Simulation and Verification of Autonomous Route Planning Behavior

by

Amrik S. Kochhar

S.B. Computer Science and Engineering S.B. Mathematics, M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 19, 2010

[June 2010]

Author ______________________

Department of Electrical Engineering and Computer Science

May 19, 2010

Certified By ______________________

Professor Munther A. Dahleh

M.I.T. Thesis Supervisor

Accepted By ______________________

Dr. Christopher J. Terman

Chairman, Department Committee on Graduate Theses

Simulation and Verification of Autonomous Route Planning Behavior

by

Amrik S. Kochhar

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 21, 2010

Abstract

This thesis presents a new dynamic traffic simulator called DUST, used to investigate new foundational theory for autonomously reconfigurable cyber-physical systems in the presence of unexpected disruptions. We focus on transportation networks to develop our methodologies since it is a prime example of a cyber-physical system that is characterized by distributed decision-making and is prone to unexpected disruptions. This new simulator investigates the route choice behavior of cars in an urban environment. It allows subjects to participate as drivers in virtual city environments directly as well as artificial intelligence algorithms. It provides a platform for experimentation with various incentive mechanisms and information dissemination protocols that are critical for emergency planning during disruptions.

Thesis Supervisor: Munzer A. Dahleh

Title: Professor

Contents

1 Introduction

1.1 Problem

This thesis is part of a larger investigation for new foundational theory for autonomously reconfigurable cyberphysical systems in the presence of unexpected disruptions. In particular, for transportation networks agents can be characterized by distributed decision making, and is a great example of a system that is prone to disruptions. Current methods for examining the effectiveness of current theories and is usually characterized by generating predictive models of car agent behavior by examining collected data and fitting parameters to the distribution modeled. However this is often inadequate - the data is often incomplete in terms of the metrics captured, such as time elapsed on a particular road, or traffic conditions on each road at a particular time of day. Other data lacks particular accuracy to make route-choice tracking easy such as cellphone based information, or simply does not cover a large enough area. Also, with natural disasters and terrorist attacks there simply is insufficient data and it is unrealistic to plan observance of such events.

Thus to help develop the theory we envision the creation of a virtual city testbed, where traffic flows can be simulated and various route choice behaviors can be implemented in artificial intelligence. Furthermore, we would like to allow direct human interaction with the simulation, to be able to record and compare real human behavior to that predicted by a theoretical model.

One of the most important applications of this project is that simulating real-time disruptions allows us to study traffic flow under adverse conditions. For example, a natural disaster or a terrorist attack that affects city streets can be simulated. The results of such a simulation can be used to test how different road systems behave in response to unexpected disruptions and improve emergency management protocols. This thesis focuses on the concrete implementation of the server component and of the learning artificial intelligence system in the simulator.

1.2 Previous research

This project is at the center of a multidisciplinary approach that includes city planning, modeling and software development. We use data analysis of map and traffic data combined with theoretical models of route choice and traffic as a basis for the test bed's programmatic behavior.

In addition to gathering our own GPS data from taxis we have gathered data from cell phones from the CarTel project[1]. For theoretical models about the natue of traffic equilibria that we wish to look further into we have [2] and [3]. These theeorize and abstract route choices and traffic flow. The software we developed here uses this information to generate a real time simulation and view state.

For our map data, we use OpenStreetMap[4]. This provides a freely available liberally licensed map with an easy to parse XML format that makes it ideal for our needs. It identifies road segments, intersections as well as giving very accurate coordinates. Thus the road network topology can be inferred from this data and imported directly into the simulator.

Two currently available traffic simulators are TRANSIMS[5] and TransCAD[6]. However neither of these support real time simulation. That is we allow for clients to remotely interact with the vehicles in the simulation and be able to influence the outcome directly. Furthermore, we allow an experimenter to change the simulation dynamically by introducing disruptions into the road system which allows for interesting situations such as simulating natural disasters, that are difficult to observe in real life. We would also like a good interface for directly modifying and subclassing AI agents' behavior programmatically, for investigating different route choice behaviors. We would also eventually like to experiment with microtrolling which is also not supported directly by these simulators.

1.3 Objectives

The goal of the Virtual City Testbed project overall is to provide a simulator to study traffic behavior in response to adverse conditions. We would like the traffic test bed to utilize road path data and develop route choice models that can control automated vehicles. We also aim to observe the way real people make driving decisions and how these change with unexpected disruptions, and to be able to present hints to users about optimal path decision changes if the same conditions rearise. In particular, we aim to create and improve a new testbed that supports real time disruptions. We allow real human agents get to interact with autonomous agents to study their route choice behavior. This thesis describes the implementation of the simulation engine and programmatic infrastructure to support different AI agent models and experiments, as well as the base model itself.

2 Methodology

2.1 System Overview

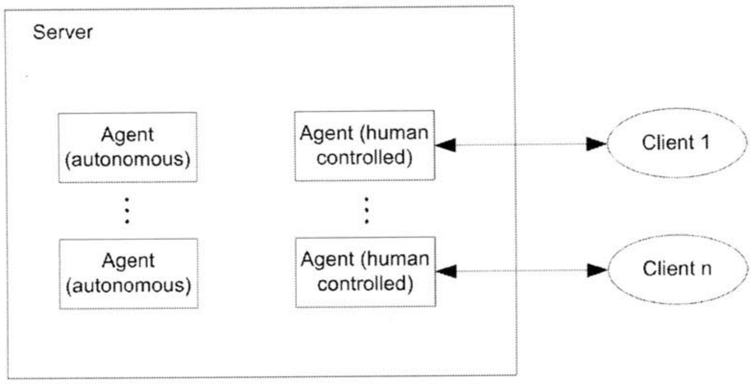


Figure 1: Client/server model

The simulation system is designed as a client/server model. The state and progress of the simulation are held on a centralized server, where multiple clients can connect and view the simulation as it unfolds. We will not discuss the client side, and instead will focus on the game server and learning components.

2.2 Server

The implementation of the server is based on a previous C++ implementation[7]. It is broken down into several packages. The `dust.server.engine` package holds the game engine fundamentals. The `dust.server.lib` package holds basic simulated objects such as Cars, Roads and Intersections. The `dust.server.learning` package holds all of the implementation of the learning strategy for autonomously controlled agents and their interfaces. The `dust.server.signal` package holds the definition and implementation of the signal/signal handler pattern used by the simulation to generate and handle various events. Finally, `dust.server.main` holds the experiments that are run.

2.2.1 Game engine

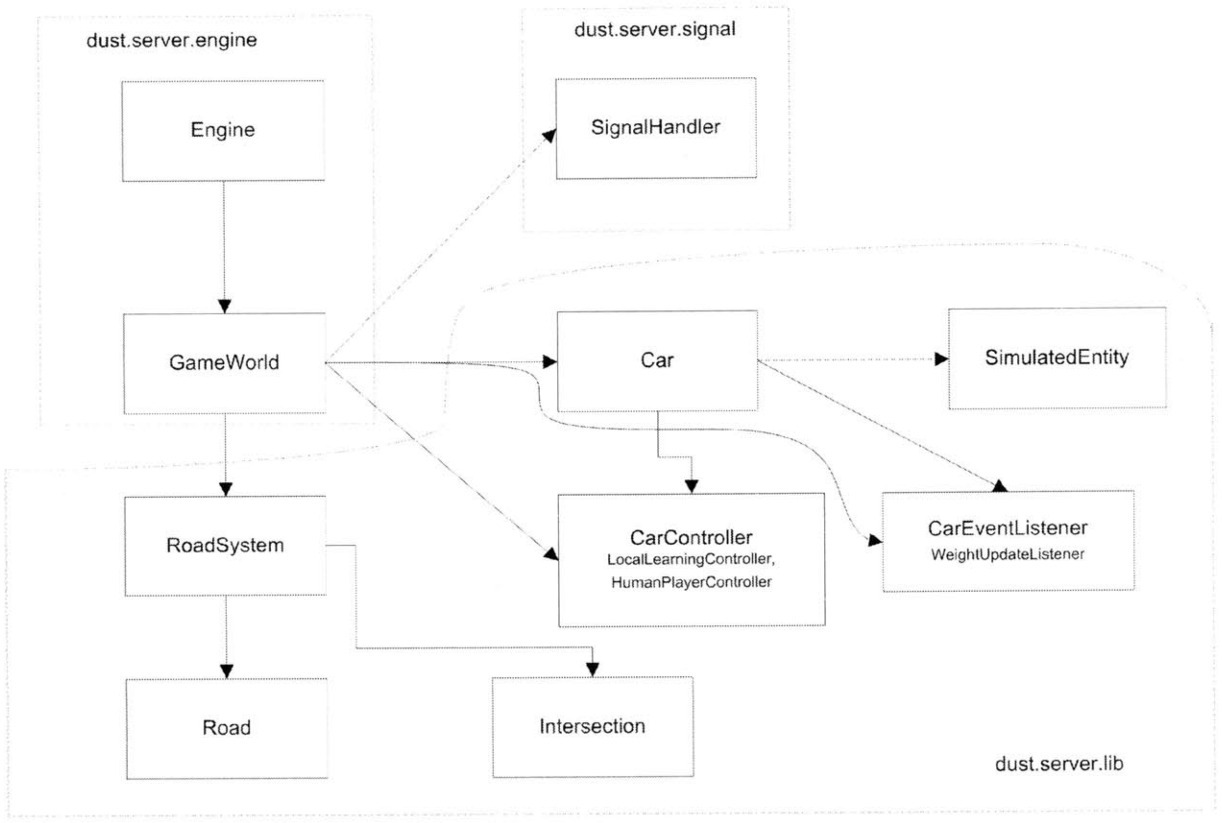


Figure 2: Server system class diagram with dependencies

The `Engine` class is the encapsulating class of the simulator. It holds an instance of the current GameWorld, which is the main data structure that stores the road system model as described below in section 2.2.2. It keeps track of the current round of the simulation and is responsible for handling the server timestep forward. It does this by presenting a method to be called sequentially called `runRound()` which steps the simulation forward by Δt the current simulation timestep specified by the configuration. Thus an experiment is flexible and can either continuously step the simulation forward, specify a round limit, or run batches of rounds. It also holds an instance of the networking server used to communicate with clients.

`GameWorld` is the container class that contains all of the data structures that pertain to the current road system and world. The GameWorld represents a layer of abstraction between the Engine and the Cars themselves by presenting an `update` method that will iteratively update all the Cars and step them forward in the simulation. Finally, the GameWorld is a SignalHandler, which will be discussed in 2.2.4. It is a singleton class, and contains an instance of the current road system as well as a list of all the present cars in the simulation. It also has the random number generator and stores the current listener for weight update events. A GameWorld is constructed using a factory pattern, where it initializes the road system to that specified by an OSM file using an external OSM file loader. It constructs the traffic flow by adding cars to the network. It does this by constructing a new Car instance and placing it on a particular road and then assigning it a destination.

Since initially all cars are autonomous agents controlled by AI, it is responsible for instantiating the correct CarController (in this case a LocalLearningController) and setting the controller as well as the listener on the car. The GameWorld is also responsible for assigning and revoking cars to human players, specified by the RemotePlayerEventListener interface. It does this by selectively maintaining a mapping of which cars are under AI control. It also passes player input received from human clients to the cars that they control.

2.2.2 Road system model

The road system model is based on Roads and Intersections. Roads are edges of the traffic graph that maintain a one-to-one correspondence with road segments in the OSM file. Intersections are the corresponding vertices of the traffic graph. Roads have a start and end intersection, a name, a speed limit and whether the road is bidirectional or one-way. Roads also know their own length as well as the cars that are currently on it, which is important for simulating traffic slowdown and spacing. The Intersections have their ID as well as a GeographicLocation, and a list of roads that touch this Intersection vertex. Both the Road and Intersection class support functionality for finding the nearest Car, which locates the nearest Car instance distancewise from this Intersection or from a particular point along a Road. This helps indicate whether a Car is being blocked by another one while traversing a Road and has to initiate spacing. It is important to note that since Roads are bidirectional, entities that traverse them have a boolean direction that is true if they travel from start to end, and false for the other direction.

The actual RoadSystem itself is constructed as a list of intersections and roads using a factory pattern. When a RoadSystem is constructed, an OSM file loader is used to first parse the OSM file. Next, the constructor iterates over the structure the OSM loader presents and constructs new

Intersections according to the IDs. It then constructs roads from the list of Roads and assigns end points by doing a reverse lookup on the OSM ID of each of the start and end points.

2.2.3 Simulated entities

A SimulatedEntity is one that must be updated at every timestep of the simulation. It presents an `update` method that must be invoked once per simulation step and updates the entity state by simulating forward a change in time of Δt.

A Car is a SimulatedEntity that is controlled by an agent in the traffic system. A Car has a controller which directs it which turn to take at each Intersection, an event listener that is used to record weight updates as well as a reference to the road it is on. The most important part of the Car class is the update method. A car firstly must keep track of the time it has spent on the current road segment. It does this by keeping a running total $\sum \Delta t$ of all the timesteps it has experienced on the current Road. Next, the Car determines whether there is a Car in front of it on the current Road going in the same direction. If there is then it computes the distance to the leader in front, otherwise it computes the distance to the next intersection. Now if the distance to the leader is less than a particular constant corresponding to a two second gap then the car slows down. If the distance is within 5 units (virtually next to each other) the Car behind stops.

When a Car arrives at an Intersection by traversing sufficient distance it must decide which of the next Roads to take. To do this it first initially picks any unblocked road that is not the same as the current one as its default choice, to allow for cars with no controller. Next it asks its CarController which of the next Roads to take, with according parameters as the Car itself, the road it is currently on, its current direction, and the current time spent. The CarController interface specifies that it must return a Road and a boolean direction that the Car can travel next to. The Car then sets this as the next Road, overriding the default choice as appropriate. Next if there is a listener installed on the Car, it is called back with the parameters of the Car the old road, the new road, the direction of travel and the time spent. This allows a listener to be able to record the time elapsed and use it for later updates to the simulation such as increasing edge weights of roads that have consistently slow time of traversal.

2.2.4 Signals

There is an implementation of a Signal/SignalHandler pattern in the simulation server. This is used for triggering edge weight updates in the traffic system asynchronously. The Update-PushSignal is generated by an experiment for example to indicate that it would like the currently stored edge weight updates to be pushed into the traffic graph. The GameWorld instance is the SignalHandler that dispatches incoming signals to the appropriate method. When an Update-PushSignal is handled in the GameWorld it triggers the GlobalPathInfo to receive all the updates that the CarEventListener has recorded previously. It then clears all the updates in the listener queue. This approach allows for easy extensibility as new signals can be easily added and triggered anywhere. For example a signal could be created and triggered when a particular car enters an intersection or we could trigger a signal to indicate the start of an earthquake that disrupts particular roads in sequence.

Signals are also used to control traffic flow. When modeling a flux of cars through the network we generate a `SourceSignal` which indicates to add a car to the road network. It takes as parameters a road, a direction and a destination and it places a Car at that location and initializes its controller with the target destination. The corresponding signal to remove a Car from the network is called a `SinkSignal`. It takes as parameters the Car to be removed from the network. A SourceSignal is usually generated by the experiment framework perioudically to control when and where Cars enter the road network. This is usually specified by a rate of arrival constant which counts the number of rounds of simulation between injection of cars. However, SinkSignals are usually generated automatically, for example when a Car reaches its destination in a CarController a SinkSignal is generated and immediately processed (using Java's built in exception mechanism) to sink the Car at its destination.

2.2.5 Human control

The human control on the server-side is implemented via interfaces. The `MessageServer` handles all the actual network communication. The simulation server implements the RemotePlayerEventListener interface, since we require functional notification of players joining and leaving the server. This is used to assign cars to human players and revoke these assignments later when they disconnect. Thus human players take control over an AI car and the control returns to the AI when they are done. The GameWorld class implements the RemovePlayerEventListener since it contains the data structure holding the current Car instances.

When a human player takes control of a car a `HumanPlayerController` instance is created. This is a CarController that determines the next road a human-controlled car will take depending on the input received by the server from the corresponding client. When a player decides an action for the Car he is controlling, a message is sent from the client containing an instance of the `PlayerInput` class, whichspecifies the angle as well as desired direction vector of the player. The GameWorld instance dispatches this to the correct HumanPlayerController, which then decides the next candidate road to take by the following algorithm which is run when the car reaches an intersection and has a CarController callback initiated.

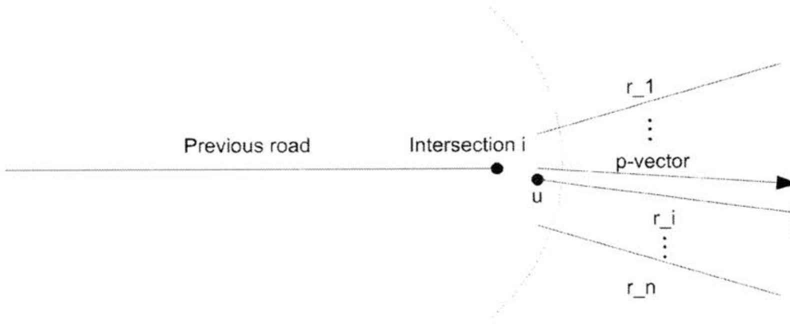


Figure 3: Diagram showing vector path selection under human control.

1. Let $\vec{p}$ be player input direction. Compute the unit vector $\hat{p}$ in the same direction.

2. For each road R_i that fans out from the intersection i, compute the vector in that road's direction. Let the end points of the road be say (u, v). This is necessarily reversed if the road under consideration may be traversed in the reverse direction, in which case the travel would start at the road's end. Assuming without loss of generality that the road is not reversed, compute the directional road vector $\vec{r_i} = (v_x - u_x, v_y - u_y)$. Note that these are positional vectors with their x and y components coming from their GeographicLocations.

3. Compute the unit vector $\hat{r_i}$ for each of the road direction vectors.

4. Compute

$$r_j := \left\{ j : \max_i \hat{p} \cdot \hat{r_i} \right\}$$

That is the road vector that maximizes the unit dot product between its direction and the player input direction. Also record whether the road is reversed or not.

This works since the angle between two vectors u and v is given by $\cos\theta = \frac{u \cdot v}{|u| \cdot |v|} = \hat{u} \cdot \hat{v}$ and so maximizing this product minimizes the angle between the two directions, thus choosing the best target road. A custom Vector2 class was implemented for this which provided the unit vector, length and dot product functions.

2.3 Learning

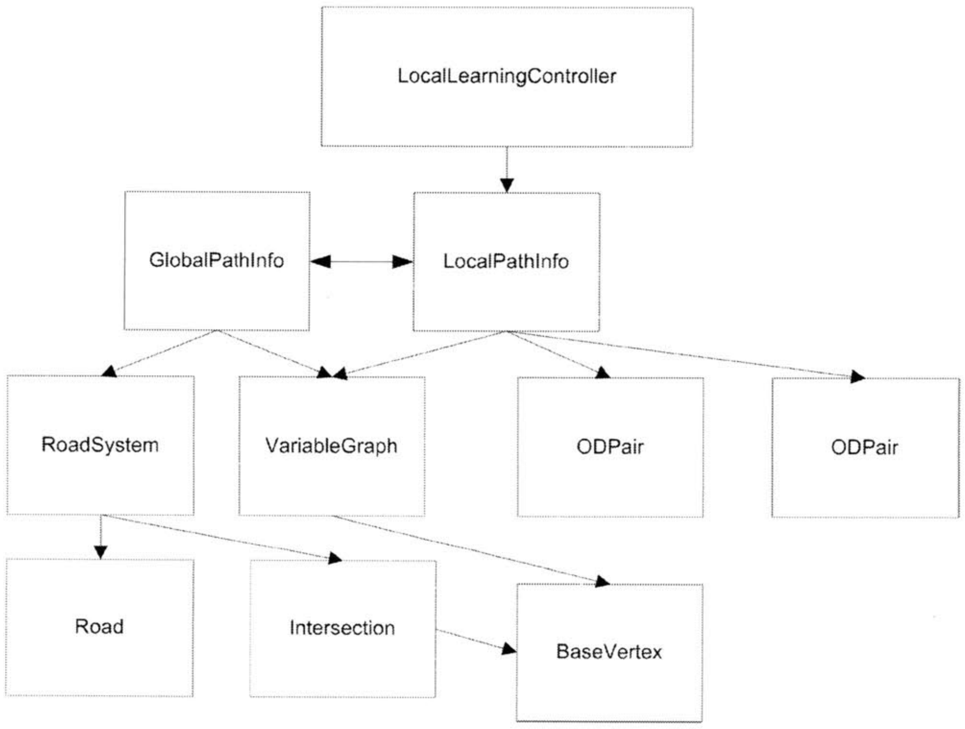


Figure 4: Learning subsystem interclass dependencies.

The learning subsystem is characterized by a division of information between *global* and *local* information. Agents when initialized are given a particular origin and destination pair (OD pair) that indicates their start and end locations. Agents start at a particular node and navigate the road system until they reach their target destinations. Upon reaching their destination they will attempt to renavigate the road mesh to their start, and thus repeat indefinitely. This simulates a flow of traffic between one particular OD pair in the network.

When a learning agent is initialized he is given the latest global snapshot of the pathing information available to him. This is copied into his local pathing information and will be used for all pathing decisions afterwards. Whenever an agent traverses a particular road in the traffic system it records an update of that particular edge weight. These updates and the protocol that is used to push them are described in section 2.3.3 in more detail. Thus the simulation evolves over time as agents traverse different roads, record and push different edge weights for these roads thus affecting the route choice of this agent as well as others.

2.3.1 Implementation

`GlobalPathInfo` is a data structure that holds the sum total of all the globally available pathing information available at any particular point in time during the simulation. It is initialized with a filename that indicates the OSM file to load. This loading procedure first constructs the associated RoadSystem and a traffic Graph is created on the RoadSystem by corresponding edges in the graph with Roads in the network, and vertices in the graph by Intersections. The edge weights are initialized according to the formula:

$$w(\text{road } i) = \frac{\text{length}(i)}{\text{speed\_limit}(i)}$$

This corresponds to the intial estimated conditions that cars will travel as fast as possible and there is no other traffic on the roads they traverse. The GlobalPathInfo contains a path information mapping - a map of OD pairs to localized information for each OD pair known as `ODPathInfo`. This ODPathInfo has a list of pairs of the form (path, probability). Thus GlobalPathInfo contains a map of origin-destination pair to a path distribution, which is a generic representation of a model for route choice selection. The distribution is initialized according to the number of paths required per ODPair - if we have k paths then the probability vector is initialized as $[1/k, 1/k, \ldots, 1/k]$ ie. uniform.

`LocalPathInfo` is the corresponding data structure that holds local information. This is a necessary extra layer due to the restriction of visibility of local traffic information and updates between agents. There is one instance of LocalPathInfo for every agent there is in the system, as opposed to GlobalPathInfo which has a singleton instance. LocalPathInfo holds a copy of the traffic graph where locally visible changes to the edge weights are applied. It also holds localized traffic information as well in the form of a localized cost function at the edges. This is a breadth-first search through the vertices starting at the current location with a fixed depth for the most up-to-date edge weight. This is used to model how much "look-ahead" a vehicle can have on local traffic conditions and make appropriate judgements as opposed to pathing chosen at the very beginning of the journey.

The `LocalLearningController` implements the CarController interface. It defines the behavior of an AI controlled simulated vehicle and must be able to decide upon road changes when a car arrives at a particular intersections ie. pathing decisions are reduced to turns at viable intersections.Pathing is based on sorting through a collection of possible *moves*. A move is a potential next road at an intersection and the associated remaining path. Since the LocalPathInfo contains locally visible traffic conditions this can also be utilized to provide a better ranking mechanism. We can apply a heuristic cost function in order to rank moves. Let M_i denote the ith move and let $r_1, r_2, \ldots, r_n$ denote the road segments along the path, w be the edge weight function and l be the local weight function. The current heuristic function is given by:

$$h(M_i) = l(r_1) + \sum_{j=2}^{n} w(r_j)$$

To preserve the probabilistic nature of the route selection, individual Move instances can inherit the probabilities associated with the Paths they came from. Now we can describe the road choice selection algorithm as follows:

1. If at start, sample a list of k appropriate paths and corresponding distribution.

2. Compute the paths that contain the current car's location by searching along the path. The beginning part of the path is set as the prefix.

3. Look ahead to determine the next possible moves by taking the path and removing the prefix

generated in step 2.

4. Compute the heuristic function on each move as described above.

5. Select the move with the lowest cost heuristic.

A `CarController` must implement a method called `roadChanged`. This method is called whenever a Car reaches an Intersection and must make a decision which is the next Road it will take. It takes as parameters the Car under consideration, the old Road the Car just came from, the default next Road in case no selection is made, the boolean direction the car is traveling, and finally the time spent on the old Road. It must return a Road and a boolean direction as a pair to indicate which direction the Car should go next.

2.3.2 K shortest paths

We still need a mechanism for choosing some subset of paths between an origin and a destination from the set of all possible paths. Our currente strategy is to select the k shortest paths through the road network from origin an destination and use these. This is convenient since this gives us a well-defined set of paths and the parameter k can be externalized for further experimentation.

The algorithm of choice for computing the k shortest paths is known as Yen's algorithm[8]. An important property of this algorithm is that it finds loopless paths which is crucial for sensible path generation as it is unrealistic for a real agent to drive in a needless loop through the road system. It works by first computing the shortest path between the origin and destination (via Dijkstra's or another suitable algorithm) and then expanding candidate paths as deviations of the current path that have the same prefix up to the ith node. The next shortest path is found on the basis of shortest paths that are longer. For a more detailed discussion on the updated algorithm used see [9].

The library implementation used for our simulator is one described in [11]. This was chosen for its simplicity in use and that it has an easy-to-use Java interface. However some intricacies arise. For example, the library causes vertices to be reduplicated whenever a new Graph is created, which prevents updates from being applied directly to the Graph. This has been worked around by

Beacon Hill
© 2010 Google
Image MassGIS, Commonwealth of Massachusetts EOEA
Gray Buildings © 2008 Sanborn
42°21'26.29" N 71°04'08.77" W elev 68 ft
Google
Eye alt 743 ft

Figure 5: One example of a shortest path generated shown in Google Earth[10].

modifying the library directly. Another problem is that there is no external way to adjust weights on edges in a Graph object on the fly, and this was done by adding a method `set_edge_weight` to the Graph to modify the underlying data structure containing the edge weights directly.

One problem that arose during implementation was that the vertices used by the Graph have their own internal ID and these IDs must exist in a purely sequential order starting from 1 to N. This creates difficulties since the original OSM file has unique but nonsequential IDs for the intersections. Thus during the construction phase of the Graph we must create a bidirectional mapping from the actual OSM IDs to a sequential numbering and use the latter for edge construction. For an intersection i let a_i denote the OSM ID and let b_i denote the sequential vertex ID. The solution is to actually construct an importable textfile of the Graph listing the edges (i, j) as (b_i, b_j), constructing the Graph and taking advantage of the fact that Intersections subclass Vertex to restore the mapping from the b_i's to the a_i's.

Notice that the k shortest path algorithm, and indeed most graph algorithms, describe paths as an ordered collection of vertices. However for our application, pathing decisions are made at the vertices (in our case Intersections) themselves. Hence we must have a vertex to edge decision conversion to make decisional sense: intersections must know the roads that they are connected to as well as roads knowing their endpoints so we can do a reverse lookup when necessary. If a car reaches an intersection t, it can examine the neighboring intersections. Let us call this set the *neighborhood* of t, $N(t)$. Since pathing and moves are generated on a vertex basis, tha actual output of the road selection algorithm is a vertex $j \in N(t)$ such that

$$j := \{i : \min_{i \in N(t)} h(M_i)\}$$

That is the j that minimizes the movement heuristic cost. We can look up a road that corresponds to an edge between t and j but there is a slight ambiguity in the case of when there is more than one road between them, so for now we simply choose the road edge with minimal weight.

2.3.3 Edge weight updates

The distributed nature of many agents pathing through a road system causes traffic conditions to vary widely over time on various roads. Thus we need a mechanism for keeping track of the latest edge weights and sending this information as updates to agents. Each agent records the functional weight of the edge it just traversed in the network. In our case this is the time elapsed on the previous road when arriving at an intersection.

`WeightUpdateListener` is a CarEventListener that is notified whenever a car arrives at an intersection and records the agent's reported edge weight. These weights are immediately applied directly to the agents local traffic graph. However these updates are not immediately applied to a shared graph, and so other agents' pathing that takes place will still use the old edge weights. A push event is used by the simulation to indicate that updates should be applied to the corresponding graph. When a push event is triggered by a signal being emitted, the updates are immediately applied to the shared graph. If there is more than one update for a particular road then the latest update is used and overrides any previous update by any other agent, for simplicity. This is useful for testing out different information dissemination protocols. In the multi-staged design, updates are first pushed between one particular OD pair, and then pushed globally later on, whereas in the prototype the OD pair layer is omitted and just the global push is used. Similarly, the latest updated weights are pulled whenever an agent embarks on a new or return journey from its source or destination to take advantage of newer information.

2.4 Infrastructure

The base implementation of the simulator is in the Java programming language[12]. Java was chosen for the second version of the simulator due to its ease of development over C++ whereas still retaining most of the performance benefits of a compiled language. Certain infrastructure was added to make the simulator much easier to modify and run on different platforms. In particular a logging system based on Log4j[13] was added to make debugging much easier and allow simulation output to be redirected to files for later perusual. The logger allows for different log levels to be set, for example show all debug output or show only errors, in addition to sectioning output by class and file. Simulation configuration files were externalized so that parameters to the simulation can be changed and tweaked without having to recompile. For example, which OSM file to load and where to output data are controlled through external configuration files. The Config class allows one to retrieve externally specified booleans, doubles and integers, whether specified in the command-line, in the Ant built script or in a file.

The build system is built using Ant[14]. This is useful since it allows us to have runtime detection of the current platform and load the appropriate native libraries dynamically. This allows the simulation to run seamlessly on all supported platforms (Mac, Windows and Linux), which was a problem with the C++ version of the simulator. It also makes setting up the project for development much more easily as all the necessary libraries are already included and no paths need to be set, and new ant targets can be defined to run experiments repeatably.

3 Results

3.1 Experiment 1

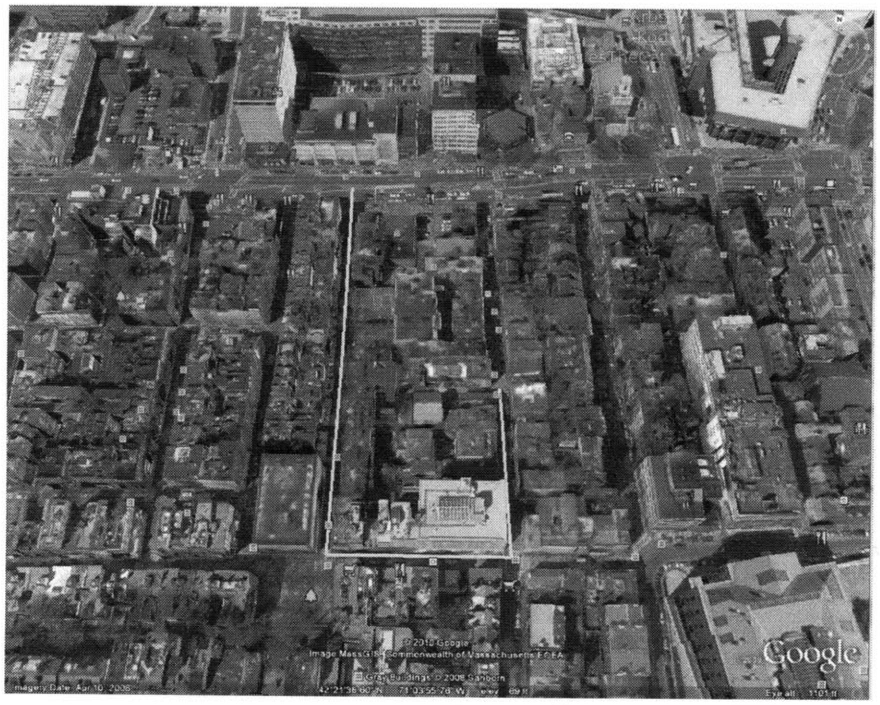
In our experiment we have a simulated flow of one AI controlled agent that moves continuously between an origin and destination pre-specified in the simulation. They are chosen randomly but with a fixed random seed of 121 to give the experiment repeatability. These locations corresponds to Beacon Hill in downtown Boston. The value of k, the number of shortest paths to generate is fixed at 5. The AI controlled agent moves initially according to the shortest path. This is since the route choice selection algorithm should choose that with the minimum move cost heuristic and the initial conditions assume there is no traffic flow on the graph. However the time elapsed on each road edge is deliberately skewed to make it appear as if the car is traveling very slowly ie. experiencing a lot of traffic on these segments. The simulation is run for 100000 rounds with the default step rate. No update is sent until the 50000th round ie. halfway through the simulation, when an UpdatePushSignal is created and sent to the GameWorld. we observe the route change behavior in the AI controlled agent.

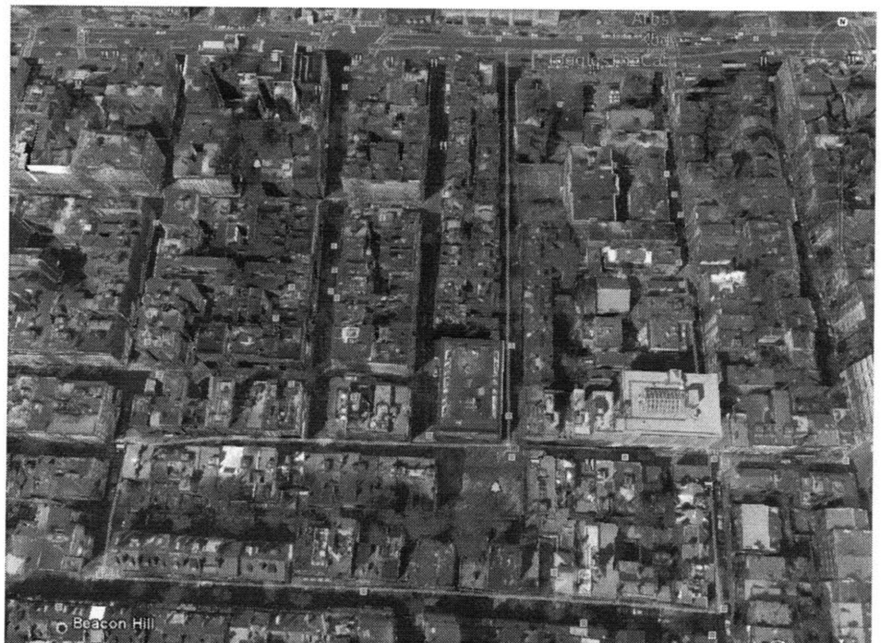
The first picture shows the path taken during the first 8 traversals, and the second picture the path with the next 7 traversals. During traversals 0 through 7 before the updated weights are pushed we see that the agent attempts to navigate what is ostensibly the shortest path between the origin and the destination. However during traversals 8 through 14 the path is drastically altered as the agent uses the newly updated weights and must take a potentially longer path around very slow (and thus high weight) edges.

3.2 Experiment 2

In this experiment our goal is to investigate the dynamical properties of the simulated system. We set up a net traffic flow between a source and a sink and observe the traffic density on a specified road network. Consider the following road network set up:

We construct a flow of traffic by initializing some rate of arrival of cars entering a particular vertex. They are queued on the road containing this vertex. They then attempt to traverse the road


(a) Initial autonomous agent shortest path



(b) Updated path reflecting new edge weights

Figure 6: The change in pathing of an autonomous agent in response to adverse simulated traffic conditions.

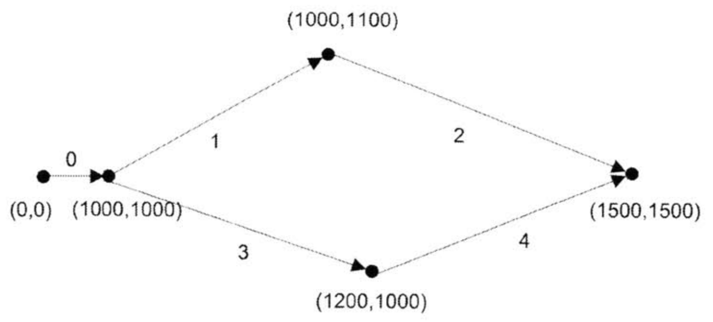
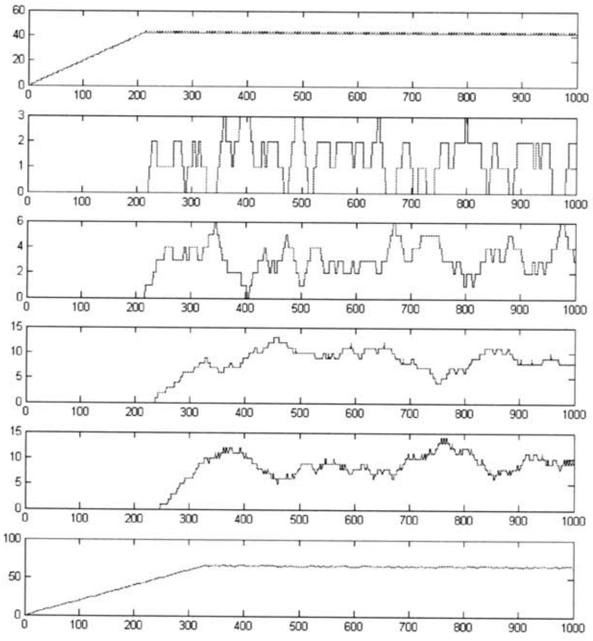


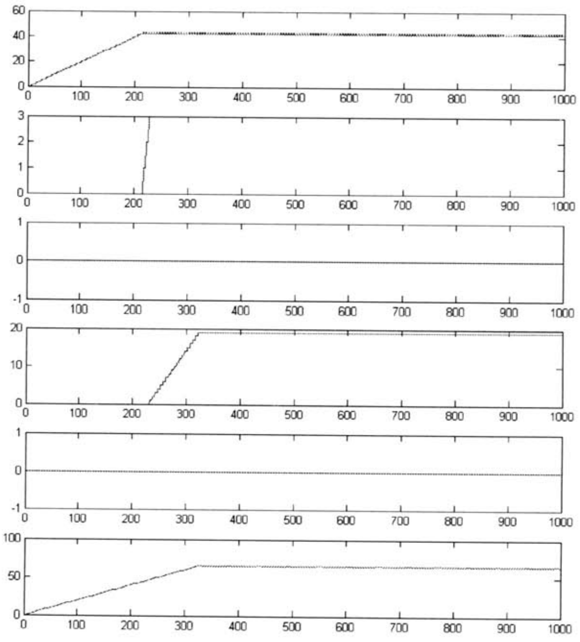
Figure 7: Network used in this experiment. The roads are labeled 0 through 4 and intersection coordinates are indicated.

network until they reach their destination after which they are sunk immediately. We can examine the congestion levels of this flow by looking at the number of cars on a few links along the network. In this experiment's case we can simply look at all the road links to get a full picture of the congestion. We examine the dynamical behavior for both heuristically choosing the best possible move at every step versus choosing randomly among the best k moves, and examine the differences. The simulation was run for 1000 seconds to see the asymptotic state. Edge weight updates were pushed every 10 seconds and each measurement corresponds to 1 second of simulated time elapsed. The first link is constructed deliberately long so as to be able to hold a significantly long queue of cars waiting to enter the diamond test portion of the graph. Every tick on the t axis repsents one second of simulated time. The y axis for each of the graphs is simply the number of cars on a particular link. Each graph generated has 6 subgraphs. The first 5 correspond to the number of cars on each of the links 0 through 4, and the last subgraph is the total number of cars in the road network. This increases every time a car is sourced and decreased whenever it is sunk. The top left figure represents the random choice selection algorithm with an arrival rate of 1 car every 5 seconds. The top right represents the heuristic best choice with the arrival rate 1 car every 5 seconds. The bottom left is the random choice with arrival rate 1 per 10 seconds and the bottom right is best heuristic choice with arrival rate 1 per 10 seconds. These different arrival rates are referred to as 1/5 and 1/10 respectively.

One aliasing effect noticed was that once a segment gets a bad reputation by being particularly slwo for the last car to traverse it before the update push event, then it may never be utilized again. This effect does not occur with the random choice method. Notice how the utilization of links in the random choice method has slight oscillations due to natural variation in the number of cars

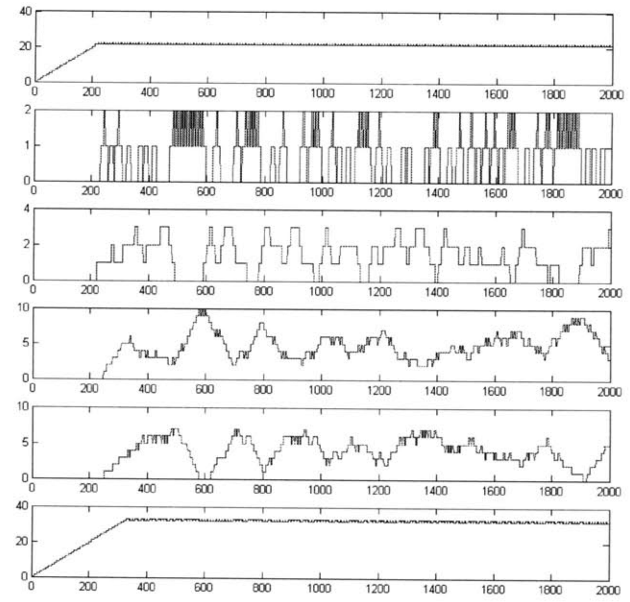


(a) Random move choice

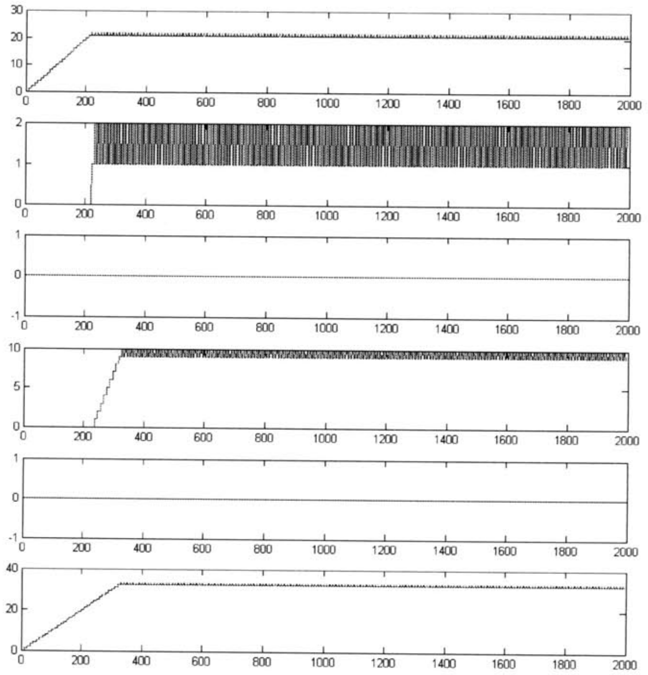


(b) Best heuristic move choice

Figure 8: Road link density in the network for random and best first heuristics with rate of arrival $\alpha = 1/5$



(a) Random move choice



(b) Best heuristic move choice

Figure 9: Road link density in the network for random and best first heuristics with rate of arrival $\alpha = 1/10$

on each of the two legs, and eventually the simulator gets saturated as indicated by the bottom graph. The oscillations are a little more prominent on the 1/10 random graph. However for the best path heuristic it simply chooses the shorter of the two paths and due to successful navigation it sticks with utilizing only 2 of the segments and not using the other two at all. Notice that both heuristic methods, random and shortest, get the road network saturated at the same level, 70 cars for 1/5 and 35 cars for 1/10 respectively.

4 Conclusion

In this thesis we have presented the foundations for an extensible traffic simulator with an implementation of a learning strategy for autonomously controlled agents. We have demonstrated effective route choice modeling and path generation and demonstrated a workable experimental framework for subsequent testing and expansion. Future work would be to implement new probabilistic learning algorithms within the framework and to generate larger scale tests that involve multiple human agents interacting at the same time within the simulation. We would also like to have a system to script disruptions in a controlled manner and this could perhaps use the signal system introduced in 2.2.4.

5 Appendix

5.1 Configuration options

Property	Default value	Description
dust.server.onewaystreets	true	If false, treat all streets as bidirectional
dust.server.random.seed	1337	Random seed to use for simulation
dust.server.osmfile	data/boston_subset.osm	OSM file to load road system from
dust.server.timestep	1	Change in time to simulate in each simulated step
dust.server.num_cars	64	Number of cars to initialize in the simulation
dust.server.kspexport.osm	data/boston_subset.osm	OSM file to load road system from
dust.server.kspexport.csv	input_odpairs.csv	Comma delineated file of OD pairs to find paths between
dust.server.kspexport.kval	5	Number of shortest paths to compute
dust.server.kspexport.output	output_paths.csv	KSP exporter destination filename

5.2 Experimentation framework

The experimentation framework is designed to make it easy to add experiments. To do so create a new main class under the `dust.server.main` package. Construct a new instance of the Config class from the command line arguments and set it as the global config. Next, create a new Engine based on the server osm file and specify the number of cars. Do any other setup required and then to run rounds of the simulation simply call the method `Engine.runRound()`. This can be constructed in a while loop to make it easy to run many rounds at once. Signals such as UpdatePushSignals can also be generated and sent to the simulation here.

5.3 K Shortest Paths Exporter

An exporter for the k shortest paths algorithm was written to help another part of the project with MATLAB analysis. It takes as input a comma separated value or CSV file that contains on each line a new origin destination pair. So inputting 500 OD pairs would create a CSV file with 500 lines each containing two long ID values. This input file is specified as a configuration option as described in section 5.1. The value of k is also specified in the configuration file. The output file is another CSV file, where each line is a path. The line takes the form of origin, destination, vertex 1 (which is the origin), vertex 2, and so on until we get to the destination. Each of these values is the intersection ID specified by the OSM file. Thus if we let n be the number of od pairs and k the number of paths we desire we expect to have nk lines of output, each an individual path. It is also possible to output paths in KML (Google Earth) format using the KMl export function provided.

References

[1] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen K. Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.

[2] D. Fudenberg and D. K. Levine. Learning-theoretic foundations for equilibrium analysis. 2008. Working paper.

[3] J. R. Marden, G. Arslan, and J. S. Shamma. Joint strategy fictitious play with inertia for potential games. *IEEE Transactions on Automatic Control*, 54(2):208–220, 2009.

[4] Mordechai (Muki) Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008. `doi:http://dx.doi.org/10.1109/MPRV.2008.80`.

[5] R. J. Beckman, C. L. Barrett, K. B. Berkbigler, K. R. Burris, B. W. Bush, S. D. Hull, J. M. Hurford, P. Medvick, D. A. Kubicek, M. Marathe, J. D. Morgeson, K. Nagel, D. J. Roberts, L. L. Smith, M. J. Stein, P. E. Stretz, S. J. Sydoriak, K. Cervenka, and R. Donnelly. Transportation analysis simulation system (transims) - the dallas-ft. worth case study, 1997.

[6] Caliper Corporation. Transcad: a geographic information and transportation planning system. `http://www.caliper.com/`.

[7] Oleg I. Kozhushnyan. Virtual city testbed. Master of engineering thesis, Massachusetts Institute of Technology, 2010.

[8] Jin Y. Yen. Finding the k shortest loopless paths in a network, 1971.

[9] Ernesto de Queiros Vieira Martins, Ernesto Queir, Vieira Martins, Marta Margarida, and Marta Margarida Braz Pascoal. A new implementation of yen's ranking loopless paths algorithm, 2000.

[10] Google. Google earth: geographic information visualizer. `http://earth.google.com`.

[11] Yan Qi. An implementation of k-shortest path algorithm. `http://code.google.com/p/k-shortest-paths/`.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[13] Apache Foundation. Log4j: Apache logging services for java. `http://logging.apache.org/log4j/`.

[14] Apache Foundation. Ant: a pure java build tool. `http://ant.apache.org`.