Iterative Route Planning for Modular Transportation Simulation

Bryan Raney* and Kai Nagel†
Dept. of Computer Science, ETH Zürich
CH-8092 Zürich, Switzerland

April 2, 2002

Abstract

The TRANSIMS (TRansportation ANalysis and SIMulation System) project is a large scale transportation system project produced by Los Alamos National Laboratory for transportation planning. In TRANSIMS, all processes are represented on the microscopic level. These processes range from decisions of individuals about their daily activities all the way to signal operations and traffic movements. TRANSIMS consists of several modules, some of which are listed here:

- *Route planner*, which generates travel plans for each driver.
- *Micro-simulation*, which executes all plans simultaneously and in consequence computes the interaction between different travelers, leading e.g. to congestion.
- *Feedback:* The above modules are interdependent. For example, plans depend on congestion but congestion depends on plans. This is solved via an iterative method, where an initial plans set is slowly adapted until it is consistent with the resulting travel conditions.

As part of the eventual goal of implementing the TRANSIMS software for all of Switzerland, we are running simulations on a test-case with the Switzerland transportation network. We use a similar simulation framework as found in TRANSIMS, but with our own, simpler versions of the three modules.

We discuss the operation and interaction of these modules, and bring to light a combined flaw in our route planner and feedback modules. This flaw initially caused several unrealistic simulation results, such as freeways being avoided by vehicles in favor of lower-capacity roads. We illustrate several improvements made to the modeling logic of the modules in an effort to correct these problems, and compare simulation results from the various methods.

We also discuss the results of our most substantial improvement, which is the addition of a database that gives each driver a "memory" of its past routes from earlier iterations, plus the performance of those routes. When a new plan-set is generated, each driver chooses a route from those in its memory, based on their relative performance. This solution appears to be very robust, because it does not depend on having a route planner that works perfectly all the time.

1 Introduction

There is an emerging consensus that large scale transportation simulations consist of several cooperating software modules, some of them being:

- **Traffic simulation module** – This is where travelers move through the street network by walking, car, bus, train, etc.
- **Modal choice and route generation module** – The travelers in the traffic simulation usually know where they are headed; it is the task of this module to decide which mode they take (walk, bus, car, bicycle, ...) and which route.

*raney@inf.ethz.ch
†nagel@inf.ethz.ch

- **Activity generation module** – The standard cause why travelers are headed toward a certain destination is that they want to perform a specific activity at that location, for example work, eat, shop, pick someone up, etc. The activity generation module generates synthetic daily plans for the travelers.

- **Life style, housing, land use, freight, etc.** – The above list is not complete; it reflects only the most prominent modules. For example, the whole important issue of freight traffic is completely left out. Also, at the land use/housing level, there will probably be many modules specializing into different aspects.

- In addition, there need to be **initialization modules**, such as the synthetic population generation module, which takes census data and generates disaggregated populations of individual people and households. Similarly, it will probably be necessary to generate good default layouts for intersections etc. without always knowing the exact details.

The above modules interact, and the interaction goes in both directions: for example, (the execution of) plans lead(s) to congestion, yet (the expectation of) congestion influences plans. Any large scale transportation package needs to resolve this logical deadlock in a meaningful way.

Reality seems to approach the issue of feedback by a slow system-wide learning process: People pre-plan major pieces of their life (like when and where they work) a long time in advance and normally only re-adjust small pieces of their schedules when needed [1]. More precisely, they pre-plan and re-adjust on many time scales, where the time scale is related to the magnitude of the adjustment: workplaces and home locations are re-adjusted on time scales of several years, while the decision to make a detour to buy some ice cream may happen within seconds. In consequence, a simulation system is faced with two challenges:

1. Modeling adaptation and learning on all time scales – In principle, a transportation simulation should simulate several thousand days in sequence, and the decisions of the individual people should unfold on their particular time scales as pointed out above. In particular, travelers should be able to replan while en route. While this sounds simple in principle, it is difficult in practice, because one wants to avoid a large monolithic software package and thus to separate the traffic flow simulation from the strategic decision-making of the travelers. This becomes particularly relevant for parallel transportation simulations, since now the strategic planning needs to be separated from the traffic simulation also for performance reasons. This is not the topic of this paper; see [2, 3] for more information.

2. Behavioral realism vs. fast relaxation – In practice, simulating several thousand days in sequence is difficult to do because of computational resource limitations. It is also questionable if this would yield useful results without a deep understanding of the learning dynamics. As a reaction to this, mathematical modeling of transportation scenarios, as well as of economics in general, in the past has relied on the notion of a Nash or User Equilibrium (UE). As is well known, in a UE no traveler can improve by unilaterally changing her/his behavior. The advantage is that this prescribes a state of the system and it does not matter how the computational system finds it – as opposed to a realistic modeling of the transient learning dynamics. Today, we however increasingly recognize that socio-economic systems do not operate at a User Equilibrium point; for example, for the housing market it is assumed that the system is permanently in the transients [4].

This second point is the focus of this paper. Our approach to the problem is to design a framework which admits all the different views to the problem. That is, the framework should as well converge to the User Equilibrium (assuming it is unique and an attractor – this is a difficult discussion but again outside the scope of this paper) as it should allow for experimentation with different behavioral hypotheses. We entirely concentrate on day-to-day replanning although our results will also apply to within-day replanning. In particular, we will demonstrate that the introduction of an agent database, which keeps track of agents' past strategies and their performances, will greatly improve both plausibility and robustness of the system.

Throughout this paper we use the term *agent* to refer to an entity within the simulation capable of making decision about its actions (such as the route to take from point a to point b). Since our simulation does not yet

involve land use or other non-transportation activities, an agent is presently equivalent to a *traveler*, a person using the transportation network.

The structure of this paper is as follows: Section 2 describes the specific modules we are using in this study. Section 3 introduces the traffic scenarios we are applying those modules to. Following that, Sec. 4 describes some results from the day-to-day replanning of our feedback system, which turned out to have some implausible implications. We continue the section by describing some alterations we made to the feedback mechanism to try to resolve the problems, and the results of those changes. Next we present in Sec. 5 the agent database, a completely different and more robust approach to solving the problems encountered in the previous section. We finish with conclusions in Sec. 6. For reference, we have provided Appendix A, which lists some of the source code used in our framework and with the agent database.

2 The Modules

The modules which are important for this study are the traffic micro-simulation, the router, and the feedback mechanism, which controls the interaction between the micro-simulation and the router.

2.1 Queue Micro-Simulation

As a traffic micro-simulation we use an improved version of a so-called "queue simulation" [5]. The improvements refer to an implementation on parallel computers, and to an improved intersection dynamics, which ensures a fair sharing of the intersection capacity among incoming traffic streams [6]. The details of the traffic simulation are not particularly important for this paper; we expect many traffic simulations to reproduce similar results. The important features are:

- Plans following. The feedback framework generates individual route plans for each individual vehicle, and the traffic simulation needs to have travelers/vehicles which follow those plans.

  This implies that the traffic simulation needs to be microscopic, that is, all individual travelers/vehicles are resolved. Beyond that, it does however not prescribe the dynamics; everything is possible from smooth particle hydrodynamics where particles are moved according to aggregated and smoothed quantities (e.g. [7, 8]) to virtual reality micro-simulations (e.g. [9]).

- Computational speed. We need to run many simulations of 24-hour days – usually about 50 *for a single scenario*. This means that a computational speed of 100 times faster than real time on a network with several thousands of links and several millions of travelers is desirable. Our queue simulation demonstrates that this is feasible.

- Simulation output. The framework needs a certain type of simulation output to function. These outputs are simple and do not require sophisticated programming skills or a sophisticated output subsystem of the micro-simulation (as opposed to, say, Ref. [9]). These requirements are that the traffic simulation outputs (i) the time every time a vehicle/traveler leaves a link, and (ii) a dump of the locations of each vehicle/traveler in the system in specified intervals of time. The first one is the information from which link travel times are computed; the latter is in fact necessary for debugging and visualization only.

- Congestion build-up and queue spillback. Although this is not a requirement for the framework in general, the results of the present paper depend on the fact that congestion normally starts at bottlenecks (i.e. where demand is higher than capacity), but then spills backwards into the system and across intersections. Once such congestion is there, it takes a large amount of time to resolve it; in fact, if there are N vehicles in a queue upstream of a bottleneck and the capacity of the bottleneck is C, then the amount of time to clear the queue is N/C. The model should reflect this, *and* it should reflect that physical space that the queued vehicles occupy in the system.

2.2 Router

In addition, we need a router, i.e. a module that generates paths that guide vehicles/travelers through the network from a given origin to a given destination. In addition, the vehicles/travelers have starting times, and the router needs to be sensitive to congestion in the sense that it tends to avoid congested links.

The router we have used for the present study is based on Dijkstra's shortest-path algorithm, but "shortness" is measured by the time it takes an agent to travel down a link (road segment) in the network. These times depend on how congested the links are, and so they change throughout the day. This is implemented in the following way: The way a Dijkstra algorithm searches a shortest path is by expanding, from the starting point of the trip, a network-oriented version of a wave front. In order to make the algorithm time-dependent, the speed of this wave front along a link is made to depend on when this wave front enters the link.

That is, for each link l we need a function $c_l(t)$ which returns the link "cost" (= link travel time) for a vehicle entering at time t. This information is taken from a run of the traffic simulation. In order to make the look-up of $c_l(t)$ reasonably fast, we aggregate over 15-min bins, during which the function is kept constant. That is, for example all vehicles/travelers entering a link between 9am and 9:15am will contribute to the average link travel time during that time period.

2.3 Feedback

Finally, we need the feedback mechanism to couple router and traffic simulation. Initially, we plan all trips based on free speed travel times, and feed the traffic simulations with those plans. From then on, every time a traffic simulation run completes, the route planner uses the traffic simulation output to update the travel-time (cost of utilization) associated with each link in the network. After the route planner updates its view of the network, it generates new plans for a subset (typically a randomly selected 10%) of the drivers, and the entire updated plan-set is fed back into the micro-simulation for another run. We repeat this process as many times as necessary (about 50) until the system "relaxes". Relaxation is as of now not measured by a quantitative criterion, but via judging visualizer output. This will eventually change.

Figure 1 gives an idea of the improvement in the system brought about by the iterative scheme. The figure shows two snapshots of vehicle positions in the Gotthard scenario, described in Sec. 3. The left side of the figure shows a snapshot of the vehicles in the midst of the initial iteration (number 0), 2-3 hours after all vehicles have left their starting locations, for their common destination. In this iteration demand is not known, so each traveler assumes free speed travel times, and chooses a route as if it is the only driver in the network. Thus, the freeways are all in use, and no alternative routes have been explored. The right side of the figure shows the same situation, but 49 iterations later. Here, the drivers take into account the congestion caused by other vehicles on the roadways, so many more routes are explored. In the 49th iteration, fewer travelers take the "middle" paths through the Alps (such as the Gotthard tunnel) than in the 0th iteration, instead electing to take the western or eastern paths.

3 The Scenario

The goal of our work is a full 24-hour simulation of all of Switzerland, including transit traffic, freight traffic, and all modes of transportation. This will involve about 7.5 million travelers, and more than 20 million trips (including short pedestrian trips etc.). A more short-term goal is a full 24-hour simulation of all of car traffic in Switzerland. For this, we will have about 10 million trips.

Our network consists of 10 572 nodes and 28 622 links. This network is provided by the Swiss transportation planning authorities. Besides the standard attributes for geographical location and length, the links have speed, capacity, and type attributes. As of now, no street layout, not even the number of lanes, is part of that information; also, no information about traffic signals is known. This makes using the TRANSIMS micro-simulation difficult since it needs that information. This is one of the reasons why we use our queue simulation as described above.

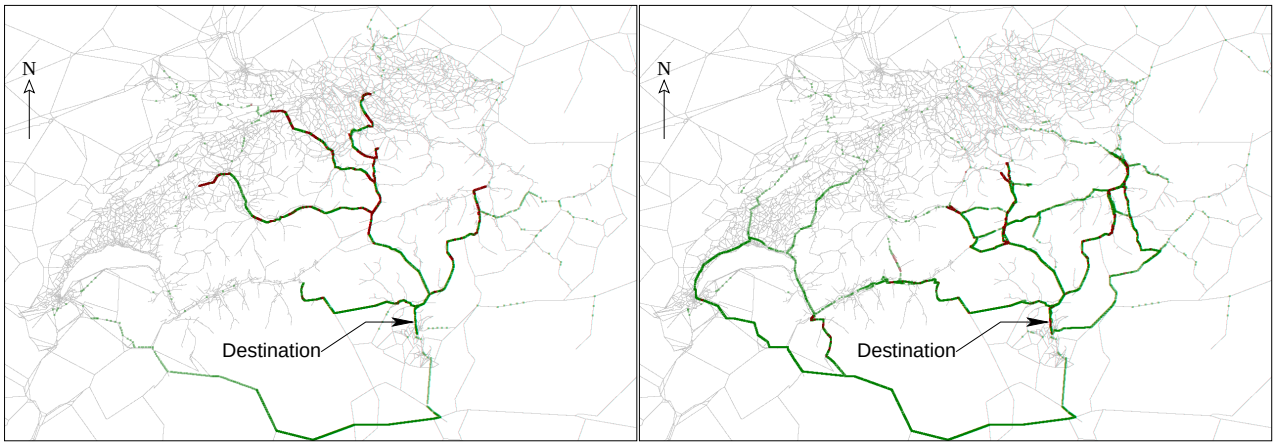

Figure 1: Example of relaxation due to feedback. **LEFT:** Iteration 0 at 9:00 – all travelers assume the network is empty. **RIGHT:** Iteration 49 at 9:00 – travelers take more varied routes to try to avoid one another.

In order to test our modules and our framework, we use a so-called **Gotthard scenario**. In this scenario, 50'000 travelers/vehicles start, with a random starting time between 6am and 7am, at random locations all over Switzerland, and with a destination in Lugano/Ticino. Although this scenario has some resemblance with vacation traffic in Switzerland, its main purpose is to test the congestion dynamics of the micro-simulation, and its interaction with the feedback framework. This will become clear later in the text.

4 Link Travel Time Feedback

Even within the framework as described above, there is considerable flexibility in how to interpret the different pieces. One of these pieces is how to aggregate the link travel times: While the traffic simulation generates link entry and exit times for each individual vehicle, the router needs link traversal times as a function of link entry time. As pointed out above, these latter times also need to be aggregated in order to reduce computational overhead.

One issue is if to use link entry or link exit times as the basis for aggregation. The way the router works, one would like the average travel time of all vehicles entering during a specific period of time. In terms of simulation logic, this is awkward since one needs to keep information about when the last vehicle belonging to such a batch has actually left the link.

As a result, TRANSIMS averages over vehicles *leaving* the link during a specific period of time. This has however the disadvantage that now the averaged information is no longer consistent with the router – for example, a link travel time for vehicles *exiting* a link between 9 and 9:15 is not the same as a link travel time for vehicles *entering* a link between 9 and 9:15.

This issue can be addressed by "backdating" [10], that is, one calculates the respective link entering times. TRANSIMS does that after the averaging has taken place. For example, assume that the average link travel time for vehicles exiting between 9 and 9:15 is 10 min, and the average link travel time for vehicles exiting between 9:15 and 9:30 is 15 min. By backdating, one would arrive at the result that all vehicles entering between 8:50 and 9:05 need 10 min, and all vehicles entering between 9:00 and 9:15 need 15 min. This clearly leads to gaps and overlaps; TRANSIMS uses piece-wise linear functions to interpolate between the periods.

In our approach, we decided to completely separate the aggregation from the micro-simulation. That is, the micro-simulation is asked to output event information every time a vehicle leaves a link (this is information that also the TRANSIMS traffic simulation can generate). A post-processing step then aggregates this data into the information needed by the router.

For the post-processing, we use a pair of AWK scripts. The first script, (see Sec. A.1 for listing), reads the

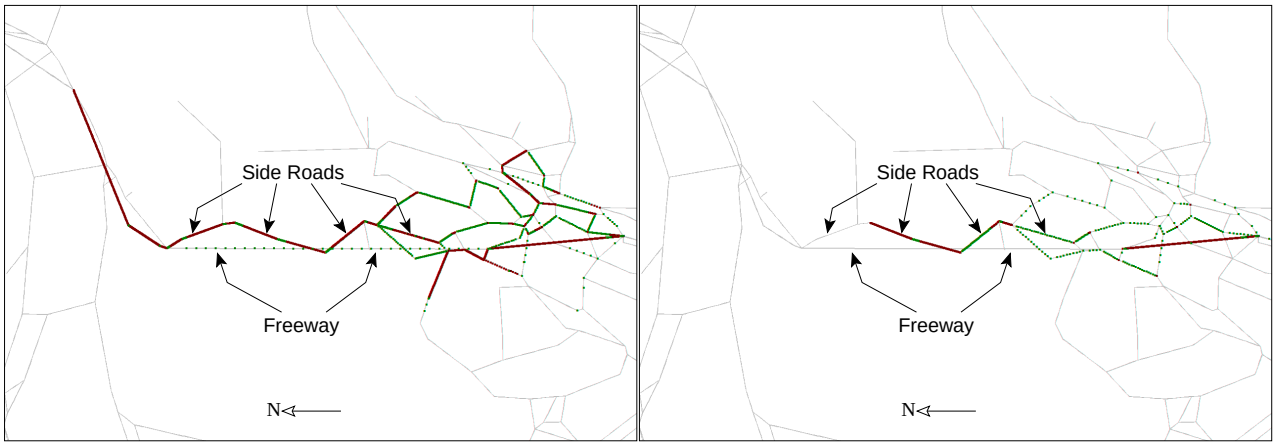

Figure 2: A freeway and side roads with the original travel time feedback strategy at 19:00 (left) and 20:00 (right). The side roads contain many vehicles while the freeway contains very few or none.

events file produced by the micro-simulation, filters the events marking vehicles exiting links, and compiles them together into an intermediate file that lists, for each vehicle, the time it entered and exited each link in its plan. It also creates a second file that lists the arrival times of each vehicle at its destination. Coupled with the (already known) starting times of the travelers' routes, this second output file enables each traveler to calculate the total travel time of its plan. The second script (see Sec. A.2 for listing), aggregates the output of the first script, to determine the average travel-time on the links. For each link in the network, this script keeps a running count of the number of vehicles who entered the link during each time bin of the day; as well as a running sum of the total amount of time that group of entering vehicles spent on the link. Dividing the sum by the count for each link and time bin combination gives the average travel time for that link during that time bin.

4.1 Initial Results

We ran the above setup with the Gotthard scenario. In this section we present the initial results of that simulation.

For the following, we concentrate on an about 50 km $\times$ 100 km section north of Lugano. For better exposition, the orientation of the plots will be rotated by 90 degrees, so that Lugano now is to the right and the Alps are to the left. Fig. 2 shows snapshots of the situation at 19:00 and at 20:00. These and all other snapshots are after 49 feedback iterations. In general, the vehicles are jammed up because there are bottlenecks inside Lugano for the vehicles to reach their destination.

The implausible feature of these plots is that there are traffic jams on the side roads while the freeway is empty. Note that there is no en-route replanning, and so the plan-following vehicles are stuck with their plans for the whole duration of their trips.

After further investigation, we found that the problem was caused by the fact that the router will not react "fast enough" if traffic is moving well at the beginning of the time bin, but not at its end. Cars that are on that link at the beginning of the time bin will leave sooner than the router expects, but those placed at the end of the time bin will leave later than expected.

As an example, suppose a link L has a free-speed travel-time of 3 minutes, and the router is considering routing two agents A and B on that link during the time bin from 7:00 to 7:15. Suppose further that L is close to free-flowing at 7:00, but gets congested by 7:15. Its average travel-time during this time bin is calculated to be 5 minutes.

If agent A starts out on the link near the beginning of the time bin, say 7:03, it has a clear ride and will be off the link in, say, 4 minutes. If agent B starts out on the link closer to the end of the time bin, say 7:10, it

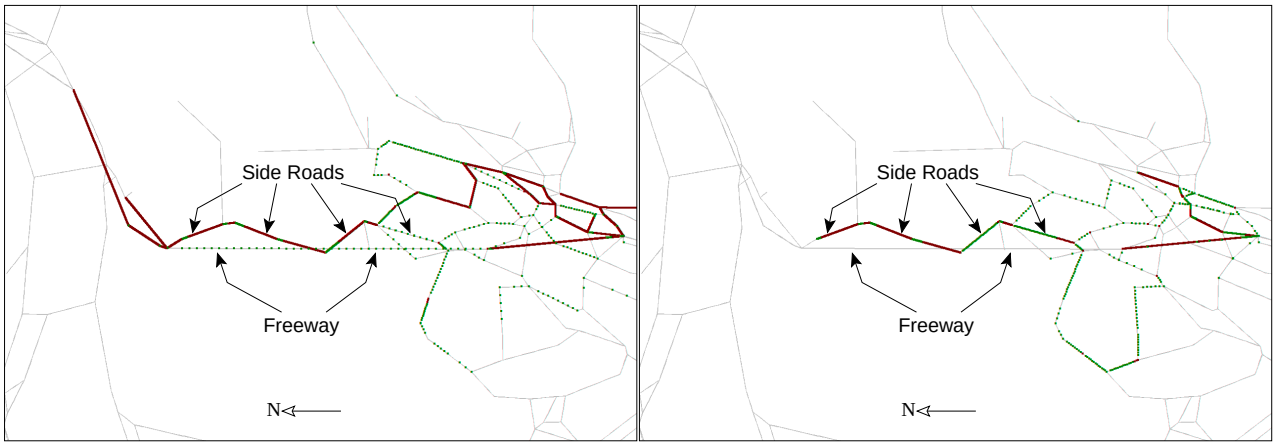

Figure 3: A freeway and side roads with the offset time bins strategy at 19:00 (left) and 20:00 (right). The side roads contain many vehicles while the freeway contains very few or none.

gets into that congestion and has a longer travel time, say 9 minutes. The result is that agent A is one minute ahead of the router's schedule for it, while B is 4 minutes behind schedule.

Overall, there are four cases:

- Congestion building up, and vehicle early in time bin. Then the vehicle will be faster than the router thinks. The vehicle will be faster, and since congestion is just building up, it will also be faster in other parts of the system, thus amplifying the initial error.
- Congestion building up, and vehicle late in time bin. The the vehicle will be slower than the router thinks. The vehicle will fall behind, and since congestion is building up, it will fall behind further in other parts of the system, thus amplifying the initial error.
- Congestion going away, and vehicle early in time bin. Then the vehicle will be slower than the router thinks. The vehicle will fall behind, but by falling behind will encounter less congestion, which will limit how much it falls behind.
- Congestion going away, and vehicle late in time bin. Then the vehicle will be faster than the router thinks. The vehicle will be faster, but by being faster it will encounter more congestion, which will limit how far ahead of schedule it is.

From this description it is clear that in particular the first two cases are a problem since the dynamics tends to amplify the errors. In order to test our hypothesis, we describe two modifications to the router in the following.

4.2 Offsetting the Time Bins

How do we fix this problem? Since the problem seems to be the router's reaction to a link's transition from free-flowing to congestion, we consider giving the router an "early warning" about impending congestion build up. We do this by simply offsetting all the time bin data so that it is ahead of reality by one bin. This causes the router to use the 7:15-7:30 time bin information when it is calculating link costs between 7:00 and 7:15. That way, it will start instructing agents to avoid congested links *before* those links actually get congested. This strategy will also cause the router to place more vehicles on links undergoing transitions from congested to free-flowing at an earlier time. Based on the reasoning above, however, this situation should not cause too much of a problem.

Figure 3 shows the outcome of this strategy. We can see that the freeway is still emptying earlier than the side roads. This strategy, by itself, does not seem to help us at all in this case.

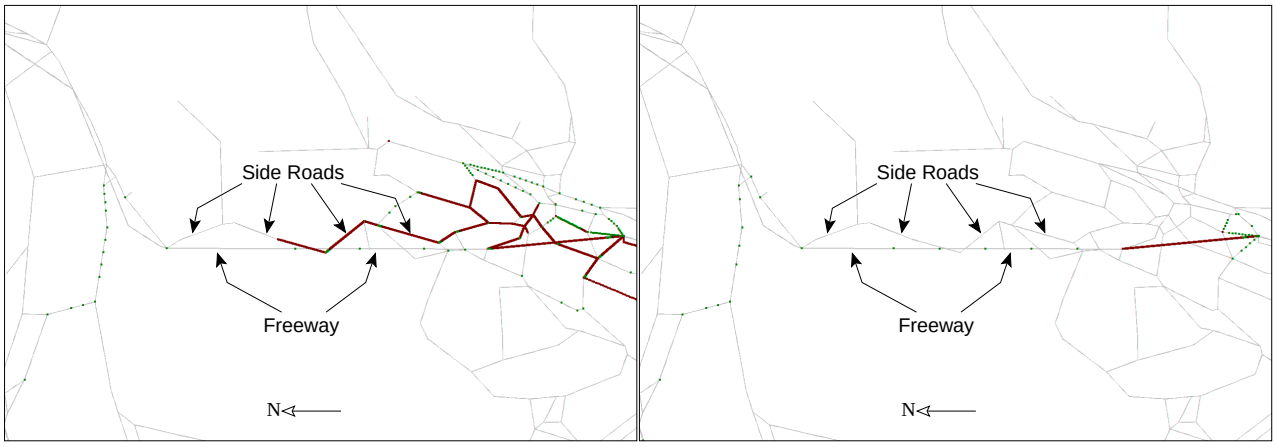

Figure 4: A freeway and side roads with the maximum travel time strategy at 19:00 (left) and 20:00 (right). At 19:00 the side roads contain some vehicles while the freeway is mostly empty. At 20:00 the side roads are now empty while the freeway contains a few vehicles.

4.3 Maximum vs. Average Travel-Time

Another issue with the data used by the router is that it is an average of the travel times experiences by the vehicles. As stated above, if the router under-predicts the travel-time for an agent on a link during a time bin, that agent will be behind schedule. But, if the router over-predicts, then it is not a big problem. Instead of giving the router an early warning, we alter the router's view of the links so that it pays more attention to the travel times of those vehicles who experienced congestion on the links. In other words, we bias the data against congested links. The simplest way to do this is to take the *maximum* travel-time experienced on each link during each time bin, rather than the average.

Figure 4 shows the result of this strategy. This strategy also does not fix the problem because the freeway still practically empties earlier than the side roads. In this case, however, we notice that a few vehicles use the freeway after the side roads are clear. But the number of vehicles on the freeway is too small compared to the side road. We seem to be getting some improvement, at least.

4.4 Combining Maximum and Offset

Neither offsetting the travel times data, nor biasing it toward the maximum reported travel time seemed to completely fix the problem of the implausible results. We now try, as a new strategy, the combination of the two. We take the maximum travel times instead of the average, *plus* we offset the resulting travel times data by one bin. This should improve the "early warning" to the router given by the offset method, since only the most delayed drivers will be the ones reporting their experiences to the router.

Figure 5 shows the output from this result. As we can see, the side roads finally empty before the freeway does, as we expected from the beginning.

4.5 Conclusion

After enough analysis, "combining maximum and offset" finally solved the problem. We essentially had to greatly exaggerate the router's view of the links undergoing transition from free-flowing to congested regimes, so that it could react in time to move travelers away from those links to avoid the congestion.

This solution was tailored for this specific problem, however. If there are other routing problems that we discover at a later time, we may have to adjust our travel time reporting strategy again. Such adjustments could conflict with the current method, bringing back the problem of empty freeways with congested side roads. We would like a more robust solution, which can work even if flaws exist in the router or the feedback system.

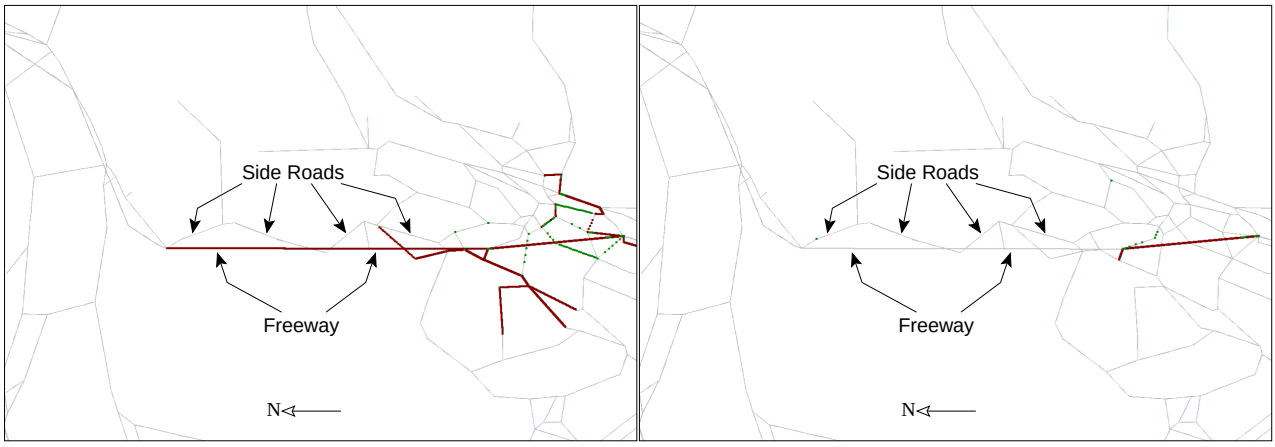

Figure 5: A freeway and side roads with the combined offset time bins with maximum travel time strategy at 19:00 (left) and 20:00 (right). The side roads are finally empty, while the freeway now contains vehicles. This is what is expected from the scenario.

In the next section we present an alternative solution to the maximum and/or offset strategies, which moves away from adjusting the travel times reporting, to adjusting the behavior of the travelers.

5 The Agent Database

5.1 Concept

In the above methods, all agents forgot their previous plans when new ones were created, on the assumption that the new ones were always better than the old ones. But, if the router is flawed, or not obtaining the proper information, this might not (always) be true. So, we now give the agents a *memory* of their past plans (decisions), and the outcome (performance of plans) of those decisions. We allow them to choose their new plan based on the performance of the routes in their memory. New, untested routes from the router iteration are given top priority, but if an agent has tried all of his/her plans before, then he/she chooses one by comparing their performance values. This strategy means that more than our original 10% replanning fraction of the agents will change their plans at a given iteration. These changes will be "informed" decisions, though – not random exploration.

By giving the agents a memory, we must give them a way to select remembered routes. For a given plan – as a whole – we can find the total time taken to traverse the route. This will be a measure of the performance of the route. Agents can compare performance of remembered routes, and choose one based on performance information, without knowing anything else about the routes.

The idea here is that we don't need to fix the router to be perfect, as long as it generates reasonable routes most of the time. We can use the original router and travel time reporting strategies (averaged travel times and non-offset time bins), and still get behavior that makes sense.

In comparison, TRANSIMS also uses a database, called the "Iteration Database," which stores information about agents and their experiences from previous iterations. This database is meant to be used to choose specific sets of agents for replanning, but to our knowledge, does not store previously discarded routes for later re-use. [9]

5.2 Implementation of the Agent Database

We introduce a database into the iteration framework to give the agents memory of their plans. Currently, this database is implemented in MySQL, an open-source relational database management system.Each time the router generates a new (initial or updated) plan-set, those plans are added to the database, along with

plans table:

agent	plan_num	is_new	start_time	plan
1	1	0	25200	<text string 1>
1	2	1	25200	<text string 2>
2	1	0	25380	<text string 3>
⋮	⋮	⋮	⋮	⋮

travel_times table:

agent	plan_num	travel_time
1	1	462
1	2	0
2	1	1047
⋮	⋮	⋮

flags table:

agent	plan_num	flag
1	1	1
1	2	0
2	1	1
⋮	⋮	⋮

Figure 6: Example tables in the agent database. The "agent" and "plan_num" fields are combined into the primary key for all three tables. The "plan" field of the plans table contains a text string consisting of link identifiers and other information that the router requires.

the identifying number of their corresponding agent, and the starting time of the plan. The database also stores, for each plan, the most recently measured travel time (performance measurement) made by the agent for that plan; and a flag that, when true, marks the plan as being the one used by its agent in the most recent micro-simulation. For new, untried plans generated by the router, the travel-time is considered to be zero, and the agent is forced to always choose that plan next. See Fig. 6 for an example of how the database stores information, and Sec. A.4 for the actual MySQL code used to interact with the database.

Once the new set of plans has been entered into the database, the travel times table is joined with the flags table and output into a file. This file is read by a script which uses the travel-times information to make the choice for each agent of its next plan. See Sec. A.3 for the listing of this script. The script writes a new file with updates to the flags table, which is then written into the database. The flags indicate that the plan is chosen in the current iteration. Once the database knows which plans to choose, it writes that set of plans (only the ones with flag $= 1$) to the input file for the micro-simulation, and the micro-simulation is executed.

After the simulation is finished, its events output is parsed into entry and exit times for each agent on each link of their route. These entry and exit times are aggregated into the 15 minute time binned travel-times, which are used by the router to generate its next 10% planset. (Please see Sec. 4 for a more detailed description of these scripts, or Secs. A.1 and A.2 for the listings of them.) At this time another file is created that indicates the arrival time of each agent at its destination link. This file is read back into the database, the plan start times are subtracted from the arrival times, and the travel-times are updated. This only occurs for plans which are flagged in the flags table as having been used in the last iteration. In other words, only one plan per agent is updated with the travel time.

At this point, the database is ready for the next iteration, when the router will again generate a new set of plans that must be entered into the database.

5.3 How plans are actually chosen based on performance

The only detail left out of the above explanation is how the performance (total travel-time) information is used by the agents to choose their plan for the next iteration.

Each agent uses the following model to compare the *utility functions* of its remembered plans. This function is defined as the relative probability, P, of choosing a given plan i (out of p plans) for an agent a:

$$P(tt_{a,i}) := \exp(-\beta \cdot tt_{a,i}) \tag{1}$$

where β is an empirical constant, and $tt_{a,i}$ is the total travel time known by agent a for its plan i. This resembles both a Boltzmann distribution in physics and a logit model in discrete choice theory [11].

Equation 1 is only a relative probability; in order to have the probabilities for all p plans of agent a sum to 1, we must normalize the probabilities. Let P' be the normalized probability:

$$P'(tt_{a,i}) := \frac{P(t_{a,i})}{\sum_{i=1}^{p} P(t_{a,i})} \tag{2}$$

Next, we calculate the cumulative probability sums:

$$SC_{a,i} := \sum_{j=1}^{i} P'(t_{a,j}) \tag{3}$$

Agent a next draws a random number, $r \in [0, 1)$. It then chooses plan i such that $SC_{a,i}$ is as large as possible, but is less than r:

$$SC_{a,i} < r \leq SC_{a,i+1} \tag{4}$$

The end result of these calculations is that agents are most likely to choose the plan with the highest performance, second-most likely to choose the plan with the second highest performance, etc. Since a plan's performance is overwritten by new tries of that plan, if the plan improves its performance, it is more likely to be chosen in the future. If it's performance degrades upon reuse, it will be tried less often in the future.

The value of β determines how likely it is that a "non-best" plan will be chosen. For the Gotthard scenario, we chose the value of β so that about 90% of the agents, in the initial iterations at least, would choose their best possible plan. In other words, we allowed only 10% (of the 90% who were *not* replanned in the current iteration) to retry "non-best" plans. Specifically, we set β to be $\frac{1}{360}$. This allows the relaxation to progress rapidly in the early iterations, and gives agents the ability to occasionally give "non-best" plans the chance to improve.

5.4 Results of Agent Database on the Gotthard Scenario

Figure 7 shows the results of using the original strategy from Sec. 4 plus the agent database, with plans selected as described above. As one can see, the freeway problem is avoided when the agents have memory of their plans. If the router starts putting too many agents on the side roads, some will eventually try out an old plan that used the freeway and find that it has a good performance, so will likely use that plan again. As long as they remember one or more plans that utilize the freeway, the agents can decide for themselves to use it, bypassing the side road choice of the router. Thus, the agent database gives an added flexibility and robustness to the system, so that even with a flawed router or feedback mechanism, the results come out satisfactorily.

This value of β we chose seemed to work well, but future work will likely need to explore the outcome of other values for this constant.

6 Conclusion

The purpose of this paper and this study is to demonstrate that for multi-module transportation simulations, not only is the functionality of the single modules important, but also how they interact. In particular, an agent-based implementation of the interfaces between the modules is capable of correcting for artifacts in the modules. An agent-based representation means that travelers are considered as agents, which have a memory of different strategies and their respective performances. In general, they chose the strategy with the best performance, but from time to time re-try one of the other strategies just to check if its performance is still unchanged. Also from time to time, new strategies are generated and added to the pool.

In this particular example, we apply this approach to route feedback for dynamic traffic assignment. The problem was that the router uses aggregated feedback information from the micro-simulation, and that this

Figure 7: A freeway and side roads with the agent database strategy at 19:00 (left) and 20:00 (right). As with the "max and offset" strategy, the side roads are emptying, while the freeway contains vehicles. This shows the agent database is a solution to the freeway problem.

aggregation with most plausible algorithms lead to artifacts in the resulting traffic. Specifically, the router under-estimated long distance travel times, leading to the fact that the router assumed the existence of congestion for later parts of the trip while in fact the congestion was long gone. This resulted in travelers using the side roads where the freeway would have been much better. The use of the agent data base solves this problem *without any changes in the router*. That is, even when the router consistently generates faulty plans, the agent database approach will compensate for this as long as at least some of the routes are plausible.

The approach was implemented using MySQL as a data base, and perl/awk as scripting languages. Further details are given in the text.

Acknowledgments

We would like to thank the Swiss regional planning authority (Bundesamt für Raumentwicklung) and Milenko Vrtic at the Institute for Transportation Planning (IVT) of ETH Zürich for providing the Switzerland network; Andreas Völlmy for the Gotthard scenario initial plan set data; and Nurhan Cetin for the parallel queue simulation.

References

[1] S. T. Doherty and K. W. Axhausen. The developement of a unified modelling framework for the household activity-travel scheduling process. In *Verkehr und Mobilität*, number 66 in Stadt Region Land. Institut für Stadtbauwesen, Technical University, Aachen, Germany, 1998.

[2] K. Nagel. Distributed intelligence in large scale traffic simulations on parallel computers, in preparation. See www.inf.ethz.ch/personal/nagel/papers.

[3] K. Nagel. Routing in iterated transportation simulations, in preparation. See www.inf.ethz.ch/personal/nagel/papers.

[4] P. Waddell, A. Borning, M. Noth, N. Freier, M. Becke, and G. Ulfarsson. Microsimulation of urban development and location choices: D esign and implementation of UrbanSim. *Networks and Spatial Economics*, in press.

[5] C. Gawron. An iterative algorithm to determine the dynamic user equilibrium in a traffic simulation model. *International Journal of Modern Physics C*, 9(3):393–407, 1998.

[6] N. Cetin and K. Nagel. Parallel queue model approach to traffic microsimulations. In *Swiss Transport Research Conference*, Monte Verita, Switzerland, March 2002.

[7] DYNAMIT, 1999. Massachusetts Institute of Technology, Cambridge, Massachusetts. See its.mit.edu.

[8] H.S. Mahmassani, T. Hu, and R. Jayakrishnan. Dynamic traffic assignment and simulation for advanced network informatics (DYNASMART). In N.H. Gartner and G. Improta, editors, *Urban traffic networks: Dynamic flow modeling and control*. Springer, Berlin/New York, 1995.

[9] TRANSIMS, TRansportation ANalysis and SIMulation System, since 1992. See transims.tsasa.lanl.gov.

[10] I. Porche and S. Lafortune. On combined dynamic traffic assignment and traffic-responsive signal control problem. *Transportation Research C*, submitted. Also at www.eecs.umich.edu/ porche/.

[11] M. Ben-Akiva and S. R. Lerman. *Discrete choice analysis*. The MIT Press, Cambridge, MA, 1985.

A Source Code

A.1 read_events.awk

This script reads the events output of the micro-simulator and converts it into entry time and exit time pairs for each vehicle on each link. It also outputs the arrival times of the agents at their destinations.

Interface:

Type	**Name**	**Comment**
Input File	events.trv	traveler events file from micro-simulator
Output File	events.start_end	the starting and ending times for each vehicle on each link in its plan
Output File	END_TIMES	when vehicles finished their routes

```
#!/bin/awk -f

# This script reads a SINGLE (consolidated) SORTED events file and
# figures out when each car entered and exited each link in its plan.

BEGIN {
    OFS = "\t";
    print "VEHICLE", "LINK", "ENTRY", "EXIT";
}

# main pattern -- executed for each line of input file
{

    # Skip header line(s)
    if ( $1 == "TIMESTEP" ) {
        next;
    }

    timestep = $1 + 0;
    vehicleid = $2;
    link = $3;
    fromnode = $4;              # ignored for now
    flag = $5;

    # Assuming file is sorted by timestep.

    # Store information for END_TIMES output
    if ( first_time[vehicleid] == "" ) {
        first_time[vehicleid] = timestep;
        last_time[vehicleid] = timestep;
    } else if ( timestep > last_time[vehicleid] ) {
```

```
        last_time[vehicleid] = timestep;
    }

    if ( last_time[vehicleid] < first_time[vehicleid] ) {
        print "Something is wrong! ABCDEFG" > "/dev/stderr";
    }

    # Ignoring initial link entry, since we do not know where the
    # parking accessory really is on the link.

    # flag == 2 means a "normal" link exit.
    if ( flag == 2 ) {

        if ( older_time[vehicleid] != "" ) {

            print vehicleid, old_link[vehicleid],
                     older_time[vehicleid],
                     old_time[vehicleid];
            if ( old_time[vehicleid] <= older_time[vehicleid] ) {
                print "Something is wrong! ZYEW" > "/dev/stderr";
            }

        }

        older_time[vehicleid] = old_time[vehicleid];
        old_time[vehicleid] = timestep;
        old_link[vehicleid] = link;

    }

}

END {
    for ( v in last_time ) {
        print v, last_time[v] > "END_TIMES";
    }
}
```

A.2 parse_link_times_entry.awk

Interface:

File Type	Filename	Comment
Input File	events.start_end	the starting and ending times for each vehicle on each link in its plan, from read_events.awk
Output File	summary.tim	travel-times file for the router

```
#!/bin/awk -f

# Read the output of read_events.awk, and transform it into something
# resembling a TRANSIMS travel-times summary file, for reading by the
# router.

# this is for where tt is time_bin of ***ENTRY*** time, not exit time

# figure out which 15-minute time bin to store data into
function calc_time_bin(time) {

    # want times to map like so:
    #      ...21600 => 21600
    # 21601...22500 => 22500
    # 22501...23400 => 23400

    # this is the original time-binning strategy; subtract 1 to get
    # offset (so that 7:15 read from the input file goes into 7:00's
    # bin
    if ( ( time % 900 ) == 0 ) {
```

```
        return int(time / 900) - 1;
    } else {
      return ( int( time / 900 ));
    }
}

function print_data() {

    print ll , -1 , (tt+1) * 900 , count[ll, tt]+0 , sum[ll, tt]+0 ,
           -1 , -99 , -1 , 0 , 0 , -1;
}


BEGIN {
    OFS = "\t";
    SUBSEP = OFS;

    min_time_bin = 10000.0;
    max_time_bin = -10000.0;

    min_link = 100000000.0;
    max_link = -1.0;

    # expected format of TRANSIMS travel times files
    # we aren't using most of these fields
    print "LINK" , "NODE" , "TIME" , "COUNT" , "SUM" , "SUMSQUARES" ,
          "TURN" , "LANE" , "VCOUNT" , "VSUM" , "VSUMSQUARES" ;
}


NR > 1 {

    vehid = $1;
    link  = $2 + 0.0;
    entry_time = $3 + 0.0;
    exit_time = $4 + 0.0;

    travel_time = exit_time - entry_time;

    time_bin = calc_time_bin(entry_time);

##  For MAXIMUM strategy, replace the 2 lines below with
##  if ( travel_time > max[link, time_bin] ) {
##     max[link, time_bin] = travel_time;
##     count[link, time_bin] = 1;
##  }
##  ... and replace "sum" everywhere with "max"

    count[link, time_bin] ++;
    sum[link, time_bin] += travel_time;

#    sumsquared[link, time_bin] += ( travel_time * travel_time );

    links_seen[link] = 1;
    time_bins_seen[time_bin] = 1;

    if ( link > max_link ) {        max_link = link;    }
    if ( link < min_link ) {        min_link = link;    }

    if ( time_bin > max_time_bin ) {    max_time_bin = time_bin;  }
    if ( time_bin < min_time_bin ) {    min_time_bin = time_bin;  }

    # let user keep track of how far into the input file we are
    if ( NR % 100000 == 0 ) {
        print "line: "NR >> "/dev/stderr";
    }

}
```

```
END {

  # go through the time bins of the day
  for ( tt = min_time_bin ; tt <= max_time_bin ; tt ++ ) {

    # go through the links of the network
    for ( ll = min_link ; ll <= max_link ; ll++ ) {

      if ( ( ll, tt ) in count ) {

        print_data();
        # once a link is outputted, it should continue to be
        # outputted, to show that it is empty
        output_link[ll] = 1;

      } else if ( ll in output_link ) {

        # deal with links that have vehicles on them for more than 15
        # minutes
        count[ll,tt] = count[ll,tt-1];
        sum[ll,tt] = sum[ll,tt-1] - count[ll,tt]*900;

        if ( sum[ll, tt] <= 0 ) {
          sum[ll, tt] = 0;
          count[ll,tt] = 0;
        }
# (the logic behind the above is that, as soon as the queue should be
# resolved, we report zero vehicle entries so the link is unreported.)
# (The router uses free-speed travel-times for links during time bins
#  they are unreported for a time bin.)
# the router also ignores links with 0 count

        print_data();
      }
    }
  }
}
```

A.3 pick_plans.exp-Bt.awk

This script performs the decision-making of the agents. For each agent, it chooses one of the plans remembered by that agent based on the performance of the remembered plans. See Sec. 5.3 for the decision description.

Interface:

File Type	Filename	Comment
Input File	travel_times.out	travel times and flags output from database
Input Parameter	seed	seed for the random generator
Output File	flag_update.in	update of flags table for database

```
#!/bin/awk -f

# Reads a file of agent, plan_num, travel_time, and flag.
# Chooses a new plan_num for each agent based on the travel_time.

# The probabilistic version -- chooses plan_num based on utility
# function exp(-beta*travel_time).

BEGIN {
#1    beta = 1.0/3600.0/6.0;
#2    beta = 1.0/3600.0;
#3    beta = 1.0/1000.0;
    beta = 1.0/360.0;
    OFS = "\t";
    assert( (seed != ""), "I need a seed value!");
```

```
    srand(seed);
}

function assert(is_true, msg) {
    if ( ! is_true ) {
        print "ERROR (agent="old_agent"): "msg > "/dev/stderr";
        error = 1;
        exit(1);
    }
}

function choose_plan_num(chosen_last,p) {

    assert( ( ( 1 in tt ) && ( 1 in fl ) ), "No plans?");

    sum = 0;
    chosen_last = 0;

    p = 1;
    while ( p in tt ) {
        if ( tt[p] == 0 ) {
            return p;           # ALWAYS choose brand-new plans (tt=0)
        }
        prob[p] = exp(-beta * tt[p]);
        sum += prob[p];
        assert( ( p in fl ), "Plan " p " is in tt but not fl!");
        if ( fl[p] == 1 ) {
            assert( (chosen_last == 0) , "Too many chosen plans!");
            chosen_last = p;
        }
        p++;
    }
    max_p = p - 1;

    assert( ( chosen_last != 0 ), "No chosen plans!");

    p = 1;
    sum_list[0] = 0;
    while ( p in tt ) {
        norm_prob[p] = prob[p] / sum;
        sum_list[p] = sum_list[p-1] + norm_prob[p];
        p++;
    }

    if ( sum_list[max_p] != 1 ) {
        sum_list[max_p] = 1;
    }

    r = rand();
    p = 1;
    while ( r >= sum_list[p] && ( (p+1) in sum_list) ) {
        p++;
    }

    assert( (p <= max_p && p > 0),
            "final p ("p") is out of bounds; max_p="max_p);

    return p;
}

function update_flags(pn,p,q) {
    pn  = choose_plan_num();
    for ( p in tt ) {
        print old_agent, p, p == pn;
    }
    delete tt;
}

NR == 1 {
```

```
    old_agent = $1;
}

# Assuming input is sorted by agent ($1) then by plan_num ($2)

{
    agent = $1;
    plan_num = $2;
    travel_time = $3;
    flag = $4;

    if ( agent != old_agent ) {

        # choose a plan_num for the agent and print its new flags
        update_flags();

    }

    tt[plan_num] = travel_time;
    fl[plan_num] = flag;

    old_agent = agent;
}

END {

    if ( error == 1 ) {
        exit(1);
    }

    # choose a plan_num for the agent and print its new flags
    update_flags();

}
```

A.4 SQL code for Agent Database

A.4.1 Create Database

```
# Create the database and set up the tables
# To be executed just once, at the beginning of the iteration.

DROP DATABASE IF EXISTS agent_db ;

CREATE DATABASE IF NOT EXISTS agent_db ;

USE agent_db ;

# store the plans themselves; we have some minimal information about
# the plan plus the actual plan stored as a text string (which is what
# the simulator reads and the router outputs)

# is_new tells us that the plan has not been tried yet; its default is
# 1, so that newly added plans are automatically marked is_new

CREATE TABLE plans (
        agent           INT UNSIGNED NOT NULL DEFAULT 0,
        plan_num        INT NOT NULL AUTO_INCREMENT,
        is_new          TINYINT UNSIGNED NOT NULL DEFAULT 1,
        start_time      INT UNSIGNED NOT NULL DEFAULT 0,
        plan            TEXT NOT NULL,
        PRIMARY KEY ( agent, plan_num )
) ;

# store the most recent performance (utility?) of the plans
CREATE TABLE travel_times (
        agent           INT UNSIGNED NOT NULL DEFAULT 0,
        plan_num        INT NOT NULL DEFAULT 0,
```

```
        travel_time     INT NOT NULL DEFAULT 0,
        PRIMARY KEY ( agent, plan_num )
) ;

# the flag tells whether or not the plan has been chosen by the agent
# for this iteration
CREATE TABLE flags (
        agent           INT UNSIGNED NOT NULL DEFAULT 0,
        plan_num        INT NOT NULL DEFAULT 0,
        flag            TINYINT UNSIGNED NOT NULL DEFAULT 0,
        PRIMARY KEY ( agent, plan_num )
) ;
```

The “flag” and “travel_time” attributes are stored in separate tables because MySQL doesn’t allow the database to update a table using information from that table. So, either large temporary tables must be used, or the information used to update a table must be stored in a separate table.

A.4.2 Read Plans into Database

After creating the database, the router is run to generate some plans. Plans are converted into a format suitable for reading and saved under the file “plans.for.db”.

```
# Read (new/initial) plans into database

UPDATE plans SET is_new = 0 WHERE is_new <> 0;

LOAD DATA LOCAL INFILE 'plans.for.db'
INTO TABLE plans
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n\n'
        ( agent,
          start_time,
          plan ) ;

# The plans are automatically marked as new (default of is_new is 1)

# Add entries to flags that correspond to the new plans.

INSERT INTO flags
SELECT  agent, plan_num, 0
FROM    plans
WHERE   is_new = 1;

# Add entries to travel_times that correspond to the new plans.

INSERT INTO travel_times
SELECT  agent, plan_num, 0
FROM    plans
WHERE   is_new = 1;
```

A.4.3 Output Travel Times

```
# Output entire (updated) travel-times table, so the external script
# can choose the new set of plans.

# Also, output the flag so we know which plan was chosen last time (if
  any)

LOCK TABLES travel_times READ, flags READ ;

SELECT
        travel_times.agent,
```

```
        travel_times.plan_num,
        travel_time,
        flag
INTO OUTFILE '/iteration/output/location/travel_times.out'
FROM
        travel_times,
        flags
WHERE
        travel_times.agent = flags.agent AND
        travel_times.plan_num = flags.plan_num
GROUP BY
        agent,
        plan_num ;

# not unlocking here -- that will be done when we read the flags.
```

The output of the above is then processed by pick_plans.exp-Bt.awk to update the flags (see next sub-section). (see Sec. A.3).

A.4.4 Update Flags and Output Plans

```
# Choose new set of plans based on performance, and update flags

CREATE TEMPORARY TABLE tmp (
        agent           INT UNSIGNED NOT NULL DEFAULT 0,
        plan_num        INT NOT NULL DEFAULT 0,
        flag            TINYINT UNSIGNED NOT NULL DEFAULT 0,
        PRIMARY KEY ( agent, plan_num )
) ;

# READ lock of travel_times is overridden here

LOCK TABLES flags WRITE , plans READ ;

LOAD DATA LOCAL INFILE 'flag_update.in' INTO TABLE tmp ;

REPLACE INTO flags
SELECT  *
FROM    tmp ;

DROP TABLE tmp ;

# Output chosen plans

SELECT plan
INTO OUTFILE '/iteration/output/location/plans.out'
FIELDS TERMINATED BY ','
ESCAPED BY ''
LINES TERMINATED BY '\n\n'
FROM
        plans,
        flags
WHERE
        plans.agent = flags.agent AND
        plans.plan_num = flags.plan_num AND
        flags.flag = 1 ;

UNLOCK TABLES ;
```

A.4.5 Update Travel Times

After the micro-simulator is run, the events files are parsed. The END_TIMES file created by read_events.awk (see Sec. A.1) is used here to update travel times of the plans in the database.

```
# Update travel-times

# First run read_events.awk to create END_TIMES file

CREATE TEMPORARY TABLE end_times (
        agent           INT UNSIGNED NOT NULL DEFAULT 0,
        end_time        INT NOT NULL DEFAULT 0
) ;


LOAD DATA LOCAL INFILE 'END_TIMES' INTO TABLE end_times ;

LOCK TABLES travel_times WRITE , flags READ , plans READ ;

# We're overwriting old travel times with new ones; we could also
# average or something to not lose the old information completely

REPLACE INTO travel_times
SELECT
        flags.agent,
        flags.plan_num,
        ( end_times.end_time - plans.start_time ) AS travel_time
FROM
        flags,
        plans,
        end_times
WHERE
        flags.agent = plans.agent AND
        flags.agent = end_times.agent AND
        flags.plan_num = plans.plan_num AND
        flags.flag = 1 ;


DROP TABLE end_times ;
```