Dynamic Routing Algorithm Selection in MANETs through Reinforcement Learning

Robert D. Grant (author) and Agoston Petz
The University of Texas at Austin
Austin, TX 78712
{bgrant,agoston}@mail.utexas.edu

***Abstract*—In this paper we apply reinforcement learning to the problem of routing algorithm selection in mobile ad-hoc networks (MANETs). We hoped to show results indicating that an agent dynamically switching routing algorithms using a learned policy could outperform agents using a fixed routing algorithm, but our results are inconclusive. We arrive at these results through simple discretization of sensed network context and Q-learning of state-action values. Our results could likely be improved with more complex function approximation using, for example, neural networks.**

I. Introduction

Mobile Ad-Hoc Networks (MANETs) differ from traditional wired networks in many ways. They "have no fixed routers; all nodes are capable of movement and can be connected dynamically in an arbitrary manner. Nodes of these networks function as routers which discover and maintain routes to other nodes in the network," [1]. Because of these differences, many routing algorithms have been developed specifically for MANETs; these can be categorized into *proactive* and *reactive* algorithms.

Proactive algorithms periodically send out requests to populate their routing tables, while reactive algorithms wait until routes are needed before finding them; in programming language terms, this can be thought of as *lazy* routing. Reactive algorithms are intended to require less overhead, especially in MANETs where routes often change; however, they incur a cost in latency the first time a route is needed. For applications that are latency sensitive, including realtime applications such as voice and video, there could be situations where proactive algorithms are still preferable, for example when node speed is slow and routes are changing slowly. Alternately, for fast moving nodes and highly varying routes, overhead considerations may give reactive protocols the advantage.

In this paper we consider a well known proactive routing algorithm, Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) [2], and the latest evolution in a series of reactive routing algorithms, Dynamic MANET On-Demand Routing (DYMO) [3]. DYMO is based on the well-known reactive algorithm Ad Hoc On-Demand Distance-Vector Routing (AODV) and also borrows from Dynamic Source Routing (DSR) [4]. We refer specifically to DYMO-FAU, a version of DYMO implemented in the OMNeT++ simulator [5].

Our goal in this paper was to show that the simple strategy of switching between routing algorithms via a policy learned through reinforcement can outperform the strategy of using a single routing algorithm. While the idea of using reinforcement learning (RL) in routing has been examined in such papers as [6] and [7], these papers define a routing protocol from the ground up. Our simpler strategy is more like the idea outlined in [8]; there the authors explicitly define switching policies for scenarios where they believe one routing algorithm becomes superior to another. Using reinforcement learning, we automatically derive such polices and as a side-effect learn about the performance characteristics of the employed routing algorithms. Though our results are inconclusive, they are a stepping stone for further work.

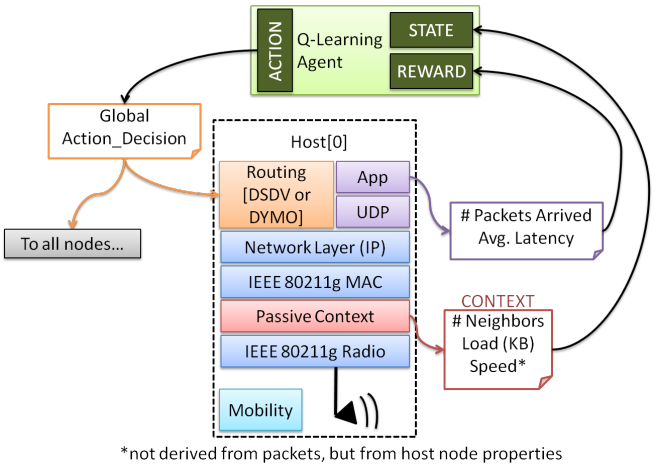

Fig. 1. System architecture

II. System Architecture

The top-level architecture of our system is shown in Figure 1. Our learning agent, implemented in PyBrain [9], generates experience by running network simulations in the `inetmanet` framework of the OMNeT++ network simulator [10].

A. Learning Framework

We used the reinforcement learning framework in PyBrain for our learning-agent implementation. PyBrain is an object-oriented machine-learning framework written in Python, with classes corresponding to familiar aspects of the reinforcement learning problem (Figure 2). We derived our own versions of

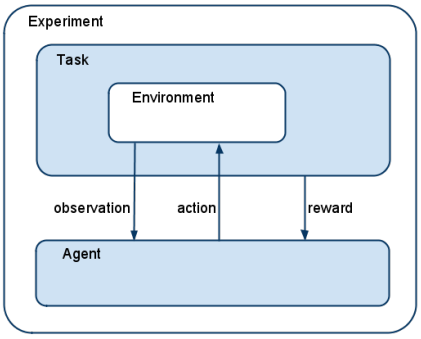

Fig. 2. RL in PyBrain

the PyBrain `Environment` and `Task` classes wherein we defined how the agent obtains observations from its *sensors* and how it receives rewards (see `manet_learner.py`). Using this framework was somewhat restricting, but it will allow us to easily try other RL algorithms in future work.

The key methods in our derived `OmnetEnvironment` class are `performAction()` and `getSensors()`. `performAction()` runs a perl script `rlRunExperiment.pl` which creates an OMNeT++ initialization file and runs a network simulation with a chosen action (DSDV or DYMOFAU). This simulation returns state and reward data which `getSensors()` parses and passes up to our derived `OmnetTask`. In `OmnetTask`, we quantize the sensor values to discrete states, and compute reward as

$$reward = throughput - 1000(latency), \qquad (1)$$

where throughput is the number of delivered packets in a single OMNeT++ simulation, and latency is the average delay in packet delivery over the same period. This reward function emphasizes latency (as necessary for a realtime application like video or voice) while still ensuring a reasonable number of packets are delivered. The coefficient "1000" was chosen to make throughput and latency the same order of magnitude on average.

We chose three aspects of network context to form our states: *load* (KB), *node speed* (m/s), and *number of neighbors*. *Load* is the total amount of traffic seen by our source node in a single network simulation, *node speed* is the fixed speed at which all nodes in a simulation travelled, and *number of neighbors* is the total number of unique neighbor nodes sensed by our source in one simulation.

Since these variables are continuous (or discrete but widely varying), we chose to quantize each of these variables to three state levels: low, medium, and high. To form reasonable bins for these variables, we ran a sample set of 1000 simulations and calculated the first and second tertiles for each state range. We used tertiles instead of simply dividing the range by three to ensure outliers did not skew our bins; we quantized the range so that equal numbers of samples fell into each bin. These bins are calculated in `get_param_ranges.py`.

Parameter	Initial	Min	Max	Δ
Number of Nodes	12	5	20	+/- 1 (every 6 runs)
Speed (mps)	5	0	10	+/-0.1
Packet Size (B)	500	4	1000	+/- 10
Send Freq. (pkts/sec)	0.1	0.01	100	+/- 0.1

TABLE I
PARAMETER VARIATIONS FOR LESS-PERTURBED SIMULATION

B. Network Simulation

We use the OMNeT++ network simulator to generate experience for our learning agent. "OMNeT++ is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators," [10]. In particular, we use framework called `inetmanet` that is specifically designed for simulating MANETs and includes implementations of many routing algorithms such as DSDV and DYMO. Additionally we use the passive context sensing module developed by Petz et. al [11] to observe state from our source node in a realistic way.

From PyBrain's point of view, OMNeT++ is encapsulated inside the `Environment` class as shown in Figure 2.

C. Fault Tolerance and Intermediate Results

Since the learning process took quite a long time (days), and since we rely on a large amount of research-quality code shipped with the OMNeT++ simulator, our code was written to restart after the occasional OMNeT++ segfault and to write intermediate results to disk after every ten simulator runs. This worked well for safety reasons and also allowed us to load and examine intermediate results from disk as our agents were still learning. Since we periodically serialized all important agent state, we were also able to restart agent learning after examining their Q-tables for convergence when we thought it necessary.

III. EXPERIMENTS

A. Methodology

In all of our experiments, we used the PyBrain Q-learning agent with the `EpsilonGreedyExplorer` and the following parameters:

$$\alpha = 0.1$$
$$\gamma = 0.9$$
$$\epsilon = 0.05$$

Our OMNeT++ wrapper code stored various network parameters between runs and varied them by adding a random delta each run. This ensured that parameters would vary smoothly between runs, reflecting realistic state transitions. We began by varying parameters more widely before switching to the settings in Table I.

In all our experiments we execute ten-second runs of the OMNeT++ simulator to gather each state observation and reward. Nodes follow the Random Waypoint mobility model.

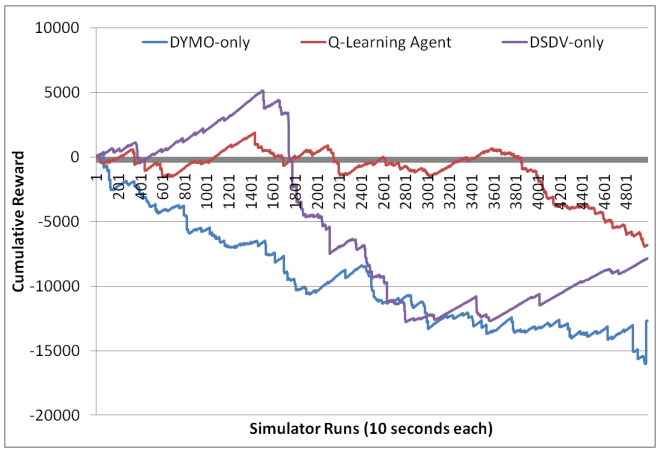

Fig. 3. Policy after 5000 runs

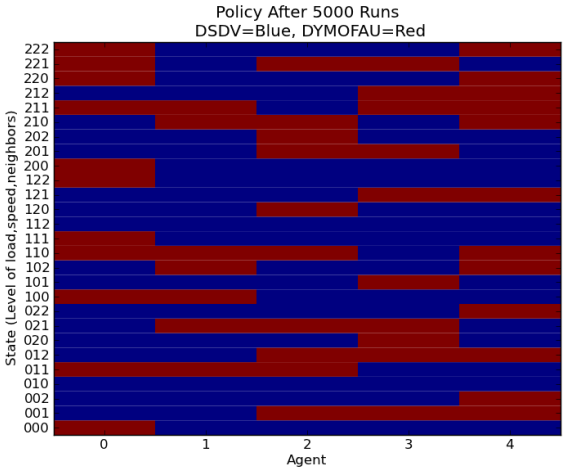

Fig. 4. Policy after 5000 runs

Since the simulator is restarted for every simulation, the routing tables and ARP caches are cleared on all nodes between runs. This affects our experimental validity particularly in the case where our learner does not decide to switch routing algorithms, and it strongly penalizes algorithms with long setup time. We settled on this implementation after problems with other methods, as described in Section V.

B. Results

In Figure 3 we see the cumulative reward of Q-learning agents (mean of 5 agents) vs DSDV- and DYMO-only agents (mean of three agents) after 5000 simulator runs (5000 observations). This graph shows the performance of agents before we switched to the *less perturbed* parameters as shown in Table I. The policies for these agents are shown in Figure 4.

After switching to these smoother and (we believe) more realistic state transitions, we see the results in Figure 5, which shows the cumulative reward of single agents. This graph seems promising, as several of the agents already seem to do better than both fixed-algorithm policies, and all the agents do better than the DYMO-only policy by the end. The policies, as shown in Figure 6, also seem promising, as there are several states across which all agents already agree.

We assumed that with more runs, agents would converge to the same optimal policies, but after 50000 runs, this does not seem to be the case. The polices for two agents after 50000 runs are shown in Figure 7. Only 50% of the Q-values are the same, which would occur by random chance. The cumulative reward also shows this inconclusive result (Figure 8); the cumulative reward of the two learning agents and the DSDV-only policy cluster together, all dominating the DYMO-only policy by the end. The learned policies do not clearly dominate DSDV-only.

In Figures 9, 10, and 11 we show the derived Q-tables for different levels of load, node speed, and number of neighbors, to see what the one agent learned about the state-action space. We compute these graphs by treating the Q-values as expected values and combining them accordingly. Unfortunately, DSDV

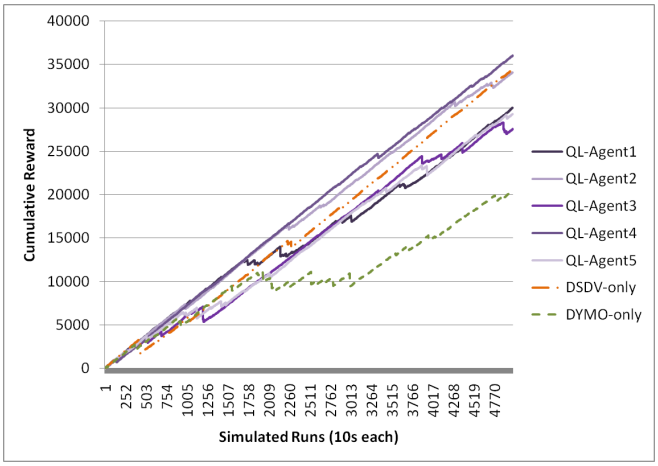

Fig. 5. Policy after 5000 runs

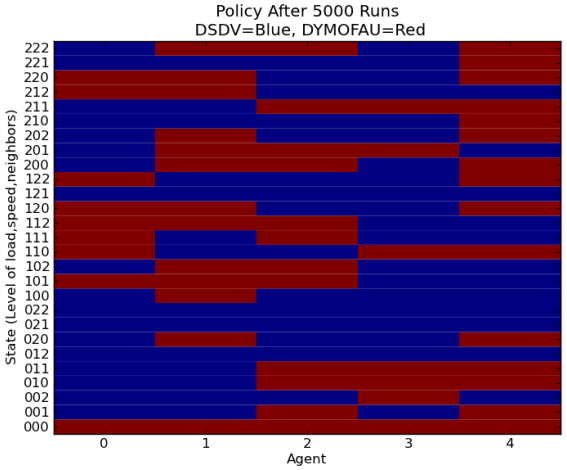

Fig. 6. Policy after 5000 runs, smoother changes

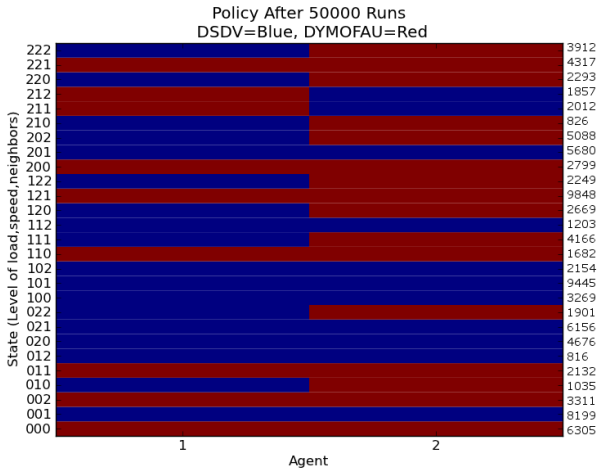

Fig. 7. Policy after 50000 runs

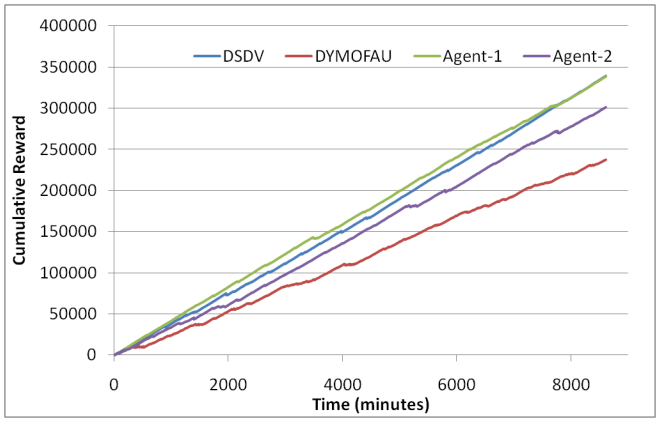

Fig. 8. Cumulative Reward after 50000 runs

is shown as having a higher expected-future-reward in all three graphs, for all levels. The agents do still choose DYMO for some state values (as seen in Figure 7), so none of our three state variables are obviously non-useful in selecting a routing algorithm.

Finally the polices for two agents after over 70000 runs are shown in Figure 14. By just looking at this plot, the agents do not seem to be quickly converging to exactly the same policy, and these runs took approximately 36 hours each. However, examining their Q-tables in Figures 12 and 12 shows them to be closer than the policies indicate.

Our hypothesis is that in most all states in our parameterization, DSDV is superior, and in the rest, DSDV and DYMO perform equivalently. This would explain why in the cumulative reward graphs the learning agents performance is similar to DSDV-only even though the agents still sometimes choose DYMO in their policies.

We doubt that this is intrinsic to the algorithms; it is more likely a result of our coarse quantization of the state space, and possibly that we didn't include other network statistics

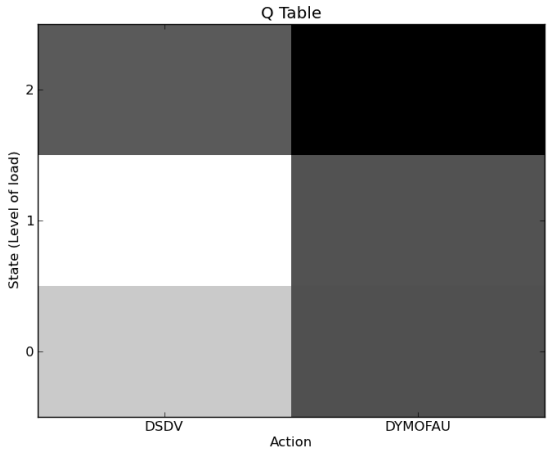

Fig. 9. Q-table for load

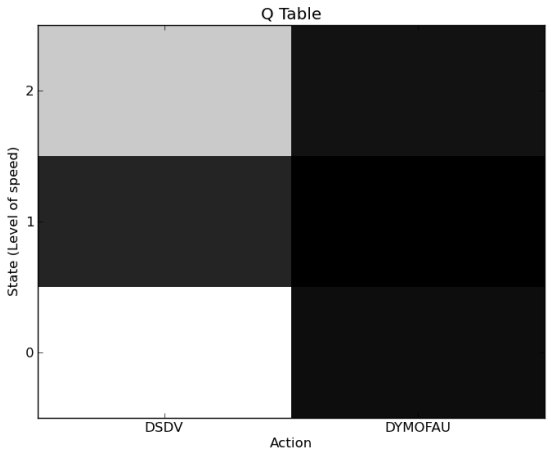

Fig. 10. Q-table for speed

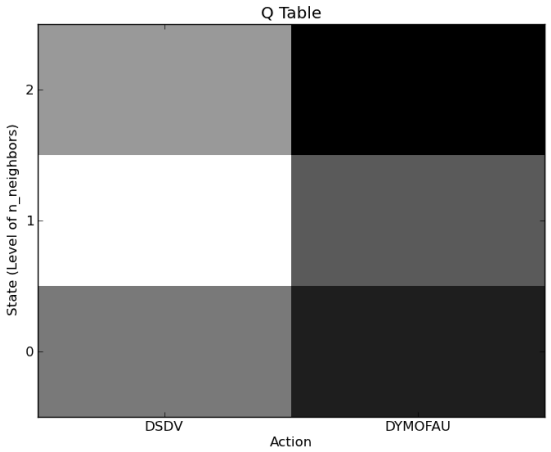

Fig. 11. Q-table for neighbors

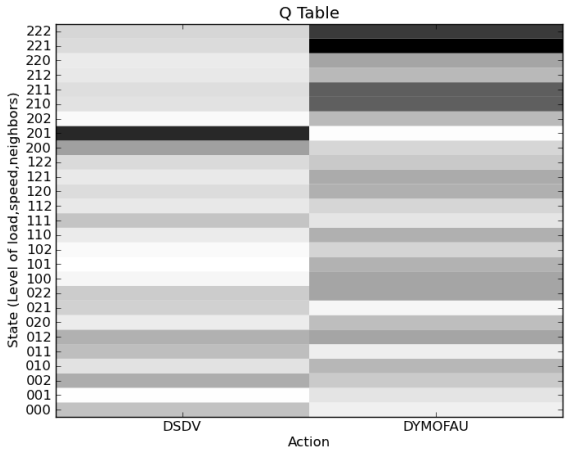

Fig. 12. Q-table for speed

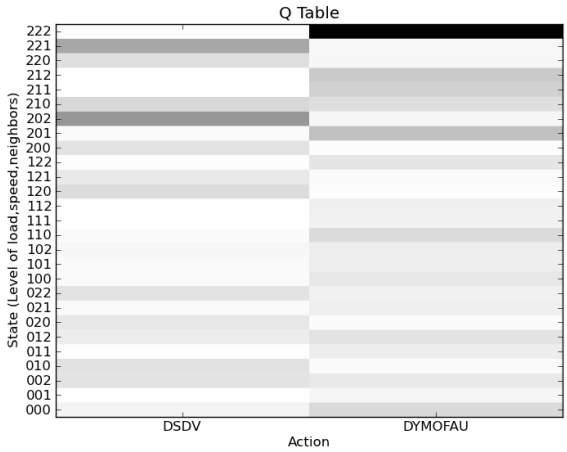

Fig. 13. Q-table for speed

like *jitter*. Papers such as [4] come to the opposite conclusion about the ordering of the two algorithms, which is another reason to suspect our parameterization.

IV. Is this a Markov Decision Process?

An initial question about this domain was whether it is really a Markov Decision Process (MDP) or a degenerate MDP as we suggested in our proposal. More specifically, in our system does the chosen action affect which state occurs next, or is the domain better modeled as a graph of bandits? Qualitatively, of our state variables, we believe that `load` and `neighbors` could be affected by routing algorithm choice. We hypothesize that DSDV could increase load and (in some situations) discover more neighbors because of its proactive nature. `node speed` is set independently of action. We believe we could measure this by calculating

$$p(state|action) \tag{2}$$

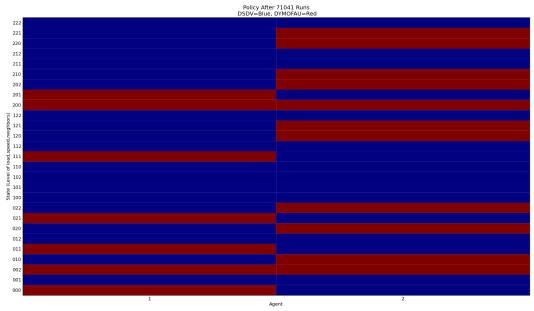

Fig. 14. Policy after 70000 runs

for all states for an agent that always explored. After a sufficient amount of experience, if

$$p(state|DSDV) = p(state|DYMO) \tag{3}$$

for all states (to some tolerance), then we would consider it shown that this domain is a degenerate MDP. Unfortunately we did not have time to run such an agent.

V. Problems

We encountered a number of practical problems before converging on our final implementation.

A. Network Simulation

Our originally proposed idea was to switch between a standard MANET routing algorithm (such as DSDV or DYMO) and epidemic routing [12]. This would have been a slightly nicer problem, as the difference between these algorithms and their intended scenarios is more stark than between DSDV and DYMO. The practical problem in this proposal turned out to be finding a network simulator that contained implementations of both a standard MANET routing algorithm and epidemic routing.

OMNeT++ has plenty of MANET routing algorithm implementations, but epidemic routing was only implemented for an old version of the simulator that passive context sensing does not work with.

The NS-3 simulator seemed promising [13], as the RapidNet fork seemed to have implementations of many routing algorithms [14] [15]. Unfortunately, these were mostly proofs-of-concept for their protocol language, "Network Datalog", and they were undocumented and not flexible enough for experimentation.

After deciding that switching between DSDV and DYMO would be equally interesting from a learning point of view, if more difficult, we returned to the OMNeT++ simulator and `inetmanet` framework that we had the most experience with. Our original architecture involved switching routing protocols dynamically within an OMNeT++ simulation by enabling and disabling routing stacks within each node. While conceptually very straightforward, this approach turned out to intractable in the time we had since routing protocols seem

to store away state that is non-obvious from the abstractions presented by the simulator. While we seemed to be able to dynamically change routing protocols on the surface, the algorithms didn't behave as expected and often crashed. We then settled on our final strategy of restarting the simulator for every action, ensuring we simulated "smooth" state transitions by keeping node positions and other state externally in text files and propagating this state to the next run through simulator initialization.

B. Learning

We also had some practical problems finding a learning framework to use. We initially intended to implement our learning algorithms in C++ within a network simulator, so we searched for C++ reinforcement learning frameworks in hopes of getting well tested code and easy to swap algorithms. Both the Reinforcement Learning Toolbox 2.0 [16] and LibPG [17] looked reasonable, but both had compilation problems. We submitted several issues to the bugtracker for LibPG with no response.

After settling on our system architecture we were free to look for RL frameworks in other languages. PyBrain became the obvious choice, because it was one of the few that ran without problems. Since we only used Q-learning in these experiments it may have been more trouble than it was worth, but in future work we will be able to leverage other algorithms written for PyBrain.

VI. Future Work

There is much room for future work on this topic, including

1) trying other continuous function approximation methods and related learning algorithms (e.g. neural-fitted q-learning in PyBrain),
2) implementing epidemic routing and pursuing our initial proposal, and
3) running our algorithms on real robots in the PHAROS testbed to remove simulator bias[18].

References

[1] E. Royer and C.-K. Toh, "A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Communications*, vol. 6, no. 2, pp. 46–55, 1999. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=760423

[2] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," in *Proceedings of the conference on Communications architectures, protocols and applications*. ACM, 1994, p. 244. [Online]. Available: http://portal.acm.org/citation.cfm?id=190336

[3] I. Chakeres and C. Perkins, "Dynamic MANET on-demand (DYMO) routing," *IETF draft*, 2010.

[4] N. Sivakumar and S. K. Jaiswal, "Comparison of dymo protocol with respect to various quantitative performance metrics," in *Proceedings of IDT Workshop on Interesting Results in Computer Science and Engineering*, October 2009.

[5] C. Sommer, "Dymo-fau: Dynamic manet on demand (dymo) routing for omnet++," 2011. [Online]. Available: http://www7.informatik.uni-erlangen.de/~sommer/omnet/dymo/

[6] P. Stone, "Tpot-rl applied to network routing," in *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[7] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," *Advances in Neural Information Processing Systems*, 1994.

[8] S. Zhao, "Policy-based adaptive routing in ad hoc wireless networks," Spring 2004, rutgers ECE 572 Final Project Report.

[9] CogBotLab and Idsia, "Pybrain v0.3 documentation: Reinforcement learning," November 2009. [Online]. Available: http://pybrain.org/docs/tutorial/reinforcement-learning.html

[10] OMNeT++ Community, "Omnet++." [Online]. Available: http://www.omnetpp.org

[11] A. Petz, T. Jun, N. Roy, C.-L. Fok, and C. Julien, "Passive network-awareness for dynamic resource-constrained networks," in *To be published at the 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS '11)*, 2011.

[12] A. Vahdat and D. Becker, "Epidemic routing for partially-connected ad hoc networks," *Citeseer*, 2000.

[13] ns-3 project, "The ns-3 network simulator."

[14] V. Nigam, L. Jia, A. Wang, B. T. Loo, and A. Scedrov, "An operational semantics for network datalog," 2010.

[15] B. T. Loo, "The design and implementation of declarative networks," Ph.D. dissertation, University of California, Berkeley, 2006.

[16] G. Neumann, "Reinforcement learning toolbox 2.0." [Online]. Available: http://www.igi.tugraz.at/ril-toolbox/general/overview.html

[17] D. Aberdeen and O. Buffet, "Libpg." [Online]. Available: http://code.google.com/p/libpgrl/

[18] C. Fok, A. Petz, D. Stovall, N. Paine, C. Julien, and S. Vishwanath, "Pharos: A Testbed for Mobile Cyber-Physical Systems," The University of Texas at Austin, Tech. Rep., 2010.