MOU 130: Feasibility study of fully autonomous vehicles using decision-theoretic control
Final Report

Jeffrey Forbes, Nikunj Oza, Ronald Parr, Stuart Russell
Computer Science Division
University of California, Berkeley

Abstract

This project investigated the feasibility of constructing an autonomous vehicle controller based on probabilistic inference and utility maximization. We believed, and still believe, that such methods are required for autonomous operation in unconstrained mixed human/automated traffic.

The project has been successful in identifying the major technical issues to be resolved. Contrary to our expectations, several significant theoretical and algorithmic advances were required in order to create an inference system capable of handling vehicle monitoring in a real-time fashion. New methods were also developed for learning probabilistic models from data, and for learning control policies given reward/penalty feedback.

Our experience in hand-coding controllers for high-level control suggests that it will be quite easy to attain a fairly high level of performance, equivalent to perhaps safe operation over a 40-mile trip. However, we require safe operation over, say, 100,000 miles or more. We believe that testing and refinement against an incident scenario library, combined with reinforcement learning techniques for controller optimization, will enable us to reach this goal; however, we do not yet have a mechanism for verifying that the goal has been reached, since each test would require about three months of simulator time.

Thus, we believe on the basis of our experience in this project that the *high level* driving problem can be handled satisfactorily, although much remains to be done in terms of real-time operation.

Contents

1 Introduction

Previous research into autonomous vehicles has concentrated on low-level sensing and control, and has achieved a reasonable degree of success in providing basic driving capabilities (following a lane, keeping a safe distance, tracking other vehicles). At the same time, technology for real-time Bayesian decision-making under uncertainty has developed to the point where it is reasonable to consider the integration of high-level and low-level systems. Our objective in this project was to evaluate the applicability and potential impact of real-time decision-making technology to the problem of guiding a vehicle in unrestricted traffic, including the development and demonstration of prototypes in simulation. Although it is a difficult technical problem, autonomous operation in unrestricted traffic has the potential to provide a feasible, "non-invasive" migration path towards high-throughput automated traffic with no modification of existing infrastructure. It can also provide "on-demand", low-overhead bus, taxi and rental-car operation, an efficient local feeder system for mass transit, and greater transportation access for a wide segment of the population. Robust, rigorous techniques for real-time sensing and decision-making under uncertainty can also be expected to contribute to the development of accurate driver-behaviour simulators, situation surveillance methods, probabilistic data fusion, and sensor failure detection and recovery.

Our general approach to the driving problem began from the position that driving in real traffic involves making complex decisions under uncertainty. Because of complexity, uncertainty, and the discrete nature of the high-level problem, traditional control solutions are not appropriate. Predictions of traffic behaviour, for example, rely on high-level analysis of the intentions of other drivers, which may depend in complex ways on current road conditions.

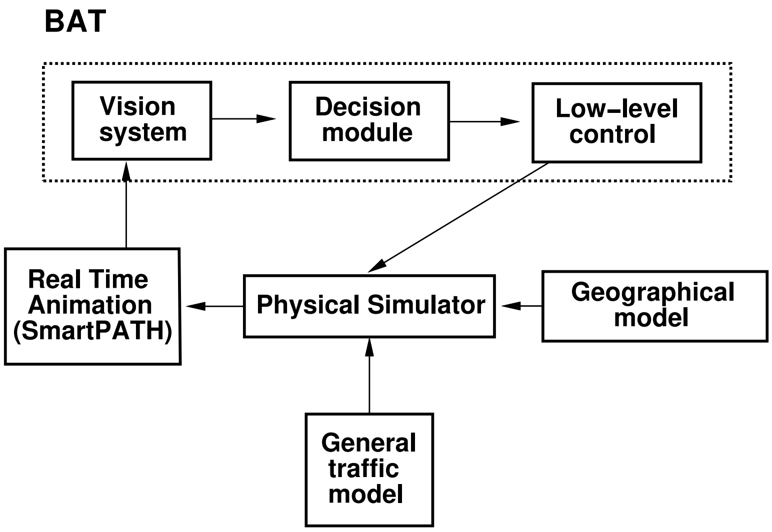

Figure 1: Overall design of development system

Driving involves tradeoffs among multiple goals. We cannot, for example, maximize safety, since that would require remaining stationary in a parking garage. Normal driving behaviour involves risks: passing a slow-moving truck on the freeway involves assuming it will not decide to push one across the central divider. Optimal driving therefore involves maximizing expected utility with a multiattribute utility function subject to significant uncertainty as to the possible outcomes of the available actions. To approximate this, we developed a Bayesian decision-making architecture (the *decision module* or *controller*); it controls a vehicle we call the BAT (Bayesian Automated Taxi). In the course of the project, we developed a variety of BAT controllers, testing them in a microscopic freeway traffic simulator with realistic vehicle dynamics that we constructed for the purpose. Figure 1 shows the basic structure of the development environment.

The following items describe the original research plan (in italics) together with a brief summary of results and deviations from plan.

1. *Interface an independent vehicle control module to SmartPATH simulator.*
   The SmartPATH system available at the beginning of the work turned out to be optimized for other tasks and unsuitable for our needs—for example, lateral motion was not represented and the simulator did not allow a clean separation between vehicle dynamics and controller decisions. We therefore developed out own simulator system, described in Section 2, providing modular sensor interfaces between simulator and controllers.

2. *Identify appropriate discrete descriptors for current traffic situation; define levels of simulator "realism" and plan migration path towards realistic visual input.*
   An interface between the vision subsystem and the controller was defined in which the vision system provides positions and velocities of all visible vehicles (as might be obtained from a Kalman filter tracking system). A set of discrete variables was defined for describing vehicle behaviours [Huang *et al.*, 1994].

3. *Develop sensing architecture , incorporating multiple sensors and Bayesian data fusion using the HUGIN probabilistic inference system.*
   A sensing architecture was developed and demonstrated using dynamic probabilistic networks (DPNs) [Huang *et al.*, 1994; Forbes *et al.*, 1995c]. The HUGIN inference package turned out to be inadequate for real-time inference. We developed and implemented new probabilistic inference algorithms for DPNs (Section 3.4 and [Kanazawa *et al.*, 1995b]).

4. *Demonstrate data fusion, sensor failure detection, graceful degradation, integration with low-level vision component.*
   The sensing architecture was shown to be capable of sensor fusion over time, and also capable of automated sensor failure detection based on deviation from predicted sensor inputs. We designed a closed-loop integration with the SmartPATH rendering system, which was modified to generate stereo image pairs for input to the stereo vision system designed under MOU 257. As yet, the stereo vision system tracks only lane markers and not vehicles, so we have been unable to conduct any experiments with real vision sensing.

5. *Encode existing driver behaviour models to allow prediction of behaviour of other vehicles. Carry out statistical analysis of real video data libraries to estimate the associated probability parameters.*
   Existing driver behaviour models turned out to be completely inadequate, and we were forced to develop models from scratch. We developed DPN models by hand, and also developed and implemented an automated DPN learning algorithm [Russell *et al.*, 1995] to learn models from observed behaviours. Experiments with this learning system are ongoing (Section 4) using simulated vision data from our traffic simulator. A real vision capability has been developed as part of the Roadwatch project, but we are still in the process of accumulating the necessary digitized sequences.

6. *Implement the real-time Bayesian control system, including nominal-plan-following, emergency reflexes, local lookahead lane selection.*
   A Bayesian controller was implemented using DPN inference combined with a decision tree representation of the control policy, where the variables tested in the decision tree are marginal probabilities on hidden state variables computed by the DPN (Section 5.1). A nominal plan-following capability was implemented, specifying target lanes for the vehicle at each point on its journey. Both emergency response and lane selection were performed by the decision tree controller; lookahead decision making is computationally intractable given the present state of the technology. We developed two low-level

controllers for handling steering and acceleration. One was developed using hill-climbing in parameter space to optimize a PD controller (Section 5.2.2); the other used reinforcement learning to train a neural net controller (Section 5.2.3). Both were able to follow and change lanes accurately and safely over a wide range of speeds and curvatures.

7. *Demonstrate control system operation driving in simulated passive traffic.*
   The first demonstrations took place in 1994 [Forbes *et al.*, 1995c]. Since then, we have made steady progress on both low-level and high-level control, yielding good driving behaviour even in dense, active traffic (Section 6).

8. *Incorporate route-planning algorithms with interface to local (lane-change, turn) decision-making.*
   Route following, with interface to local decisions, has been implemented. Route planning has not been implemented "onboard," but several algorithms have been implemented separately.

9. *Define and study feasibility and safety testing methodologies for deployment, including possible use or development of accident-scenario libraries and failure models.*
   The principal difficulties with simulator testing are 1) lack of realism of sensory input; 2) lack of realism in behaviour of other vehicles; 3) very long intervals between incidents as driving quality improves. Item 1 will remain a problem until the stereo vision system becomes usable. The basic methodology we have adopted to handle item 2 is to bootstrap—as the BAT controller is improved, the "drone" vehicles constituting the bulk of the traffic adopt the BAT control strategies; then the BAT improvement cycle continues. To deal with item 3, we have developed a scenario library of those occasions in which the BAT crashes, loses control, leaves the roadway, or engages in undue lateral or longitudinal acceleration or deceleration. These incidents are automatically detected and stored for later replay as part of BAT testing.

Overall, the BAT project has encountered a number of unexpected technical challenges, most of which have been addressed through the developed of new inference, learning, and control algorithms. The fundamental hypotheses of the BAT project have been qualitatively confirmed: that assessment of probabilities and utilities is a promising method for handling the driving problem; that it is necessary to infer information about the intentions of other vehicles in order to drive safely; and that simulator testing is an appropriate development route for the first phase of autonomous vehicle research.

2 The BAT Simulator

2.1 Overall Design

In developing the BAT simulator, we have tried to make it as flexible and extensible as possible. At the same time, our design reflects our focus on developing controllers for autonomous vehicles that operate in unrestricted traffic. Given this emphasis, we developed the following requirements:

- In general, traffic scenarios for testing controllers should resemble real-world traffic patterns. This means that the user should be able to specify a highway topology as well as probability distributions for generating a range of traffic from free-flowing to congested levels, plus accidents, stalls, and other unexpected events. It also means that the traffic should consist of various types of vehicles, each with parameterizable behaviors.

- Since there are many kinds of cars and since they operate in a wide range of physical conditions, e.g., clear vs. rainy weather, the simulation should allow controllers to be tested with different vehicle dynamics and models of the physical environment.

- The simulation should allow sensor models with varying levels of detail. These can be models of real-world sensors or of abstract sensors. For example, a simple, noisy radar sensor could be compared with a hypothetical sensing system that returns the positions and velocities of all neighboring vehicles without measurement noise or bias.
- Controller development should be expedited with a facility for detecting and recording traffic incidents. As a controller is improved, its effectiveness can be assessed by observing its performance in previously saved traffic scenarios.
- The software for the simulator itself should be easy to manage. As features are added and as different controllers are developed, the impact of such improvements on the system be as localized as possible.

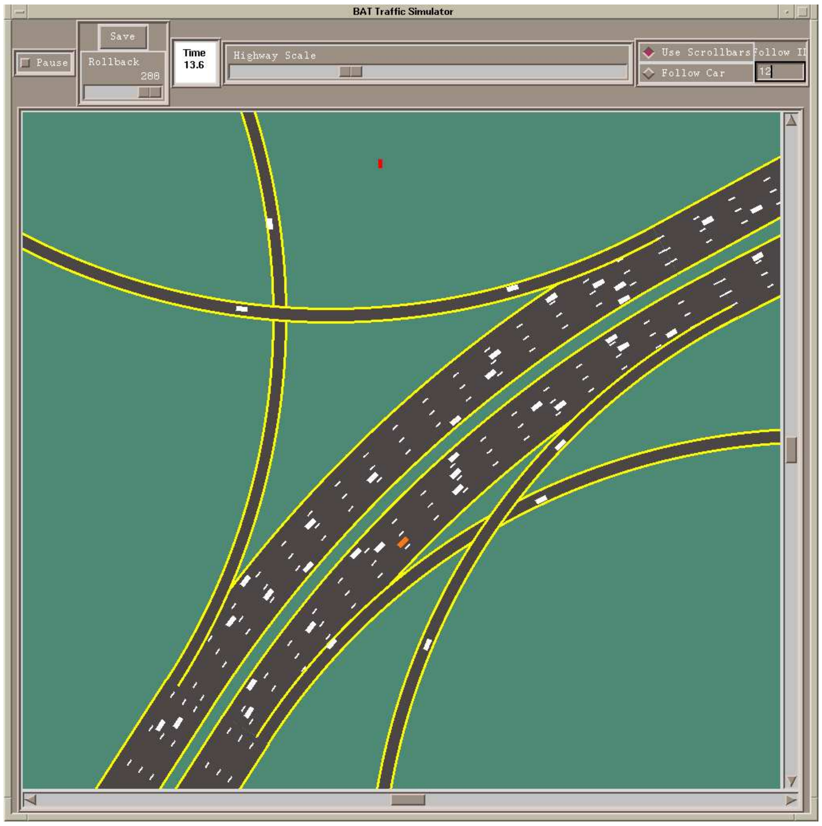

Figure 2: **Graphical front end to the BAT simulator:** The user can display information about and follow different vehicles. The user can also zoom in, zoom out, or focus on a particular area. Control buttons allow saving the state of a situation so that the situation can be recalled in the future.

Along with the above requirements, we also recognized a number of areas that were less critical to our simulation environment:

- Because our sensor models do not rely on the graphical representation of the simulation, our graphical display is adequate (see Figure 2) but does not offer the same level of detail found in other simulators.
- Because our effort has primarily been a simulator-based feasibility study, it has not been critical to build support for integration of our code with actual vehicles.

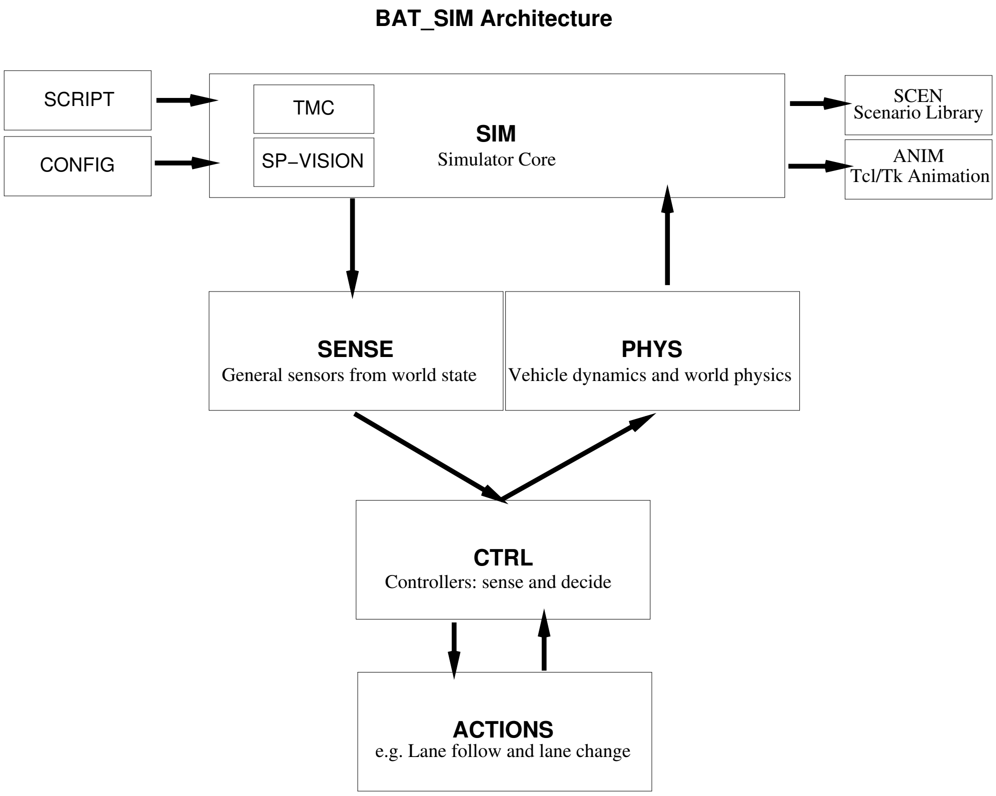

Figure 3: The BAT Simulator Architecture

The modules of the BAT simulator and their interactions are described schematically in Figure 3. The core module, SIM, runs the main simulator loop. It also maintains and updates the state of each vehicle in the simulation. When a simulation is initiated, the SIM module parses the SCRIPT file, which specifies the traffic distribution and simulation parameters. SIM also parses the CONFIG file, which describes the highway configuration. During the simulation, the SIM module can pass highway and traffic information to the ANIM module for interactive display, as well as to the SCEN module for saving "interesting" scenarios for future use. The SENSE module provides the controllers with percepts based on the world state maintained by the simulator. The CTRL module uses these percepts and other state information to return a controller output in the form of a steering, throttle and braking decision to the PHYS module. The PHYS module is responsible for simulating the vehicle dynamics and world physics. The ACTIONS module provides generic actions such as lane following and lane changing for the controller in order to reduce needless duplication of work for vehicle controller designers. Finally, the COMM module can perform network socket communication with other processes, including the SmartPATH animator, and the MISC module provides utility functions for use by any of the other modules. The figure also includes two applications built on top of the simulator: a TMC (Traffic Management Center) and SP_VISION, an extension that uses the SmartPATH animator to simulate a stereo vision system.

2.1.1 Sensor and Physics Models

The sensing and physics modules provide a layer of abstraction between the controllers and the world state as maintained by the simulator. The sensing routines provide the controllers with a consistent representation of the world state in the simulation. The information generally consists of information about the car itself and the cars which are within its sensor range. There are a number of different types of sensing and levels of detail. For example, the simple though not realistic sensor model may provide simulator information directly to a vehicle controller. A more sophisticated model may simulate noise, specific kinds of sensors, limited sensor range, and may infer additional information about neighboring cars.

The physics routines maintain and update the kinematic state of the vehicles based on the control decisions of the controller. The controllers pass their steering, throttle, and braking decisions to the physics module. The physics module also handles all other interactions of the controller with the environment. There are a number of different levels of physics which correspond to different models of dynamics. The most complicated model that we use is for a front-wheel drive automobile using 18 state variables. These variables describe the dynamics of the chassis, engine, tire, suspension, and steering, throttle, and brake actuators. (The hypothetical vehicle from which the parameters are taken is a Lincoln Town car engine with a Toyota Celica chassis, although other vehicle-specific parameters could be used as well. This model of vehicle dynamics provides an accurate representation of the physical plant of an automobile[Tomizuka *et al.*, 1995; Hedrick *et al.*, 1993; Peng, 1992].)

2.1.2 Lower-level Library of Action Routines

The actions module provides a library of routines for calculating lower level control decisions. These routines provide a controller-independent application of a control law based on a particular vehicle state. Developing new controllers is thus much easier because the programmer can simply plug-in already developed low-level control decisions and concentrate on choosing higher-level actions. Some of the actions which currently exist are:

1. Lateral actions
   (a) Follow lane number **i**
   (b) Change **left/right**
2. Longitudinal actions
   (a) Maintain speed of **n** $meters/sec$
   (b) Track vehicle in front at distance **n** $meters$
   (c) Achieve acceleration of **n** $meters/sec^2$

Some of the actions are continuously parameterizable. For example, tracking the vehicle in front at distance **n** meters which will return a throttle decision that keeps the car **n** meters from the vehicle in front. Other actions, such as changing lanes, are discrete.

To create the action routines, we generally used a proportional plus derivative controller where:

$$\delta(output) = K_p e + K_d * de/dt$$

and where *output* is a low-level control decision. The error is denoted by e, which may, for example, be the distance from the center of the lane we want to follow during lane following. The parameters K_p and K_d are constants which we determined using a technique called hill-climbing search. In this method, K_p and K_d are set to arbitrary initial values and are iteratively improved, based on how they affect a controller's performance in a training scenario. For example, poor performance in lane following results from diverging from the center of the lane, excessive lateral acceleration, and off-road excursions.

2.1.3 Scripting and Configuration

In order to fully test the control algorithms, the simulation must include realistic and challenging traffic patterns and highway topologies. The simulation scripts handle the creation of the traffic. A script contains a list of car generation commands, each of which describes probability distributions for vehicle type (including controller, size, and color), the time a vehicle enters the freeway, and the goals of a vehicle. A script also contains commands to control various simulation parameters. These may involve something as comprehensive as the length of the simulation or as particular as the variance on minimum following distance for a specific kind of controller. The script file format is described in Appendix A.3.

Highway networks in the BAT simulator are described as a set of highways connected by junctions. A highway is defined as an uninterrupted stretch of road with a constant number of lanes. The highways themselves are constructed from straight and circular arc segments. The junctions provide a way to add onramps, offramps, changes in the number of lanes, splits, merges and most other highway formations. The highway configuration files are compatible with SmartPATH version 2.0 (without the extensions for AHS, such as automated versus manual lanes). Appendix A.4 describes our configuration file format, and the SmartPATH User's Manual also gives details [Eskafi and Khorramabadi, 1994].

2.1.4 Scenario Generation

Scenario generation and incident detection facilitate the development and testing of controllers. The simulation can be instrumented so that when a particular "incident" occurs, the state of the simulator is rolled back a specified amount of time and saved. By "incident," we mean any sort of interesting traffic event, which may include crashes or just dips in the performance of a particular vehicle. A user can also roll back time and save the state manually while viewing the simulation in progress. In order to roll back the state of the simulation, the simulator and the individual controllers must periodically save their state. These saved states together constitute a traffic scenario. Once a scenario has been saved, the user may choose to vary some parameters or make other changes and then restart the simulator from that point.

A scenario can be saved for future testing to a scenario library. We are currently working towards creating a large scenario library to facilitate verification and measurement of performance of a controller on a number of difficult situations. Appendix A.2 discusses the use of the scenario saving and replay facilities.

2.2 Controllers

Given the design of the modules in the simulator, a controller needs to have four major routines:

1. **Initializer**: Called upon creation of the car. Initializes and returns the controller data structure.
2. **Controller**: Called every time step. Senses the environment using a sensor module and outputs its decision to one of the physics modules.
3. **Recorder**: Called at some specified interval. Records data needed for rollback of the system.
4. **Restarter**: Called up restart of a scenario. Sets the controller state to the recorded state.

Appendix A.1 further details the process of creating a controller.

There are a number of different controllers in the system. These include: a vehicle which stalls, simple lane followers, the Bayesian Automated Taxi itself, and "smart" drones. The parameterized stochastic smart drones have perfect sensing and can provide a number of interesting traffic patterns. The drones drive by trying to achieve a particular (possibly varying) target speed while driving toward its destination. In order to reach its destination, a drone sets a target lane on each highway. Besides the general goals of target speed and lane, the drone's actions are influenced by various driving parameters, some of which are regularly sampled from a probability distribution to bring about stochastic behavior. The drone controllers can be used as a basis from which to build controllers with more realistic percepts.

2.3 Implementation and Results

Although the current version of the BAT simulator uses very little of the code from the first working version, which was developed in late 1994, this version does share some characteristics with its predecessor. In particular, the simulator itself is written in C, and the graphical user interface is implemented in Tcl/Tk. To enhance the modularity of the system while maintaining compatibility with existing code, we created a framework that facilitates a more object-oriented structure of the system.

2.3.1 Extensions

The modular structure of the simulator has allowed us to implement several substantial extensions to the basic simulator itself. We have developed an interface with the SmartPATH system that allows us to display 3-D movies of traffic simulations in the animator. Furthermore, we can also obtain dual images from the SmartPATH animator to simulate input to a stereo vision sensor system. This makes it possible to quickly generate stereo vision data for vehicle controllers without having to actually drive an instrumented vehicle in real traffic.

A major extension that we have implemented is the construction of a TMC (traffic management center) graphical display on top of the BAT simulator itself. This gives users the ability to interactively add fixed video cameras alongside highways in the highway network. These cameras provide information about vehicles observed in their fields of view, and this information is reported in the TMC display. The Roadwatch project, a related research project at UC Berkeley that aims to perform wide-area traffic surveillance and analysis, uses this tool to display information such as average vehicle speed in a lane, average link travel time (the time it takes a vehicle to travel from point A to point B), and origin-destination counts (the number of vehicles that enter the highway network at some origin, X, and proceed to some destination, Y).

2.3.2 Performance

The performance of the BAT simulator is affected primarily by the physics model and by whether or not the display of the scene is turned on. With a simplified physics model and the display turned off (the data can be saved for replay later), the simulator can handle on the order of a thousand cars in real-time. This makes it possible to perform batch simulation runs, and we have done this in the context of applying machine learning techniques to constructing a vehicle controller. An example of this was the hill-climbing search described in Section 2.1.2. Turning on the display adds significant computational costs to running the simulation because Tcl/Tk is not well-suited for real-time animation. However, this is not a significant problem because most of our testing does not require the display.

3 Monitoring the Traffic State

3.1 Modeling Traffic

The driving problem can be modeled formally as a POMDP (partially observable Markov decision process). In a POMDP, the optimal decision is a function of the current belief state—the joint distribution over all possible actual states of the world. The problem can be divided into two parts: updating the current belief state, and making a decision based on that belief state.

In order to make appropriate control decisions, the BAT must have accurate information about its own state and the state of its environment. For example, the BAT must know its own position, velocity, and intentions, and it must monitor those of its neighboring vehicles. It must also monitor road and weather conditions, since they may significantly affect the BAT's ability to drive.

The state of the BAT's environment is only partially observable. Sensor information for variables such as vehicle positions and velocities may be incomplete and noisy, while driver intentions and road conditions may not be directly measurable at all. Thus, the BAT cannot make decisions based merely upon its latest sensor readings. Rather, it must maintain estimates for the random variables that together represent the state of the world, and it must make its decisions based upon the joint probability distribution over all those variables. In the rest of this section, we will outline our approach to maintaining the current belief state.

3.2 Dynamic Probabilistic Networks

To maintain the BAT's current belief state, we employ *dynamic probabilistic networks* (DPNs). Probabilistic networks are directed acyclic graphs in which nodes represent random variables (typically discrete) and arcs represent causal connections among the variables [Pearl, 1988]. Associated with each node is a CPT (conditional probability table) that provides conditional probabilities of the node's possible states given each possible state of its parents (or the prior probabilities if the node has no parents). Probabilistic networks offer a mathematically sound basis for making efficient inferences under uncertainty. The conditional probability tables provide a natural way to represent uncertain events, and the semantics of the updated probabilities are well-defined. Knowledge of causal relationships among variables is expressed by the presence or absence of arcs between them. Furthermore, the conditional independence relationships implied by the topology of the network allow the joint probability distribution of all the variables in the network to be specified with exponentially fewer probability values than in the full joint distribution. When specific values are observed for some of the nodes in a probabilistic network, posterior probability distributions can be computed efficiently for any of the other nodes using a variety of inference algorithms [Pearl, 1988].

DPNs allow for reasoning in domains where variables take on different values over time [Dean and Kanazawa, 1988]. Figure 4 shows the general structure of a DPN. Typically, observations are taken at regular 'time slices,' and a given network structure is replicated for each slice. DPNs model their domains as partially observable Markov processes, so nodes can be connected not only to other nodes within the same time slice but also (and only) to nodes in the immediately preceding or immediately following slice. The Markov property states that:

$$P(State_{t+1}|State_t, State_{t-1}, State_{t-2}, \ldots) = P(State_{t+1}|State_t)$$

In other words, the future is independent of the past given the present. As long as the BAT's representation of the world conforms to this property, the BAT need not maintain the history of its percepts to predict the next state, since the accumulated effect of its observations is captured in its current belief state.

The CPTs for the set of arcs proceeding from one time slice to the next form the BAT's state evolution model. This model quantifies how the BAT believes the actual state of the system will evolve over time. The CPTs for the set of arcs proceeding into nodes whose values are typically observed at each time slice

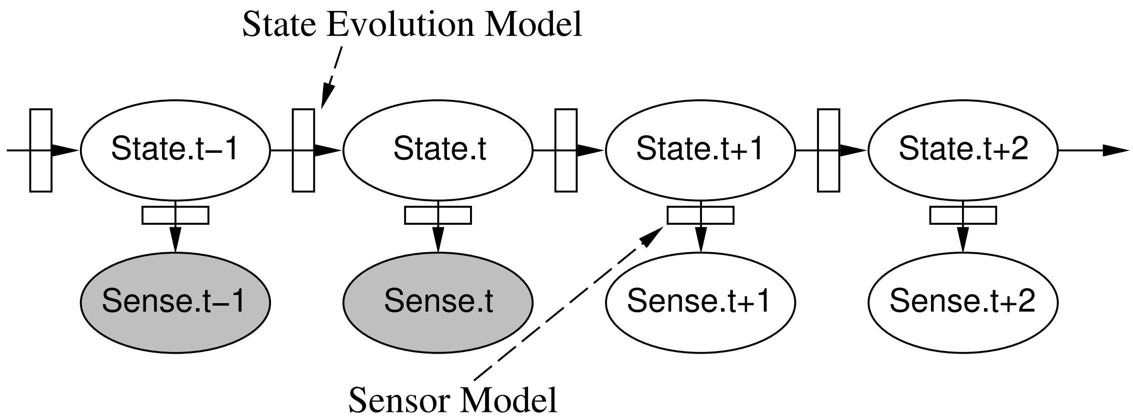

Figure 4: The structure of a dynamic probabilistic network. The ovals denote sets of state nodes or sensor nodes. The arcs going from one slice to the next form the state evolution model, and the arcs going into the sensor nodes form the sensor model. The shaded ovals denote observations available when predicting the state at time $t+1$.

form the BAT's sensor model. This model quantifies the likelihood of making a set of observations of the world given the actual state of the world.

Since a given variable may be measured by more than one sensor, the BAT must be able to integrate multiple sensor readings in computing its estimate of the variable's value. Because sensors are usually conditionally independent of each other given the variable that they measure, Bayesian updating within a probabilistic network easily fuses the readings from several sensors, automatically returning an updated posterior probability distribution for the value of the measured variable.

3.3 Network structure

As implemented, the BAT monitors each vehicle tracked by the sensor system with a separate DPN. Each network contains nodes for sensor observations, such as vehicle position and velocity, as well as nodes for predicting driver intentions, such as whether the driver intends to make a lane change or to slow down.

Like a Kalman filter, each network computes probability distributions for a vehicle's position and velocity based on both its latest observations and its previous state estimate (which reflects the influence of all previously observed evidence). Unlike a Kalman filter, which is limited to Gaussian distributions, the network predictions can be arbitrarily distributed. For example, if a vehicle were approaching some debris directly in front of it, the network could predict that the vehicle would move either to the right or to the left (but not straight) in order to avoid the debris. Also, the network could easily incorporate additional sensor information. If the sensor system recognized that a vehicle was flashing its right turn signal, the network could make predictions that biased the vehicle's position towards the right.

An alternate, perhaps preferable, approach to vehicle monitoring would utilize one large scene network for all relevant vehicles. Because this greatly increases the computational complexity of inference operations and requires run-time modifications to the network structure, we chose to use separate networks for each vehicle. To incorporate the influence of nearby vehicles, each network contains nodes corresponding to those vehicles. For example, the Front Clear and Front Speed Diff nodes in Figure 5 refer to "the space between this vehicle and the vehicle in front," and "the speed difference between this vehicle and the vehicle in front," respectively. Since the vehicle in front of or behind a given vehicle may change, these *indexical* nodes do

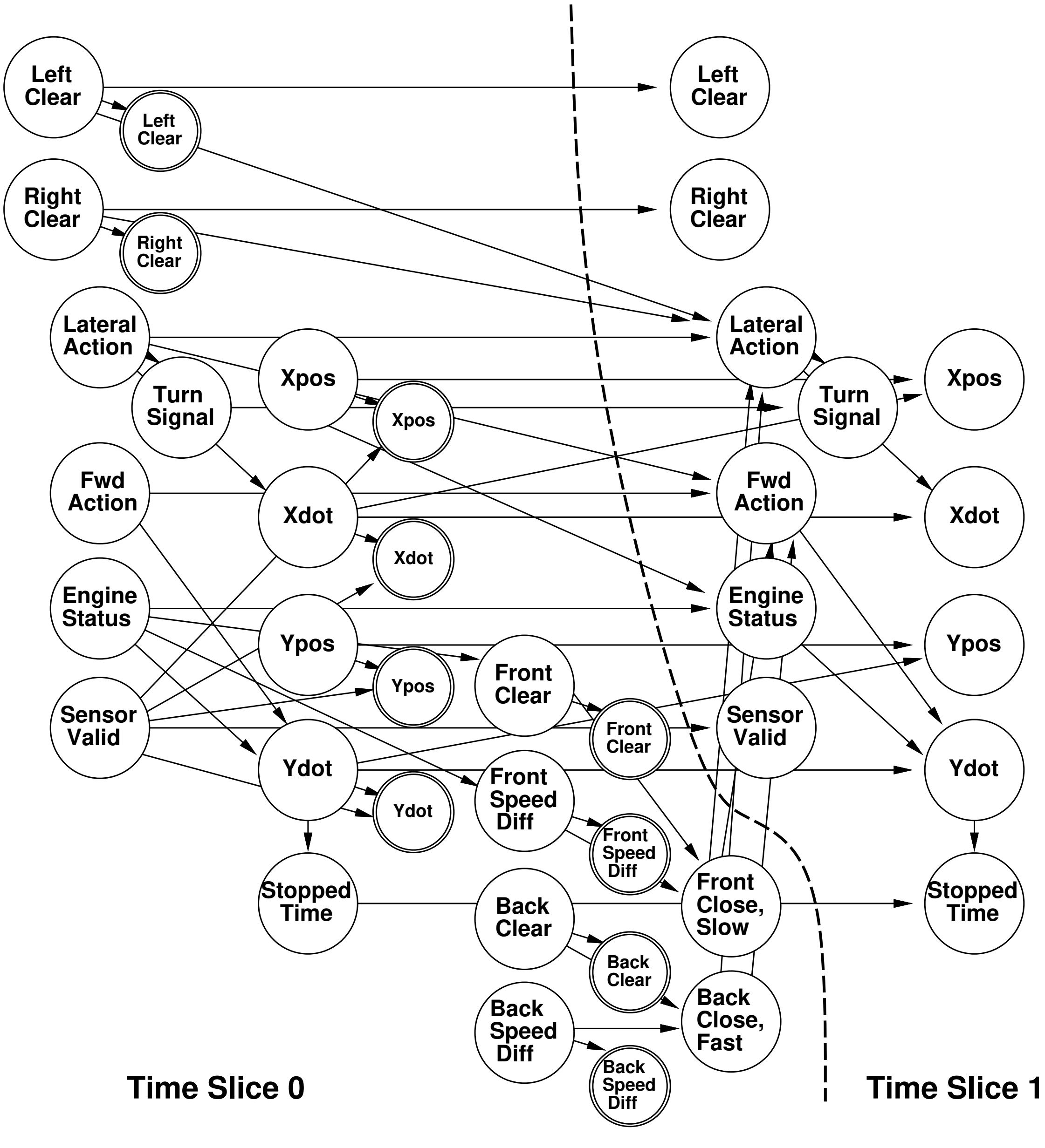

Figure 5: Dynamic probabilistic network for one vehicle, including inter-slice arcs. The smaller nodes with thicker outlines denote sensor observations.

not correspond to a specific vehicle. Instead, a preprocessing step using sensor data determines the spatial relationships among the vehicles and then sets the node states accordingly. Figure 5 shows an example vehicle network for one time slice, along with the inter-slice links to the next time slice.

3.4 Stochastic Simulation

Stochastic simulation algorithms compute probabilities by counting how frequently events occur in a series of simulation trials. Using stochastic simulation in probabilistic networks, one can approximate the posterior probabilities. Essentially, each sample corresponds to a "possible world"; if a particular fact appears in more possible worlds, then that fact is more likely. We can use the samples to indirectly represent the belief state.

Monitoring an environment with a DPN puts more restrictions on the behavior of the algorithms. First, the number of trials must be constant, so that real-time monitoring in a dynamic environment is possible. If the number of sampling trials grows, the time required to do inference will also increase. Second, the error in the approximations must be bounded over time. The systems may be running for hours, days, or years, so if the error diverges, the inferred state values will become useless after a certain period of time. In stochastic simulation, given a set of samples, one can compute the belief state, so effectively, the samples can be used themselves to calculate the belief state. After getting new evidence, the algorithm extends the samples forward by one time step, so that this new extended set of samples is a surrogate for the belief state at the next time step. Because of this representation, one does not need rollup; since the samples at any point uniquely determine the belief state, there is no need to change the network and its probability tables to absorb evidence.

3.4.1 Likelihood Weighting

In stochastic simulation, one typically generates samples recursively from roots to descendants based on the conditional probability distributions. The problem is that since evidence nodes, nodes representing evidence such as sensor readings, are leaf nodes, we may sample a "sensor reading" that is different from the actual sensor reading.

An algorithm called likelihood weighting fixes this by weighting the samples differently based on their agreement (likelihood) with the evidence. That way, random samples which do not represent the world state are relatively ignored, while those possible worlds which seem likely will have higher weights. However, stochastic simulation with likelihood weighting is still problematic because when samples are propagated forward, it is based primarily on the state evolution model. The sensor readings, because they are leaf nodes, do not participate in extending the samples. For this reason, the likelihood weighting samples will diverge further and further from reality. An increasing number of samples will have low weights, thus the overall "sample mass"; the sum of all of the sample weights, will be lower. The cache mass represents the effective number of samples, since many low-weighted samples can provide inaccurate measurements, since the algorithm will be relying on a smaller number of samples for the results. Thus, after a relatively small number of time steps, the posterior probabilities inferred by likelihood weighting are not reliable.

3.4.2 Evidence Reversal & Survival of the Fittest

The problems of likelihood weighting can be fixed by evidence reversal and survival of fittest sampling. In evidence reversal (ER), through network arc reversal operations, we can make evidence nodes be "local" root nodes, where the only parents are in previous time slices. Then, when we extend the samples, they will reflect the evidence. These arc reversal operations may increase the connectivity of the network, but since calculating node values only requires a CPT lookup, there is no significant decrease in speed of inference.

Another method is to selectively extend samples based on their likelihood given observed evidence. In survival of fittest (SOF) sampling, we randomly choose whether or not to extend a sample based on its weight. Samples with high relative weight are extended more frequently compared with those of low relative weight. In this way, the samples indirectly reflect the evidence better.

An obvious idea is to combine ER and SOF. In this way, the results are better than either of the techniques separately.

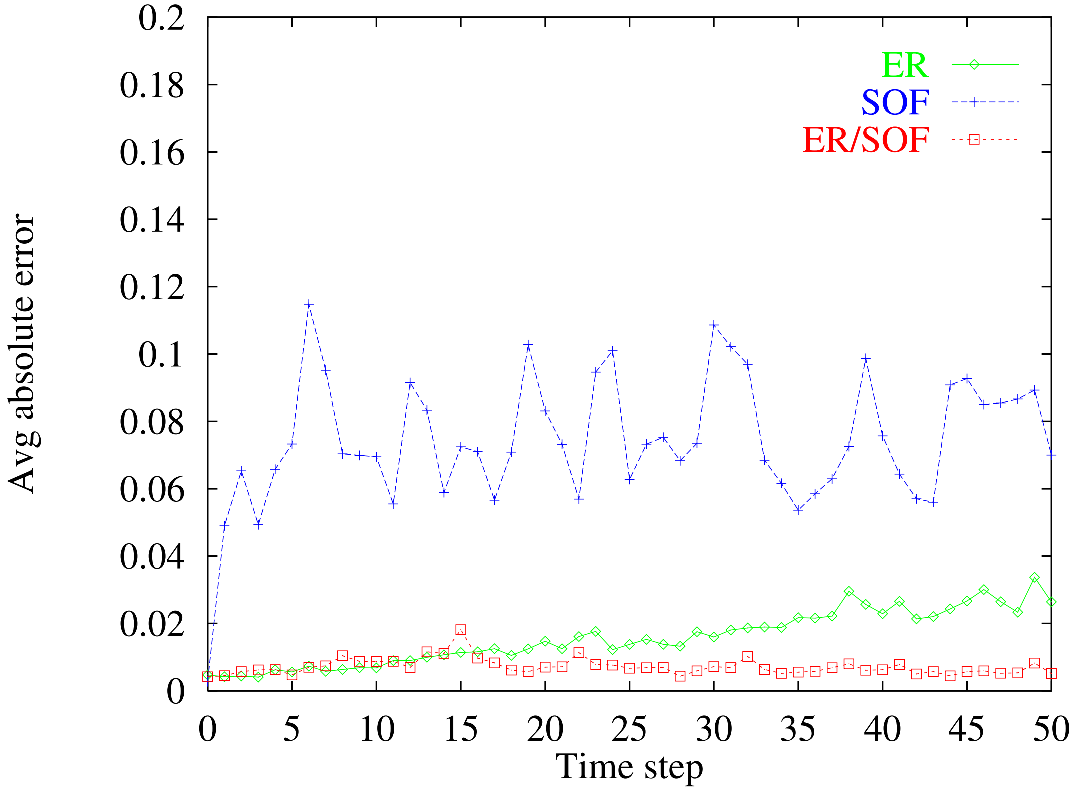

Figure 6: Various sequential Stochastic Simulation techniques

In our experiments, the error of ER/SOF appears not to diverge. On top of this, ER/SOF is inherently fast because it avoids the expensive run-time calculations involved with rollup, so it is a very effective and practical algorithm for maintaining the belief state.

3.5 Parallel Implementation

The algorithms were written in Split-C and run on a Thinking Machines CM-5 with 32 processors. Since Split-C can be used on multiple platforms, the system should be portable. Much of the core code was copied from the sequential version developed by Keiji Kanazawa including the Tcl interface that enables a great deal of user-specified functionality.

The parallel implementation of stochastic simulation using likelihood weighting (P-LW), was relatively straightforward. Each processor had its own copy of the net and ran its own independent trials. At the end of the local trials, the scores (the weighted sampled values for the state nodes) are summed across all of the nodes, Then the posterior probabilities can be computed by normalizing the scores.

The parallel version of ER/SOF (P-ER/SOF) proceeds the same as P-LW. Evidence reversal is done in the initialization stage or off line. In survival of fittest, the number of times a sample is propagated is proportional to its weight relative to the sample mass. In sequential SOF, the sample mass is global for all samples, but in the parallel version, each processor only has access to the local samples. We can propagate each sample relative to the local cache mass instead. The properties of the algorithm can change due to this local selection though. The relation is unclear because it is not known how the error varies with the number of trials - linearly, exponentially or whatever. The variance in error for a number of processors should be lower than on just one, however.

If a processor has set of generally "poor" samples, that is samples which are unlikely given the evidence, that processor will not be able to pick a better sample from the global pool with local selection. In this case, the processor will effectively be computing fewer trials and will not be performing its share of useful computation. Thus parallelism will decrease due to this load imbalance. Drastic load balancing measures are probably not necessary due to the existence of local selection and also because any extra communication can significantly reduce the parallelism of an "embarrassingly parallel" task.

A general scheme would be to periodically rebalance the global cache mass minimizing the time spent rebalancing wherever possible. How often rebalancing should be done could be determined experimentally, the values would probably vary depending on the network topology. To do a "rebalance" step, the processors are ordered and placed schematically in a two dimensional torus as shown below. The processors on the edges are actually connected to the other side.

3.6 Parallel Results

Parallel likelihood weighting exhibits almost linear speedup as seen in Figure 7, even on a relatively small number of trials (5000). P-ER/SOF did well too, but because of added global cache mass calculation and redistribution, the speedup was somewhat less.

Next, we varied the number of trials with the number of processors (Figure 8). Likelihood weighting once again showed almost perfect speed up. Once again, P-ER/SOF did only slightly poorer.

The parallel implementation yields enough speedup to make simulation over large networks possible in real time. Monitoring complex stochastic may now be a feasible problem. Assuming ER/SOF does not diverge, parallel ER/SOF on dynamic probabilistic networks can be used effectively to maintain the belief state over time.

4 Dynamic probabilistic network learning

One of the primary criticisms aimed at probability-based systems is the difficulty of obtaining meaningful probability information. This is particularly relevant in the BAT project, since the numbers will be used for life-and-death decisions. For this reason, we developed learning algorithms for probabilistic networks [Russell *et al.*, 1994] and extended them to handle dynamic probabilistic networks [Russell *et al.*, 1995; Binder *et al.*, 1997]. This section describes preliminary work on the problem of learning DPNs that will serve as driver models for monitoring the behaviour of other vehicles.

Creating driver models involves identifying the variables essential to predicting human driver behavior, using a learning algorithm to automatically construct a DPN model from data, and assessing how well the model

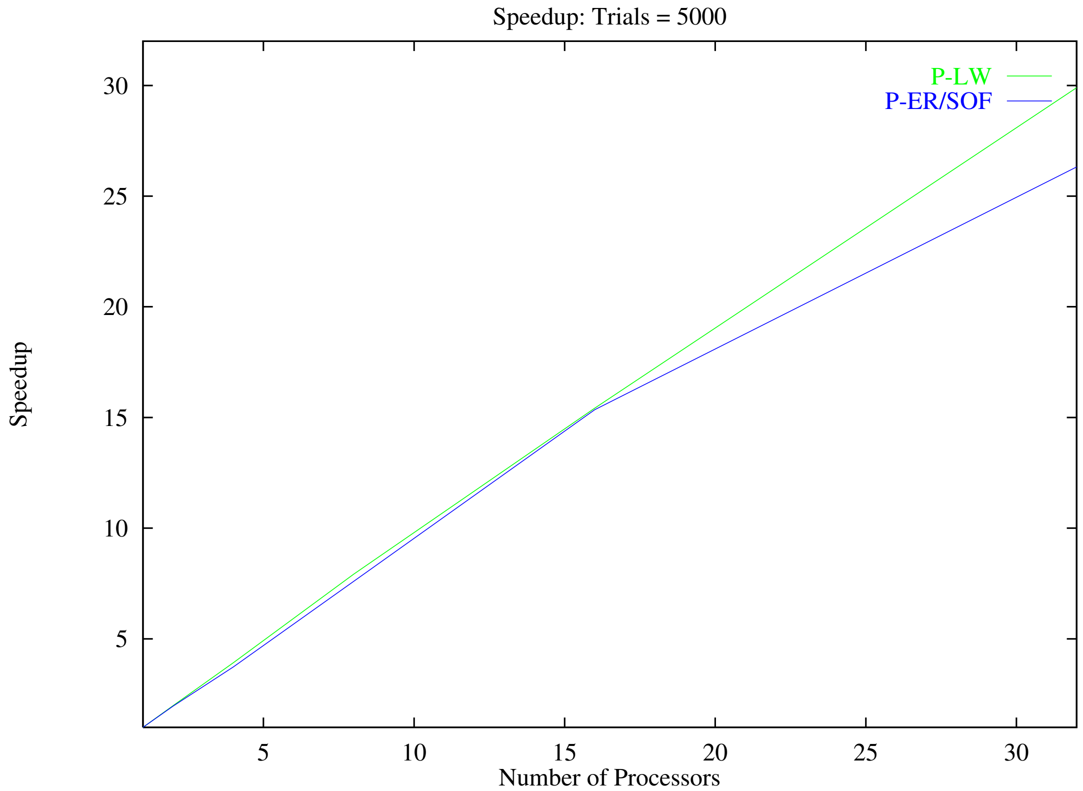

Figure 7: Speedup.

predicts drivers' actions. The data might come from simulator cameras, video surveillance cameras [Huang *et al.*, 1994], or onboard cameras. In this case, as discussed above, we used data from simulator cameras.

The type of model we are constructing is considered to be a microscopic model in that it models the behavior of individual vehicles. There have been several attempts to create microscopic models of driver behavior. Most of these are deterministic models in which the acceleration of a vehicle is assumed to be some function of (at a minimum) its own velocity, its relative velocity with respect to the car in front, and the distance to the car in front (a good summary of these is given in [Benz, 1993]). However, traffic flow is stochastic in nature (e.g., drivers often behave differently even in similar situations); therefore, a stochastic model is more appropriate. There has been some work on creating probabilistic models [Niehaus and Stengel, 1994];however, this work does not incorporate percepts over time to assess unobservable aspects of the current state, and the models have not been validated.

We first discuss two DPN models we have constructed so far; the first model does not have any hidden state nodes while the second one does. We explore what behaviors our model without hidden state has learned and explain what we expect to gain through the addition of hidden state.

4.1 Simplified network models used for learning experiments

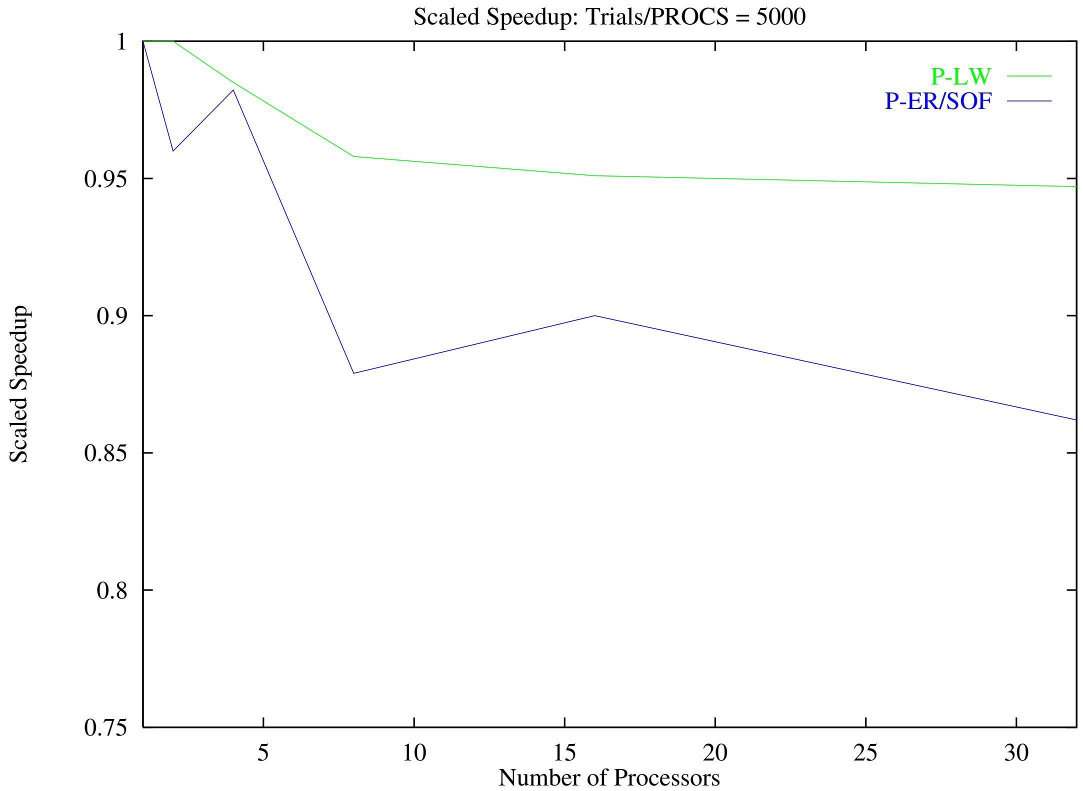

Figure 8: Scaled Speedup.

4.1.1 Model Without Hidden State

The first network that we created is shown in Figure 9. Both this and the network with hidden state have variables with discrete values. The nodes, their meanings, and possible values are as follows.

CarLeft: Is there a car to the left?

Possible Values: True, False

CarRight: Is there a car to the right?

Possible Values: True, False

cif: How far ahead (in meters) is the car in front (if any)?

Possible Values: $(0, 2], (2, 5], (5, 10], (10, 25], (25, 50], > 50$

relspeed: What is our relative speed (meters/second) with respect to the vehicle in front (if any)?

Possible Values: $< 0, [0, 1), [1, 2.5), [2.5, 5), > 5$

$\dot{x}$**:** Lateral Velocity (meters/second) (positive velocity is toward the right).

Possible Values: $< -1, [-1, -0.5), [-0.5, -0.25),$
$[-0.25, 0.25), [0.25, 0.5), [0.5, 1), > 1$

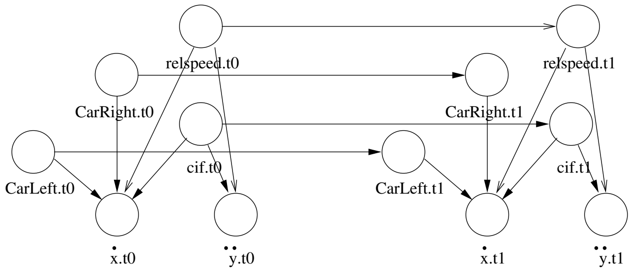

Figure 9: Network with no Hidden State

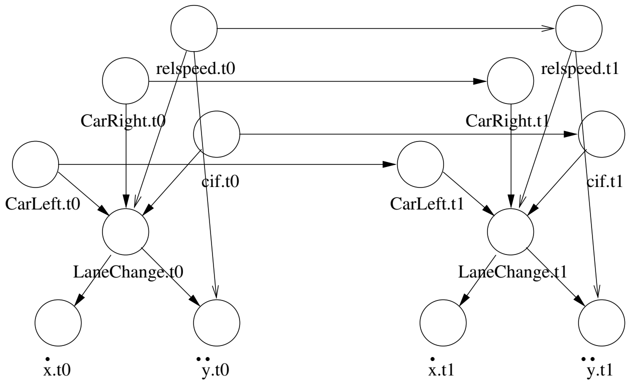

Figure 10: Network with hidden state: the LaneChange nodes are not observable.

$\ddot{y}$**:** Acceleration

Possible Values: $< 0, \approx 0, > 0$

Of course, not all of these variables are observable at all times. Because we obtain our data using cameras mounted on the side of the highway, there are occasionally cars for which the cameras cannot see the car in front either due to occlusion or because the car in front is outside the camera's field of view.

4.1.2 Model with Hidden State

The second network that we created is shown in Figure 10. The nodes, their meanings, and possible values are the same as above with one addition.

LaneChange Unobservable

Possible Values A, B

This node is intended to represent the hidden state of whether the driver is intending to change lanes or not, which is why we named it "LaneChange." Of course, because the value of this node is never observable, we cannot force a particular real-world meaning upon this node. What we hope is that, after learning, we can examine the CPTs of the hidden nodes and the children of the hidden nodes and find real-world interpretations of the hidden nodes that are consistent with the CPTs. This is exactly what we plan to do for the "LaneChange" node. Our hypothesis is that the addition of this node will improve prediction—it reflects the fact that driver's intentions are relevant aspects of hidden state that carry over from one time step to another.

We will to use this method to learn aspects of driver preference such as the preferred following distance or preferred minimum size of the gap in a neighboring lane when changing lanes. For example, given the network above, if we learn how the lane changing behavior depends on the distance to the car in front, we would also know the preferred following distance of the vehicles. We can also use this method to learn higher-level actions such as overtaking and following. By using traffic data to learn these models and, ultimately, inducing an interpretation upon our hidden nodes, our method effectively will find those behaviors and attributes that are relevant to understanding real traffic and driver behavior rather than our own conception thereof. As a result, our models may even find novel methods of evaluating traffic and driver behavior.

4.2 Learning results

For each DPN we created, we modified the simulator to return the observable data (and placeholders for unobservable or missing data) in the form required by the belief network learning and inference system Facilitator, developed in our group by John Binder. We learned the conditional probability tables (CPTs) of the dynamic probabilistic networks using the simulator data by instantiating 30 slices of the network—where each slice represents one reported camera observation of a vehicle and each training example consists of 30 reported camera observations of the same vehicle—and learning this as a static network with the caveat that the CPTs corresponding to connections between nodes in different time slices remain the same across all time slices.

4.2.1 Learning results with observable network

We learned the network using one scenario and tested its performance on another similar scenario (similar flow and density), yielding the learning curve shown in Figure 11. The learning curve gives the log-likelihood of the data from the test scenario given the learned model, where one data case is 30 reported camera observations of one vehicle. The learning curve approaches its maximum log-likelihood quite quickly, indicating that the traffic is fairly consistent across the entire scenario.

We can analyze what this network has learned in more detail by examining the conditional probability tables. One example is the node for lateral velocity. Let us examine the case where the car in front is from 25 to 50 meters ahead and the speed of the vehicle is less than that of the car in front. Figure 12 shows the lateral velocity distribution for the case where there are cars immediately to the left and right, which precludes the possibility of a lane change. The distribution indicates the correct behavior, which is to continue straight ahead (the car would slow down as necessary depending on the relative velocity with respect to the car in front).

Figure 13 gives the distribution for the case where there is a car on the left but no car on the right, and Figure 14 gives the distribution for the case where there is no car on the left and a car on the right. As indicated in Section 6, the traffic scenario that we used to learn these driver models has vehicles that are heavily biased toward changing lanes to the left rather than the right. For the case where there is a car on the left but no car on the right, the cases where $P(\dot{x} < -0.25)$ correspond to completion of a left lane change, for example from lane 3 to lane 2, which places the vehicle next to another vehicle in lane 1. During the completion of that lane change, negative values of $\dot{x}$ and the presence of a car to the left are reported.

Figure 15 gives the distribution for the case where there are no cars on either side. In this case, the cars we have observed are clearly biased toward changing lanes to the left rather than to the right. This is consistent with the convention of passing vehicles on the left-hand side where traffic is generally faster.

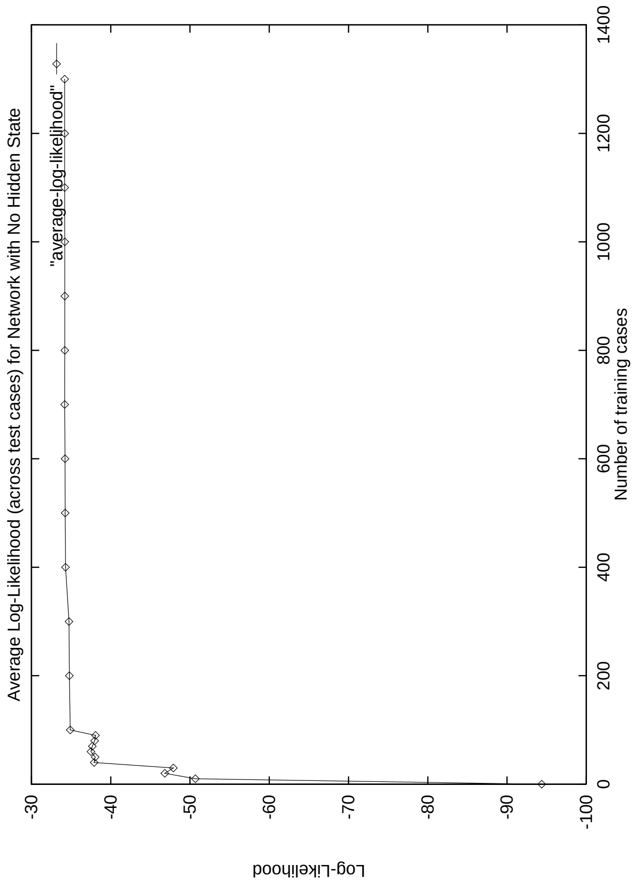

Figure 11: Learning Curve for Network with no Hidden state

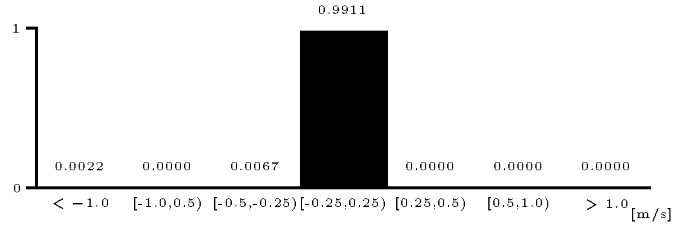

Figure 12: Probability of $\dot{x}$ (lateral velocity) given CarLeft=True, CarRight=True, cif=25-50m, relspeed < 0.

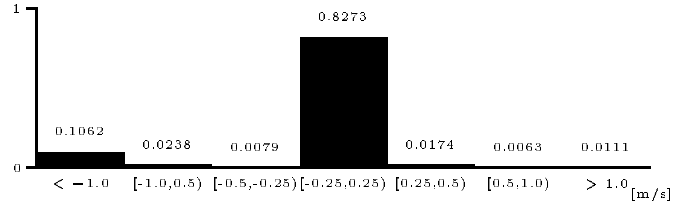

Figure 13: Probability of $\dot{x}$ (lateral velocity) given CarLeft=True, CarRight=False, cif=25-50m, relspeed < 0.

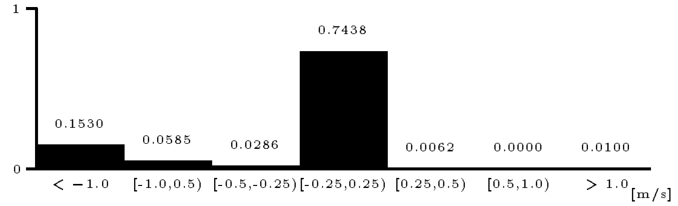

Figure 14: Probability of $\dot{x}$ (lateral velocity) given CarLeft=False, CarRight=True, cif=25-50m, relspeed < 0.

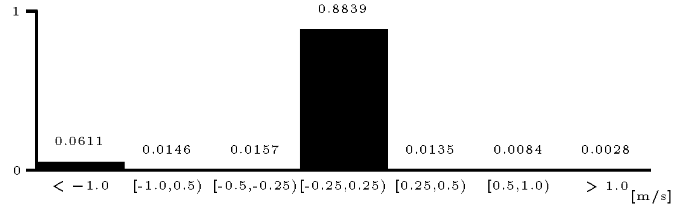

Figure 15: Probability of $\dot{x}$ (lateral velocity) given CarLeft=False, CarRight=False, cif=25-50m, relspeed < 0.

5 Decision Making

In this section, we describe our approach to real-time BAT decision making. As we noted earlier, the decision problem corresponds to a POMDP. Computing the policy for POMDPs is PSPACE-complete, and exact solutions can be obtained only for tiny state spaces. Our approach to real-time BAT decision making is to find approximate solutions. We are undertaking three separate approaches. They are (1) bounded lookahead using dynamic decision networks, which incorporate action nodes and an explicit utility function; (2) hand-coded, explicit policy representations, such as decision trees, that take as input the joint probability distribution encoded in the DPN; and (3) supervised learning and reinforcement learning methods for solving the POMDP, in which we learn a policy representation, a utility function on belief states, or an action-value function on belief-state/action pairs.

5.1 Hand-Coded Approaches

Our second approach combines the DPN state evolution model with a decision tree. The decision tree is a tree of binary if-then-else constructs where the test predicates are computed from the joint distribution computed by the DPN. Each leaf of the tree is a decision. This obviously yields an effective, real-time policy, but constructing the decision tree is a difficult task. Other researchers, for example Lehner and Sadigh [Lehner and Sadigh, 1993], have examined the creation of decision trees from influence diagrams, but only for static problems. In such cases, the decision tree nodes test fully determined evidence variables. If this method is applied to dynamic problems, one may be forced to test the entire percept sequence.

Our approach involves testing the current belief state instead. Although this is potentially much smaller, optimality in a POMDP requires that the tests define regions in the joint probability space rather than regions in the marginal probability space for each variable. We have found this extremely unintuitive, and so have used tests on marginals of individual variables as an approximation. To date, this has been reasonably effective. We have implemented several hand-constructed decision trees, which have the following general structure (each "predicate" here is actually a complex set of probability thresholds on specific variables, and each "action" a subsidiary decision tree):

```
if imminent collision
    Avoid collision
else if no vehicle in front or fast vehicle in front
    Maintain target speed
else if slow vehicle in front
    if can change lanes to faster lane
        Pass
    else
        Maintain safe following distance
```

Figure 16: Excerpt of Basic driving decision tree

We found that having major modes for the driving state such as changing lanes or exiting the highway made. The reason for these modes is that they override the basic driving behavior. For example, if a controller starts changing lanes, it should continue changing lanes until it has reached the next lane unless there is some threat. After initiating a lane change to the left, it is not advisable to change lanes to the right because that lane appears to be faster now. Furthermore, when changing lanes, the vehicle may have some acceleration trajectory to get into a gap between two cars in the adjacent lane. These modes formed the top level of the decision tree, and they dispatch control off to the different modules. The problem which these modes cause is that there may be cases where it is necessary to switch modes and it is difficult to do so cleanly and not have oscillations back and forth between modes.

Some of the driving modes are described below:

Basic Maintains lane and target speed while passing vehicles where necessary. An example of the basic drivng structure is given in Figure 16.

Continue Lane Change The controller tries to predetermine hazards before changing as much as possible in making a lane change. In determining whether a lane change is possible, a acceleration trajectory is also computed.

Move to Target This mode occurs when the trying to exit or enter a highway. The controller may need to slow down or speed up to move into a gap in traffic before the junction.

To a large extent, the behavior of the vehicles is governed by its target lane and target speed. The target speed is just set upon initialization of the vehicle. The target lane is set depending on the location of its destination highway. Each vehicle has access to a map object which gives the possible routes from highway A to highway B. Below, we have given an example of how we act depending on the distance to the junction.

In general the implementation of the longitudinal control was independent of that of the lateral control. For more complicated maneuvers such as swerving to avoid an obstacle, this assumption would no longer hold.

5.2 Hierarchical Control

The controllers are able to specify high level actions such as lane following instead of the actual steering control decisions, for example. However, the low level control is still an important issue in the BAT project. We have found that a hierarchical approach to making driving control decisions is essential for ease of debugging and testing and from a conceptual viewpoint. The driving problem seems to be to intricate for one monolithic control law to handle. Instead, a hierarchical control where the low level steering, throttle, and braking control decisions are computed by actions and the correct actions are determined by a higher level decision module. Above that level, there will also eventually be a planning component. A problem which must be addressed with hierarchical systems is the inevitable interactions between different levels. For example, if the the controller decides to change lanes, it must monitor the progress of the lane change maneuver and abort the lane change if necessary. Graceful switching of control between different actions has been a significant challenge throughout the project.

5.2.1 Control Laws

To create the action routines, we generally used a proportional plus derivative controller where:

$$\delta(output) = K_p e + K_d * de/dt$$

and where *output* is a low-level control decision. The error is denoted by e. In the case of lane following, we used two types of error, the lateral error (e_x) which was the vehicles deviation from the center of the lane, and the heading error (e_h) which was the angle by which the cars heading differed from the road markers heading at the current point on the road. To directly take into account different velocities (v) and the curvature of the road (C) defined as the signed inverse of the radius of curvature. Thus, the lane following control is defined as:

$$\delta(output) = K_{p_x} e_x + K_{p_h} e_h + +K_{d_x} * de_x/dt + K_{d_h} * de_h/dt + K_c C/v$$

5.2.2 Hill Climbing

The control laws provided a framework for the implementation of the control decisions from the higher level actions, but the actual parameters, K_p and K_d above, for these must be obtained by a tedious and possibly ineffective trial and error process. We were able to find reasonable control parameters using iterative improvement algorithms. The control would be run for a length of time at different speeds and then be given a score based on its performance. For example, in the case of the lane following controller, the error accumulated with the aggregate distance outside the center of the lane with additional penalties for velocity across the highway. The control parameters would be varied in some direction by some step size. Whenever the error was less in a certain direction, . If the performance could not be improved by moving in any direction, then a minima had been reached. In general, the minima which were reached were local, so we would randomly restart the hill climbing process in another region of the parameter space.

5.2.3 Reinforcement Learning

We developed a BAT controller that used *reinforcement learning* to improve its performance over time. Reinforcement learning is a general method that is capable of learning a utility function in environments with delayed feedback. Feedback can take the form of a positive signal when the destination is reached, or a negative signal when an undesirable outcome, such as a crash, is obtained. The feedback is delayed in the sense that the action or actions that led to a particular outcome may not have been the actions that most recently preceded the outcome. Stepping on the brake, for example, does not cause crashes even though it is the action that most frequently precedes a crash.

The reinforcement learning problem can be formalized in the framework of Markov Decision Process (MDP), in which the environment is divided into states and a stochastic transition function describing the effects of actions or control inputs upon the environment state. A reward function is also defined over states, allowing us to assign a positive reward or feedback signal to desirable states and a negative reward or punishment for undesirable states. The value of time is often captured through the use of a discount factor, $0 < \gamma < 1$, which discounts future rewards versus current rewards. We say that an optimal control strategy for an MDP maximizes the expected, discount sum of rewards received:

$$E(\sum_t \gamma^t r_t)$$

where, r_t is the reward received at time t.

If the state space of an MDP is small, then we can use a procedure called *Q-learning* that is guaranteed to converge to the optimal control strategy. Q-learning works by maintaining an estimate, $Q(s, a)$, of the value of taking action a in state s. When a transition from state s to state t is observed under action a, the Q estimate is updated according to the following rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma V(t) - Q(s, a)]$$

where R is the immediate reward received, α is the learning rate, and $V(t)$ is the value of state t, determined by our current estimate of the best action in state t. ($V(t) = \max_a Q(t, a)$).

For the driving environment, it is impractical to maintain a table for the cross product of every state of the simulator and every possible control input. In such cases, the standard approach is to use a generalized function approximation method to represent the Q-function. We replace $Q(s, a)$ with a generalized function, $F(s, a, w)$, where $w = (w_1, w_2, \ldots w_k)$ is a vector of weights or parameters for F. We chose a neural network for F, so w corresponded to the weights and thresholds in our neural network.

The update rule for Q-learning with a function approximator is best explained in two steps. The first step determines the discrepancy between the value assigned to the current state and the value estimate for the next state:

$$E \leftarrow R + \gamma V(t) - Q(s, a).$$

The weights in the function approximator are then updated according to the following rule:

$$w \leftarrow w + \alpha \bigtriangledown E$$

We did some experiments using a neural network to represent our Q-function and using the sensor space, the space of all possible sensor inputs, as the state space for the learning problem. We restricted our investigation to a simple subproblem - steering the vehicle to stay within a lane on a curved road. There were no positive rewards in this test, but penalties were given for excessive lateral acceleration or angular velocity and a penalty was assessed based upon how far the vehicle was from the center of the lane. The neural network represented the value of different steering controls in different sensor states as a continuous function and the optimum input at any time was estimated using a gradient ascent strategy.

Our initial empirical results for this method were quite promising. With only the simple reward function described above and no human intervention, the reinforcement learning controller quickly learned to stay centered as it smoothly turned left and right on an asymetrical figure-eight shaped track.

We believe that it is possible to extend these results to more complicated driving scenarios. The addition of other vehicles to scenario will require a more complicated representation of both the environment and the controller's belief state about the objects in that environment. We expect that in the future, a dynamic belief network will represent this information and be used as an input for the neural network Q-function approximator. This architecture, which we call the DPN/RL architecture, will form the basis for the next phase of the project.

6 Results

6.1 Specific traffic and freeway configurations used

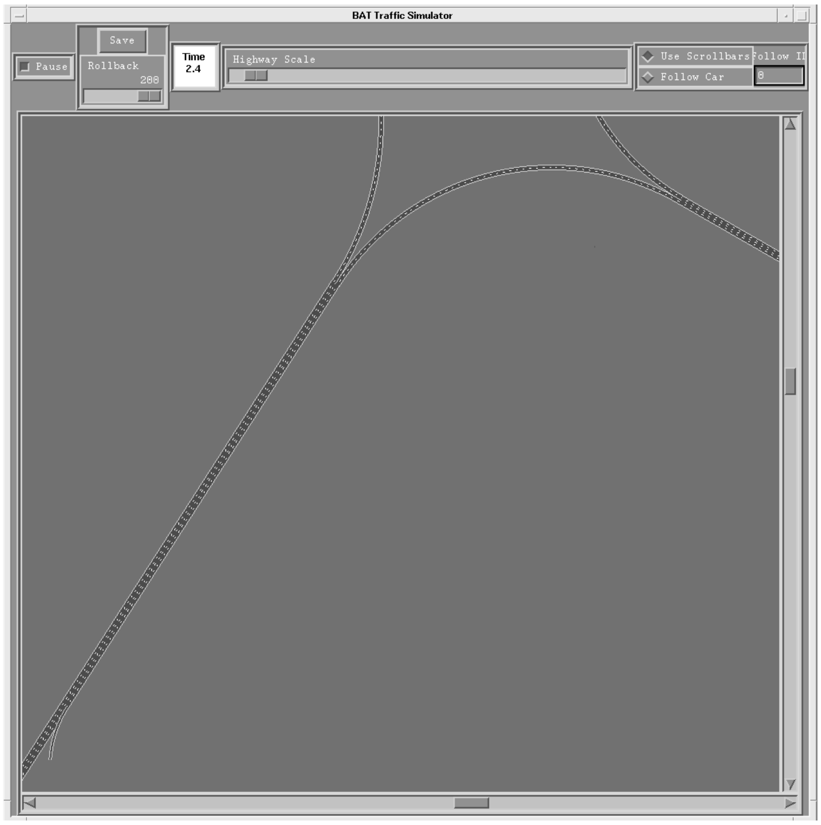

Figure 17: **Bay Bridge area freeway network**

As explained earlier, the simulator allows us to create very diverse freeway configurations and traffic scenarios. One example freeway network is shown in Figure 17. The view shown here is similar to the interchange between Interstate 80 (the lower freeway), Interstate 580 (above the current view), the Bay Bridge (the freeway on the right side), and the Powell Street onramp (at the bottom of the figure) to Highway 80 in the San Francisco Bay Area. Another example freeway configuration is Figure 18. This network was designed specifically as a rigorous testbed for reinforcement learning a lane-following controller Section 5.2.3. We have similarly designed numerous other freeway networks either to support traffic scenarios involving many cars or to observe and/or learn the behavior of specific vehicle controllers.

We created traffic scenarios that we used to learn models of driver behavior in Section 4 by first generating slow vehicles in the three right lanes out of the four lanes and then generating faster vehicles following them. Therefore, in order to proceed at their desired speed, the faster vehicles had to change lanes to the left, ultimately into the left lane so that they could pass the slow traffic in front. Faster vehicles were continually generated, so there was a significant amount of lane changing and the left lane occasionally became a bottleneck as faster cars in all three of the right lanes wanted to get into the left lane to pass the slow traffic.

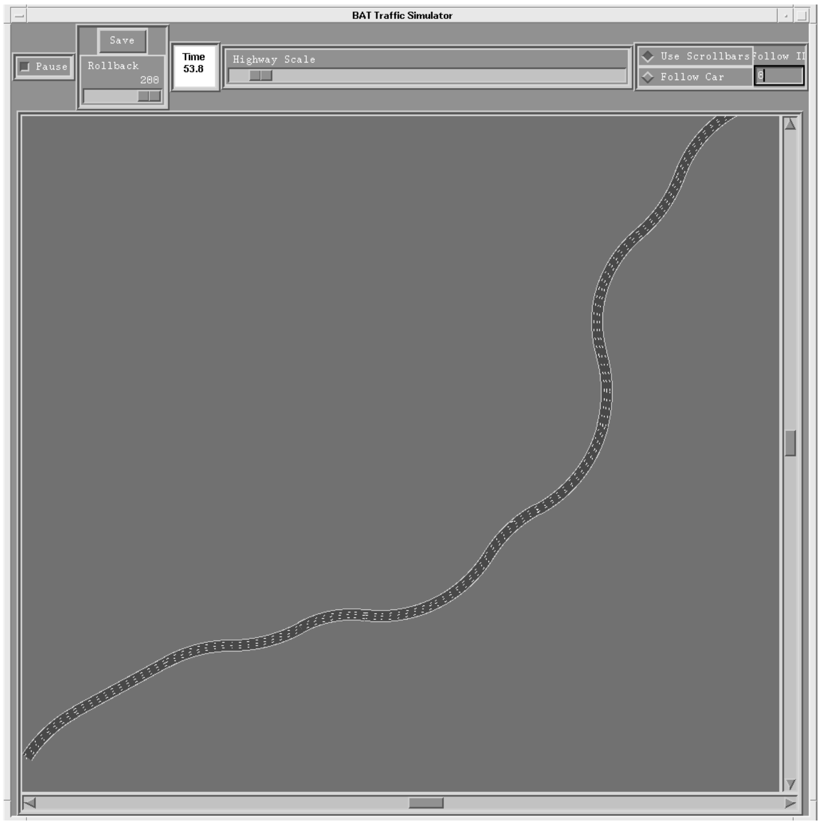

Figure 18: **Freeway used for reinforcement learning of lane-following controller**

6.2 Controllers tested

We incrementally improved the controllers by changing some of the predicates and in some cases modifying the structure of the decison trees itself. For the most part, we tried to keep the low level actions consistent. Here are the the controllers we tested and a few distinguishing features.

BAT0 The initial version converted from the controllers which the drones used.

BAT1 Adding the driving modes allowed the system to better follow goals such as finding an exit or continuing a lane change. Improved the front vehicle tracking action.

BAT2 Improved change lanes and lane following actions to handle curves and higher speeds better.

BAT3 Reduced sizes of "buffers," areas around other vehicles which make the controller more risk averse in changing lanes or following other vehicles. We did this because the controllers would not change lanes at times when it was necessary and possible because it perceived a dangerous situation.

BAT4 More agressive lane changing with the help of better checks to see if a lane change is safe and the improvement of the action which adjusts the throttle to achieve a certain acceleration.

There were many other control strategies which were tried, but this section highlights the general testing cycle that we use to develop viable controllers. After running a controller on the scenario library a number of times, we try a number of different techniques to improve the performance.

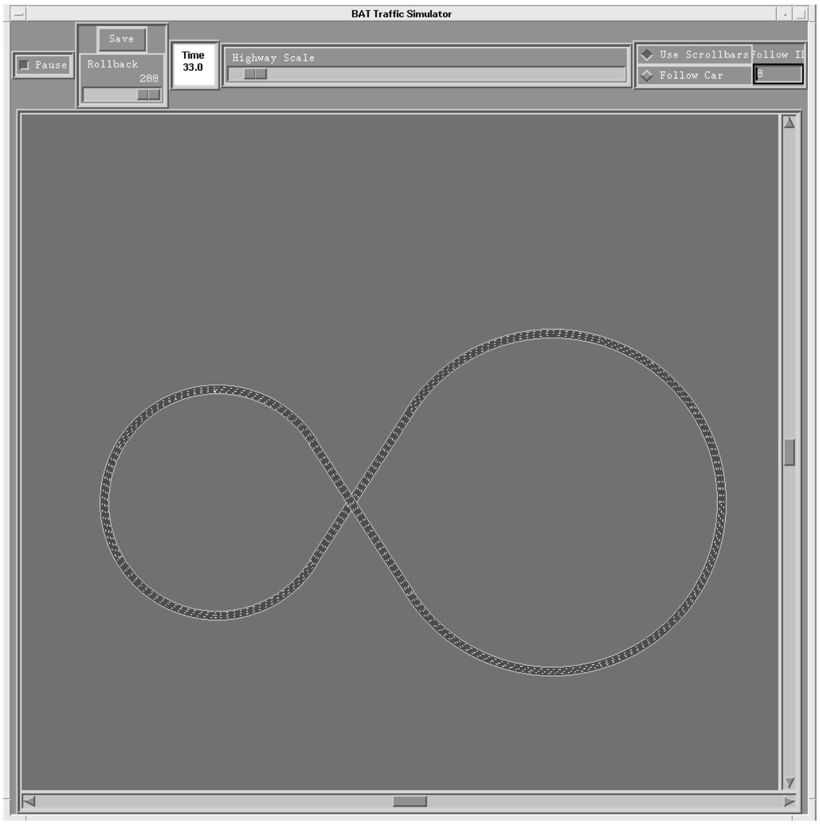

Figure 19: 2-loop track used to generate congested traffic.

6.3 Controller performance metric

The reinforcement learning algorithms of Section 5.2.3 use a reward function whose cumulative sum over a sequence of states gives an estimate of the driving quality exhibited. The reward function used for the BAT is shown in Table 1. Often, it is difficult to objectively compare the behavior of different controllers. For example, the mean time to failure fails as a performance metric because it does not take into account progress to the destination, excessive jerk by the controller, and many other factors. Since each controller may take a different path in a particular simulation run, it is important to give them some reasonably objective quantifiable measure by which they can be judged.

The simulator grades each controller according to this reward function. At every time step, a vehicle can be assigned reinforcement which can be used to improve the controller or just as a means of comparison. With this approach, we can work toward building the decision theoretic agent which drives so as to maximize utility. By changing values in the reward function, one can bring about different kinds of behavior. For example, giving more negative weight to excessive acceleration would favor a controller that produced very smooth driving behavior.

6.4 Controller performance

The hill climbing search described in Section 5.2.2 was used to improve the parameters for the PD control. Figure 20 shows a graph of the improvement achieved for a lane following control using hill climbing. Each

Incident	Reward or Penalty per time step	
progress/arrival	+\$0.0025	60mph
crash	-\$10.00 to -\$10000.00	
excessive lateral acceleration	-\$$(a^2)/150$	
excessive braking/acceleration	-\$$(a^2)/150$	
fuel	-\$0.0001	
time	-\$0.001	
speeding/too slow	-\$0.0001	65mph
	-\$0.003	75mph
	-\$0.012	85mph
straddling lane marker	-\$0.001	
too close	-\$0.1	delta t = 0.2s
	-\$0.002	delta t = 0.5s
off road	-\$0.1	

Table 1: **Reward function**: for each time step. The time step size is 0.1 seconds.

trial was run on the curved track described earlier. The graph shows the mean error from driving in one lane at a number of different velocities. The error is defined for lane following as the distance from the center of the lane summed over all the time steps. In this scenario, the controller had a maximum of 100 seconds to complete a 1700 meter course.

We tested the controllers (BAT0, BAT1, BAT2, BAT3, BAT4) over a number of scenarios.

1. Free flowing traffic on a straight road
2. Using the 2 loop track in Figure 19, there were three staggered slow ($5m/s$) vehicles which blocked three of the four lanes. This script generated congestion in the loop as the simulation continued to create cars in each lane at a rate of 2000 new vehicles per lane per hour. The controller had target of reaching the "fast" lane and continuing at its target speed ($20m/s$).
3. Using the simulated bay area highway network, the vehicle had to cross the highway through steady traffic (between 1000 and 2000 vehicles created per lane per hour) to its target highway.

The maximum length of each scenario ranged in length from 30 seconds to 300 seconds. The third scenario caused the most trouble becuse it was very difficult to find the gaps in the traffic to change into and then change with proper acceleration trajectory. Also, the other vehicles had their own goals, so the vehicle had to avoid them. None of the controllers modelled the state of the other vehicles, so they would, for example, not anticpate other vehicles changing lanes. The performance of the controllers measured by the above reward function is shown in Figure 21.

7 Future Work

The significant work that has been done thus far with the BAT project has yielded a system that allows us to test a variety of new control strategies in a variety of traffic scenarios. However, we would like to do some additional work to add even more realism to our simulator, facilitating the goal of deployment into real vehicles on real highways.

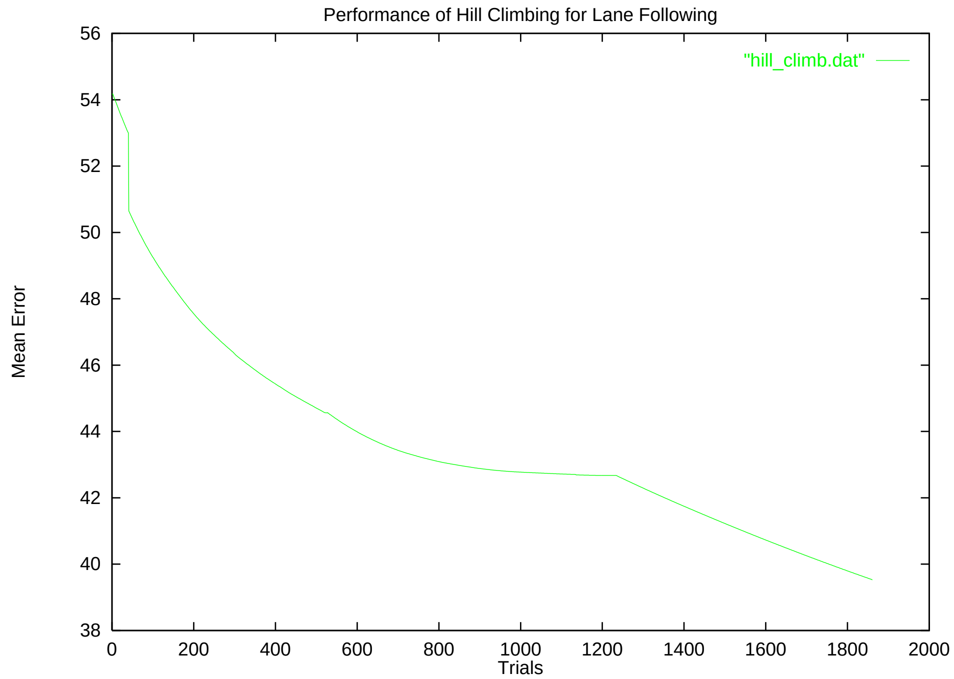

Figure 20: Improvement of lane following control through hill climbing search.

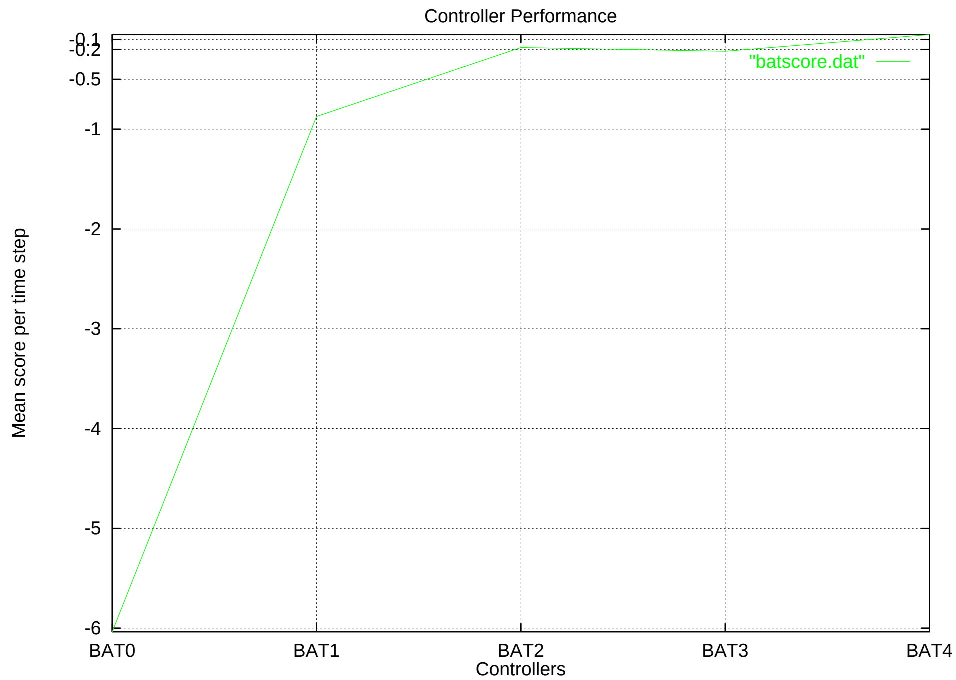

Figure 21: Controller improvement and refinement

The future directions of the BAT project include:

- Use the SmartPATH 3.0 Animator to produce simulated video input to a stereo vision system to create a realistic sensor module for the controllers.
- Release the BAT simulator to the research community, so that more scenarios can be created and there can be a means for comparing different control algorithms in different projects.
- Use dynamic probabilistic networks with continous variables. This will allow us to more accurately model aspects of the driver state that are continuous such as velocity and distance to a neighboring car.
- Use genetic programming to generate better low-level controllers. Genetic programming has had success in developing control laws for (nonlinear?) systems[Koza, 1992b]. For example, genetic programming was used to induce a control for backing a tractor-trailer truck into a spot in simulation[Koza, 1992a]. The program which we would be trying to induce would have as an input the same error variables as before and the output would be the control decision. Genetic programming has the advantage of being an iterative improvement algorithm which could start with our current form and parameters and eventually learn an arbitrarily complex nonlinear mapping.
- Apply hierarchical reinforcement learning techniques to driving as described in Section 5.2.3. Specifically, we would first like to use reinforcement learning to create low-level controllers for basic driving actions such as staying in the current lane and changing lanes. Once those are learned, we plan to use reinforcement learning to create high-level controllers whose output would be one of these basic driving actions given the traffic situation.
- Provide for modeling of specific AHS paradigms such as platooning—the scheme in which vehicles form a chain within a dedicated lane with very low inter-vehicle spacing and with the lead vehicle in charge of determing what to do.
- Add planning component to include goals such as desired destination along with driving parameters such as target speed and target highways. Conceptually, this level would lie on top of the high-level controllers created using reinforcement learning as described above.

8 Conclusions

The project has been successful in identifying the major technical issues to be resolved in constructing an autonomous vehicle controller based on probabilistic inference and utility maximization. Contrary to our expectations, several significant theoretical and algorithmic advances were required in order to create an inference system capable of handling vehicle monitoring in a real-time fashion. New methods were also developed for learning probabilistic models from data, and for learning control policies given reward/penalty feedback.

Our experience in hand-coding controllers for high-level control suggests that it is quite easy to attain a fairly high level of performance, equivalent to perhaps safe operation over a 40-mile trip. However, we require safe operation over, say, 100,000 miles or more. We believe that testing and refinement against an incident scenario library, combined with reinforcement learning techniques for controller optimization, will enable us to reach this goal; however, we do not yet have a mechanism for verifying that the goal has been reached, since each test would require about three months of simulator time.

Thus, we believe on the basis of our experience in this project that the *high level* driving problem can be handled satisfactorily, although much remains to be done in terms of real-time operation. So far, we have not addressed the interaction between low-level and high-level issues, most crucially how to handle failures of sensors and effectors. At present, for example, we do not expect the stereo vision system to be very reliable in detecting the existence, location, and speed of other vehicles. Further research is required to establish whether multiple-mode, redundant sensing is a satisfactory solution, or whether the control strategies can be adapted to drive safely even with sensors of dubious quality. Certainly, human sensing is far from perfect and human driving patterns are adapted to cope.

A Simulator Usage

A.1 Controller Format

How to create a controller:

1. Make a new file for your controller, e.g. drone.c. Here's where the actual code for the controller will be written.
2. Make a header file for that controller, e.g. drone.h. In here, put the declarations for the exported function for the controller. The only necessary exported functions are: the initializer, controller, restarter, and recorder.

   e.g.

   ```
   #ifndef _ctrl_drone_h
   #define _ctrl_drone_h
   void* drone_initializer(int id, int hwy, int lane_num, double velocity,
                           int argc, char** argv);
   void* drone_controller(void* data);
   void* drone_restarter(int id, char *info);
   void* drone_recorder(int where, void *data);
   #endif
   ```

3. Required Functions
   (a) Initializer
      - Called upon creation of the car and should initialize and return the controller data structure
      - ```
        void* drone_initializer(int id, int hwy, int lane_num,
                                double velocity, int argc, char** argv);
        ```

   (b) Controller
      - Called every time step with its control data passed as an argument. The controller should sense the environment through a sensor module and output its decision through one of the physics modules.
      - e.g. `void* drone_controller(void* data);`

   (c) Recorder
      - Called every time step to record data needed for rollback of system. There are two different modes specified by the `where` argument. One is when the recorder needs to output the controller state to the rollback file for restart, and the other mode is when it needs to make a separate copy of the state for keeping a number of time steps in memory, so one can "rollback" the simulation a number of ticks.
      - e.g. `void* drone_recorder(int where, void *data);`

   (d) Restarter
        ```

- Called up on a restart of a scenario. The restarter takes the aforementioned data from the recorder as input and initializes the controller state to the point where it was when it was recorded.
- e.g. `void* drone_restarter(int id, char *info);`

4. Sensing

   - The sensing routines give the controllers some accurate and consistent representation of the world state in the simulator
   - The information generally consists of information about the car itself and a list of the cars which are within its sensor range.
   - There are a number of different "levels" of sensing which can correspond to different models of inputs.
     - Direct access to simulator structures
     - Exact information about car's state along with data on each of the vehicles in visible range
     - Noisy version taking into account occlusion of other vehicles'
     - Simulated vision system using SmartPATH animation as vision input
   - The sense module also provides a number of useful routines such as the current simulation time, the distance to the next junction, and any other input which might be useful for the controllers.
   - Important Calls

     ```
     void* SEN_InitSenseData(int id, int level);
     void* SEN_GetSenseData(int id);
     ```

5. Physics (Vehicle Dynamics)

   - There are a number of different levels of physics which correspond to different models of dynamics.
   - The physics module also provides routines for any effect a controller wants to have on the environment such as setting a turn signal.
   - Important Calls

     ```
     void PHY_InitCarPhysics(int id, int level);
     void PHY_UpdatePhysics(int id);
     int PHY_set_decision(int id, double d_tire_ang, double d_accel, double d_brake);
     ```

6. Quit routines: `typedef void (*quit_routine_t)(void*);`
   e.g `void DroneQuitRoutine(void *data);` The call `SEN_SetCarQuitRoutine(DroneQuitRoutine)` means that `DroneQuitRoutine` is called whenever the controlled vehicle dies either due to a crash or some other calamity or the end of the simulation.

The drones which provide the traffic in the simulation have a number of stochastic parameters. Among the parameters are reaction time, the frequency of random lane changes, deviation from the target speed, following distance, and a number of other parameters.

The stochastic paramters are define in the script files with a mean and standard deviation. For example to say that a car should randomly change lanes (for no reason) on average once every 5 minutes with a standard deviation of two minutes, you'd put

```
stochastic-param random_change_time 5.0 2
```

By using these stochastic parameters and varying the goal highways of the drones, one can produce realistic traffic complete with traffic jams.

A.2 Scenario Saving and Replay

We define a "scenario" as a specific situation on a specific highway– the positions of a set of cars and their internal state as well as the arrangement of the highway and any other scenario configuraion.

The bat simulator allows users to save a highway scenario for later use. This is most useful when controllers do bad things, like crash or drive off the road. If we save the state of the world a few seconds before the crash, we can later restore that state (a "scenario") and run it with an improved version of the controller, that may be able to handle the situation in more reasonable fashion. In this fashion, a library of scenarios can be built up against which any controller can be tested.

The simulator periodically saves the state of the simulator, controllers, and vehicle dynamics to memory. The default interval for saving the state is once every second (10 time steps). The default amount of time recorded is twenty seconds (200 time steps). There are two ways to save the state of the simulator

1. During a scenario, the user can press the save button whenever they want to save the state of the system. The user is able to specify how far into the past the system state should be saved. Presumably, when an "interesting" situation occurs, one would want to save the state some time before the situation and replay the situation possibly after changing some control parameters. The amount that the simulator is rolled back can varied using a sliding control.

2. The simulator can also do automatic scenario detection. The user can add a command to detect various incidents of the form:

   ```
   detect <detect_type> <detect_event1> <detect_event2> ...
   ```

   where `detect_type` can either be ALL, a vehicle id number, or a type of controller and `detect_event` are optional arguments which give what events to detect. Specifying no `detect_event` means that the simulator will detect all possible events for that `detect_type`. Other possible events can be `CRASH`, `OFFROAD`, or `SCORE` which indicates that the performance score discussed in Section 5.2.3.

Three scripts implement this ability to save, restore and re-run scenarios, `save_scen2`, `get_scen`, and `run_scen`.

1. The `save_scen2` script

   The `save_scen2` script saves scenarios and works in the following way:

   It takes one argument, the "rollback" file created when the scenario was noticed. It is called in this way from the `$BAT_SIM/sim` directory:

   ```
   > save_scen2 ../scen/<rollbackfile>.rbk
   ```

   Example:

   ```
   > save_scen2 ../scen/curves230.rbk
   ```

   It makes a directory named `$BAT_SIM/scen/<rollbackfile>.dir`. for the new scenario.

   It then copies this rollback file and also copies the script (".bat"), highway configuration (".config"), and tcl default files ("bat.tcl" and "ms.tcl") files into this directory. The names of the script and config files are retreived from the rollback file, where they are stored when the rollback file is created.

   The script also makes links from this directory to a library files in a shared library directory. This was conceived as an alternative to copying all library files each time a scenario is saved, and saves considerable disk space.

   Finally, `save_scen` puts a script in this directory named `get_scen` which is a means for retrieving the scenario later on.

2. The `get_scen` script

   The `get_scen` script restores and re-runs scenarios in the following manner:

   `get_scen` takes any number of parameters, each being the full pathname of a controller object file. For example

   `> get_scen $BAT_SIM/ctrl/lane_follower.o $BAT_SIM/ctrl/bat/control.o`

   or

   `> get_scen`

   The object files are then compiled together with the library files created by `save_scen` to put together a new binary, which is located in the `$BAT_SIM/scen/<rollback>.dir/bin` directory. The controllers in this new binary will act either a) as they acted before the scenario was saved or b) as the controllers in the object file, if they were specified.

   `get_scen` also puts a script, `run_scen` into the bin directory which takes care of all the details of re-running the scenario.

3. The `run_scen` script

   After running the `get_scen` script, all is in place to re-run the scenario. Change directory into bin, and execute the command `run_scen`, i.e.

   ```
   > cd $BAT_SIM/scen/<rollbackfile>.dir
   > run_scen
   ```

   After this, the scenario will re-run.

A.3 Script File Format

How to make a bat/script file (file with extension .bat):

1. Vehicle Specifications

   The .bat file allows you to specify what types of vehicles enter the freeway network. It allows you to specify the type of vehicle (including controller type, size, and color), the time the vehicle enters the freeway, and optionally the path the vehicle is to take.

   There are two methods for creating cars:

   (a) The command 'make-car-type' is formatted as follows:

   ```
   'make-car-type' <originating hwy> <lane> <start time>
      <end time> <interval> <velocity> <controller type> <other args>
   ```

   'make-car-type' creates cars at a regular time interval specified by `<interval>`, and starting at time `<start time>` and ending at time `<end time>` as measured from the start of the simulation. All the cars created via this command have the same controller type (`<controller type>`) and move at the same velocity (`<velocity>` in meters per second). `<other args>` represents arguments that controllers may take when cars of that type are initialized. For example the

Sbat controller takes in a series of directions which include target speed, target lane, and target highway.

Examples:

```
#                    hwy   ln  beg  end  intvl vel   cntrl other args

       make-car-type  Hwy1  1   0.0  15.0 3.0   20.0  LaneFollower
       make-car-type  Hwy2  2   0.0  0.0  1.0   35.0  PerceptFollower
       make-car-type  Hwy1  2   30.0 30.1 10.0  23.0  Stall 82 -2
       make-car-type  Hwy1  3   0.0  4.0  1.0   45.0  PerceptFollower
       make-car-type  Hwy1  1   0.0  20.0 4.9   30.0  Sbat 30.0 4 Hwy2right
```

(b) To create cars in mass, use the 'generate-cars' routine. The 'generate-cars' routine allows you to create a distribution of cars starting at one location of the freeway system. Unlike the command make-car-type, the command generate-cars does not necessarily generate identical cars. You must designate the freeway name and lane number, density (average number of cars in that lane per hour), and the earliest and latest times at which cars should be generated. Other required information includes probability distributions over vehicle characteristics such as color, type, controller_given_type, length_given_type, width_given_type, speed_given_controller, and paths. "controller_given_type" is an example of a conditional distribution, for which the user must specify a probability distribution over controllers for each possible type of car listed in the "type" distribution.

All the information in a generate-cars command must be on one line.

This is the format of the generate-cars command:

```
'generate-cars -highway <hwy> -lane <ln> -density <#cars-per-hour>
 -beg <time> -end <time> -colors <% color1 % color2>
 -types <% type1 % type2>
 -controller_given_type <type1: % controller1 % controller2
                        <type2: % controller1 % controller2>
 -length_give_type <type1: % length1 % length2 type2: % length3>
-width_given_type <type1: % width1 % width2 type2: % width3>
-speed_given_controller <controller1: % speed1 % speed2>
                        <controller2: % speed3 % speed4>'
-paths <% path1 % path2 >
```

Here are some of the available variable choices:

```
-color:  red, LightBlue, bisque, purple
-type:  normal_car, motorcycle, light_truck, heavy_truck, bus
-controller:  LaneFollower, Sbat, Stall, PerceptFollower
```

Paths are created using the add-path command:

```
'add-path <pathname> { hwy1 lane1 hwy2 lane ...}'
```

<pathname> is the unique name for the path. Inside braces, one gives a sequence of highways and lane numbers that signify the path. For example, the above path would start in <lane1> of "hwy1" and go through <lane2> of "hwy2." Any highway named in a path must be connected via a junction to both the previous and subsequent highways in the path.

The easiest way to understand the generate-cars command is to follow an example. Here is one example:

```
    #lane 1 on Hwy1
        generate-cars -highway Hwy1 -lane 1 -density 1200 -beg 0.0 -end
105.0 -colors 30% red 30% LightBlue 20% bisque 20% purple -types 60%
normal_car 2% motorcycle 27% light_truck 10% heavy_truck 1% bus
-controller_given_type normal_car: 40% LaneFollower 60% Sbat
motorcycle: 50% LaneFollower 50% Sbat light_truck: 40% LaneFollower
60% Sbat bus: 50% LaneFollower 50% Sbat heavy_truck: 60% LaneFollower
40% Sbat -length_given_type normal_car: 33% 3.5 33% 4.0 33% 4.5
motorcycle: 100% 2.0 light_truck: 15% 3.5 45% 4.0 40% 4.5 bus: 100%
7.0 heavy_truck: 10% 4.0 70% 4.5 20% 7.0 -width_given_type motorcycle:
100% 1.0 normal_car: 33% 1.5 33% 1.75 34% 2.0 light_truck: 50% 1.75
50% 2.0 heavy_truck: 50% 2.0 50% 2.5 bus: 100% 2.5
-speed_given_controller LaneFollower: 25% 23 50% 24 25% 25 Sbat: 25%
23 50% 24 25% 25
```

The above command generates cars on lane 1 of "Hwy1" at a rate of 1200 cars per hour starting at the beginning of the simulation and ending at 105.0 seconds after the simulation has started. The distribution over colors specifies that 30% of the cars should be red, 30% should be light blue, 20% should be bisque, and 20% should be purple.

2. Camera Specifications

   The cameras are used to simulate traffic data collection using surveillance cameras.

   To add a camera to the highway, use the command 'make-camera' with the following format:

   ```
   'make-camera <x-coordinate> <y-coordinate> <camera-heading>'
   ```

   Example:

   ```
   make-camera   22.50    0.50  1.18
   make-camera 3982.00  181.67  1.86
   ```

   `<camera-heading>` is measured in radians counterclockwise where zero radians means the camera is pointed directly to the right.

   Cameras can also be added using the mouse while the simulator is running. (see the section on the graphical display for more information on cameras)

3. Miscellaneous parameters

   This section allows you to set miscellaneous parameters such as drawing intervals, thresholds, following distances, etc.

   For each parameter begin on a new line and type

   ```
   'set <parameter name> <parameter value>'
   ```

   To set the name of the configuration file to be used with your bat file, type 'set-config-file <name>.config'.

The simulator will know to use `<name>.config` without needing to specify it on the command line. The command line `-c` argument will override this parameter setting.

There are many more parameters that can be set in the .bat file as well as the file .batrc.

A.4 Configuration File Format

How to make a config file (configuration file with extension .config):

The configuration file is a file which allows the user to create a highway system in which the cars will drive.

A highway is a set of interconnected segments, each with a specified number of lanes, length, and radii of curvature.

There are 3 parts to the config file.

1. Segment configuration
2. Junction definition
3. Segment geometry

I. SEGMENT CONFIGURATION

This portion of the config file defines each of the segments of the highway system in the following manner:

A. The first line at the top of the section is as follows:

```
<number of highways> <lane width [meters]>
```

`<number of highways>` is the total number of highways in the system. `<lane width>` is the width of all lanes in all highways in meters.

B. Segment Format: (each is separated by a space)

```
line 1: <highway name> <length> <@default section length> <@illegal distance>
line 2: <number of lanes>, <@autolanes>,<@entrances>, <@exits>.
```

(Any item with a '@' sign in front of it is not used currently. Generally, zeros are used as placeholders for any unused fields.)

Below is an example segment configuration section:

```
SECTION CONFIGURATION
34 4    // Number of highways, lane width(U)
// For the lines below, use the following format.
// hwy name, length(U), default section length(U), illegal distance(U)
// number of lanes, @autolanes (U) @entrances, @exits
Onramp1pre 100 100 0
4 0 0 0
```

```
DumbartonOnramp 100 100 0
1 0 0 0
Dumbarton 3700 3700 0
4 0 0 0
Onramp1 1000 1000 0
4 0 0 0
```

The first line indicates the start of the segment configuration section. The second line indicates that there are 34 highways in this system and each lane is 4 meters wide. The next two lines indicate that a 100 meter long highway named "Onramp1pre" with 4 lanes is to be created.

II. SECTION JUNCTION

This portion describes how the highways interconnect. The meeting points are called 'junctions'.

For each junction, indicate which lanes of which highways meet. Begin each junction description with the header

```
'JUNCTION' <junction name>
```

(Any word in quotation marks needs to be typed verbatim into the file.) After the header, there needs to be one line of the following form for each lane in the junction:

```
'LANE' <junction lane number> <origin highway> <lane number in origin highway>
       <destination highway> <lane number in destination highway>.
```

The junction lane number represents the lane number of the junction itself. The other entries represent the two highways that meet and the lanes that meet. For example, if "HwyOld," a two lane highway, and a one lane onramp called "Onramp1" meet and become "HwyNew," a three lane highway, the junction would be specified as follows:

```
JUNCTION 0
LANE 1 HwyOld  1 HwyNew  1
LANE 2 HwyOld  2 HwyNew  2
LANE 3 Onramp1 1 HwyNew  3
```

In the example, we see that lane 1 of Onramp1 (its only lane) becomes lane 3 (the rightmost lane) of HwyNew. Similarly, lanes 1 and 2 of HwyOld continue and just become lanes 1 and 2 (the two left lanes) of HwyNew.

III. SECTION GEOMETRY

Each segment is made up of subsegments which are either arcs or straight lines. For each major segment, one first needs to declare how many subsegments there will be, and then define each of the subsegments individually.

A. The major segment declaration (first line) is of the following form:

```
<highway name> <# of subsegments> <init X> <init Y> <init Z> <heading>
```

`<highway name>` must be a highway name given in the CONFIGURATION SECTION. The fields `<init X>`, `<init Y>`, `<init Z>`, and `<heading>` are required for the first highway in a sequence of connected segments. For each subsequent segment, do not put values for these four fields. Also, the Z-coordinate is meaningless in the BAT simulator. It is present in order to maintain some consistency with the SmartPATH 3-D simulator configuration files).

B. Subsegment information (lines after each major segment declaration):

If the subsegment is an arc, then the format of the line is:

```
<"A"> <length> <radius of curvature> <direction of
turn> <@gradient>
```

If the subsegment is straight, then the format is:

```
<"L"> <length> <@gradient>
```

`<length>` and `<radius of curvature>` are measured in meters. The radius is measured from the left edge of the road.

Example geometry configuration section:

```
SECTION GEOMETRY
// <hwyname> <#subsegments> <x> <y> <z> <heading>
// straight subsegment: <"L"> <length> <@gradient>
// curved subsegment: <"A"> <length> <radius of
curvature> <direction of turn> <@gradient>
// Since "Onramp1pre" is the first in a sequence of
// connected segments, we have, after the number of
// segments (1), the starting point of the highway
// (0,0,0) and the initial heading (1.0 radians).
Onramp1pre 1 0 0 0 1.0
L 100 0
DumbartonOnramp 1
A 100 100 R 0
Dumbarton 3
A 7 50 R 0
L 179 0
A 24 50 R 0
Onramp1 1
L 1000
```

References

[Benz, 1993] Thomas Benz. The microscopic traffic simulator as (autobahn simulator). In *Proceedings of the 1993 European Simulation Multiconference*, Lyon, France, 1993.

[Binder *et al.*, 1997] John Binder, Daphne Koller, Stuart Russell, and Keiji Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, to appear, 1997.

[Dean and Kanazawa, 1988] Thomas Dean and Keiji Kanazawa. Probabilistic temporal reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 524–528, St. Paul, Minnesota, 1988. American Association for Artificial Intelligence.

[Eskafi and Khorramabadi, 1994] Farokh Eskafi and Delnaz Khorramabadi. Smartpath user's manual. Technical Report 94-02, California PATH/UC Berkeley, 1994.

[Forbes *et al.*, 1995a] J. Forbes, T. Huang, K. Kanazawa, and Stuart Russell. The batmobile: Towards a bayesian automated taxi. In *International Joint Conference on Artificial Intelligence*, 1995.

[Forbes *et al.*, 1995b] J. Forbes, T. Huang, K. Kanazawa, and Stuart Russell. The batmobile: Towards a bayesian automated taxi. In *International Joint Conference*, 1995.

[Forbes *et al.*, 1995c] Jeff Forbes, Tim Huang, Keiji Kanazawa, and Stuart Russell. The BATmobile: Towards a Bayesian automated taxi. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, August 1995. Morgan Kaufmann.

[Hedrick *et al.*, 1993] J. Hedrick, D. McMahon, and D. Swaroop. Vehicle modeling and control for automated highway systems. Technical Report UCB-ITS-PRR-93-24, California PATH/UC Berkeley, 1993.

[Huang *et al.*, 1994] T. Huang, D. Koller, J. Malik, G. Ogasawara, B. Rao, S. Russell, and J. Weber. Automatic symbolic traffic scene analysis using belief networks. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 966–972, Seattle, Washington, August 1994. AAAI Press.

[Kanazawa *et al.*, 1995a] K. Kanazawa, D. Koller, and S. Russell. Stochastic simulation algorithms for dynamic probabilistic networls. In *Uncertainty in Artificial Intelligence*, 1995.

[Kanazawa *et al.*, 1995b] Keiji Kanazawa, Daphne Koller, and Stuart Russell. Stochastic simulation algorithms for dynamic probabilistic networks. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, Montreal, 1995. Morgan Kaufmann.

[Koza, 1992a] John R. Koza. A genetic approach to finding a controller to back up a tractor-trailer truck. In *Proceedings of the 1992 American Control Conference*, volume III, pages 2307–2311, Evanston, IL, 1992. American Automatic Control Council.

[Koza, 1992b] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[Lehner and Sadigh, 1993] Paul E. Lehner and Azar Sadigh. Two procedures for compiling influence diagrams. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence (UAI-93)*, Washington, D. C., 1993. Morgan Kaufmann.

[Niehaus and Stengel, 1994] A. Niehaus and R. F. Stengel. Probability-based decision making for automated highway driving. *IEEE Transactions on Vehicular Technology*, 43(3), 1994.

[Parr and Russell, 1995] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *IJCAI*, 1995.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, California, 1988.

[Peng, 1992] Huei Peng. *Vehicle Lateral Control for Highway Automation.* PhD thesis, UC Berkeley, 1992.

[Russell *et al.*, 1994] Stuart J. Russell, John Binder, and Daphne Koller. Adaptive probabilistic networks. Technical Report UCB/CSD-94-824, Computer Science Division, University of California at Berkeley, July 24 1994.

[Russell *et al.*, 1995] Stuart Russell, John Binder, Daphne Koller, and Keiji Kanazawa. Local learning in probabilistic networks with hidden variables. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1146–52, Montreal, Canada, August 1995. Morgan Kaufmann.

[Tomizuka *et al.*, 1995] M. Tomizuka, J.K. Hedrick, and H. Pham. Integrated manuevering control for automated highway system based on a magnetic reference/sensing system. Technical Report UCB-ITS-PRR-95-12, California PATH/UC Berkeley, 1995.