# Adaptive Intelligent Scheduling for ATM Networks

Raman K. Mehra, B. Ravichandran, and Richard S. Sutton[1]
Scientific Systems Company Inc.
500 West Cummings Park, Suite 3000
Woburn, MA 01801

**Abstract**

Asynchronous Transfer Mode (ATM) is a fast emerging information technology that promises to provide inter-operable multi-media services for commercial and defense applications. Unlike commercial broadband networks which ATM was originally designed for, defense or tactical ATM networks must be able to traverse low-rate transmission links. To allow more flexible and efficient use of this limited bandwidth resource, optimal traffic management in ATM networks is critical. This paper demonstrates the feasibility of developing Self-Learning Adaptive (SLA) scheduling algorithms using Reinforcement Learning (RL). This technique was applied to simulated ATM data and proved to be more efficient than fixed static scheduling methods.[1]

## 1 Introduction

Today's information explosion is finding its way into the Department of Defense's tactical communication networks. Asynchronous Transfer Mode (ATM) is a fast emerging information technology that promises to provide inter-operable multi-media services to the warrior. Unlike commercial broadband networks which ATM was originally designed for, tactical ATM networks must be able to traverse low-rate transmission links that are typically between 256 kb/s and 4.096 Mb/s. These low-rate tactical transmission systems require more efficient ways to handle cell-based ATM technology. Current tactical multiplexing techniques do not provide the flexibility for dynamically allocating bandwidth that commercial ATM technology promises. Methods must be provided for freeing up the idle channel bandwidth on tactical links so that it can be made available to support additional multi-media and high speed data services. This will provide an efficient method for utilizing all the available bandwidth while not degrading existing services.

Traffic on most ATM based networks is partitioned into classes based upon their quality of service requirements. A traffic class can be defined by its maximum allowable queuing delay and a minimum cell loss probability. Thus, congestion can be simply defined as a failure to meet the quality of service requirements. In switching nodes, congestion control is typically achieved through multiple output buffers and queue management schemes. Where there are multiple service classes, the output controller must decide in what sequence to service the queues in order to maximize the number of cells transmitted subject to the quality of service constraints. For example, a simple control policy may be to service all highest priority cells first and, only when that queue is empty, accept lower priority cells. This problem of cell scheduling is addressed in this paper. A detailed version of the paper can be found in [8].

Service disciplines for cell scheduling with multiple traffic classes that have been proposed often assume knowledge of the entire busy period in advance, or do not take into account the size of the load on each queue. In addition, scheduling algorithms may require heuristics along with causal algorithms to reduce the computational cost. Fortunately, patterns of ATM cell arrivals are highly correlated, a feature which we have exploited with an adaptive scheduling approach which uses an adaptive algorithm based on Reinforcement Learning.

## 2 Reinforcement Learning

Reinforcement learning (RL) is a collection of mathematically principled methods for approximately solving stochastic optimal control problems. RL methods are novel combinations of dynamic programming (DP) methods, stochastic approximation methods, and learning methods developed by artificial intelligence and neural network researchers. RL is based on classical optimal control methods and inherits much of their mathematical structure, but also extends them in three ways. 1. RL methods can learn optimal behavior directly from experience, if necessary; they do not require complete knowledge of the problem in order to solve it. 2. RL uses function approximation methods such as neural networks to generalize across states. 3. RL uses Monte Carlo sampling to direct computation towards the most relevant parts of the state space. The above extensions enable RL

[1]Richard S. Sutton is with the University of Massachusetts, Amherst, MA. This work was supported by Rome Laboratory F30602-95-C-0181: Alan Akins - Technical Monitor.

methods to effectively solve very large stochastic decision problems that would be intractable using conventional exact methods. In reinforcement learning problems, a scalar value called a *payoff* is received by the control system for transition from one state to the other. The aim of the system is to find a control policy that maximizes the expected future discounted sum of payoffs received, known as the *return*. The value function is a prediction of the return available from each state:

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_k\} \quad (1)$$

where $r$ is the payoff received for the transition from state $s$ to $s'$ and $\gamma$ is the discount factor ($0 \leq \gamma \leq 1$).

Consider learning the value function for a Markov chain. These problem are often solved using algorithms based upon dynamic programming [1, 2] and involves storing information associated which each state, then updating the information in one state based upon the information in subsequent states. For predicting the outcome of a Markov chain, the learning algorithm:

$$V^{\pi}(s) \leftarrow r_t + \gamma V^{\pi}(s') \quad (2)$$

is equivalent to the TD(0) algorithm [3]. A Markov chain is a degenerate Markov Decision Process (MDP) for which there is only one possible action to choose from in each state. Thus, considering a MDP, for which there is more than one action in each state an incremental form of value iteration is:

$$V^*(s) = \max_a E_a\{r + \gamma V^*(s')\} \qquad \forall s \in S \quad (3)$$

Watkins [4] introduced the idea of *Q–Learning* where the idea is to learn the value of a state-action pair $(s, a)$ rather than to learn the value of a state $(s)$ alone. The prediction is updated with respect to the predicted return of the next state visited:

$$\hat{Q}^{\pi}(s, a) \leftarrow r + \gamma \max_{a \in A} \hat{Q}^{\pi}(s', a) \quad (4)$$

The update equation for this case is:

$$\hat{Q}^{\pi}(s, a) = \hat{Q}^{\pi}(x, a) + \alpha[r + \gamma \max_{a \in A} \hat{Q}^{\pi}(s, a) - \hat{Q}^{\pi}(s, a)] \quad (5)$$

This has been shown to converge for finite–state MDP problems when a lookup table is used to store values of the Q–functions [4]. Once the Q–functions has converged, the optimal policy is to take the action in each state with the highest predicted return; this is called the *greedy* policy. In formulating Q-Learning, it is assumed that $\max_{a \in A} \hat{Q}^{\pi}(s, a)$ provides the best estimate of the return of the state $s$. Sutton [5] along with Rummery and Niranjan [6] argues against this and present a case for using $\hat{Q}^{\pi}(s', a')$ instead of $\max_{a \in A} \hat{Q}^{\pi}(s, a)$ and use:

$$\hat{Q}^{\pi}(s, a) \leftarrow r + \gamma \hat{Q}^{\pi}(s', a') \quad (6)$$

to update the prediction. In this case the update equation is:

$$\hat{Q}^{\pi}(s, a) = \hat{Q}^{\pi}(s, a) + \alpha[r + \gamma \hat{Q}^{\pi}(s, a) - \hat{Q}^{\pi}(s, a)] \quad (7)$$

Sutton calls this algorithm SARSA because you need to know *State Action Reward State Action* before an update is performed. Rummery and Niranjan call this algorithm Modified Q–Learning.

Sutton [5] has applied SARSA to continuous-state control problems. These include *puddle world, mountain car*, and *acrobot*. In puddle world, the object is navigate around puddles in a 2D terrain en-route from a start to a finish position. In mountain car, a car must climb up a mountain, but the engine is too weak to accelerate directly up the slope, hence, it must first move away from it. The acrobat is a two link under actuated robot. The first joint cannot exert torque, but the second joint can. The object is to swing the endpoint above the bars by an amount equal to one of the links. In all of these problems SARSA reports robust performance.

# 3 ATM Scheduling Using Reinforcement Learning

Schwartz [7] developed an adaptive scheduling algorithm based on an urgency function that maximizes the long term payoff from using a given policy. His learning algorithm is based on the TD(0) algorithm, which does not involve any optimization. The resulting algorithm is, therefore, not optimal in the sense of Dynamic Programming. We reformulate the urgency scheduling problem and use the SARSA learning algorithm.

The incoming traffic into an ATM switch is stored in buffers corresponding to their traffic class, and the problem of scheduling corresponds to deciding what sequence to service the buffers in order to maximize the cells transmitted subject to the quality of service constraints. The states $(x)$ are a combination of both the length $(n)$ of a buffer and the waiting time $(\tau)$ of the packet at the head of the buffer, i.e. $x = (n, \tau)$. Since the waiting time information for a call is not available and the queue lengths can be large, we model the state as representing some combination of buffer occupancy fractions. For simplicity consider the case where there are only two traffic classes and we quantize the buffer occupancy corresponding to each traffic class into 3 classes, corresponding to empty, half full, or a completely full buffer. For this example there are nine states as shown in Table 1.

| | | B2 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| B1 | 0 | 1 | 2 | 3 |
| | 1 | 4 | 5 | 6 |
| | 2 | 7 | 8 | 9 |

Table 1: States Representing Buffer Occupancy

Buffer 1 occupying B1 can be 0,1,or 2. Likewise B2 has three classes. And, state 1 represents the case where B1 = 0 and B2 = 0. state 2 represents the case where B1 = 0 and B2 = 1 etc. At each state we would like the adaptive scheduling algorithm to service the buffers in an optimal fashion.

SARSA attempts to obtain a DP feedback policy by associating with each state an action that specifies the optimal choice of service for that state. Table 2 shows an outline of the SARSA algorithm we use. This version of the algorithm is a simplified version of the SARSA algorithm described in [5] and does not use eligibility traces or CMAC tilings because of the size of the state space.

Step 1. Initialize $Q(s,a)$, $\forall s \in S, a \in A(s)$

Step 2. Start Trial: Initialize $s, a$

Step 3: Take action $a$.
Observe resultant reward, $r$, and state $s'$

Step 4: Choose next action:
$a' = \epsilon$-*greedy-policy*$(s', Q)$

Step 5: Use TD to update value:
$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$

Step 6: If $s'$ is terminal end,
else set $s = s'$ and $a = a'$ and goto to Step 3

Table 2: The SARSA algorithm

*Example 1: 9 States*

We begin with applying the SARSA algorithm to a simplified problem. The network model we use is as follows: Packets arrive from two traffic classes and are sorted into two buffers. Each buffer has three states Thus the state space is described as in Table 1. The packets arrive with probabilities as shown in Table 3 In this example I1 corresponds to an input that fills up half of buffer 1 At the output, only one unit can be drained per time step. This is akin to specifying the output link capacity as half of the buffer size per decision step. Thus at each time step the controller must choose between one of two actions. Action One: remove a unit from B1, and Action Two: remove a unit from B2. We also design these actions such that when Action One is chosen and B1 is empty, then it will drain one unit from B2. Thus, these actions can be generalized and specified as Action One: Drain B1 per the link capacity and then drain B2. Action Two: Drain B2 per the link capacity and then drain B1. We also place a priority on B1 over B2 and stipulate that it is twice as expensive to drop a unit from B1 compared to B2. Thus, observe that a *highest priority first policy* will always choose Action One if there are one or more units of packets in B1. An optimal policy will specify either Action One or Action Two depending on the given state. A purely *random policy* will randomly decide between Action One and Action Two independent of the state. Note that both the random policy and the highest priority first policy are valid policies, even though they are not optimal policies under all circumstances. As per the description in Table 2 we have described the *states* and *actions*. We now describe the specification of *rewards*. The reward corresponds to the function we want to optimize: in this case it is the loss function $J$ defined as $J = \frac{1}{w1+w2}[w2 * L1 + w2 * L2]$, where L1 = ( Units Dropped in B1 / Units Received in B1) and L2 = (Units Dropped in B2 / Units Received in B2). We also place a priority on B1 over B2 and stipulate that it is twice as expensive to drop a unit from B1 compared to B2, thus $w1 = 2$ and $w1 = 1$. Now the reward $r$ at each time step is: $r = w2 * d1 + w2 + d2$ where $d1$ is the number of units from $B1$ dropped in that time step and $d2$ is the number of units from $B2$ dropped in that time step. The network model we use is as follows: Units of packets arrive from two traffic classes and are sorted into two buffers. The state space corresponding to three levels of buffer occupancy is described as in Table 1. The inputs arrive with probability per time step as shown in Table 3. At the output, only one unit is drained per time step. In this example the average input arrival rate is one unit per time step. However, congestion can occur because with probability 1/4 one unit can arrive into B1 and one unit can arrive into B2, and there is no guarantee that either B1 or B2 will have room for these units. At each time, step the controller chooses between one of two actions. Action One: Drain B1 per the link capacity and then drain B2. Action Two: Drain B2 per the link capacity and then drain B1. We also place a priority on B1 over B2 and stipulate that it is twice as expensive to drop a unit from B1

| | | I2 | |
|---|---|---|---|
| | | 0 | 1 |
| I1 | 0 | 1/4 | 1/4 |
| | 1 | 1/4 | 1/4 |

Table 3: Input Arrival Probability Distribution

compared to B2, thus $w1 = 2$ and $w2 = 1$. We use SARSA algorithm to compute the optimal policy and compare it against a *highest priority first* (HPF) policy that always chooses Action One if there are one or more units of packets in B1. Figure 1 shows the cost functions from a sample run. The network was run for 20000 time steps. Notice that it took about 2000 time steps for SARSA to learn, and from that point on it behaves better than HPF. The loss rate by using the SARSA policy is 20% lower than the loss rate using HPF.

Figures 4 and 5 show "snapshots" of examples using the SARSA and Highest Priority First policy, respectively. Note that these snapshots are only for 100 time steps and they do not reflect the time average performances as shown by the loss (J) function plots. Each figure has six panels. The top left panel shows the traffic coming in to B1. The top right shows the traffic coming in to B2. The middle left panel shows the buffer occupancy in B1. The middle right panel shows the buffer occupancy in B2. The bottom left panel shows the buffer occupancy in B2. The bottom right panel shows the buffer occupancy in B2.

*Example 2: 36 States*

The input and output parameters for this model are similar to the parameters in Example 1. However, the buffer occupancy states are quantized into 6 levels of occupancy. Thus, the state space is larger and has 36 states. Figure 2 shows the Loss functions from a sample run. The network was run for 20000 time steps. Notice that it took about 2000 time steps for SARSA to learn, and from that point on it behaves better than HPF. The loss rate by using the SARSA policy is 33% lower than the loss rate using HPF.

*Example 3: 36 States, Unequal Arrival Rates*

Example 3 is similar to Example 2 except that, the input traffic model is changed and is shown in Table 4. In this case, the arrival rate for B2 is greater than the arrival rate for B1. In this example the average arrival rate is 1.5 units per time step (0.5 units per time step for B1 and 1.0 units per time step for B2). Since the drain rate is 1 unit per time step, congestion will occur In this case SARSA took about 60000 time

| | | I2 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| | 0 | 1/6 | 1/6 | 1/6 |
| I1 | 1 | 1/6 | 1/6 | 1/6 |

Table 4: Input Arrival Distribution

steps to converge. These Q values are saved and the simulation is started again with Q values initialized to the saved Q values. This is run for about 5000 time steps. Figure 3 shows the Loss functions for the last 500 time steps from a sample run. Observe that SARSA has lower loss than HPF.

# 4 Simplified Implementations of Reinforcement Learning

Reinforcement Learning (RL) policies are optimal, but require a large amount of computation. In this section we describe some techniques to implement the RL policies by approximating them with simple heuristic policies.

First, we examine the policies found by the SARSA RL algorithm. The RL actions corresponding to each state for Example 1 are shown in Table 5. For

| | | B2 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| | 0 | A1 | A2 | A2 |
| B1 | 1 | A1 | A1 | A2 |
| | 2 | A1 | A1 | A1 |

Table 5: Optimal Policy Learned by Sarsa for Example 1

Example 2, the RL actions for each state are shown in Table 6. Similarly, the RL actions for Example

| | | B2 | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | A1 | A2 | A2 | A2 | A2 | A2 |
| | 1 | A1 | A1 | A1 | A1 | A1 | A2 |
| B1 | 2 | A1 | A1 | A2 | A1 | A1 | A2 |
| | 3 | A1 | A1 | A1 | A2 | A2 | A2 |
| | 4 | A1 | A2 | A2 | A2 | A1 | A2 |
| | 5 | A1 | A1 | A1 | A1 | A1 | A1 |

Table 6: Optimal Policy Learned by Sarsa for Example 2

3 are shown in Table 7. We would like to examine

| | | B2 | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | A1 | A2 | A2 | A2 | A2 | A2 |
| | 1 | A1 | A1 | A2 | A2 | A2 | A2 |
| B1 | 2 | A1 | A2 | A1 | A1 | A2 | A2 |
| | 3 | A1 | A1 | A2 | A2 | A2 | A2 |
| | 4 | A1 | A2 | A2 | A2 | A2 | A2 |
| | 5 | A1 | A1 | A1 | A1 | A1 | A1 |

Table 7: Optimal Policy Learned by Sarsa for Example 3

if these actions intuitively make sense. To verify this let us look at the actions taken when the buffers are full. For all three examples the optimal actions drain B1 when B1 is full, and B2 when B2 is full, and favors B1 when they are tied. Further in Example 3, look at the scenario when B2 has 4 units of packets in it. In this example, because the peak arrival rate for B2 is 2 units of packets, it is possible for the B2 to

overflow even when it is not full. Therefore, for some combination of buffer occupancy fractions, action A2 is optimal.

One simple rule that performs close to the learned strategies is a *proportional feedback* policy, under which the drain rates are proportional to buffer occupancy levels. In this case the choice of an action is determined by computing the following. Drain from B1 : $Linkmax * w1 * b1/(w1 * b1 + w2 * b2)$ and Drain from B2 : $Linkmax * w2 * b2/(w1 * b1 + w2 * b2)$ where b1 is the occupancy level of B1 and b2 is the occupancy level of B2. If ties are broken in favor of the longest buffer, we can derive actions that are close to the learned strategies. The action maps for each example based on the above proportional feedback policy are shown in Tables 8 and 9. (The action maps are the same for Examples 2 and 3). Notice that for

| | | B2 | | |
|---|---|---|---|---|
| | | 0 | 1 | 2 |
| B1 | 0 | A1 | A2 | A2 |
| | 1 | A1 | A1 | A2 |
| | 2 | A1 | A1 | A1 |

Table 8: Proportional Feedback Policy for Example 1

Example 1 the performance of proportional feedback and SARSA are identical. SARSA performs better

| | | B2 | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| B1 | 0 | A1 | A2 | A2 | A2 | A2 | A2 |
| | 1 | A1 | A1 | A2 | A2 | A2 | A2 |
| | 2 | A1 | A1 | A1 | A1 | A2 | A2 |
| | 3 | A1 | A1 | A1 | A1 | A1 | A1 |
| | 4 | A1 | A1 | A1 | A1 | A1 | A1 |
| | 5 | A1 | A1 | A1 | A1 | A1 | A1 |

Table 9: Proportional Feedback Policy for Examples 2 and 3

than proportional feedback for Examples 2 and 3 because it deals more effectively with the case when B2 is full. This can be seen via the action map in Table 9: Proportional Feedback does not serve buffer B1 as often as Highest Priority First, but does serve buffer B1 more often than SARSA.

In these experiments we evaluated the performance of RL learned policies as compared to a highest-priority-first (HPF) policy and derived a proportional feedback (PF) policy as a close approximation to that found by RL. One remaining concern is that RL methods might suffer from an initial period of low performance while learning occurs. We also explored several ways in which this initial *learning transient* can be minimized or eliminated, by *priming* RL with an initial PF policy. Finally, we noted that SARSA performed better that HPF when the buffer lengths were the same. We also experimented with the case where the buffer lengths were varied. In these examples we assigned the higher priority queue a longer buffer length, and observed that the performance of SARSA over Higher Priority First was even better. These details can be found in [8].

# 5 Conclusions

This paper demonstrates the feasibility of developing Self-Learning Adaptive (SLA) scheduling algorithms using Reinforcement Learning (RL). This technique was applied to simulated ATM data and proved to be more efficient than static methods. Using RL policies as a guide, a simple proportional feedback policy has been identified that performs close to the RL policies and is easy to implement.

# References

[1] D. P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[2] S. Ross. *Introduction to Stochastic Dynamic Programming.* Academic Press, New York, 1983.

[3] R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 1988.

[4] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[5] R. S. Sutton Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding In *Advances in Neural Information Processing Systems 8*, Cambridge, MA. MIT Press. 1996.

[6] G. A. Rummery and M. Niranjan. "On–line Q–Learning using Connectionist systems" Technical Report CUED/F-INFENG/TR 166 Cambridge University Engineering Department, 1994

[7] D. B. Schwartz, "Learning from rare events: dynamic cell scheduling from ATM networks", *Telecommunication Application of Neural Networks*, Kluwer, 1993

[8] R. K. Mehra, B. Ravichandran, R. S. Sutton, S. Ghosh, and R. Jain. "Adaptive Intelligent Scheduling for ATM Networks" Scientific Systems Company Inc. Woburn, MA. Contract F30602-95-C-0181. Rome Laboratory Technical Report RL-TR-96-25. January 1996.

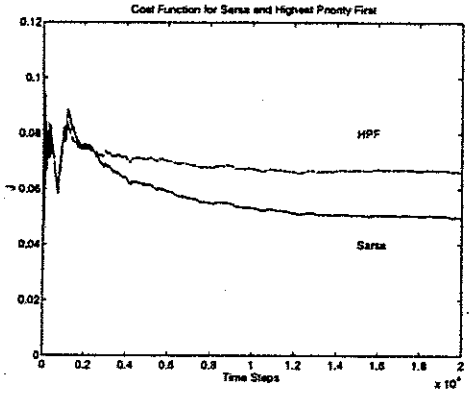


Figure 1: Loss Function for Sarsa and Highest Priority First (HPF) Example 1

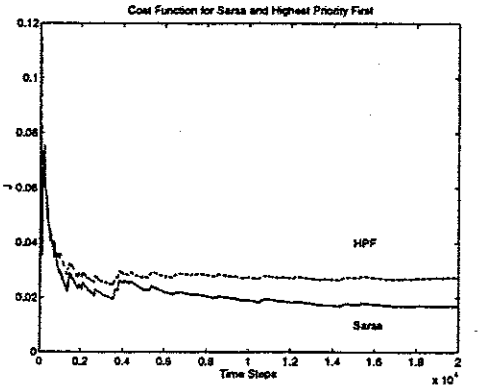


Figure 2: Loss Function for Sarsa and Highest Priority First (HPF) Example 2

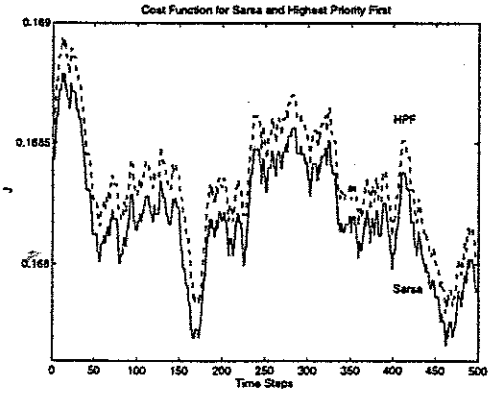


Figure 3: Loss Function for Sarsa and Highest Priority First (HPF) Example 3

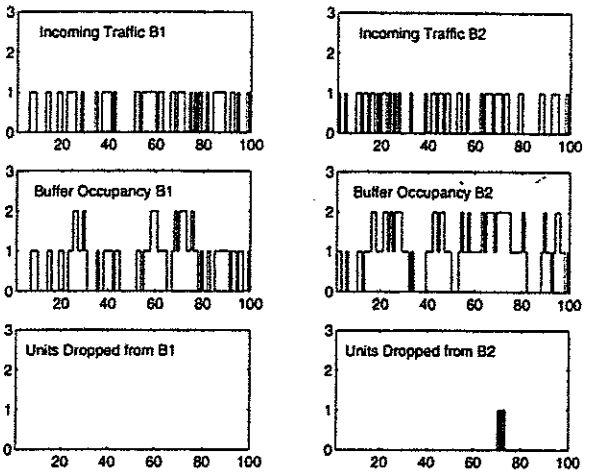


Figure 4: Snapshot of Example 1 using SARSA

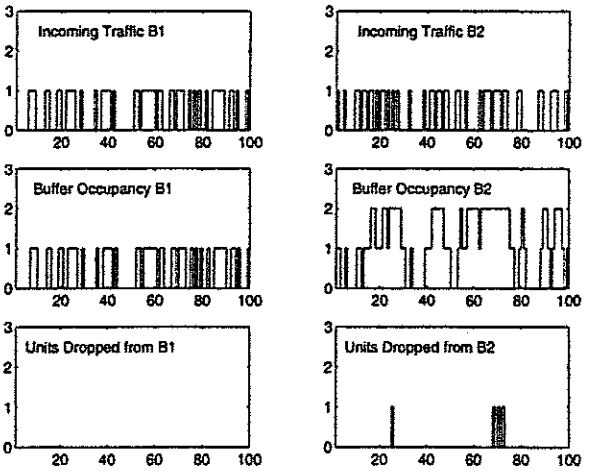


Figure 5: Snapshot of Example 1 using Highest Priority First