CS440 Project: Ship Steering Using Reinforcement Learning

Kyriakos Chatzidimitriou

December 3rd, 2004

Abstract

This report describes the work made for the semester project in the CS440 course of the Computer Science Department in Colorado State University. The project deals with the use of reinforcement learning to a problem from the control literature. The main goal is to steer a ship through a sequence of gates in the minimum amount of time. For constructing the state-action policy of the agent, the Sarsa algorithm, with linear function approximation and tile coding, was applied to four different settings of the same problem and promising results were derived in all four cases. The present report discusses the implementation issues, the fine-tuning of the algorithm and the obtained results.

Contents

1 Introduction

Controlling automated systems and processes is one of the common problems engineers have to face. Their ultimate goal is to design and implement a controller that will operate the process or system at hand, adhering to specifications such as optimality, safety, efficiency, economy and profitability among others. In order to declare success, engineers are required to model the process and the environment in the form of state equations and then design the appropriate controller with the above specifications in mind.

An alternative approach is that of an autonomous system (agent) that eventually learns to operate by itself with respect to the desired objectives. One approach that accomplishes this task is that of reinforcement learning, a methodology that was proposed by scientists in the artificial intelligence domain, who took into account research conducted by animal psychologists. In reinforcement learning the agent learns the desired behavior by interacting with its environment, and modifying its decision-making through reward signals (positive and negative).

In order to demonstrate the reinforcement learning paradigm, a test case from the optimal control research [1] will be used, which is steering a ship through multiple gates in the minimum amount of time. In computer science, the ship steering problem was approached by Rosenstein and Barto [2], who used an actor-critic architecture combined with a supervised learning module. In their discussion they are claiming that for reinforcement learning by itself, it is virtually impossible to solve this problem optimally, but give a very close to optimal solution. This task has been also used in cognitive science research, studying how humans derive cognitive skills, controlling event-driven systems with long time lag [3].

The report consists of five sections. The first section presents the problem of ship steering, provides the model of the ship and concludes with the objectives of the project. Section two introduces the reinforcement learning algorithm and the specific elements that had to be fine tuned. In section three the implementation issues of the project are discussed, while in section four the results are presented and interpreted. Finally the report ends with the conclusions and a paragraph concerned with future work applicable to the ship steering problem.

2 Problem Description

The main goal of the problem is to steer a ship, which is cruising at constant speed, through a sequence of gates. The ship has a particular start position that is defined in the specifications of the problem and has to be maneuvered to reach its goal position, which is a single gate. If in between the start and the goal position there are more gates, the ship has to pass through each one of them, before finishing successfully. Figure 1 shows a simple example of a route that the ship has followed to reach the goal gate. One important issue is the fact that there is a time lag between changes in the desired turning rate and the actual rate, modelling the effects of a real ship's inertia and resistance to water [4].

2.1 Mathematical Formulation

The following equations and constraints provide the model of the ship (plant) as it is sailing cross the ocean:

State :

x, y the coordinates of the ship (m)

θ the orientation of the ship (degrees)

$\dot{\theta}$ the actual turning rate of the ship (degrees/s)

Control :

r the desired turning rate of the ship (degrees/s)

Constraints :

$|r| < 15$ *degrees/sec* ship cannot turn faster than 15 degrees/s

Initial :

$x[0] = 0.5$

$y[0] = \theta[0] = \dot{\theta}[0] = 0$

Equations of motion :

$x[t+1] = x[t] + \Delta V \sin\theta[t]$

$y[t+1] = y[t] + \Delta V \cos\theta[t]$

$\theta[t+1] = \theta[t] + \Delta\dot{\theta}[t]$

$\dot{\theta}[t+1] = \dot{\theta}[t] + \Delta(r[t] - \dot{\theta}[t])/T$

Parameters :

T 5 time constant of convergence to desired turning rate

V 3 m/s

Δ 1.0 sampling interval

Controller inputs-outputs :

Control interval 1.0 s (equal to sampling interval, Δ)

Input $x[t], y[t], \theta[t], \dot{\theta}[t]$

Output $r[t]$

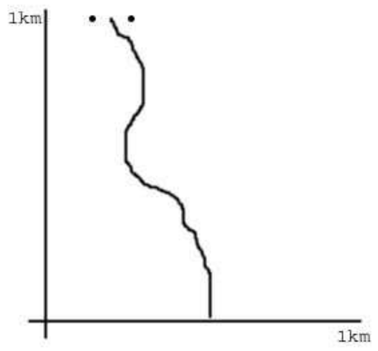

Figure 1: Simple example where a ship is navigating its way through a single gate.

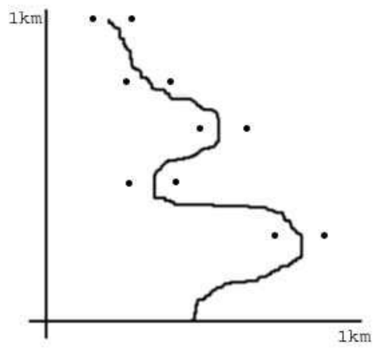

Figure 2: A more complicated example. The ship has to maneuver through a sequence of gates.

2.2 Problem Objectives

The project's objectives follow an incremental level of complexity and difficulty. The first test case is very straightforward. The ship starts at coordinates $(x, y) = (0.5, 0)$ moving straight up ($\theta = 0, \dot{\theta} = 0$). The goal is to generate sequences of r values that steer the center of the ship through sea into the gate at coordinates **(0.2, 1.0)** and **(0.3, 1.0)** within a minimum amount of time.

The next problem is more challenging by adding extra gates. For example, using the same final gate as before, four gates are added with sides at **(0.8, 0.3)** and **(0.9, 0.3)**, **(0.3, 0.6)** and **(0.4, 0.6)**, **(0.5, 0.7)** and **(0.6, 0.7)** and **(0.3, 0.8)** and **(0.4, 0.8)**, as shown in figure 2. Now, planning for future encounters with gates should be part of the current control decision, because the ship's position and orientation, as it moves through one gate, can greatly affect the ease of navigating through successive gates [4].

Finally, in both of the above cases the problem becomes a little bit more complex if constant water current is added(for example water current of speed 1.0 m/s coming from South-East). The problems with the water current are formulated as a simple modification to the plant's equations. In this case the agent alternates the sequence of r values in order to take into account the water current.

3 Methodology

For the simple and for the more complicated examples (more gates, environment disturbances), the methodology followed was that used by Sutton in the acrobot control problem task [5]. That is applying the *Sarsa* algorithm with linear function approximation using tile coding and replacing traces. In such a setting the *Q-values* are going to be represented as linear functions of binary features or tiles. Both examples have in common the fact that they are control task problems with continuous state variables.

The Sarsa algorithm belongs in the family of *Temporal Difference (TD)* algorithms, like *Q-Learning.* The Sarsa algorithm is considered an *On-Policy* algorithm, which means that the updating of the value depends on the choice that has been made by the policy (experience-based), whereas an *Off-Policy* algorithm will update the value regardless of what choice was made, separating learning from current behavior. In the rest of this section the approach in the various elements of the algorithm is going to be presented.

3.1 Algorithm Description

The pseudocode of the algorithm is provided below. The tilings in the algorithm are represented as Θ.

```
Initialize all $\Theta$ and $e$ to 0.
 **Repeat** (for each episode):
    $s \leftarrow$ initial state of episode
    **Repeat** (for each *step* in the episode):
        **For all** $\alpha \in A(s)$:
            $F_\alpha \leftarrow$ set of features present in s, $\alpha$
            $Q_\alpha \leftarrow \sum_{i \in F_a} \Theta(i)$
        $\alpha \leftarrow argmax_\alpha Q_\alpha$
        **With probability** $\epsilon : \alpha \leftarrow$ a random action $\in A(s)$
        Take action $\alpha$, observe reward $r$, and next state $s'$.
        **if**($s'$ is terminal):
            $done \leftarrow$ TRUE.
            **if**($failed$):
                **For all** $i \in F_\alpha$:
                 $\Theta(i) \leftarrow$ negative reward
            **else:**
                **For all** $i \in F_\alpha$:
                 $\Theta(i) \leftarrow$ positive reward
        **if**($step > 1$):
            **For all** $i \in F_{\alpha_{old}}$:
                $\Theta(i) \leftarrow \Theta(i) + a \cdot (r + \gamma \cdot Q_\alpha - Q_{\alpha_{old}})$
            **Back up:**
        $\alpha_{old} \leftarrow \alpha$, $Q_{\alpha_{old}} \leftarrow Q_\alpha$, $s \leftarrow s'$, $F_{\alpha_{old}} \leftarrow F_\alpha$
    **until not** *done*.
```

The learning rate α was set to 0.1 in all testcases and the discount factor γ was set to 1, so no discount was used. All tilings had an initial value of 0.0. Finally the greedy policy factor ϵ was set to 0.05 for the single gate and to 0.01 for the multiple gates problems. When the number of episodes was approaching a pre-specified limit, it was set to 0.0. This choice provided the insurance that the agent would always terminate successfully, after exploration has been made.

3.2 Engineering Tiles

The tilings used, sliced the four state variables in a variety of simple ways. One tiling included all four dimensions x, y, θ and $\dot{\theta}$. The dimensions x, y were sliced into 20 intervals, θ was sliced into 30 intervals and $\dot{\theta}$ was sliced into 31 intervals. Two other tilings included the three dimensions with x, y being in both. For these two, the one with θ was sliced into 50 x 50 x 90 tiles and the one with $\dot{\theta}$ was sliced into 50 x 50 x 31 tiles. Finally, another three tilings were used for dimensions x, y with 500 tiles in each dimension, using a 0.0005 offset to the right and to the top. The total amount of tiles was in the end approximately $1,500,000$. This amount of tiles was be mapped into a single action out of the total 31. These are the available actions if the turning rate is cut into 1 degree/s intervals. Special care was taken for θ and $\dot{\theta}$ so that they never exceed their range of "legal" values. That is $[0, 359]$ degrees and $[-15, 15]$ degrees/s respectively. All intervals were equal.

3.3 Reward Shaping

In order to guide the process of learning more effectively, naive reward shaping was used. By using the reward shaping methodology, we can incorporate domain knowledge into the learning process. First of all each episode could fail in three ways:

1. The simple case of the ship going out of bounds.
2. The ship turning back. In this case the new y has a lower value than the old y. There is no reason to continue the episode since it will delay the learning process (the ship can go into cycles) and not prove useful in our goal of obtaining the minimal amount of time in reaching the goal.
3. The ship passes through the virtual line that is defined by the two gate points. This of course is used only in the multiple-gate test case. The reason is the same as in number 2.

If any of the three case above happens, then the episode is consider as a failure and a reward of -10 is applied. If the episode ends succesfuly, a reward of $+\frac{1.0}{\sqrt{(x_{goal}-x_{init})^2+(y_{goal}-y_{init})^2}}$ is given for both the single gate and the multiple gate problems. Using these fractions the closer the ship reaches the gate from its initial condition, the smaller the steps and therefore the smaller the time to goal. For the intermediate step of the episode the reward is set to 0.0.

4 Implementation Issues

All the algorithms were implemented using the Java programming language. The code was written from scratch, in order to fully comprehend the underlying concepts of the algorithm involved. Output was be directed to console output, to files (for graphical representations) and to a graphical user interface (GUI) in order to visualize the route and the maneuvers of the ship. All graphical figures were derived using `gnuplot` and the report was written using LaTeX.

In total four classes were defined. The first one is:

```
public class ShipSteering {
  private static float ALPHA = 0.1f;
  private static float GAMMA = 1.0f;
  private static float EPSILON = 0.01f;
  public static void main(String[] args) throws Exception {
    float[][][][][] thetaAll = new float[20][20][30][31][31];
    float[][][][] thetaXYO = new float[50][50][90][31];
    float[][][][] thetaXYT = new float[50][50][31][31];
    float[][][] thetaXY = new float[500][500][31];
    float[][][] thetaXY2 = new float[500][500][31];
    float[][][] thetaXY3 = new float[500][500][31];
     /* Implementation of the SARSA algorithm as seen in the previous section */
  }
}
```

This is the main class. It is also the class that includes the learning algorithm. As arguments the testcase (simple or hard) and the water current speed are passed. Instead of using double numbers, float numbers were used, minimizing memory requirements.

Our second class maintains information about the state of the ship:

```
public class State {
  private double x = 0.5;
  private double y = 0.0;
  private double orientation = 0.0;
  private double turnRate = 0.0;
  private boolean first = false;
  private boolean second = false;
  private boolean third = false;
  private boolean fourth = false;
  public boolean isTerminal() {...}
  public int fail() {...}
  public State getNextState(int action) {
     double oldx = getX();
     double oldy = getY();
     double oldTurnRate = getTurnRate();
     double oldOrientation = getOrientation();
     double oldOrRad = Utilities.getRad(oldOrientation);
     double newx = oldx + DELTA * SPEED * Math.sin(oldOrRad);
     double newy = oldy + DELTA * SPEED * Math.cos(oldOrRad);
     double newOrientation = Utilities.fix(oldOrientation + DELTA * oldTurnRate);
     double desiredTurnRate = Utilities.getDesiredTurnRate(action);
     double newTurnRate = oldTurnRate + ((DELTA * (desiredTurnRate - oldTurnRate))/ 5.0);
     return new State(newx, newy, newOrientation, newTurnRate);
  }
...
}
```

This class except from the common get and set methods, has also methods for answering questions like, if the state results in failure and in which positions (that is which gate combination), or if the state is terminal. It also includes the capability of making state transitions.

The third class is a utilities class that mainly contains static methods for calculating indices and manipulating degrees:

```
public class Utilities {
  public static int[] getIndicesAll(State state) {...}
  public static int[] getIndicesXYO(State state) {...}
  public static int[] getIndicesXYT(State state) {...}
  public static int[] getIndicesXY(State state, double offset) {...}
  public static double getDesiredTurnRate(int action) {
    return (((double)action) - 15.0);
  }
  public static double getRad(double degrees) {
    return (degrees/360.0) * 2.0 * Math.PI;
  }
  public static double fix(double degrees) {
    if(degrees < 0) {
      degrees += 360.0;
    } else if(degrees >= 360) {
      degrees -= 360.0;
    }
      return degrees;
   }
...
}
```

Finally the fourth class is a simple GUI class for displaying the route of the ship:

```
public class GUI extends JFrame {
...
  class BoardPanel extends JPanel {
    public void paint(Graphics screen) {
      screen.setColor(Color.WHITE);
      screen.fillRect(0, 0, size * CELL_SIZE, size * CELL_SIZE);
      screen.setColor(Color.BLACK);
      // gate 1
      screen.fillRect(400, 150, 2 * CELL_SIZE, 2 * CELL_SIZE);
      screen.fillRect(450, 150, 2 * CELL_SIZE, 2 * CELL_SIZE);
      // gate 2
      screen.fillRect(150, 300, 2 * CELL_SIZE, 2 * CELL_SIZE);
      screen.fillRect(200, 300, 2 * CELL_SIZE, 2 * CELL_SIZE);
      // gate 3
      screen.fillRect(250, 350, 2 * CELL_SIZE, 2 * CELL_SIZE);
      screen.fillRect(300, 350, 2 * CELL_SIZE, 2 * CELL_SIZE);
      // gate 4
      screen.fillRect(150, 400, 2 * CELL_SIZE, 2 * CELL_SIZE);
      screen.fillRect(200, 400, 2 * CELL_SIZE, 2 * CELL_SIZE);
      // gate 5
      screen.fillRect(100, 499, 2 * CELL_SIZE, 2 * CELL_SIZE);
      screen.fillRect(150, 499, 2 * CELL_SIZE, 2 * CELL_SIZE);
      for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
          // Draw cell contents
          if (sea[i][j]) {
             screen.setColor(Color.GRAY);
             screen.fillRect(i, j, 2*CELL_SIZE, 2*CELL_SIZE);
          }
        }
      }
    }
  }
...
}
```

This inner class is used to paint the contents of the panel. This specific class is used for the multiple gates problem. After painting the panel white and the gates black is using the sea variable to trace the coordinates the ship has visited. The sea variable is filled from the ShipSteering class when it is running, executing the learned policy. As a final comment on the implementation, JVM arguments `-Xms` and `-Xmx` where used to increase the memory allocated for the program to run.

5 Results and Discussion

For evaluating the Sarsa algorithm and the choices made in the parameters of this algorithm, plots are going to be used, displaying the learning process of the algorithm as episodes increase in number. The two basic utility metrics are success rate and time to goal. The first one signifies the rate of succesful routes over the episodes. It is going to be averaged in bins of $10,000$ episodes. The second benchmark is the time period it takes the ship to go from its initial state to the goal state. Again it is going to be averaged in bins of $10,000$ episodes. In all cases $2,600,000$ episodes were used. The greedy policy factor ϵ was set to 0 after $2,500,000$ episodes. Finally the GUI representation of the route will be used to display that none additional, time consuming and unecessary moves-actions are being made.

5.1 One Gate - No Current

For the one gate with no water current problem, the minimum amount of time learned was 350 seconds. Figures 3 and 4, clearly display a steady learning procedure. The goal of obtaining a constant success rate equal to 1.0 was accomplished, while the time to goal was continuously improving reaching also an integer constant value. From figure 5 it is obvious that no additional moves are generated except the necessary ones, even though the solution might not be the optimal one. It is clear though that the solution is an acceptable sub-optimal one.

5.2 One Gate - With Current

When water current was added to the problem's specifications as shown in figure 8 the minimum amount of time to reach the goal was increased to 361 seconds. Again the plots exhibit a steady learning procedure, except when near the point of setting the greedy policy factor to zero. Despite this setback the algorithm converges to a very good solution that is shown in figure 8. The ship learns to steer against the current and when appropriate, lets the current to drive it towards the gate.

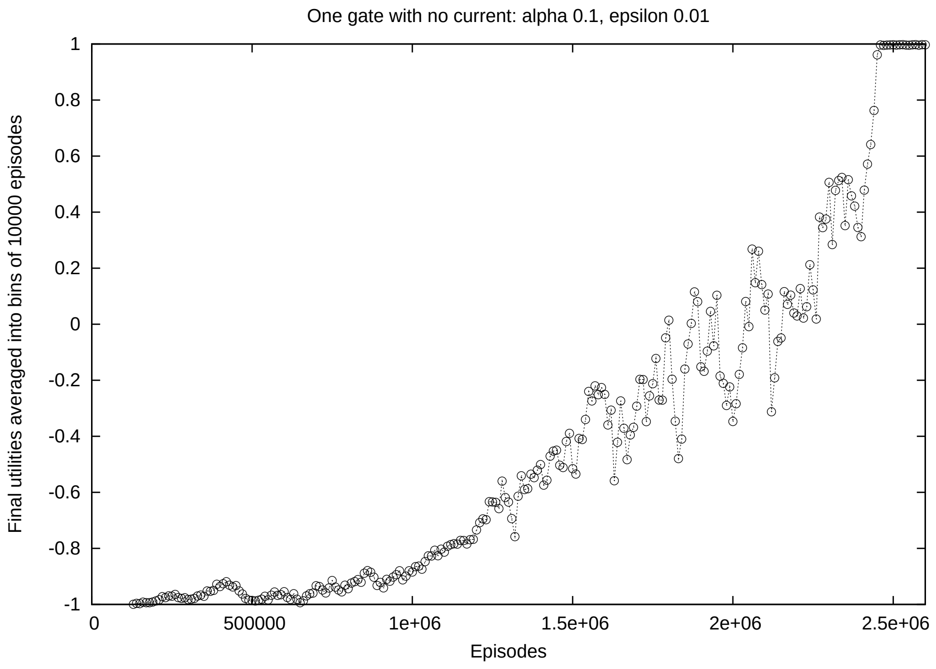

Figure 3: Success rate: single gate - no current. Converged at maximum success rate.

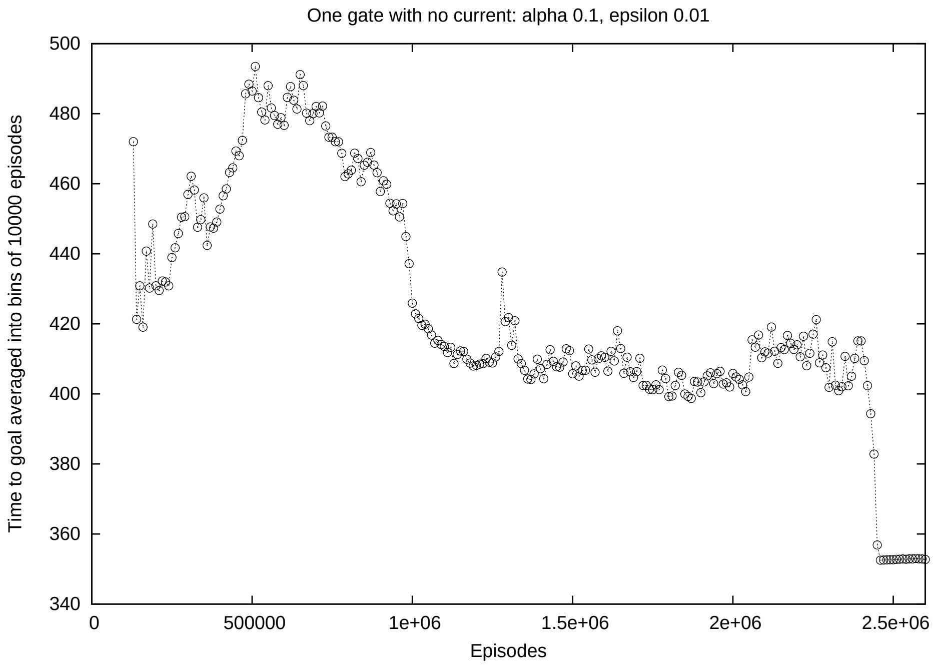

Figure 4: Time to goal: single gate - no current. Converged at time to goal equal to 350 seconds.

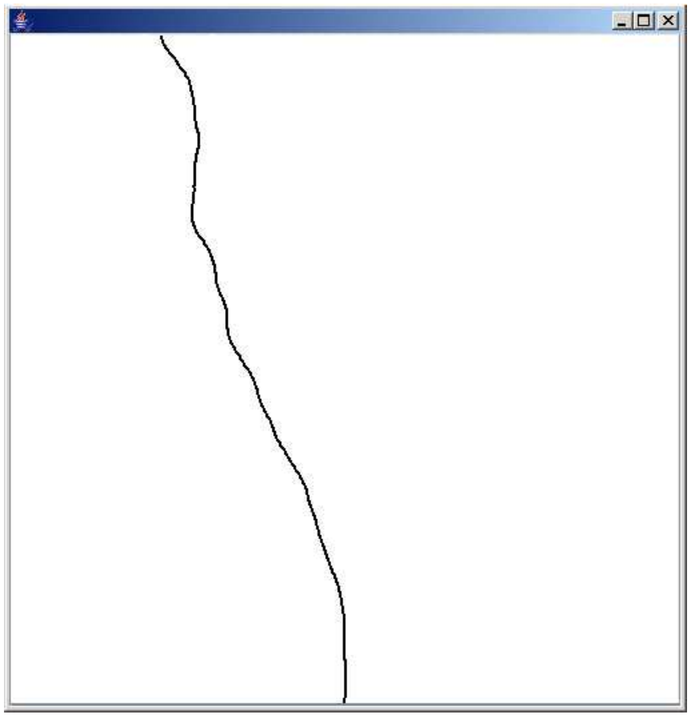

Figure 5: The ship is navigating its way through a single gate with no current.

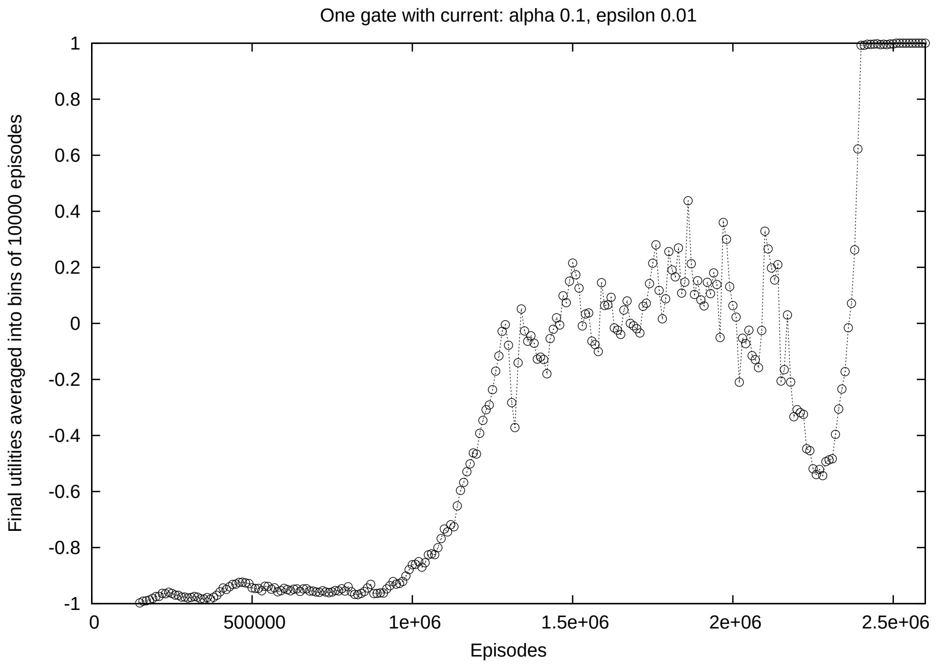

Figure 6: Success rate: single gate - with current. Converged at maximum success rate.

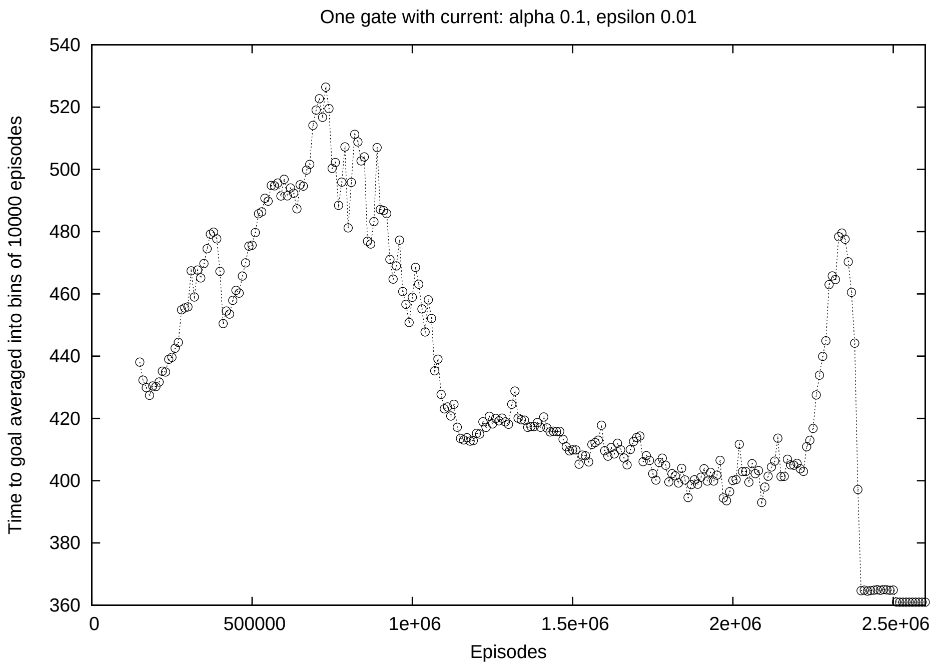

Figure 7: Time to goal for single gate - with current. Converged at time to goal equal to 361 seconds.

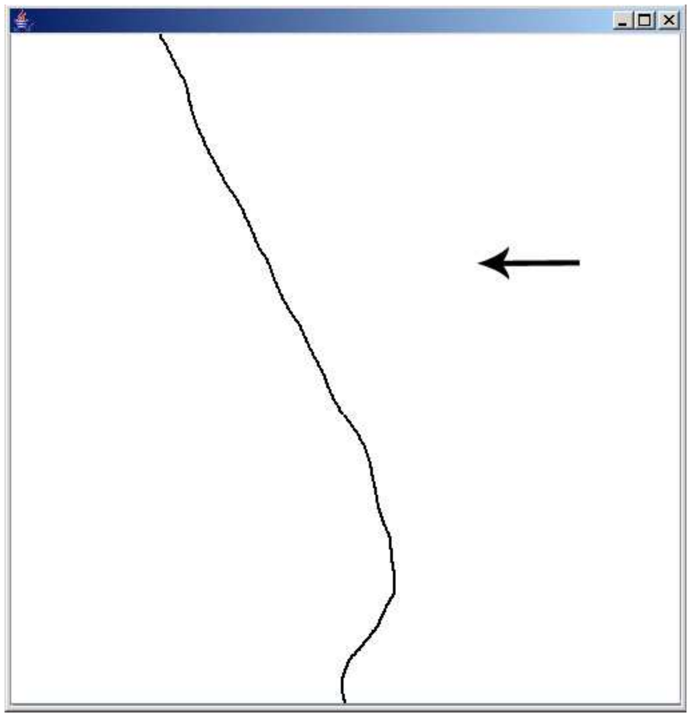

Figure 8: The ship is navigating its way through a single gate with current.

5.3 Multiple Gates - No Current

Even though one would expect for the elements of the algorithm to require again fine tuning for this complicated environment, the set of parameters used in the single gate problem, proved to be efficient for this also. The minimum time for the cruise was 617 seconds. The first gate was passed in the first $10,000$ iterations the second after $370,000$, the third after $610,000$ and finally the fourth after $1,020,000$ iterations. The more complex environment, the more simple learning curves were produced as one can see in figures 9 and 10. One reason is the fact that when we have the first success, the agent has already figured how to pass the previous gates and it is left with only a fraction of the whole problem that can be solved easily and quickly.

Another interesting observation can be found by inspecting figure 11. After passing the first gate, the ship does not rush straight into the second gate, but instead learns to take a more "open" turn that will help it pass the third gate more easily. This is also the case at the beginning of the cruise, where the ship instead of rushing into gate one, it goes wider.

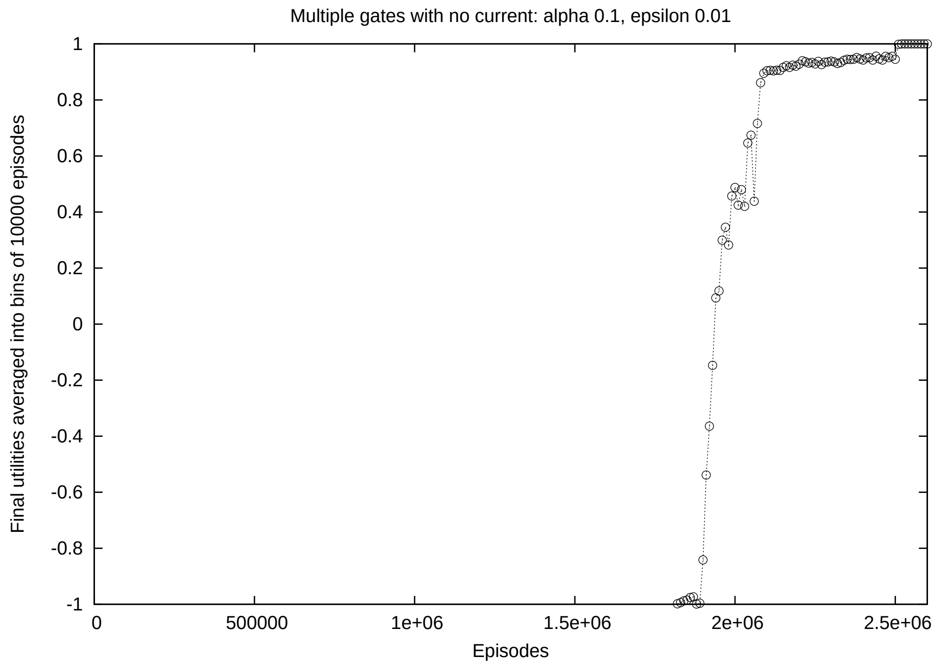

Figure 9: Success rate multiple gates - no current. Converged at maximum success rate.

5.4 Multiple Gates - With Current

Again using the same configuration, water current was added as shown in figure 14. The minimum time learned for cruising through the five gates was 702 seconds. The first gate was passed in the first $10,000$

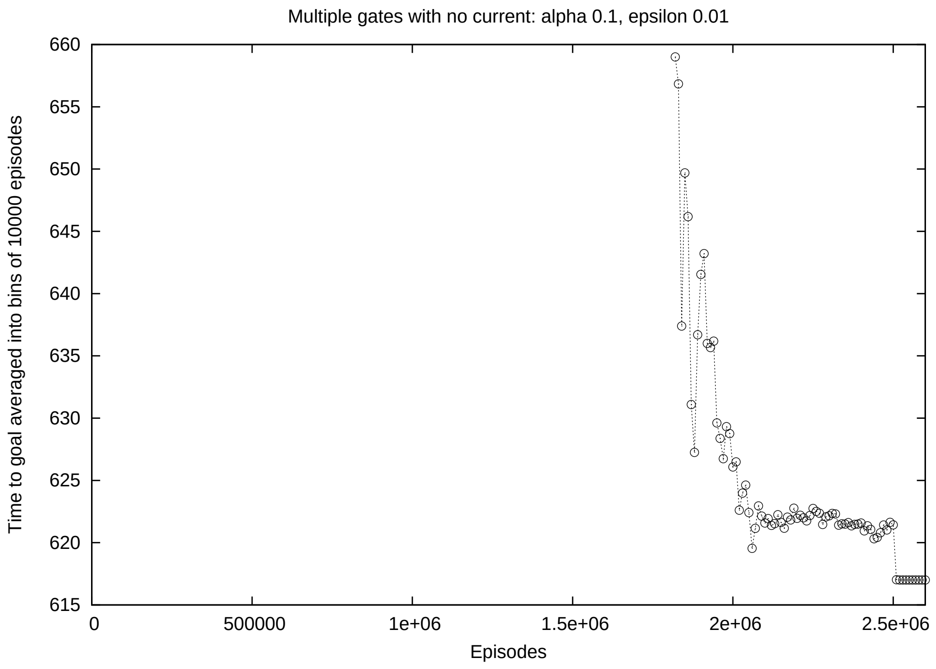

Figure 10: Time to goal for multiple gates - no current. Converged at time to goal equal to 617 seconds.

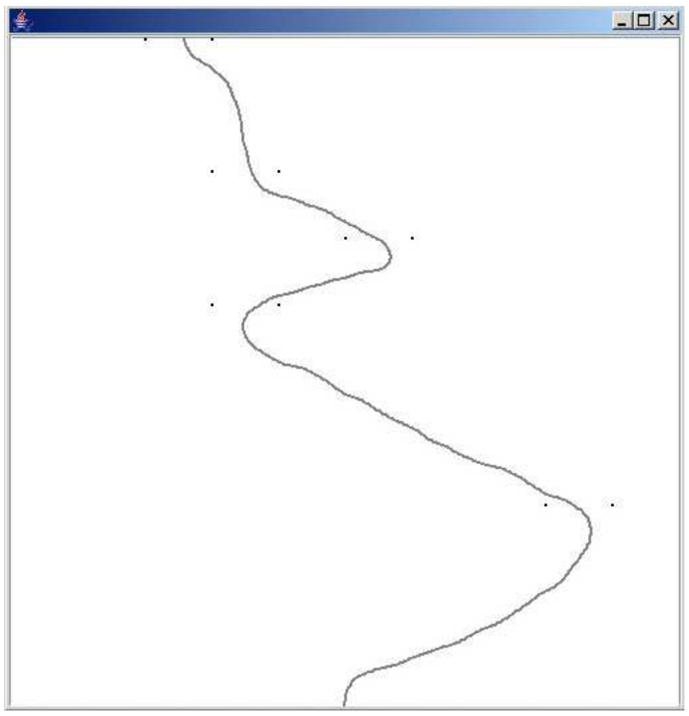

Figure 11: The ship is navigating its way through multiple gates with no current. It learns to take "wider" turns that will help its encounters with the next gates.

iterations the second after $420,000$, the third after $650,000$ and finally the fourth after $1,150,000$ iterations. Once more the learning curves, in figures 12 and 13, were quite smooth.

The effects of the water current are represented in figure 14. In the ship's route from gate one to gate two, one can observe that the current forces the ship out of its intended course (figure 11). This disturbance is the reason that learning algorithm yields a longer amount of time to goal.

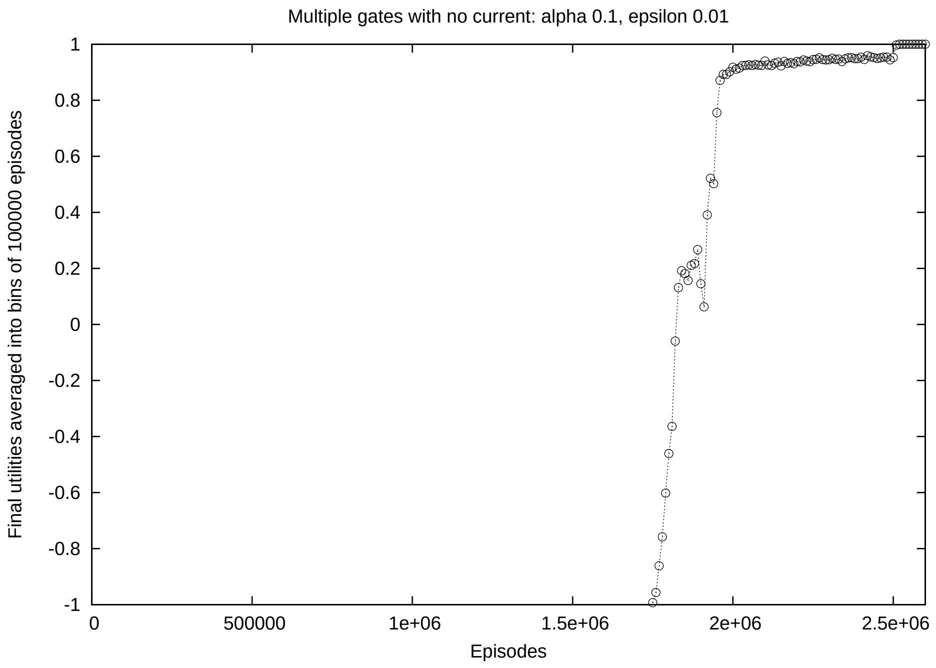

Figure 12: Success rate multiple gates - with current. Converged at maximum success rate.

6 Conclusion and Future Work

Reinforcement learning is a very robust paradigm, and as one expected it provided very good solutions to every problem encountered. In all the cases the algorithm converged slowly. This can be attributed to the fact that the continuous state space even when it becomes finite by slicing the state variables, results in a vast amount of state-action values. The use of tilings helps to alleviate this requirement, but still the amount is indeed very large.

The main problem encountered was that of finding a good tiling representation and an appropriate configuration of the parameter set. Many different tilings and parameters were used in many different combinations, in order to derive the best ones. Values going to infinity and poor time from start to goal were often the case, even though it is obvious that good solutions exist in the hypothesis space [6]. For example

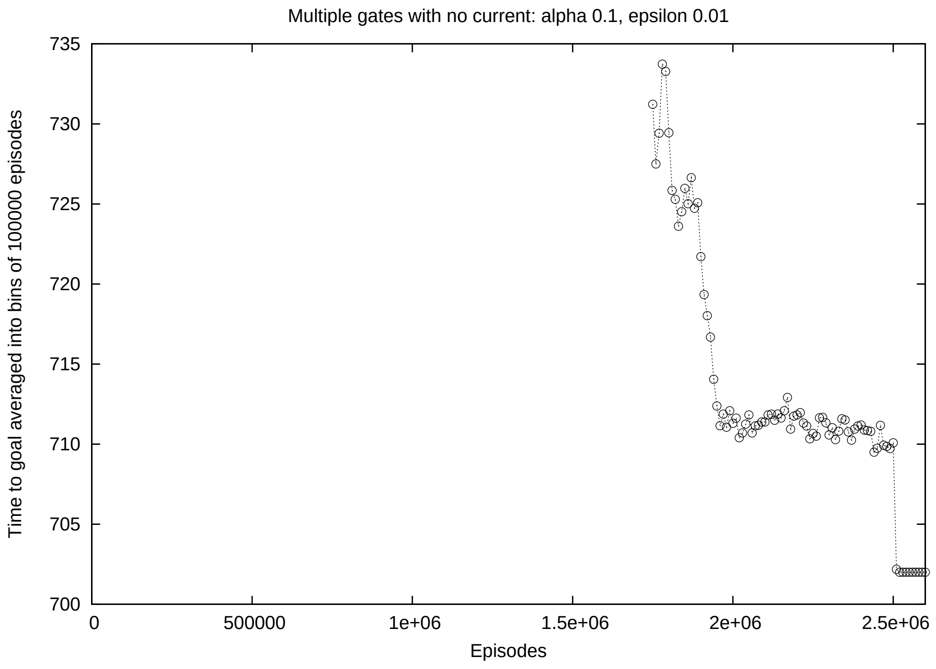

Figure 13: Time to goal for multiple gates - with current. Converged at time to goal equal to 702 seconds.

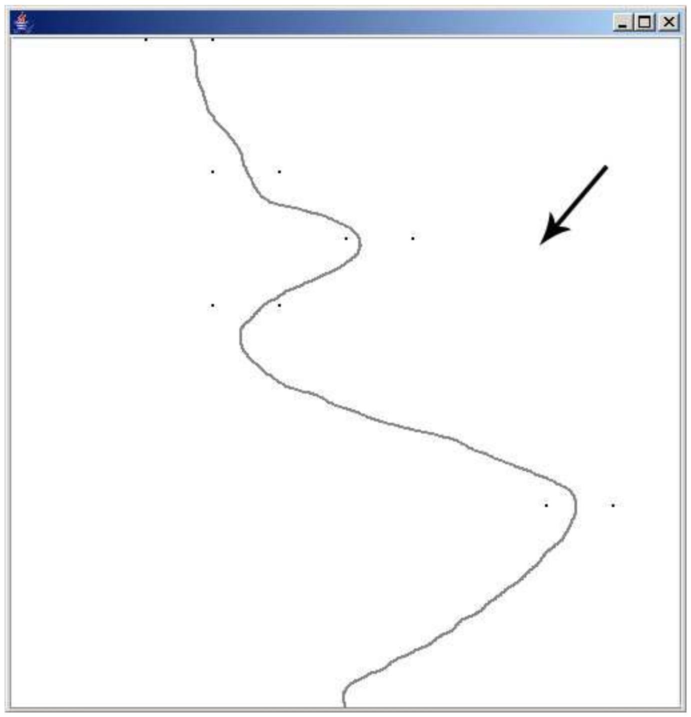

Figure 14: The ship is navigating its way through multiple gates with current. Even though the current pushes the ship off-course, it manages to cruise through all the gates.

by changing only the ϵ value in the single gate problem a poor solution was obtained as shown in figure 15.

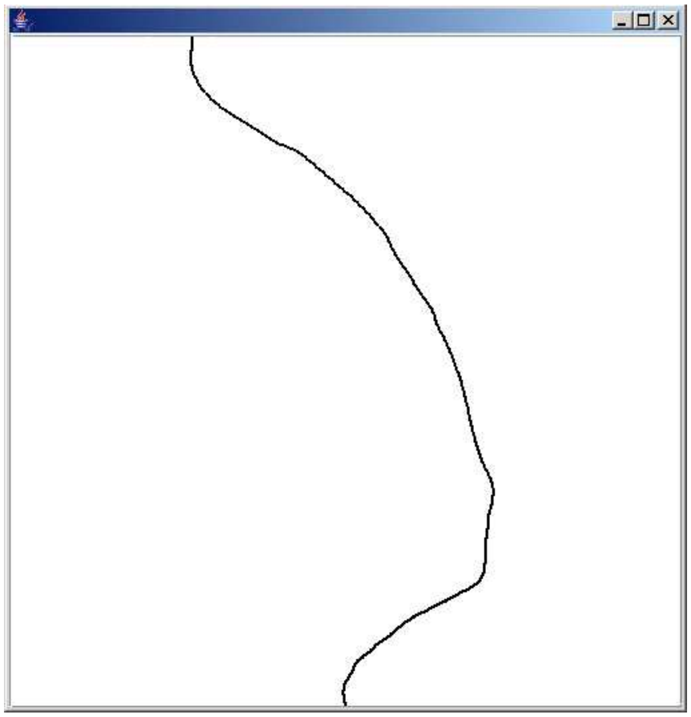

Figure 15: Finding a poor route to the single gate problem by just switching one parameter in the algorithm's configuration set.

What could be pursued further in respect to the ship steering problem, would be the introduction of neural networks as function approximators instead of tilings. Neural networks could be used for training the agent over a large set of single gate examples and then simply provide the controller a sequence of gates. Using past experience the ship could drive its way through all the gates. Another interesting extension includes deeper understading of what constitutes an appropriate model-representation of the state space, when the later is continuous. For example, heuristics exists for designing a feedback controller or a neural network model. How this could be extended for constructing appropriate tilings. Finally, techniques for speeding up the process of learning could be proved to be very useful, especially in domains with a continuous state space.

In overall the whole project was a great introduction to the area of reinforcement learning in general and in specific regarding the control of systems. The ship steering problem provides the means to extend the same way of thinking and implementation to numerous different control problems that can make use of the reinforcement learning methodology. Solving problems by just having a simple model of the plant and have the agent learn its way through is really challenging and interesting. It is also amazing to see intelligence emerge, by using a simple example of life such as giving rewards and penalties to either good or bad actions.

References

[1] Bryson, A .E., and Y. Ho, *Applied Optimal Control : optimization, estimation, and control*, Hemisphere Publishing Corporation, New York, 1975.

[2] M.T. Rosenstein, M. T., and A.G. Barto, `Supervised Learning Combined with an Actor-Critic Architecture,` Technical Report 02-41, Department of Computer Science, University of Massachusetts, Amherst, 2002.

[3] Anzai, Y., *Cognitive Control of Real-Time Event-Driven Systems*, Journal of Cognitive Science, 8, 221-254, 1984.

[4] Anderson, C. W., and T. Miller, *A Challenging Set of Control Problems*, Neural Networks for Control, 475-508, MIT Press, 1990.

[5] Sutton, R. Y., and A. G. Barto, *Reinforcement Learning : an introduction* , MIT Press, Cambridge, Mass., 1998.

[6] Russel, S., and P. Norvig, *Artificial Intelligence: A Modern Approach, 2nd Edition*, Prentice Hall, 2003.