Artificial Intelligence Project 4: Design Document

Group 11: Lisa Peters, Janette Rounds & Monica Thornton

1. Description of the Problem

Reinforcement learning is a branch of machine learning in which an agent uses observed rewards to learn an optimal (or close to optimal) policy for the given environment (Russell & Norvig, 2003). A hallmark of reinforcement learning is that this learning is accomplished without giving the agent prior knowledge of the environment or the reward function and instead the agent relies on feedback (reinforcement) about their actions. This feedback allows the agent to learn how to conduct itself in the given environment by learning the actions that maximize its cumulative rewards. Reinforcement learning has a long history of use in game playing and robot control applications, and the software developed for this project pays homage to those roots. This project focuses on the design and implementation of reinforcement learning algorithms that will be applied to the racetrack problem. The racetrack problem is a standard control problem where the goal is to control the movement of a racecar attempting a time trial on a prespecified racetrack.

In this work, we will be testing agents that employ two different algorithms, Value Iteration and Q-learning, on three racetracks of varying difficulty. For the agents to perform well on these racetracks, they will need to determine an appropriate policy for each of the racetracks. The algorithms implemented for this project take different approaches to determine an optimal policy, and to test the merits of each approach we will keep track of metrics related to the training and performance of the racecar associated with each algorithm. The architecture we will employ to build our software, along with a UML diagram depicting the relationships between the major classes is provided in Section 2 of this work. Details regarding the implementation of the Value Iteration and Q-learning algorithms can be found in Section 3 of this document, and our plan to evaluate the performance of these algorithms will be outlined in Section 4.

2. Software Architecture

A model of our software is provided in Figure 1. We want to have a Track and an agent that simulates actions on that Track. A potential pitfall of this approach is that changes can be made to the Track unintentionally. We plan to circumvent this by using a Mediator software design pattern (Freeman, Robson, Bates, & Sierra, 2004). The Mediator pattern defines a way for objects to interact. In our case, the objects are the learners or the `Driver` class, and the `Track` class. We have defined an immutable `Track` class, and we have also defined a `Car` class to perform actions on the track. The `Driver` only accesses the `Track` through the `Car`. The `Car` class stores position and velocity information and performs collision detection with the walls of the track. Finally, our learners (`ValueIteration` and `QLearning`) inherit from the `Driver` class.

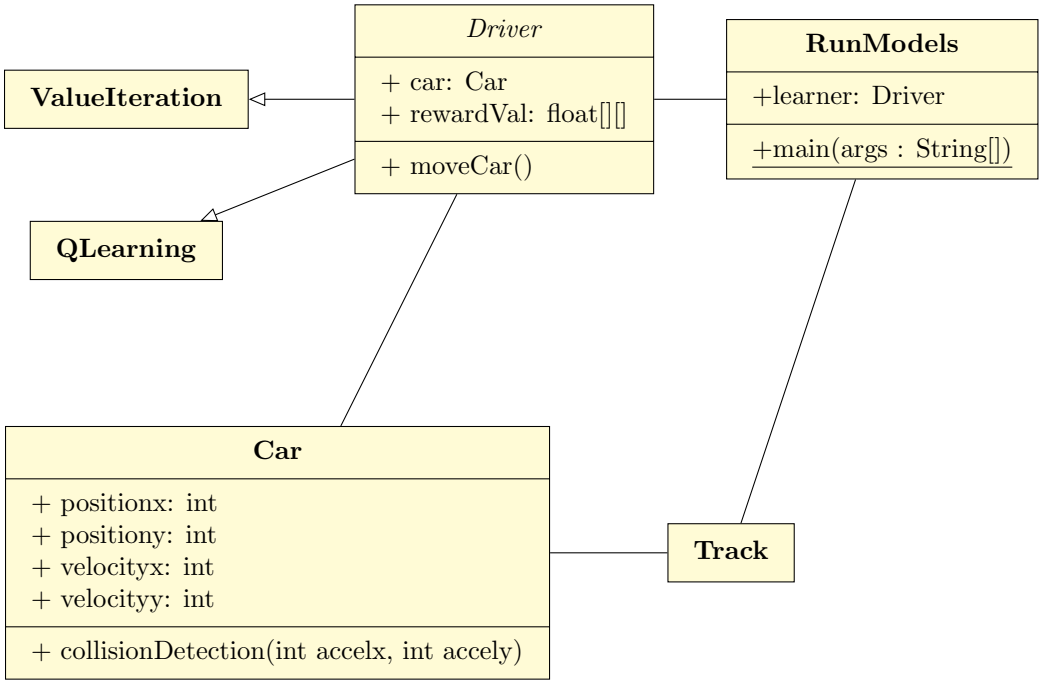

Figure 1: UML diagram that outlines the architecture of the described software.

3. Design Decisions

In the following subsections, we outline a number of the key design decisions we have made when implementing our software while the decisions directly related to our experimental design are presented in Section 4 of this document. Additionally, although not depicted in Figure 1, our software will also include classes for running experiments (including tuning experiments) and printing results. Time permitting we will also include unit tests for all of the major functions, to ensure that the software continues to perform as expected.

3.1 Simulation

Each sample consists of picking a random starting position and random velocity that moves in the general direction of the finish line. At that state, the driver will find a policy that maximizes the reward, based on a single action (which we call a move). Each iteration consists of a sample and a trial, where we run a car through the racetrack and evaluate the number of moves before the car finishes using the policies learned so far. Since the policy may involve the car repeatedly running into a wall rather than making moves that direct the car towards the finish, we will set a maximum number of crashes that a car can make. After examining the tracks, we determined that a car can run into a wall 100 times per trial, before we declare that the driver of the car is drunk. After the driver is declared drunk, the trial is over. After each iteration, we report the number of moves and crashes the car made for the trial. As we increase the number of samples, the car should crash less, and the driver will drink less and make better decisions so the number of moves should decrease.

3.2 Value Iteration

The first algorithm we will implement is called Value Iteration, and as its name implies, it iterates through states and actions in order to find the utility, and thus the optimal policy, for each state. The original implementation was done by (Sondik & Smallwood, 1973) but we will implement it as shown in (Russell & Norvig, 2003). This algorithm uses the Bellman Equation, shown below, to calculate utilities.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

In the above equation $U(s)$ and $R(s)$ is the utility and rewards of state s respectively, γ is the discount factor to prioritize short-term and long-term rewards, and $P(s'|s,a)$ is the transition function detailing the probability of reaching state s' from state s having taken action a. In our implementation, we will tune γ for optimal results.

Essentially, the Bellman Equation shows the relationship between the utility of a state and its neighbors, namely that a state's utility is defined as its immediate reward plus the utility of the next state as long as that the agent takes the optimal action to maximize utility payouts and factoring in the probability of reaching that state and the discount factor. Importantly, for every state, there exists a unique Bellman Equation (Bellman, 1957). Value Iteration takes advantage of this fact in developing an optimal policy, by iteratively updating the Bellman Equation for each state, given the Bellman update equation:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(a).$$

This update calculates the policy $\pi(s)$ under the hood, using the utility function of the previous state to calculate the utility function of the current state. All initial utilities are set to 0 and the algorithm iterates until the difference of $U_{i+1}(s)$ and the right-hand side of the equation are below a certain threshold, indicating that the solution is within an acceptable error of being absolutely correct. This specific threshold is given as:

$$\delta < \frac{\epsilon(1-\gamma)}{\gamma}.$$

3.3 Q-learning

The second algorithm we will implement is Q-learning, which presents an alternative strategy for calculating the optimal policy. Whereas Value Iteration requires a transition model and knowledge of the rewards for all states in the environment, the active temporal difference agent constructed by the Q-learning algorithm has no such requirements. This is an importance difference, given that there are many cases (i.e. state-spaces that are too large to enumerate repeatedly) in which it might not be reasonable for the agent to have access to this information in advance. The Q-learning algorithm employs reinforcement learning to learn a Q-function, which gives the expected utility for performing a certain action in a given state. Our implementation of the Q-learning agent will be based primarily on the algorithm presented in our text (Russell & Norvig, 2003).

For the Q-learning agent to perform well on these racetracks, it will need to balance exploration of the environment with the exploitation of the knowledge they have gathered. In other words, exploration is necessary to maximize the cumulative reward, but too much exploration can result in the agent attempting to maximize short-term reward at the expense of cumulative reward. The ϵ-greedy method is a well-established and often very effective approach to balancing these factors, and is what we will use in our implementation of the Q-learning algorithm (Watkins, 1989). The ϵ-greedy method determines the amount of exploration done by the agent via the generation of a small, random number m where $m \in (0, 1)$. If $m \leq \epsilon$, the agent will make an action selection greedily, otherwise the agent will select a random action. To encourage exploration of the racetrack early on, we will start with a smaller value of ϵ, and gradually increase it over time. When using the ϵ-greedy method, an appropriate value for ϵ is central to the performance of the algorithm - and this means that the ϵ value needs to be carefully tuned (Tokic & Palm, 2011).

The Q-learning update equation includes additional tunable parameters, specifically the learning rate η and the discount factor γ. The learning rate η determines the extent to which the most recently acquired information will replace older information, and in some cases convergence requires that η be annealed. Valid values for η are between 0 to 1, where a value of 0 will ensure that the agent does not learn anything, and a value of 1 would make the agent consider only the most recently information. The discount factor γ helps the agent gauge the value of future rewards. As with η, valid values for γ are between 0 and 1, with a value of 0 influencing the agent toward considering only short-term rewards, and a value of 1 pushing the agent toward focusing on long-term rewards. With respect to ϵ, γ and η, we will tune each of these parameters on a fourth track that we derived especially for this purpose. This tuning track will include both straight sections and at least one curve to mimic the L, O and R tracks we will use when testing the algorithms.

4. Experimental Design

Our plan to determine the performance of Value Iteration and Q-learning involves the collection of multiple metrics. These metrics are the number of moves a car needs to get to the finish line, and the number of crashes that happen along the way. We will construct fitness curves to demonstrate the improvement of the agents on the racetrack problem over time. Additionally, we will use statistical comparisons on our metrics for the agents at multiple pre-specified iteration intervals. We have three tracks on which the agents can perform their time trials. The first and simplest track is the L-track which contains a single curve. The O-track has four curves. Finally, the R-track also has four curves. The R-track is our most complex track. We anticipate that the number of curves can be seen as a way to evaluate the complexity of the track.

Additionally, we will compare the performance of our agents using two different crash effects on the R track. The first type of crash effect will be that once a crash happens, the velocity returns to 0 for x and y components and the position of the car is returned to the nearest position on the track that is not a wall. The second type of crash has the same effect on velocity. However, the car is returned to the start position rather than a position on the track. Finally, we will compare the performance of both types of agents on all three tracks using the first type of crash effect. We want to ensure that we are comparing number

of crashes and number of moves independent of one another between the agents. Our hope is that by tracking both of these metrics and evaluating them separately, we can discover if the agents have differing strengths and weaknesses.

As there is some non-determinism in acceleration for the agents, we will run each agent 10 times for each comparison. Time permitting, we will increase the number of times we run each agent in order to improve the certainty of our statistical comparisons.

References

Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, *6*(5), 679–684.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.

Russell, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education.

Sondik, E. J., & Smallwood, R. D. (1973). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, *21*(5), 1071–1088.

Tokic, M., & Palm, G. (2011). Value-difference based exploration: Adaptive control between epsilon-greedy and softmax.. In Bach, J., & Edelkamp, S. (Eds.), *KI 2011: Advances in Artificial Intelligence*, Vol. 7006 of *Lecture Notes in Computer Science*, pp. 335–346. Springer.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK.