foreach k $lanes {
                flow_wire [expr $LEN - 1] $k
                puts "Added flow wire $k"
            }
        }
    }
```

```
}

proc tcl_add_gens {} {
    #This routine places car generators at the entrances of the
    # 1st n-1 sections. It also places car generators on the
    # first lanes
    set cargen_lanes [tcl_cargen_lanes]
    foreach i $cargen_lanes {
        tcl_add_car_gen $i
        puts "Added car_gen $i"
    }
}

proc tcl_add_car_gen {lane_name} {
    #Carunit component generators
    veh_generator vg
    cf_generator aicc cfg
    longc_generator platoonlc log cfg
    lanec_generator lancg
    latc_generator simple lag lancg
    pathc_generator human_pc pg
    set cargen_name $lane_name
    cargen special $cargen_name $lane_name vg log lag pg
    max_to_generate $cargen_name 20
}

proc tcl_cargen_lanes {} {
    # This routine finds the lanes that should have cargens
    set highways [sinfo domain]
    foreach i $highways {
        set secs [sinfo highway $i]
        set middle_secs [lrange $secs 1 [expr [llength $secs] -2]]
        foreach j $middle_secs {
            set entry_lane [join "$j 0" .]
            lappend cargen_lanes $entry_lane
        }
        #Now find the first lanes
        set first_sec [lindex $secs 0]
        set first_lanes [sinfo section $first_sec]
        set cargen_lanes [concat $cargen_lanes $first_lanes]
    }
    return $cargen_lanes
}
```

```
proc tcl_add_exits {} {
    # The transition lane car generators will choose an exit with
    # equal probability
    #I will manually set the exit probabilities for the car
    #generators. Assuming I have only one highway
    set hwyname [sinfo domain]
    tcl_cargen_exits $hwyname.1.0 {{2 .3} {3 .3} {4 .4}}
    tcl_cargen_exits $hwyname.1.1 {{1 .6} {2 .2} {3 .1} {4 .1}}
    tcl_cargen_exits $hwyname.2.0 {{3 .35} {4 .65}}
    tcl_cargen_exits $hwyname.3.0 {{4 1}}
    tcl_cargen_exits $hwyname.1.2 {{1 .3} {2 .4} {3 .2} {4 .1}}
}

proc tcl_cargen_exits {cgname exits} {
    set hwyname [sinfo domain]
    foreach i $exits {
        set exit [lindex $i 0]
        set prob [lindex $i 1]
        cargen_exits $cgname $hwyname.$exit $prob
        puts "cargen_exits $cgname $hwyname.$exit $prob"
    }
}

proc tcl_start_gens {mean std_dev} {
    set cargen_lanes [tcl_cargen_lanes]
    foreach i $cargen_lanes {
        cgevent $i normal $mean $std_dev 0.0
        puts "CGEvent for $i with mean $mean"
    }
}

#Main program
tcl_highway I75 4 2
tcl_wire
tcl_add_gens
#tcl_add_exits3
tcl_add_exits1
tcl_start_gens 5 .5
integrator_step .01
stat_event 5 0.0
stopat 60
print_params 1 1 1 1
run
```

APPENDIX C

THE .h FILES IN THE C++ CLASS LIBRARY

```
#ifndef _CARACCEPTOR_H_
#define _CARACCEPTOR_H_

class CarUnit;

class CarUnitAcceptor
{
    private:
        void operator=(const CarUnitAcceptor&);

    public:
        virtual void AcceptCU(CarUnit*) = 0;
        virtual int CUAccId() = 0;
};

#endif
```

```
#ifndef _CARDISPOSAL_H_
#define _CARDISPOSAL_H_

class CarUnit;

class CarUnitDisposal
{
    // This takes care of stats collection and freeing up memory
    private:
        void operator=(const CarUnitDisposal&);
```

```
    public:
        virtual void Dispose(CarUnit* ) = 0;
        virtual void OffRoad(CarUnit* ) = 0;
        virtual int NumDispose() = 0;
        virtual int NumOffRoad() = 0;
};

class StdCarUnitDisposal : public CarUnitDisposal
{
    private:
        int car_disp;
        int car_offroad;

        void operator=(const StdCarUnitDisposal&);
        StdCarUnitDisposal(const StdCarUnitDisposal&);
    public:
        StdCarUnitDisposal();
        void Dispose(CarUnit* );
        void OffRoad(CarUnit* );
        int NumDispose();
        int NumOffRoad();
};

#endif
```

```
#ifndef _CAR_FOLLOWER_H_
#define _CAR_FOLLOWER_H_

#include "ValArray.h"
#include "DLinkedList.h"
#include "CarSimData.h"
#include "RandGen.h"
#include "Vehicle.h"

class CarUnit;
class Vision;

class CarFollower
{
    private:
        void operator=(const CarFollower&);
    protected:
        static CarSimData csd;
```

```
    public:
        enum {RAPID_ACTION=1, NORMAL_ACTION};
        virtual void Control(ValArray<CarUnit*>& CUA, double time_step,
             CarUnit* cu, int CFState=RAPID_ACTION) = 0;
        virtual void OneSideControl(ValArray<CarUnit*>& CUA,
             CarUnit* side_car, double ts, CarUnit* cu,
             int Cfs=RAPID_ACTION)= 0;
        virtual void ManySideControl(ValArray<CarUnit*>& CUA,
             DLinkedList<CarUnit*>&,double ts,CarUnit* cu,
             int Cfs=RAPID_ACTION) = 0;
        virtual void SetHeadway(double dist) = 0;
        virtual double GetHeadway() = 0;
        virtual double GetDefHeadway() = 0;
        virtual double GetTotHeadway(double speed) = 0;
        virtual int GetRange() = 0;
        virtual void SetNcc(int flag) = 0;
        static CarSimData& GetCsd();
};

class StdCarFollower : public CarFollower
{
    private:
        void operator=(const StdCarFollower&);
    protected:
        int ncc_flag;
        int side_info_flag;

        virtual double FindActJerk(double jerk,double acc, double speed,
             double ts, double max_dec);
        virtual double FindMaxAcc(double u);
        virtual double FindComfDec(double u);
        virtual double NonCollisionJerk(VehicleState& VS_nbor,
            VehicleParams& VP_nbor, VehicleState& VS, VehicleParams& VP,
             double ts);
        virtual double MaintainSpeed(VehicleState& VS, double ts);
        int FindCFState(double jerk, double jerk_ncc, int given_state);
        StdCarFollower();
    public:
        enum {NO_CHECK_NCC = 1, CHECK_NCC};
        enum {INFORMED = 1, NOT_INFORMED};
        // Dummy functions used by certain types of StdCarFollowers
        virtual void SetHeadway(double);
        virtual double GetHeadway();
        virtual double GetDefHeadway();
```

```
        virtual double GetTotHeadway(double speed);
        virtual int GetRange();
        virtual void SetNcc(int flag);
        virtual void OneSideControl(ValArray<CarUnit*>& CUA,
             CarUnit* side_car,double ts,CarUnit* cu,
             int CFState=RAPID_ACTION);
        virtual void ManySideControl(ValArray<CarUnit*>& CUA,
             DLinkedList<CarUnit*>&,double ts,CarUnit* cu,
             int CFState=RAPID_ACTION);
};

class FollowTraj : public StdCarFollower
{
    private:
        ValArray<double>& acc;
        int count;

        void operator=(const FollowTraj&);
        FollowTraj(const FollowTraj&);
    public:
        FollowTraj(ValArray<double>& A);
        void Control(ValArray<CarUnit*>&, double ts,CarUnit* cu,
             int CFState = RAPID_ACTION);
};

class CSCarFollower : public StdCarFollower
{
    // This is based on CarSim's controller
    private:
        RandGen rg;
        double time_step;

        double brt;
        double start_up_time;
        int stopngo_counter;

        double FindTrackJerk(double des, VehicleState& VS);
        double FindHeadwayJerk(VehicleState& VS_lead, VehicleState& VS,
            VehicleParams& VP, double lead_jerk, double headway);
        double StopNGo(ValArray<CarUnit*>& CUA, CarUnit* CU,
             double acc_ncc, double start_acc, double lead_speed);

        void operator=(const CSCarFollower&);
        CSCarFollower(const CSCarFollower&);
```

```
    protected:
        virtual double NonCollisionJerk(VehicleState& VS_nbor,
            VehicleParams& VP_nbor, VehicleState& VS, VehicleParams& VP,
             double ts);
    public:
        CSCarFollower();
        void Control(ValArray<CarUnit*>&, double ts,CarUnit* CU,
             int CFState = RAPID_ACTION);
        int GetRange();
};

typedef struct CCdataStruct
{
    double Kv_cc;
    double Ka_cc;
    double def_track_speed;

    double Kv;
    double Cp;
    double Cv;
    double Ca;
    double Ka;
    double lambda2;
    double lambda3; // Separation distance
} CCdata;

class CC: public StdCarFollower
{
    // This is simple cruise control. It also checks for ncc
    private:
        void operator=(const CC&);
        CC(const CC&);
    protected:
        static CCdata ccd;
        double track_speed;
        double safe_dist;

    public:
        CC(double speed = 25);
        void Control(ValArray<CarUnit*>&, double ts,CarUnit* CU,
             int CFState = RAPID_ACTION);
        // This actually sets opt_speed
        virtual void SetHeadway(double speed);
        virtual double GetHeadway();
```

```
        virtual double GetDefHeadway();
        virtual int GetRange();

};

typedef struct AICCdataStruct
{
    double Kv;
    double Cp;
    double Cv;
    double Ca;
    double Ka;
    double lambda2;
    double def_headway; // Default Separation distance
    double range; // Within this, AICC will kick in
} AICCdata;

class AICC: public StdCarFollower
{
    // This is based on Chien and Ioannou's Autonomous Intelligent
    // Cruise Control
    private:
        void operator=(const AICC&);
        AICC(const AICC&);
    protected:
        static AICCdata aiccd;
        double lambda3;

    public:
        AICC();
        void Control(ValArray<CarUnit*>&, double ts,CarUnit* CU,
             int CFState = RAPID_ACTION);
        virtual void SetHeadway(double l3);
        virtual double GetHeadway();
        virtual double GetDefHeadway();
        virtual double GetTotHeadway(double speed);
        virtual int GetRange();
};

typedef struct CPStruct
{
    int L;
    double Kp[3];
    double Kv[3];
```

```
    double Ka[3];
    double defH;
    double lambda;
} CPData;

class CP : public StdCarFollower
{
    private:
        void operator=(const CP&);
    protected:
        CPData *cpd;
        double H;

        CP();
    public:
        virtual void Control(ValArray<CarUnit*>&, double ts,CarUnit*,
             int CFState = RAPID_ACTION);
        virtual void SetHeadway(double dist);
        virtual double GetHeadway();
        virtual double GetDefHeadway();
        virtual double GetTotHeadway(double speed);
        virtual int GetRange();
};

class CP_C: public CP
{
    // Wei Ren's CP controller type c)
    private:
        static CPData *cpd_C;

        void operator=(const CP_C&);
        CP_C(const CP_C&);
    public:
        CP_C();
};

class CP_E : public CP
{
    // Wei Ren's CP controller type e)
    private:
        static CPData *cpd_E;

        void operator=(const CP_E&);
        CP_E(const CP_E&);
```

```
    public:
        CP_E();
};

class CP_G : public CP
{
    // Wei Ren's CP controller type g)
    private:
        static CPData *cpd_G;

        void operator=(const CP_G&);
        CP_G(const CP_G&);
    public:
        CP_G();
};

class CP_H : public CP
{
    // Wei Ren's CP controller type h)
    private:
        static CPData *cpd_H;

        void operator=(const CP_H&);
        CP_H(const CP_H&);
    public:
        CP_H();
};

class CP_J : public CP
{
    // Wei Ren's CP controller type j)
    private:
        static CPData *cpd_J;

        void operator=(const CP_J&);
        CP_J(const CP_J&);
    public:
        CP_J();
};

class CP_L : public CP
{
    // Wei Ren's CP controller type l)
    private:
```

```
        static CPData *cpd_L;

        void operator=(const CP_L&);
        CP_L(const CP_L&);
    public:
        CP_L();
};

typedef struct DPDataStruct
{
    double Kp1;
    double Kv1;
    double Ka1;
    double Kv_lead;
    double Ka_lead;
    double defH;
    int max_plt_size;
} DPData;

class DP : public StdCarFollower
{
    private:
        static DPData *dpd;
        double H;

        void operator=(const DP&);
        DP(const DP&);

    public:
        DP();
        void Control(ValArray<CarUnit*>&, double ts, CarUnit*,
             int CFState = RAPID_ACTION );
        void SetHeadway(double dist);
        double GetHeadway();
        double GetDefHeadway();
        double GetTotHeadway(double speed);
        int GetRange();
};

#endif
```

```
#ifndef _CARGEVENT_H_
#define _CARGEVENT_H_
```

```
#include "Event.h"
#include "RandGen.h"

class EventCal;
class CarUnitGen;

class CarGEvent: public Event
{
    private:
        void operator=(const CarGEvent&);
        CarGEvent(const CarGEvent&);
    protected:
        CarUnitGen& cg;

        CarGEvent(CarUnitGen&);
    public:
        /* Inherit the event behaviour */
};

class PerCarGEvent:    public CarGEvent
{
    // Periodic Car Generation Event
    private:
        double period;

        void operator=(const PerCarGEvent&);
        PerCarGEvent(const PerCarGEvent&);
    public:
        PerCarGEvent(CarUnitGen& CG, double p);
        int Priority() { return 1; }
        void Action(EventCal& );
};

class OneShotCarGEvent: public CarGEvent
{
    // One Shot Car Generation Event
    private:
        void operator=(const OneShotCarGEvent&);
        OneShotCarGEvent(const OneShotCarGEvent&);
    public:
        OneShotCarGEvent(CarUnitGen& CG): CarGEvent(CG) {}
        int Priority() { return 1; }
        void Action(EventCal& );
```

```
};

class UnifCarGEvent: public CarGEvent
{
    private:
        double u1, u2;
        RandGen rg; // A random number generator

        void operator=(const UnifCarGEvent&);
        UnifCarGEvent(const UnifCarGEvent&);
    public:
        UnifCarGEvent(CarUnitGen& CG, double U1, double U2);
        int Priority() { return 1; }
        void Action(EventCal& );
};

class ExpCarGEvent: public CarGEvent
{
    private:
        double lamda;
        RandGen rg; // A random number generator

        void operator=(const ExpCarGEvent&);
        ExpCarGEvent(const ExpCarGEvent&);
    public:
        ExpCarGEvent(CarUnitGen& CG, double L);
        int Priority() { return 1; }
        void Action(EventCal& );
};

class NormCarGEvent: public CarGEvent
{
    private:
        double mean, std_dev;
        RandGen rg; // A random number generator

        void operator=(const NormCarGEvent&);
        NormCarGEvent(const NormCarGEvent&);
    public:
        NormCarGEvent(CarUnitGen& CG, double Mean, double StdDev);
        int Priority() { return 1; }
        void Action(EventCal& );
};
```

```
#endif
```

```
#ifndef _CARGEN_H_
#define _CARGEN_H_

#include "ValArray.h"
#include "RandGen.h"
#include "Generators.h"

class CarUnitAcceptor;
class Vehicle;
class Vision;
class LongControl;
class LatControl;
class PathControl;

class CarUnitGen
{
    private:
        static int car_num;    // The current car ID

    public:
        virtual CarUnitAcceptor& CA() = 0;
        virtual void GenerateCarUnit(Vehicle*V = NULL,
            LongControl* longc=NULL, Vision* Vis=NULL,
            LatControl* latc = NULL,PathControl* pathc = NULL) = 0;
        virtual void AddDest(int d, double prob) = 0;
        virtual int NumCarGen()  = 0;
        virtual int GetMaxToGenerate() = 0;
        virtual void SetMaxToGenerate(int MC) = 0;
        static int TotalNumGen();
        virtual ~CarUnitGen();
};

class StdCarUnitGen: public CarUnitGen
{
    protected:
        CarUnitAcceptor& ca;

        int num_generated_so_far;
        int max_to_generate;
        typedef struct DestStruct
        {
```

```
            int dest;
            double prob_dest;
            double range;
        } Dest;

        ValArray<Dest*> vad;
        int first_time; // Used to check that the sum of the probs =1
        RandGen rg;
        int find_dest();

        void operator=(const StdCarUnitGen&);
        StdCarUnitGen(const StdCarUnitGen&);
    public:
        StdCarUnitGen(CarUnitAcceptor&);
        CarUnitAcceptor& CA();
        virtual int GetMaxToGenerate();
        virtual void SetMaxToGenerate(int MC);
        virtual void GenerateCarUnit(Vehicle*V = NULL,
            LongControl* longc=NULL, Vision* Vis=NULL,
            LatControl* latc = NULL,PathControl* pathc = NULL);
        void AddDest(int d, double prob);
        int NumCarGen();
        ~StdCarUnitGen();
};

typedef struct GensStruct
{
    VehicleGen* vg;
    LongCGen* log;
    LatCGen* latg;
    PathCGen* pg;
} AllGens;

class SpecialCUGen : public StdCarUnitGen
{
    private:
        AllGens& gens;

    public:
        SpecialCUGen(CarUnitAcceptor&, AllGens& AG);
        /* Though this method doesn't use these arguments, they were
        put in for signature compatibility. Admittedly a bad design! */
        void GenerateCarUnit(Vehicle*V = NULL,
            LongControl* longc=NULL, Vision* Vis=NULL,
```

```
            LatControl* latc = NULL,PathControl* pathc = NULL);
};

#endif
```

```
#ifndef _CARSIMDATA_H_
#define _CARSIMDATA_H_

#include "ValArray.h"

typedef struct AccDecStruct
{
    double max_acc; // Maximum allowable acceleration
    double dec;     // Comfortable deceleration
    double speed;
}    AccDec;

typedef struct ReacTimeStruct
{
    double reac_time;
    double prob;
} ReacTime;

class CarSimData
{
    private:
        ValArray<AccDec*> vaad;
        ValArray<ReacTime*> vart;
        double short_buffer;
        double long_buffer;
        double max_jerk;
        double min_jerk;

        void operator=(const CarSimData&);
        CarSimData(const CarSimData&);
    public:
        CarSimData();
        ValArray<AccDec*> *GetVaad();
        ValArray<ReacTime*> *GetVart();
        double GetShortBuffer();
        double GetLongBuffer();
        double GetMaxJerk();
        double GetMinJerk();
```

```
        ~CarSimData();
};

#endif
```

```
#ifndef _CARUNIT_H_
#define _CARUNIT_H_

class Vehicle;
class Vision;
class LongControl;
class LatControl;
class PathControl;
class Lane;

typedef struct CarUnitDataStruct
{
    int cu_id;
    int entry_num;
    int exit_num;
    int control_flag;
    int ref_count; // See constructor for meaning of this
} CarUnitData;

class CarUnit
{
    private:
        Vehicle& veh;
        Vision& vis;
        LongControl& longc;
        LatControl& latc;
        PathControl& pathc;
        Lane* lane;
        CarUnitData cud;

        void operator=(const CarUnit&);
        CarUnit(const CarUnit&);
    public:
        CarUnit(Vehicle&,Vision&,LongControl&,LatControl&,PathControl&);
        Vehicle& GetVehicle();
        Vision& GetVision();
        LongControl& GetLongC();
        LatControl& GetLatC();
```

```
        PathControl& GetPathC();
        void SetLane(Lane*, double long_pos, double lat_pos );
        Lane* GetLane();
        void DoLongControl(double time_step);
        int ToPathControl();
        void SetId(int ID);
        int GetId();
        void SetEntry(int);
        int GetEntry();
        void SetExit(int);
        int GetExit();
        void IncrRefCount();
        void DecrRefCount();
        int ToControl();
        void StopControl();
        ~CarUnit();
};

#endif
```

```
#ifndef    _COLLECTION_H_
#define    _COLLECTION_H_

#define    Collection mtCollection

#include    <assert.h>

template<class T>
class    Collection
{
    private:
        void operator=(const Collection<T> &);
    public:
        virtual int Size() const = 0;
        virtual void Set(int index, T val) = 0;
        virtual    T Get(int index) const = 0;
        virtual ~Collection() {}
};

template<class T, class C>
class    SimCollection : public Collection<C>
{
    private:
```

```
        Collection<T>    &tc;
        void operator=(const SimCollection<T,C> &c);
    public:
        SimCollection(const SimCollection<T,C> &c);
        SimCollection(Collection<T> &t);
        int Size() const;
        void Set(int index, C val);
        C Get(int index) const;
};

#endif
```

```
#ifndef _DLINKEDLIST_H_
#define _DLINKEDLIST_H_

#include <iostream.h>
#include <assert.h>

/* This file defines the classes used in the doubly linked list */

template <class T> class DLinkedList
{
    public:
        class Node
        {
            public:
             T data;
             Node* next;
             Node* prev;
        };

    private:
        DLinkedList<T>& operator=(const DLinkedList<T>& DLL);
    protected:
        Node* head;
        Node* tail;
        int size;

    public:
        DLinkedList();
//Deep copy constructor
        DLinkedList( const DLinkedList<T>& DLL);
        DLinkedList(T t);
```

```
        void AddToFront(T t);
        void Append(T t);
        void AddFromBackAt(Node* N, T t, int before=1);
        void AddFromFrontAt(Node*N, T t, int before=1);
        // Removes head and returns its data
        T PopHead();
        // Removes tail and returns its data
        T PopTail();
        // Plucks the element t out of the list
        void Pluck(T t);
        int GetSize() const;
        T GetRef(int num);
        //Returns a pointer to head
        Node* GetHead();
        //Returns a pointer to tail
        Node* GetTail();
        //Finds the node that has the data = t
        Node* FindInList(T t);
        void PrintList();
        ~DLinkedList();
};

#endif
```

```
#ifndef _DOMAIN_H_
#define _DOMAIN_H_

#include "ValArray.h"

class Highway;
class EventCal;

class Domain
{
    private:
        void operator=(const Domain& );
    public:
        virtual ValArray<Highway*>& GetHigh() = 0;
        virtual void AddHigh(Highway&) = 0;
        virtual int CarsInDomain() = 0;
};

class StdDomain:public Domain
```

```
{
    private:
        static EventCal* ec;
        void operator=(const StdDomain& );
        StdDomain(const StdDomain& );
    protected:
        ValArray<Highway*> vah;

    public:
        StdDomain(EventCal* EC);
        ValArray<Highway*>& GetHigh();
        void AddHigh(Highway&);
        int CarsInDomain();
        static EventCal* GetEC();
};

#endif
```

```
#ifndef _ENDSIM_H_
#define _ENDSIM_H_

#include <iostream.h>
#include "Event.h"

class EventCal;

// This defines an end simulation event.

class EndSim : public Event
{
    private:
        int* stop;

        void operator=(const EndSim&);
        EndSim(const EndSim&);
    public:
        EndSim(int* STOP) {stop = STOP;}
        int Priority() { return 1; }
        void Action(EventCal& EC)
        {
            *stop = 1; // Stop the simulation
        }
};
```

```
#endif
```

```
#ifndef _EVENT_H_
#define _EVENT_H_

class EventCal;

class Event
{
    private:
        void operator=(const Event&);

    public:
        virtual int Priority() = 0;
        virtual void Action( EventCal& ) = 0;
        virtual ~Event() {}
};

#endif
```

```
#ifndef _EVENTCAL_H_
#define _EVENTCAL_H_

#include "Event.h"
#include "LinkedList.h"

class Event;
class EndSim;

class CalNodeData
{
    private:
        Event *event;
        double time;

    public:
        friend class EventCal;
        friend ostream& operator <<(ostream& o, CalNodeData &CND);
};

ostream& operator <<(ostream& o, CalNodeData &CND);
```

```
class EventCal : public LinkedList<CalNodeData>
{
    private:
        double current_time;
        int trace; // Flag to print the time at each step
        int stop; //Flag to stop the simulation
        EndSim* es;

        void operator=(const EventCal&);
        EventCal(const EventCal&);
    public:
        enum{TRACE = 1};
        enum{ON = 1, OFF};
        EventCal(int t = 0);
        void SetTrace(int);
        double CurrentTime();
        void InsertByTime(Event*, double dtime);
        void EndSimAt(double );
        void Restart(); // This sets stop = 0
        void Run();
        ~EventCal();
};

#endif
```

```
#ifndef _GENERATORS_H_
#define _GENERATORS_H_

#include "ValArray.h"
#include "PathControl.h"
#include "CarFollower.h"

class Vehicle;
class SegmentControl;
class PathControl;
class LongControl;
class LatControl;
class LaneChanger;
class Vision;

/* The vehicle generators. At the moment, this is not abstract since we
have only one type of vehicle */
```

```
class VehicleGen
{
    private:
        void operator=(const VehicleGen&);
        VehicleGen(const VehicleGen&);
    public:
        VehicleGen();
        Vehicle* GenVehicle();
};

/* The car follower generators */
class CarFollowerGen
{
    public:
        virtual CarFollower* GenCF( Vision** ) = 0;
};

class FollowTrajGen : public CarFollowerGen
{
    private:
        ValArray<double>& acc;
        void operator=(const FollowTrajGen&);
        FollowTrajGen(const FollowTrajGen&);
    public:
        FollowTrajGen(ValArray<double>& A);
        CarFollower* GenCF(Vision**);
};

class CSCarFollowerGen : public CarFollowerGen
{
    private:
        void operator=(const CSCarFollowerGen&);
        CSCarFollowerGen(const CSCarFollowerGen&);
    public:
        CSCarFollowerGen();
        CarFollower* GenCF(Vision** );
};

class AICCGen : public CarFollowerGen
{
    private:
        void operator=(const AICCGen&);
        AICCGen(const AICCGen&);
    public:
```

```
        AICCGen();
        CarFollower* GenCF(Vision**);
};

class CP_CGen : public CarFollowerGen
{
    private:
        void operator=(const CP_CGen&);
        CP_CGen(const CP_CGen&);
    public:
        CP_CGen();
        CarFollower* GenCF(Vision**);
};

class CP_EGen : public CarFollowerGen
{
    private:
        void operator=(const CP_EGen&);
        CP_EGen(const CP_EGen&);
    public:
        CP_EGen();
        CarFollower* GenCF(Vision**);
};

class CP_GGen : public CarFollowerGen
{
    private:
        void operator=(const CP_GGen&);
        CP_GGen(const CP_GGen&);
    public:
        CP_GGen();
        CarFollower* GenCF(Vision**);
};

class CP_HGen : public CarFollowerGen
{
    private:
        void operator=(const CP_HGen&);
        CP_HGen(const CP_HGen&);
    public:
        CP_HGen();
        CarFollower* GenCF(Vision**);
};
```

```
class CP_JGen : public CarFollowerGen
{
    private:
        void operator=(const CP_JGen&);
        CP_JGen(const CP_JGen&);
    public:
        CP_JGen();
        CarFollower* GenCF(Vision**);
};

class CP_LGen : public CarFollowerGen
{
    private:
        void operator=(const CP_LGen&);
        CP_LGen(const CP_LGen&);
    public:
        CP_LGen();
        CarFollower* GenCF(Vision**);
};

class DPGen : public CarFollowerGen
{
    private:
        void operator=(const DPGen&);
        DPGen(const DPGen&);
    public:
        DPGen();
        CarFollower* GenCF(Vision**);
};

/* The longitudinal controller generators */
class LongCGen
{
    public:
        virtual LongControl* GenLongC(Vision**) = 0;
};

class FollowTrajLCGen: public LongCGen
{
    private:
        ValArray<double>& acc;
        FollowTrajGen* cf_gen;
        void operator=(const FollowTrajLCGen&);
        FollowTrajLCGen(const FollowTrajLCGen&);
```

```
    public:
        FollowTrajLCGen(ValArray<double>& A, FollowTrajGen* CFgen);
        LongControl* GenLongC(Vision**);
};

class CarSimLCGen : public LongCGen
{
    private:
        CSCarFollowerGen* cf_gen;
        void operator=(const CarSimLCGen&);
        CarSimLCGen(const CarSimLCGen&);
    public:
        CarSimLCGen(CSCarFollowerGen* CFgen);
        LongControl* GenLongC(Vision**);
};

class PlatoonLCGen : public LongCGen
{
    private:
        CarFollowerGen* cf_gen;
        void operator=(const PlatoonLCGen&);
        PlatoonLCGen(const PlatoonLCGen&);
    public:
        PlatoonLCGen(CarFollowerGen* CFgen);
        LongControl* GenLongC(Vision**);
};

/* The lane changer generator. This is not abstract, since there is
of its kind. */
class LaneChGen
{
    private:
        void operator=(const LaneChGen&);
        LaneChGen(const LaneChGen&);
    public:
        LaneChGen();
        LaneChanger* GenLaneCh();
};

/* The lateral controller generators */
class LatCGen
{
    protected:
        LaneChGen* lcg;
```

```
    public:
        virtual LatControl* GenLatC() = 0;
};

class SimpleLatCGen : public LatCGen
{
    private:
        void operator=(const SimpleLatCGen&);
        SimpleLatCGen(const SimpleLatCGen&);
    public:
        SimpleLatCGen(LaneChGen* LCG);
        LatControl* GenLatC();
};

class QuickLatCGen : public LatCGen
{
    private:
        void operator=(const QuickLatCGen&);
        QuickLatCGen(const QuickLatCGen&);
    public:
        QuickLatCGen(LaneChGen* LCG);
        LatControl* GenLatC();
};

/* The path controller generators */
class PathCGen
{
    public:
        virtual PathControl* GenPathC() = 0;
};

class HumanPCGen : public PathCGen
{
    private:
        void operator=(const HumanPCGen&);
        HumanPCGen(const HumanPCGen&);
    public:
        HumanPCGen();
        PathControl* GenPathC();
};

class SemiCompliantPCGen : public PathCGen
{
    private:
```

```
        PathArray& vap;
        void operator=(const SemiCompliantPCGen&);
        SemiCompliantPCGen(const SemiCompliantPCGen&);
    public:
        SemiCompliantPCGen(PathArray& VAP);
        PathControl* GenPathC();
};

class CompliantPCGen : public PathCGen
{
    private:
        PathArray& vap;
        void operator=(const CompliantPCGen&);
        CompliantPCGen(const CompliantPCGen&);
    public:
        CompliantPCGen(PathArray& VAP);
        PathControl* GenPathC();
};

class AutomatedPCGen : public PathCGen
{
    private:
        SegmentControl& sc;
        void operator=(const AutomatedPCGen&);
        AutomatedPCGen(const AutomatedPCGen&);
    public:
        AutomatedPCGen(SegmentControl& SC);
        PathControl* GenPathC();
};

#endif
```

```
#ifndef _HIGHWAY_H_
#define _HIGHWAY_H_

#include "ValArray.h"

class Section;
class Domain;

class Highway
{
    private:
```

```
        void operator=(const Highway&);
    public:
        virtual ValArray<Section*>& GetSection() = 0;
        virtual void AddSection(Section&) = 0;
        virtual void SetParentDom(Domain*) = 0;
        virtual void SetId(int ID) = 0;
        virtual int GetId() = 0;
        virtual ~Highway();
};

class StdHighway : public Highway
{
    private:
        ValArray<Section *> vas;
        Domain* dom;
        int id;

        void operator=(const StdHighway&);
        StdHighway(const StdHighway&);
    public:
        StdHighway();
        ValArray<Section*>& GetSection();
        void AddSection(Section&);
        void SetParentDom(Domain*);
        void SetId(int ID);
        int GetId();
        ~StdHighway();
};

#endif
```

```
#ifndef _INTEGRATOR_H_
#define _INTEGRATOR_H_

#include <fstream.h>

class Domain;
class Vehicle;
class Lane;

class Integrator
{
    private:
```

```
        static Integrator* TheIntegrator;
        void operator=(const Integrator&);
    public:
        virtual void SetTimeStep(double) = 0;
        virtual void Integrate() = 0;
        virtual void PrintInfo() = 0;
        virtual void PrintLongPos() = 0;
        virtual void PrintLatPos() = 0;
        virtual void PrintLongVel() = 0;
        virtual void PrintLongAcc() = 0;
        virtual double GetTimeStep() = 0;
        /* Flush all the data in the output buffer to files */
        virtual void Flush() = 0;
        static Integrator* GetIntegrator();
};

struct OutFiles
{
    ofstream pos;
    ofstream pos_lat;
    ofstream vel;
    ofstream acc;

    OutFiles();
};

class StdIntegrator: public Integrator
{
    private:
        Domain& dom;
        double time_step;
        OutFiles of;

        enum{NEW_SECTION=1, EXIT};
        void operator=(const StdIntegrator&);
        StdIntegrator(const StdIntegrator&);
    public:
        StdIntegrator(Domain& D, double time_step );
        void SetTimeStep(double);
        void Integrate();
        void PrintInfo();
        void PrintLongPos();
        void PrintLatPos();
        void PrintLongVel();
```

```
        void PrintLongAcc();
        double GetTimeStep();
        void Flush();
};
#endif
```

```
#ifndef _LANE_H_
#define _LANE_H_

#include "DLinkedList.h"
#include "CarAcceptor.h"
#include "ValArray.h"

class CarUnit;
class Section;
class CarUnitDisposal;
class TripWire;

typedef  DLinkedList<CarUnit*> CUList ;
typedef ValArray<TripWire*> WireList;

class Lane: public CarUnitAcceptor
{
    private:
        void operator=(const Lane&);
    public:
        virtual void SetId(int ID) = 0;
        virtual int GetId() = 0;
        virtual CUList& GetCU() = 0;
        virtual void SetParentSec(Section*) = 0;
        virtual Section* GetParentSec() = 0;
        virtual void AddCU(CarUnit&) = 0;
        virtual CarUnit* RemoveCU() = 0;
        virtual void PluckCU(CarUnit* cu) = 0;
        virtual void InsertCU(CarUnit* cu) = 0;
        virtual void AddAfter(Lane&) = 0;
        virtual void AddBefore(Lane&) = 0;
        virtual Lane* NextLane() = 0;
        virtual Lane* PrevLane() = 0;
        virtual void SetNext(Lane*) = 0;
        virtual void SetPrev(Lane*) = 0;
        virtual void SetLatPos(double) = 0;
        virtual double GetLatPos() = 0;
```

```
        virtual double GetWidth() = 0;
        virtual double GetRefSpeed() = 0;
        virtual void SetRefSpeed(double speed) = 0;
        virtual void PrintLane() = 0;
        virtual CarUnitDisposal* GetCUD() = 0;
        virtual void CheckWire(CarUnit&) = 0;
        virtual void AddWire(TripWire*) = 0;
        // Inherit the requirement for a CarUnitAcceptor

        virtual ~Lane();
};

class StdLane : public Lane
{
    private:
        int id;
        CUList cul;
        double width; //Width of the lane
        double lat_pos; // Lateral position of the center of the lane
        Section* sec;
        CarUnitDisposal* cud;
        /* Lane connectivity */
        Lane* next;
        Lane* prev;
        /* Trip Wires */
        WireList wl;
        double ref_speed;

        void operator=(const StdLane&);
        StdLane(const StdLane&);
    public:
        StdLane(CarUnitDisposal* CUD = NULL);
        void SetId(int ID);
        int GetId();
        CUList& GetCU();
        void SetParentSec(Section*);
        Section* GetParentSec();
        void AddCU(CarUnit&);
        CarUnit* RemoveCU();
        void PluckCU(CarUnit* cu);
        void InsertCU(CarUnit* cu);
        void AddAfter(Lane&);
        void AddBefore(Lane&);
        Lane* NextLane();
```

```
        Lane* PrevLane();
        void SetNext(Lane*);
        void SetPrev(Lane*);
        void SetLatPos(double);
        double GetLatPos();
        double GetWidth();
        double GetRefSpeed();
        void SetRefSpeed(double speed);
        void PrintLane();
        CarUnitDisposal* GetCUD();
        /* Check the trip wires */
        void CheckWire(CarUnit&);
        void AddWire(TripWire*);
        /* CarUnitAcceptor requirements */
        void AcceptCU(CarUnit*);
        int CUAccId(); // Return the Section ID
        ~StdLane();
};

#endif
```

```
#ifndef _LANECHANGER_H_
#define _LANECHANGER_H_

class CarUnit;

class LaneChanger
{
    /* This class parallels the CarFollower class used for longitudinal
    control */
    private:
        void operator=(const LaneChanger&);
    public:
        virtual void Change(CarUnit* cu, int counter, int total_count,
             int direction) = 0;
        virtual int GetAbortFlag() = 0;
        virtual void SetAbortFlag(int AF) = 0;
        virtual void Abort(CarUnit* CU) = 0;
};

class StdLaneChanger : public LaneChanger
{
    protected:
```

```
        int abort_flag;

    public:
        enum{ ABORT=1, DONT_ABORT};
        int GetAbortFlag();
        void SetAbortFlag(int AF);
};

typedef struct SimpleLCDataStruct
{
    double d1;
    double d2;
}SimpleLCData;

class SimpleLaneC : public StdLaneChanger
{
    private:
        static SimpleLCData slcd;

        void operator=(const SimpleLaneC&);
        SimpleLaneC(const SimpleLaneC&);
    public:
        SimpleLaneC();
        void Change(CarUnit* cu, int counter,int total_count,int dir);
        void Abort(CarUnit* CU);
};

#endif
```

```
#ifndef _LATCEVENT_H_
#define _LATCEVENT_H_

#include "Event.h"
#include "DLinkedList.h"

class EventCal;
class CarUnit;

class LatCEvent : public Event
{
    private:
        CarUnit* cu;
        int direction;
```

```
        static DLinkedList<LatCEvent*> Pool;

        void DELETE(LatCEvent*);
        LatCEvent(int dir, CarUnit* = NULL);
        void operator=(const LatCEvent&);
        LatCEvent(const LatCEvent&);

    public:
        int Priority();
        void Action(EventCal&);

        /* Special functions */
        static LatCEvent* NEW(int dir, CarUnit* CU = NULL);
        void SetCU(CarUnit*);
        CarUnit* GetCU();
        void SetDir(int dir);
        ~LatCEvent();

        /* This was done to get rid of warnings by g++, because my
        constructor is private. I want the NEW method to be used instead
        of the private constructor */
        friend class DummyFriend;
};

class LaneChangeEv : public Event
{
    private:
        CarUnit* cu;
        int counter;
        int direction;
        static double change_time;
        static DLinkedList<LaneChangeEv*> Pool;

        void DELETE(LaneChangeEv*);
        LaneChangeEv(int dir, CarUnit* CU = NULL);
        void operator=(const LaneChangeEv&);
        LaneChangeEv(const LaneChangeEv&);

    public:
        int Priority();
        void Action(EventCal&);

        /* LaneChangeEv specific functions */
        static LaneChangeEv* NEW(int dir, CarUnit* CU = NULL);
```

```
        void SetCU(CarUnit*);
        CarUnit* GetCU();
        void SetDir(int dir);
        static double GetChangeTime();
        ~LaneChangeEv();

        /* This was done to get rid of warnings by g++, because my
        constructor is private. I want the NEW method to be used instead
        of the private constructor */
        friend class DummyFriend;
};

#endif
```

```
#ifndef _LATCONTROL_H_
#define _LATCONTROL_H_

/* This is the second major version of LatControl. I didn't use RCS to
perform this update, becuase I wanted to old file at hand to copy from */

#include <stdio.h>
#include "ValArray.h"
#include "DLinkedList.h"
#include "Vehicle.h"

class Vision;
class CarUnit;
class LaneChanger;
class Lane;
class LongControl;
class Platoon;
class CarFollower;

class LatControl
{
    private:
        void operator=(const LatControl&);
    public:
        virtual void Control(int dir) = 0;
        virtual void MonitorControl(CarFollower*, ValArray<CarUnit*>&,
             double ts) = 0;
        virtual void DecControl(CarFollower*, ValArray<CarUnit*>&,
             double ts) = 0;
```

```
        virtual void SideCarControl(CarFollower*, ValArray<CarUnit*>&,
             double ts) = 0;
        virtual LaneChanger* GetLaneC() = 0;
        virtual void SetCU(CarUnit*) = 0;
        virtual CarUnit* GetCU() = 0;
        virtual int GetChState() = 0;
        virtual void SetChState(int state) = 0;
        virtual void AddHold() = 0;
        virtual void ReduceHold() = 0;
        virtual int GetBusyState() = 0;
        virtual void SetBusyState(int) = 0;
        virtual int GetComState() = 0;
        virtual int GetDir() = 0;
        virtual void PostNotify() = 0;
        virtual void PreNotify(CarUnit*) = 0;
        virtual void PreNearNotify(CarUnit* rear, CarUnit* front) = 0;
        virtual void PostNearNotify() = 0;
        virtual void PostProc() = 0;
        virtual int IsSafe(CarUnit* side_car, double& diff) = 0;
        virtual void Schedule() = 0;
        virtual void RemoveSide(CarUnit* ) = 0;
        virtual void AddSide(CarUnit*) = 0;
        virtual Lane* GetToLane() = 0;
        virtual void SetToLane(Lane*) = 0;
        virtual void Abort() = 0;
        virtual void OffRoadAbort() = 0;
        virtual ~LatControl() {}
};

class StdLatControl : public LatControl
{
    private:
        void operator=(const StdLatControl&);
    protected:
        int com_state;
        int change_state;
        int busy_state;
        int change_dir;
        double safe_dist;
        int num_holds; // This is the number of holds placed on a vehicle
        CarUnit *cu;
        LaneChanger* lc;
        Lane* to_lane;
        DLinkedList<CarUnit*> notify_cars;
```

```
        DLinkedList<CarUnit*> side_cars;
        CarUnit* side_car_rear;
        CarUnit* side_car_front;

        virtual void Diff(CarUnit* side_car,double& diff_now,
             double& diff_later);
        virtual void Diff2(CarUnit* side_car,double& diff_now,
             double& diff_later);
        double DistAfterBrake(VehicleState& VS, VehicleParams& VP);
    public:
        enum {LEFT = 1, RIGHT, NO_DIRECTION};
        enum {ALLOWED_TO_CHANGE=1, NOT_ALLOWED};
        enum {BUSY=1, NOT_BUSY};
        enum {TO_CHANGE = 1, NOT_TO_CHANGE};
        virtual LaneChanger* GetLaneC();
        virtual void SetCU(CarUnit*);
        virtual CarUnit* GetCU();
        virtual int GetChState() ;
        virtual void SetChState(int state) ;
        virtual void AddHold();
        virtual void ReduceHold();
        virtual int GetBusyState();
        virtual void SetBusyState(int );
        virtual int GetComState();
        virtual int GetDir();
        virtual int IsSafe(CarUnit* side_car, double& diff);
        virtual void RemoveSide(CarUnit* ) ;
        virtual void AddSide(CarUnit*);
        virtual Lane* GetToLane();
        virtual void SetToLane(Lane*);
        virtual void Abort();
        virtual void OffRoadAbort();
};

class SimpleLatC : public StdLatControl
{
    private:
        void operator=(const SimpleLatC&);
        SimpleLatC(const SimpleLatC&);

        // Makes the car a free agent
        int CheckLongState(ValArray<CarUnit*>& CUA2);
        // Makes the car a leader
        int CheckLongState2(ValArray<CarUnit*>& CUA2);
```

```
    protected:
        double wait_time;
        virtual void Scenario2(CarUnit*, ValArray<CarUnit*> &CUA2);
        virtual void Scenario3(CarUnit*, ValArray<CarUnit*> &CUA2);
        virtual void Scenario7(Platoon* , CarUnit* cu1, CarUnit* cu2,
             ValArray<CarUnit*> &CUA2);
        virtual void Scenario8(CarUnit* cu_front, CarUnit* cu_rear,
             ValArray<CarUnit*> &CUA2);
        virtual void PostNearNotify();
        virtual int CheckFarLane(ValArray<CarUnit*>&CUA2);
        virtual int CheckNearLane(ValArray<CarUnit*>&CUA1);
        virtual void FarLaneControl(ValArray<CarUnit*> &CUA2);
        virtual void NearLaneControl(ValArray<CarUnit*> &CUA1,
             ValArray<CarUnit*> &CUA2);

    public:
        SimpleLatC(LaneChanger* LC = NULL);
        virtual void Control(int dir);
        virtual void PostNotify();
        virtual void PreNotify(CarUnit*);
        virtual void MonitorControl(CarFollower*, ValArray<CarUnit*>&,
             double ts);
        virtual void DecControl(CarFollower*, ValArray<CarUnit*>&,
             double ts);
        virtual void SideCarControl(CarFollower*, ValArray<CarUnit*>&,
             double ts);
        virtual void PostProc();
        virtual void PreNearNotify(CarUnit* rear, CarUnit* front);
        virtual void Schedule();
        ~SimpleLatC();
};

class QuickLatC : public SimpleLatC
{
    /* This is similar to SimpleLatC. The difference is that if the car
    that wants to change is part of a platoon, it doesn't physically
    split away from the cars in its platoon. However, the platoon itself
    does split up into three platoons, one for the cars in front, one
    for the car that wants to change, one for the cars behind. */
    private:
        void operator=(const QuickLatC &);
        QuickLatC(const QuickLatC &);

        // No need to become a leader or free-agent
```

```
        void PrepLong();

    public:
        QuickLatC(LaneChanger* LC = NULL);
        void Control(int dir);
        ~QuickLatC();
};

#endif
```

```
#ifndef _LINKEDLIST_H_
#define _LINKEDLIST_H_

#include <assert.h>
#include <iostream.h>

/* This file defines the classes used in the linked list */

template<class T> class LinkedList
{
    public:
        class Node
        {
            public:
             T data;
             Node* next;
        };

    protected:
        Node* head;
        int size;

    private:
        void operator=(const LinkedList<T>&);
        LinkedList(const LinkedList<T>&);
    public:
        LinkedList();
        LinkedList(T t);
        void AddToFront(T t);
        void Append(T t);
        // Insert T into appropriate position in list
        void Insert(T t, int pos);
        // Removes head and returns its data
```

```
        T PopHead();
        void PopHead(T&);
        int GetSize() const;
        //Returns a pointer to head
        Node* GetHead();
        void PrintList();
        ~LinkedList();
};

#endif
```

```
#ifndef _LONGCEVENT_H_
#define _LONGCEVENT_H_

#include "DLinkedList.h"
#include "Event.h"

class EventCal;
class CarUnit;
class PlatoonLC;
class LongControl;

class PltLCStateEv : public Event
{
    private:
        CarUnit* cu;
        static double update_time;
        static DLinkedList<PltLCStateEv*> Pool;

        void DELETE(PltLCStateEv*);
        PltLCStateEv(CarUnit* CU, double Update_Time = 1.);
        void operator=(const PltLCStateEv&);
        PltLCStateEv(const PltLCStateEv&);
    public:
        int Priority();
        void Action(EventCal&);
        /* Special functions */
        static PltLCStateEv* NEW(CarUnit* CU, double Update_Time = 1.);
        static void SetUpdateTime(double UT);
        void SetCU(CarUnit*);
        CarUnit* GetCU();
        ~PltLCStateEv();
```

```
        /* This was done to get rid of warnings by g++, because my
        constructor is private. I want the NEW method to be used instead
        of the private constructor */
        friend class DummyFriend;
};

class SetLongCStateEv : public Event
{
    private:
        LongControl* longc;
        int lc_state; // This is a platoonLC general state
        static DLinkedList<SetLongCStateEv*> Pool;

        void DELETE(SetLongCStateEv*);
        SetLongCStateEv(LongControl* LongC, int LCState);
        void operator=(const SetLongCStateEv&);
        SetLongCStateEv(const SetLongCStateEv&);
    public:
        void Action(EventCal&);
        int Priority() {return 1;}

        /* Special functions */
        static SetLongCStateEv* NEW(LongControl* LongC, int LCState);
        void SetLongC(LongControl*);
        LongControl* GetLongC();
        void SetLCState(int LCS);
        ~SetLongCStateEv();

        /* This was done to get rid of warnings by g++, because my
        constructor is private. I want the NEW method to be used instead
        of the private constructor */
        friend class DummyFriend;
};

#endif
```

```
#ifndef _LONGCONTROL_H_
#define _LONGCONTROL_H_

#include "ValArray.h"

class Vehicle;
class FollowTraj;
```

```
class CSCarFollower;
class CarFollower;
class Vision;
class CarUnit;
class Platoon;
class RandGen;

class LongControl
{
    private:
        operator=(const LongControl&);
    public:
        virtual void Control(Vision&, double time_step ) = 0;
        virtual void SetCU(CarUnit*) = 0;
        virtual CarUnit* GetCU() = 0;
        virtual int GetConState() = 0;
        virtual void SetConState(int ) = 0;
        virtual void FindConState() = 0;
        virtual int GetGenState() = 0;
        virtual void SetGenState(int) = 0;
        virtual void SetBusy(int) = 0;
        virtual double GetDesSpeed() = 0;
        virtual int GetPltSize() = 0;
        virtual void MergePlt(Platoon& ) = 0;
        virtual void SplitPlt(int posn, int PhysSplit = 1) = 0;
        virtual Platoon* GetPlt() = 0;
        virtual int GetPltPosn() = 0;
        virtual int GetMode() = 0;
        virtual void SetMode(int) = 0;
        virtual int ToPathControl() = 0;
        virtual ~LongControl() {}
};

class StdLongControl :public LongControl
{
    private:
        void operator=(const StdLongControl&);
    protected:
        CarUnit *cu;
        double time_step;
        int busy_state;
        int mode;

    public:
```

```
        enum{BUSY=1, NOT_BUSY}; // busy states
        enum{LONGC=1, LATC, LATC_DEC, LATC_MONITOR}; //control modes
        virtual void SetCU(CarUnit*);
        virtual CarUnit* GetCU();
        virtual void SetBusy(int);
        virtual int GetPltSize();
        virtual void MergePlt(Platoon& );
        virtual void SplitPlt(int posn, int PhysSplit = 1);
        virtual Platoon* GetPlt();
        virtual int GetPltPosn();
        virtual int GetGenState();
        virtual void SetGenState(int);
        virtual int GetConState();
        virtual void SetConState(int);
        virtual void FindConState();
        virtual int GetMode();
        virtual void SetMode(int);
        virtual int ToPathControl();
};

typedef ValArray<double> Trajectory;

class FollowTrajLC : public StdLongControl
{
    private:
        FollowTraj* ft;

        void operator=(const FollowTrajLC&);
        FollowTrajLC(const FollowTrajLC&);
    public:
        FollowTrajLC(ValArray<double>& A, FollowTraj* FT = NULL);
        void Control(Vision&, double ts);
        int ToPathControl() {return 0;}
        double GetDesSpeed();
        ~FollowTrajLC();
};

class CarSimLC : public StdLongControl
{
    private:
        CSCarFollower* cs;
        double des_speed;
        static RandGen rg;
```

```
        double GenSpeed(); // Generates a random speed
        void operator=(const CarSimLC&);
        CarSimLC(const CarSimLC&);

    public:
        CarSimLC(CSCarFollower* CS = NULL);
        void Control(Vision&, double ts);
        double GetDesSpeed();
        ~CarSimLC();
};

typedef struct PlatoonDataStruct
{
    int max_size;
    int posn;
    Platoon* plt;
}PlatoonData;

typedef struct PlatoonLCDataStruct
{
    int control_state;
    int general_state;

    double track_speed;
    double split_dist;
    double merge_dist_tol;
    double merge_vel_tol;
    double follow_dist;

} PlatoonLCData;

class PlatoonLC : public StdLongControl
{
    private:
        PlatoonData   pd;
        PlatoonLCData plcd;
        CarFollower* follower;
        CarFollower* merger;
        CarFollower* splitter;
        CarFollower* tracker;

        void StopMerging();
        void operator=(const PlatoonLC&);
        PlatoonLC(const PlatoonLC&);
```

```
    public:
        enum {FOLLOWER=1, SPLITTER, MERGER, TRACKER};
        enum {LEADER=1, FREE_AGENT, PRE_LANE_CHANGE, OTHER};
        PlatoonLC(CarFollower* Follower, CarFollower* Merger = NULL,
            CarFollower* Splitter = NULL);
        virtual void SetConState(int flag);
        int GetConState();
        int GetGenState();
        void SetGenState(int);
        void Control(Vision& VIS, double ts);
        int GetPltSize();
        void SetBusy(int flag);
        double GetDesSpeed();
        void MergePlt(Platoon&);
        void SplitPlt(int posn, int PhysSplit = 1);

        /* PlatoonLC specific functions */
        void SetPlt(Platoon* Plt);
        Platoon* GetPlt();
        void SetBusyState(int flag);
        void SetPltPosn(int Posn);
        int GetPltPosn();
        void SetPltLCData(PlatoonLCData& PLCD);
        void GetPltLCData(PlatoonLCData& PLCD);

        void FindConState();
        ~PlatoonLC();
};

#endif
```

```
#ifndef _PATHCEVENT_H_
#define _PATHCEVENT_H_

#include "DLinkedList.h"
#include "Event.h"
class EventCal;
class CarUnit;

class PathCEvent : public Event
{
    public:
```

```
        enum{RESCHEDULE=1, NO_RESCHEDULE};
    private:
        CarUnit* cu;
        static double updt_time;
        static DLinkedList<PathCEvent*> Pool;
        int repeater;

        void DELETE(PathCEvent*);
        PathCEvent(CarUnit*, int Repeat=RESCHEDULE);
        void operator=(const PathCEvent&);
        PathCEvent(const PathCEvent&);
    public:
        int Priority();
        void Action(EventCal&);
        /* Special functions */
        static void SetUpdateTime(double UT);
        static double GetUpdateTime();
        static PathCEvent* NEW(CarUnit*, int Repeat=RESCHEDULE);
        void SetRepeat(int Repeat);
        void SetCU(CarUnit*);
        CarUnit* GetCU();
        ~PathCEvent();

        /* This was done to get rid of warnings by g++, because my
        constructor is private. I want the NEW method to be used instead
        of the private constructor */
        friend class DummyFriend;
};

#endif
```

```
#ifndef _PATHCONTROL_H_
#define _PATHCONTROL_H_

#include <fstream.h>
#include <stdio.h>
#include "Vehicle.h"
#include "Vision.h"
#include "RandGen.h"

class SegmentControl;
class CarUnit;
class Highway;
```

```
class Section;
class Lane;

class PathControl
{
    public:
        virtual void Control() = 0;
        virtual void SetCU(CarUnit*) = 0;
        virtual CarUnit* GetCU() = 0;
        virtual void SetAckState(int STATE) = 0;
        virtual void SetPathCTime(double Ptime) = 0;
        virtual double GetPathCTime() = 0;
};

class StdPathControl : public PathControl
{
    protected:
        CarUnit* cu;
        int ack_state;
        double pathc_time;
        ofstream path_file;
        int missed_flag; // If 1, then you've missed your exit
        int orig_exit_sec;

        StdPathControl(CarUnit* CU);
        void Schedule(int dir);
        int FindNumCars(CUArray& CUA);
        int NeedToExit(int& exit_param);
    public:
        enum{ ACK=1, NACK};
        enum{ EXIT_NOW=1, EXIT_SOON, DONT_EXIT};
        virtual void SetCU(CarUnit*);
        virtual CarUnit* GetCU();
        virtual void SetAckState(int STATE);
        virtual void SetPathCTime(double Ptime);
        virtual double GetPathCTime();
};

typedef struct HumanPCDataStruct
{
    double tol_dist1;
    double tol_vel1;
    double tol_dist2;
    double tol_vel2;
```

```
    double prob1;
    double prob2;
} HumanPCData;

class HumanPC : public StdPathControl
{
    private:
        static HumanPCData hpcd;
        RandGen rg;

        int NeedToOvertake(double speed);
        int NeedToChangeRight(Vehicle* V,VehicleState& VS,int ex_param);
        int FindOvertakeDir(Vehicle* V, VehicleState& VS, int& dir);
        void ChangeToOvertake(VehicleState&, Vehicle*, int dir);
        void ChangeRight();
        void operator=(const HumanPC&);
        HumanPC(const HumanPC&);
    public:
        HumanPC(CarUnit* CU = NULL);
        void Control();
};

typedef struct PathTripleStruct
{
    Highway* high;
    Section* sec;
    Lane* lane;
} PathTriple;

typedef ValArray<PathTriple*> PathArray;

class SemiCompliantPC : public StdPathControl
{
    private:
        RandGen rg;

        void ChangeWithProb(int dir);
        void operator=(const SemiCompliantPC&);
        SemiCompliantPC(const SemiCompliantPC&);
    protected:
        PathArray& vap;

        virtual Lane* FindInPath(Highway* H, Section* S);
        virtual int FindDir(Lane* current, Lane* desired);
```

```
        virtual Lane* FindDesiredLane();
    public:
        SemiCompliantPC(PathArray& VAP, CarUnit* CU = NULL);
        virtual void Control();
        virtual PathArray& GetPathTriple();
        virtual void SetPathTriple(PathArray& VAP);
};

class CompliantPC : public SemiCompliantPC
{
    private:

        void operator=(const CompliantPC&);
        CompliantPC(const CompliantPC&);
    public:
        CompliantPC(PathArray& VAP, CarUnit* CU = NULL);
        void Control();
};

class AutomatedPC : public StdPathControl
{
    /*  This class obtains the lane assignment information from the
    segment controller */
    private:
        SegmentControl& sc;
        int ass_lane_id;

        void operator=(const AutomatedPC&);
        AutomatedPC(const AutomatedPC&);
    public:
        AutomatedPC(SegmentControl& SC, CarUnit* CU = NULL);
        void Control();
};

#endif
```

```
#ifndef _PHYSICAL_H_
#define _PHYSICAL_H_

#include "Event.h"
#include "EventCal.h"
#include "Integrator.h"
```

```
class PhysicalControl: public Event
{
    private:
        int print_pos, print_vel, print_acc, print_pos_lat;
        int time_int, counter;
        Integrator& integ;

        void operator=(const PhysicalControl&);
        PhysicalControl(const PhysicalControl&);
    public:
        /* The Pos, Vel and Acc control output of those quantities */
        PhysicalControl(Integrator&, int Pos = 0, int Vel = 0,
             int Acc = 0, int Pos_lat = 0);
        int Priority();
        void SetPrintParams(int Pos, int Vel, int Acc, int Pos_lat);
        void ResetTimeInt();
        void Action(EventCal&);

};

#endif
```

```
#ifndef _PLATOON_H_
#define _PLATOON_H_

#include "DLinkedList.h"
#include "LongControl.h"

typedef  DLinkedList<PlatoonLC*> PList ;
typedef DLinkedList<PlatoonLC*>::Node* CurPlt;

class Platoon
{
    private:
        PList pl;
        static DLinkedList<Platoon*> Pool;

        Platoon();
        void Pdelete(Platoon* );
        void operator=(const Platoon&);
        Platoon(const Platoon&);

    public:
```

```
        static Platoon* Pnew();
        void AddPLC(PlatoonLC*);
        PList& GetList();
        const PList& GetList() const;
        void Split(int posn, int PhysSplit = 1);
        void Merge(Platoon& Plt2);
        int Size();
        void SetBusy(int);
        void Print();
        ~Platoon();

        /* This has been put to avoid compiler warnings about the
        private constructor */
        friend class DummyFriend;
};

#endif
```

```
#ifndef _RANDGEN_H_
#define _RANDGEN_H_

#define STORE 10
#define SEEDS_DIR "/home/decision2/deepa/Seeds"

/* This class will be used when we want multiple random number streams */
extern "C" {
    char* initstate(unsigned seed, char* state, int n);
    char* setstate(char* state);
    long random();
    void srandom(int);
}

class RandGen
{
    private:
        long state[32];
        static int num_gen;
        static int seeds[STORE];
        static int file_zipped;
        static char fname[50];

        void FindFile();
    public:
```

```
        RandGen();
        double GetRand(); // Returns a random number between 0 and 1
        double GetNorm(double mean, double std_dev);
        ~RandGen();
};

#endif
```

```
#ifndef _SECTION_H_
#define _SECTION_H_

#include "ValArray.h"

class Lane;
class Highway;
class Vehicle;

typedef  ValArray<Vehicle*> Neighbour;

class Section
{
    private:
        void operator=(const Section&);
    public:
        virtual void SetId(int ID) = 0;
        virtual int GetId() = 0;
        virtual ValArray<Lane*>& GetLane() = 0;
        virtual void AddLane(Lane&) = 0;
        virtual void SetParentHigh(Highway* ) = 0;
        virtual Highway* GetParentHigh() = 0;
        virtual int LowestLaneId() = 0;
        virtual int HighestLaneId() = 0;
        virtual Lane* LaneInSec(int ID) = 0;
        virtual double GetLen() = 0;
        virtual void SetLen(double) = 0;
        virtual void SetLongPos(double) = 0;
        virtual double GetLongPos() = 0;
        virtual ~Section() {}
};

class StdSection : public Section
{
    /* This class models a section without an entrance or an exit */
```

```
    private:
        void operator=(const StdSection&);
        StdSection(const StdSection&);

    protected:
        int id;
        /* Each section has a ValArray of lanes*/
        ValArray<Lane*> val;
        Highway* high;
        double len;
        double long_pos; // Long. pos of the start of the section

    public:
        StdSection();
        void SetId(int ID);
        int GetId();
        ValArray<Lane*>& GetLane();
        virtual void AddLane(Lane&);
        void SetParentHigh(Highway*);
        Highway* GetParentHigh();
        virtual int LowestLaneId();
        virtual int HighestLaneId();
        virtual Lane* LaneInSec(int ID);
        static int LowestOvertakeLane();
        virtual double GetLen();
        virtual void SetLen(double);
        void SetLongPos(double);
        double GetLongPos();
        ~StdSection();
};

class TransitionSection : public StdSection
{
    /* This class models a section with an entrance or an exit */
    private:
        void operator=(const TransitionSection&);
        TransitionSection(const TransitionSection&);

    public:
        TransitionSection();
        void AddLane(Lane& L);
        int LowestLaneId();
        static int ExitLaneId();
        static int EntryLaneId();
```

```
        int HighestLaneId();
        Lane* LaneInSec(int ID);
};

#endif
```

```
#ifndef _SEGMENT_CONTROL_
#define _SEGMENT_CONTROL_

#include "ValArray.h"
#include "RandGen.h"
class CarUnit;

/* Each highway segment should have one segment controller */
class SegmentControl
{
    public:
        virtual int GetAssLaneId(CarUnit*) = 0;
};

typedef ValArray<double> Assignment;
typedef ValArray<Assignment*> RowAssignment;
typedef ValArray<RowAssignment*> ODAssignment;

class ConstLaneSC : public SegmentControl
{
    //Here we assume that the no. of lanes per section is a constant
    private:
        ODAssignment oda; // See constructor for details
        RandGen rg;

        void operator=(const ConstLaneSC&);
        ConstLaneSC(const ConstLaneSC&);
    public:
        ConstLaneSC();
        ODAssignment& GetODAssignment();
        int GetAssLaneId(CarUnit*);
};

typedef ValArray<int> Partition;
typedef ValArray<Partition*> PartArray;

class PartStratSC : public SegmentControl
```

```
{
    //Here we assume that the no. of lanes per section is a constant
    private:
        void operator=(const PartStratSC&);

    protected:
        PartArray pa; // See individual constructors for details on pa
    public:
        PartArray& GetPartitions();
};

/* Monotone strats may be described using either destination or
    origin monotone partitions */

class DestMonoSC : public PartStratSC
{
    /*  This is the class of destination monotone strategies */
    private:
        void operator=(const DestMonoSC&);
        DestMono(const DestMonoSC&);
    public:
        DestMonoSC();
        int GetAssLaneId(CarUnit*);
};

class OrigMonoSC : public PartStratSC
{
    /* This is the class of origin monotone strategies */
    private:
        void operator=(const OrigMonoSC&);
        OrigMonoSC(const OrigMonoSC&);
    public:
        OrigMonoSC();
        int GetAssLaneId(CarUnit*);
};

typedef ValArray<double> SplitPerc;
typedef ValArray<SplitPerc*> SplitArray;

class PartSplitSC : public PartStratSC
{
    protected:
        // See individual constructor for the meaning of this
        SplitArray sa;
```

```
        RandGen rg;
        int FindLane(int part, int index, int entry_num);
    public:
        SplitArray& GetSplitPerc();
};

class DestSplitSC : public PartSplitSC
{
    /* This is the class of destination monotone strategies
    with splitting */
    private:
        void operator=(const DestSplitSC&);
        DestSplit(const DestSplitSC&);
    public:
        DestSplitSC();
        int GetAssLaneId(CarUnit*);
};

class OrigSplitSC : public PartSplitSC
{
    /* This is the class of origin monotone strategies
    with splitting */
    private:
        void operator=(const OrigSplitSC&);
        OrigSplitSC(const OrigSplitSC&);
    public:
        OrigSplitSC();
        int GetAssLaneId(CarUnit*);
};

#endif
```

```
#ifndef _STATCEVENT_H_
#define _STATCEVENT_H_

#include "Event.h"

class EventCal;
class StatCollector;

class StatCEvent : public Event
{
    private:
```

```
        StatCollector* sc;
        double update_time;

        void operator=(const StatCEvent&);
        StatCEvent(const StatCEvent&);
    public:
        StatCEvent(StatCollector*, double UT);
        void Action(EventCal& );
        int Priority() { return 1;}
};

#endif
```

```
#ifndef _STAT_COLLECTOR_H_
#define _STAT_COLLECTOR_H_

#include <fstream.h>
#include "ValArray.h"
class Domain;
class Lane;
class FlowWire;

class StatCollector
{
    private:
        static StatCollector* TheStatCollector;
    public:
        virtual void CollectStats(double update_time) = 0;
        virtual void AddFlow(FlowWire*) = 0;
        static StatCollector* GetStatCollector();
        virtual void Flush() = 0;
        virtual ~StatCollector();
};

struct StatFiles
{
    ofstream speed;
    ofstream conc;
    ofstream flow;

    StatFiles();
};
```

```
class TrafStatCollector : public StatCollector
{
    /* This class collects statistics on speed and concentration */
    private:
        Domain& dom;
        StatFiles sf;
        ValArray<FlowWire*> vaf;

        void PrintAvgSpeed(int hwynum, int secnum, Lane* L);
        void PrintAvgConc(int hwynum, int secnum, Lane* L);
        void operator=(const TrafStatCollector&);
        TrafStatCollector(const TrafStatCollector&);
    public:
        TrafStatCollector(Domain& D);
        void CollectStats(double update_time);
        void AddFlow(FlowWire*);
        void Flush();
        ~TrafStatCollector();
};

#endif
```

```
#ifndef _TRIPWIRE_H_
#define _TRIPWIRE_H_

class Lane;
class CarUnit;
class StatCollector;

class TripWire
{
    private:
        void operator=(const TripWire&);
    public:
        virtual int CheckTrip( CarUnit&, Lane*) = 0;
};

class EOLWire : public TripWire
{
    private:
        void operator=(const EOLWire&);
        EOLWire(const EOLWire&);
    public:
```

```
        EOLWire();
        int CheckTrip(CarUnit&, Lane*);
};

class AEWire : public TripWire
{
    private:
        void operator=(const AEWire&);
        AEWire(const AEWire&);
    public:
        AEWire();
        int CheckTrip(CarUnit&, Lane*);
};

class FlowWire : public TripWire
{
    /* This wire is used to calculate the flow past a point */
    private:
        double wire_pos_on_lane;
        int total_num;
        int num_since_last_reset;
        Lane* lane;

        void operator=(const FlowWire&);
        FlowWire(const FlowWire&);
    public:
        FlowWire(double Pos, StatCollector& SC);
        int CheckTrip(CarUnit&, Lane*);
        void ResetCount();
        int GetCount();
        int GetTotalCount();
        void AddToLane(Lane*);
        Lane* GetLane();
};

#endif
```

```
#ifndef    _VALARRAY_H_
#define    _VALARRAY_H_

#include    <stdio.h>
#include    <iostream.h>
#include    <assert.h>
```

```
#include    "Collection.h"

#define    ValArray mtValArray

template<class T>
class    ValArray : public Collection<T>
{
    private:
        T    *array;
        T    fill;
        int    size;
        int    allocated_size;
        int    block_size;
        ValArray(const ValArray<T> &);
        void operator=(const ValArray<T> &);
    public:
        ValArray(int bs=10, T f=NULL);
        // Collection interface
        int    Size() const;
        void Set(int index, T val);
        T Get(int index) const;
        // ValArray specific interface
        const T& GetRef(int index) const;
        T& GetRef(int index);
        void Resize(int ns);
        void Append(T val);
        void Delete(int index);
        void SetFill(T f);
        void SetBlockSize(int bs);
        T *DirectAccess();
        void Print();
        ~ValArray();
};

template<class T>
class    PtrArray : public ValArray<T *>
{
    private:
        PtrArray(const PtrArray<T> &);
        void operator=(const PtrArray<T> &);
    public:
        PtrArray(int bs=10, T *f=NULL);
};
```

```
template<class T>
class    VAIter
{
    private:
        ValArray<T>    &arr;
        int    cur;
    public:
        VAIter(ValArray<T> &va);
        void start();
        int done();
        void next();
        T val();
        T& operator=(const T &t);
};

#endif
```

```
#ifndef _VEHICLE_H_
#define _VEHICLE_H_

#include <iostream.h>

class CarUnit;

typedef struct VehicleParamsStruct
{
    double mass;
    double length;
    double starting_acc;
    double max_dec;
    double max_speed;
} VehicleParams;

typedef struct ActuatorStateStruct
{
    double des_jerk;
    double des_acc;
    double des_steer;
} ActuatorState;

typedef struct VehicleStateStruct
{
    double lat_pos;
```

```
    double lat_speed;
    double lat_acc;

    double long_pos;
    double long_speed; // This is the longitudinal speed
    double long_acc;
    double jerk;

    CarUnit* cu;

    double pos_on_lane; // This is the longitudinal position on the lane
    double lat_pos_lane; // This is the lateral position OF the lane
} VehicleState;

ostream& operator <<(ostream&, VehicleState&);

class Vehicle
{
    private:
        void operator=(const Vehicle&);
    public:
        virtual void SetPosLane(double) = 0;
        virtual void SetVel(double) = 0;
        virtual void SetJerk(double jerk_input,double steer_input=0.)=0;
        virtual double GetDesJerk() = 0;
        virtual double GetMaxDec() = 0;
        virtual double GetLongPos() = 0;
        virtual void SetLongPos(double) = 0;
        virtual double GetPosOnLane() = 0;
        virtual void UpdatePos(double time_step) = 0;
        virtual void SetLatPos(double pos) = 0;
        virtual void SetLatPosLane(double ) = 0;
        virtual void GetState(VehicleState& ) = 0;
        virtual void SetState(VehicleState& ) = 0;
        virtual void GetParams(VehicleParams& ) = 0;
        virtual void SetCU(CarUnit*) = 0;
        virtual CarUnit* GetCU() = 0;
        virtual void PrintState() = 0;
        virtual void PrintImpState() = 0;
        virtual int EndLane(double*) = 0;
        virtual int AtExit() = 0;
        virtual void CarFollow(double time_step) = 0;
        friend ostream& operator <<(ostream&, VehicleState&);
        friend ostream& operator <<(ostream&, ActuatorState&);
```

```
};

class PMVehicle: public Vehicle
{
    private:
        VehicleParams vp;
        VehicleState vs;
        ActuatorState as;

        void operator=(const PMVehicle&);
        PMVehicle(const PMVehicle&);
    public:
        PMVehicle();
        PMVehicle(VehicleParams&, VehicleState&, ActuatorState&);
        void SetPosLane(double pos);
        void SetVel(double vel);
        /* The steer input is in radians */
        void SetJerk(double jerk_input, double steer_input = 0.);
        double GetDesJerk();
        double GetMaxDec();
        double GetLongPos();
        void SetLongPos(double);
        double GetPosOnLane();
        void UpdatePos(double time_step);
        void SetLatPos(double pos);
        void SetLatPosLane(double pos);
        void GetState(VehicleState& );
        void SetState(VehicleState& );
        void GetParams(VehicleParams&);
        void SetCU(CarUnit*);
        CarUnit* GetCU();
        void PrintState();
        void PrintImpState();
        // Detects if the vehicle has reached lane's end
        int EndLane(double*);
        int AtExit();
        void CarFollow(double time_step);
};
#endif
```

```
#ifndef _VISION_H_
#define _VISION_H_
```

```
#include "ValArray.h"
#include "Lane.h"

class CarUnit;

typedef ValArray<CarUnit*> CUArray;

class Vision
{
    private:
        void operator=(const Vision&);
    public:
        virtual void SetCU(CarUnit*) = 0;
        virtual CarUnit* GetCU() = 0;
        virtual void SetRange(double Range) = 0;
        virtual double GetConRange() = 0;
        virtual double GetNccRange() = 0;
        virtual void FrontNbor(CUArray&, int LA =1 ) = 0;
        virtual void SideNbor(CUArray&,CUArray&,int dir,Lane**)=0;
        virtual void SideNearNbor(CUArray&, Lane* to_lane) = 0;
        virtual void SideNearNbor(CUArray&,Lane* to_lane,
             double NRange)=0;
        virtual void SideNearNbor(CUArray&, int direction) = 0;
        virtual int ExistsLeftLane() = 0;
};

class StdVision : public Vision
{
    private:
        void operator=(const StdVision&);
    protected:
        CarUnit* cu;
        double range; // This is the range used by WithinRange()
        double control_range; // This is a range used for control
        // This is a range used for ensuring non collision
        double ncc_range;
        double side_range1;
        double side_range2;

        virtual int WithinRange(CarUnit* cu1, CarUnit* cu2);
        virtual void Flank2(CUArray& CUA, Lane* L, double side_range);
        virtual void Flank(CUArray& CUA, Lane* L, double side_range);
        virtual void CheckSideRangeOld(CarUnit* cu1, CarUnit* cu2,
             int& in_range, double& diff, double side_range);
```

```
        virtual int CheckSideRange(CarUnit* cu1, CarUnit* cu2,
             double side_range);
        virtual void FindCarsFront(CUArray& CUA,CUList& list,
             double range);
        virtual void FindCarsRear(CUArray& CUA,CUList& list,
             double range);
    public:
        virtual void SetCU(CarUnit*);
        virtual CarUnit* GetCU();
        virtual void SetRange(double Range);
        virtual double GetConRange();
        virtual double GetNccRange();
        virtual void SideNbor(CUArray&, CUArray&,int direction,
             Lane** to_lane);
        virtual void SideNearNbor(CUArray&, Lane* to_lane);
        virtual void SideNearNbor(CUArray&,Lane* to_lane,double NRange);
        virtual void SideNearNbor(CUArray&, int direction);
        virtual int ExistsLeftLane();
};

class HumanVision : public StdVision
{
    private:
        void operator=(const HumanVision&);
        HumanVision(const HumanVision&);
    public:
        HumanVision(double ConRange = 15, double NccRange = 130);
        void FrontNbor(CUArray& , int LA = 1);
};

class CPVision : public StdVision
{
    private:
        void operator=(const CPVision&);
        CPVision(const CPVision&);
    public:
        CPVision(double ConRange = 15, double NccRange = 130);
        void FrontNbor(CUArray&, int LA = 1);
};

class DPVision : public StdVision
{
    private:
        void operator=(const DPVision&);
```

```
        DPVision(const DPVision&);
    public:
        DPVision(double ConRange = 15, double NccRange = 130);
        /* Here LA+1 will be taken to mean the size of the platoon */
        void FrontNbor(CUArray&, int LA=1);
};

#endif
```

APPENDIX D

THE .h FILES IN THE FRONT END

```
#ifndef    _SIM_CREATE_H_
#define    _SIM_CREATE_H_
#include <tcl.h>

// These prototypes are used by Sim_AppInit

class CarUnitGen;
class Lane;

class SimCreate
{
    private:
        /* Car generation events */
        static int add_per_cgevent(Tcl_Interp* interp, int argc,
            char* argv[], CarUnitGen *cg);
        static int add_one_cgevent(Tcl_Interp* interp, int argc,
            char* argv[], CarUnitGen *cg);
        static int add_unif_cgevent(Tcl_Interp* interp,int argc,
            char* argv[], CarUnitGen *cg);
        static int add_exp_cgevent(Tcl_Interp* interp, int argc,
            char* argv[], CarUnitGen *cg);
        static int add_norm_cgevent(Tcl_Interp* interp,int argc,
            char* argv[], CarUnitGen *cg);

        /* Car Followers */
        static int add_follow_traj(Tcl_Interp* interp,int argc,
            char* argv[]);
        static int add_cs_car_follower(Tcl_Interp* interp,int argc,
            char* argv[]);
```

```
static int add_aicc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_cp(Tcl_Interp* interp, int argc, char* argv[]);
static int add_dp(Tcl_Interp* interp, int argc, char* argv[]);

/* Long controllers */
static int add_ftlc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_cslc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_plc(Tcl_Interp* interp, int argc, char* argv[]);

/* Lateral controllers */
static int add_simple_latc(Tcl_Interp* interp, int argc,
    char* argv[]);
static int add_quick_latc(Tcl_Interp* interp, int argc,
    char* argv[]);

/* Path controllers */
static int add_human_pc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_sc_pc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_c_pc(Tcl_Interp* interp, int argc, char* argv[]);
static int add_auto_pc(Tcl_Interp* interp, int argc, char* argv[]);

/* Segment controllers */
static int add_const_lane_segc(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_dest_mono_segc(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_orig_mono_segc(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_dest_split_segc(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_orig_split_segc(Tcl_Interp* interp, int argc,
     char* argv[]);

/* Car Follower generators */
static int add_follow_traj_gen(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_cs_car_follower_gen(Tcl_Interp* interp, int argc,
     char* argv[]);
static int add_aicc_gen(Tcl_Interp* interp, int argc, char* argv[]);
static int add_cp_gen(Tcl_Interp* interp, int argc, char* argv[]);
static int add_dp_gen(Tcl_Interp* interp, int argc, char* argv[]);

/* Long. control generators */
static int add_ftlc_gen(Tcl_Interp* interp, int argc, char* argv[]);
```

```
    static int add_cslc_gen(Tcl_Interp* interp, int argc, char* argv[]);
    static int add_plc_gen(Tcl_Interp* interp, int argc, char* argv[]);

    /* Lat. control generators */
    static int add_simple_latc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);
    static int add_quick_latc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);

    /* Path control generators */
    static int add_human_pc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);
    static int add_sc_pc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);
    static int add_c_pc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);
    static int add_auto_pc_gen(Tcl_Interp* interp, int argc,
        char* argv[]);

    /* Car generators */
    static int add_std_cargen(Tcl_Interp* interp, int argc,
        char* argv[], Lane*);
    static int add_special_cargen(Tcl_Interp* interp, int argc,
         char* argv[], Lane*);

public:

    // Commands that create things to do with the highway
    static int add_hwy(ClientData cd, Tcl_Interp *interp,
         int argc, char *argv[]);
    static int add_sec(ClientData cd, Tcl_Interp *interp,
         int argc, char *argv[]);
    static int add_lane(ClientData cd, Tcl_Interp *interp,
          int argc, char *argv[]);

    // Events
    static int cargen_event(ClientData cd, Tcl_Interp *interp,
          int argc, char *argv[]);
    static int lat_event(ClientData cd, Tcl_Interp *interp,
          int argc, char *argv[]);

    // Core car unit components
    static int add_vehicle(ClientData cd, Tcl_Interp *interp,
          int argc, char *argv[]);
```

```
        static int add_cf(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_long_control(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_lat_control(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_path_control(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);

        // Other car unit components
        static int add_segc(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);

        // Various generators
        static int add_veh_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_cf_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_longc_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_lanec_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_latc_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_pathc_gen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_cargen(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);

        // Statistics collection
        static int add_stat_event(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_flow_wire(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);

        // Miscellaneous structures
        static int add_traj(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_path_array(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
};

#endif
```

```
#ifndef    _SIM_DATA_H_
#define    _SIM_DATA_H_

#include "sim_table.h"
#include "inc_data.h"

class SimData
{
    public:
        // Things that there are only one of
        static Domain        *domain;
        static CarUnitDisposal      *CD;
        static Integrator        *integ;
        static EventCal      *EC;
        static PhysicalControl      *PC;
        static StatCollector      *StC;

        // Highway information
        static SimHashTable<Highway> highways;

        // Core car unit stuff
        static SimHashTable<Vehicle> vehicles;
        static SimHashTable<CarFollower> car_fol;
        static SimHashTable<LongControl> longc;
        static SimHashTable<LatControl> latc;
        static SimHashTable<PathControl> pathc;

        // Information needed by core car unit
         static SimHashTable<SegmentControl> segc;

        // Various generators
        static SimHashTable<VehicleGen> veh_gen;
        static SimHashTable<CarFollowerGen> cf_gen;
        static SimHashTable<LongCGen> longc_gen;
        static SimHashTable<LaneChGen> lanec_gen;
        static SimHashTable<LatCGen> latc_gen;
        static SimHashTable<PathCGen> pathc_gen;
        static SimHashTable<CarUnitGen>  cargens;

        // Miscellaneous structures
        static SimHashTable<Trajectory> traj;
        static SimHashTable<PathArray> path_array;
};
```

```
#endif
```

```
#ifndef _SIM_DO_H_
#define _SIM_DO_H_

#include <tcl.h>

class SimDo
{
    private:
        static int started_simulation;

    public:
        // commands that do stuff
        static int integrate(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int generate_car(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int run(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
};

#endif
```

```
#ifndef    _SIM_INFO_H_
#define    _SIM_INFO_H_
#include <tcl.h>


class Info
{
    private:
        static int info_domain(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_highway(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_section(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_lane(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_traj(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_pa(Tcl_Interp* interp, int argc, char* argv[]);
        static int info_carunit(Tcl_Interp* interp, int argc, char* argv[]);
```

```
    public:
        static int sim_info(ClientData cd, Tcl_Interp *interp, int argc,
            char *argv[]);
};



#endif
```

```
#ifndef _SIMSET_H_
#define _SIMSET_H_

#include <tcl.h>

class SimSet
{
    public:
        static int connect(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int start_physical_at(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int integrator_step(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int set_trace(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int set_cargen_exits(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_to_traj(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int add_to_pa(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int states_car(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int print_params(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int update_times(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int max_to_generate(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
        static int stop_sim_at(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
};
```

```
#endif
```

```
#ifndef _SIM_TABLE_H_
#define _SIM_TABLE_H_

#include <string.h>
#include    <tcl.h>

// This structure will contain some utility functions and global
// variables specific to this application.  Because everything will
// be a static member/method, we don't have to worry about
// collision with other packages (e.g. TCL/Tk)

template<class T> class SimHashTable
{
    protected:
        Tcl_HashTable    table;
        Tcl_HashEntry    *current;
        Tcl_HashSearch    search;
    public:
        SimHashTable();
        void GotoStart();
        int AtEnd();
        int Next();
        Tcl_HashEntry *Current();
        const char *CurKey();
        const char* GetKey(T* obj);
        T *CurVal();
        T* Get(const char *name);
        int FindObj(Tcl_Interp* interp,char* name,T** obj,char* descrip);
        int FindKey(Tcl_Interp* interp,T* obj,char** name,char* descrip);
        int Add(T *obj, const char *name);
        void Clean();
        ~SimHashTable();
};

#endif
```

```
#ifndef _SIM_UTIL_H_
#define _SIM_UTIL_H_

#include <tcl.h>
```

```
class Highway;
class Section;
class Lane;
class CarUnit;

#define CMD_CHECK(sc,str,num) (c==sc) && (strncmp(argv[num],str,length)==0)

class Util
{
    public:
        static int check_carnum(Tcl_Interp* interp, char* hwyname,
            char* secnum, char* lanenum, char* carnum, CarUnit** carp);
        static int check_lanenum(Tcl_Interp* interp, char* hwyname,
            char* secnum, char* lanenum, Lane** lanep);
        static int check_secnum(Tcl_Interp* interp, char* hwyname,
            char* secnum, Section** secp);
        static int check_hwyname(Tcl_Interp* interp, char* hwyname,
             Highway** highp);
        static int check_int(Tcl_Interp* interp,char* str, int* intPtr);
        static int check_double(Tcl_Interp* interp, char* str,
             double* doublePtr) ;
        static int clear(ClientData cd, Tcl_Interp *interp,
              int argc, char *argv[]);
};


#endif
```

VITA

Deepa Ramaswamy was born in Secunderabad, India, on the 21st of October, 1968. She received a B.Tech degree in Electrical Engineering from the Indian Institute of Technology, Madras, in 1989 and was awarded the Siemens Prize for having the highest GPA in her class. She received an M.S. degree in Electrical Engineering (Control systems) from the University of Illinois at Urbana-Champaign in 1992. She is a student member of the IEEE. She plans to take a position at Ford Motor Company as a product development engineer after graduation. Her interests include intelligent transportation systems, control systems, object-oriented design and simulation.

Publications:

- D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Lane assignment on an automated highway," in *Proceedings of the 1994 American Control Conference* (Baltimore), pp. 413-417, Jun. 1994.
- D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Partitioned strategies for lane assignment in multi-lane AHS," in *Proceedings of the 33rd IEEE Conference of Decision and Control* (Lake Buena Vista), pp. 2938-2943, Dec 1994.
- J. Medanic, D. Ramaswamy, W. R. Perkins, and R. Benekohal, "Partitioned lane assignment strategies for balancing excess lane capacity on AHS," in *Proceedings of the 1995 American Control Conference*, (Seattle), pp. 3581-3585, June 1995.
- D. Ramaswamy, J. Medanic, W. R. Perkins and R. Benekohal, "Combining lane assignment with route guidance in Corridor systems", *To be presented at the 34th IEEE Conference of Decision and Control* (New Orleans), Dec. 1995.
- D. Ramaswamy, J. Medanic, W. R. Perkins, and R. Benekohal, "Lane assignment on automated highway systems," accepted for publication in *IEEE Transactions on Vehicular Technology.*

• S. H. Lu, D. Ramaswamy, and P. R. Kumar, "Efficient scheduling policies to reduce mean and variance of cycle-time in semiconductor manufacturing plants," in *IEEE Transactions on Semiconductor Manufacturing*, pp. 374-388, Aug. 1994.