CS255 Artificial Intelligence Coursework: Robocode Robot Tank Report

Danielle O'Sullivan - 1402161

Summer Term 2016

1 Abstract

This report will outline the choices and concepts used in building a Robocode robot tank, which will aim to showcase key Artificial Intelligence notions through its behaviour and performance. The task was distributed by the University of Warwick Department of Computer Science and the robot referred to throughout this document, `NishiMiki`, was built by Danielle O'Sullivan (1402161, Data Science, Statistics).

2 Introduction

This document will discuss my bot `NishiMiki`'s "Strategy Design & Implementation" which will describe how it should perform, along with the theory behind why it works. There will also be a detailed "Justification" section, to supplement the Strategy Design & Implementation section, which will consider why each choice was made (or was not made in certain instances) and how it has been implemented into the bot's behaviour.

"Robocode is a programming game where the goal is to code a robot battle tank to compete against other robots in a battle arena."[1] The Robocode environment consists of a space where robots exist within in order to compete, called the "battlefield", which is of fixed size which can be changed only before a new battle takes place; and the robots themselves are modelled as 36×36 pixel squares and each bot will act accordingly to how its owner has designed its behaviour. The owner/player has no direct control of their robot once a battle begins and so each player must program their specific bot with strategies of how

to react and behave in certain situations within the battlefield before it enters the fray - it is hence equipped with its very own Artificial Intelligence in order to appear to exhibit 'intelligent' or 'rational' decision making (where we define "rational action" to be the choice of action that maximises the expected value of the performance measure given the percept sequence to date - this concept gives us a multitude of design strategies which we will discuss in the following section).

3 Strategy Design & Implementation

We now move onto the exploration of the basis for the design of `NishiMiki`. The overall concept from which I decided how I chose to program my bot was derived from a simple Rule-Based System.

A Rule-Based System has three main components: a *knowledge base*, an *inference engine* and a *working memory*. In the most basic of systems, a knowledge base consists of permanent data which encodes the storage of information - in this particular context, the knowledge base consists of rules (hence a Rule-Based System) where the rules are in the form shown below.

```
if (e.getDistance() < 100 && getEnergy() >= 60) {
        turnGunRight(rotateGun);
        fire(3.0);
}
```

We can clearly see that the above code checks if a certain scenario meets specific requirements and if it evaluates `TRUE`, then it will execute the body of the code. This is a standard `if else` statement with the usual Boolean logic which just so happens to exhibit the properties of a rule. The next component of a Rule-Based System is the inference engine, which tells the bot how to react, i.e. it applies the rules to the working memory (a temporary memory store). This is demonstrated with the executable body of the `if else` statement.

Let's look at `NishiMiki` in further detail. The complete program is broken down into subsystems of methods which interact with each other when different scenarios occur within the Robocode environment. The most used, or more precisely, the most important method within `NishiMiki` is the `onScannedRobot()` method which tells the bot precisely how it should behave when it spots another robot. We shall discuss the *movement* and *targeting* behaviours for the aforementioned `onScannedRobot()` method.

3.1 Movement

Firstly, `NishiMiki` keeps track of the enemy's energy which is important for a few reasons - the most obvious reason is due to the fact that a robot is unable to "see" bullets so it must guess to the best of its ability. The code below shows the calculation that stores and updates the energy of an enemy bot.

```
double energyChange = initialEnergy - e.getEnergy();
initialEnergy = e.getEnergy();
```

By way of elimination, when a small energy drop (between 0.1 and 3.0) is detected, this can, in most cases, be identified as the event that the other bot has fired a bullet. And thus the second reason allows `NishiMiki` to execute a number of actions based off this result, mainly with regards to its attempt to dodge a predicted forthcoming bullet. It was observed that many professional bots, that compete in the Roborumble[1] (the vast majority, if not all professional bots, extend `AdvancedRobot` - unlike the `Robot` class, the *body*, *gun* and *radar* movements are independent in this class.), successfully implement a "Wave Surfing"[2] technique in order to dodge bullets - due to the nature of the `Robot` class, this strategy for movement was technically extremely difficult, if not near impossible to replicate to a high standard in the `Robot` class and so the decision to restrict `NishiMiki` to relatively simple movements was a necessary one. The movements that `NishiMiki` demonstrates are:

- Halving the distance between itself and an enemy bot;
- Oscillating from side to side.

`NishiMiki` would only halve the distance between itself and an enemy bot when it could afford to risk losing energy by being hit upon approaching (the constraints set out in this rule is if `NishiMiki`'s energy is above 60.0 points and when the distance between bots is further than 100 pixels away). This specification of distance, and then the halving of it, is in order to give `NishiMiki` a higher success rate of hitting the opponent (the logic behind this was that the closer you are to an object then the faster an item would reach another point whilst also reducing the time for the other bot to escape). The second type of movement, oscillation from side to side, is executed out by the movement of `NishiMiki` to become perpendicular to the enemy bot and then rapid successions of `ahead(...)` and `back(...)` motions to dodge any bullets that may be heading its way.

[1] `http://robowiki.net/wiki/RoboRumble`
[2] `http://robowiki.net/wiki/WaveSurfing`

3.2 Targeting

In the default `run()` method, `NishiMiki` rotates its gun through 360 degrees in one direction and then the other - this means that it is automatically looking for bots as even if it were to be sat still, it would never just become a `SittingDuck`[3]. The point of this movement is solely to trigger the `onScannedRobot()` method.

Once `onScannedRobot()` is called, the way `NishiMiki` tries to target (keep track of scanned robots) is to try and keep the gun (and hence radar) pointing at the opponent as much as possible. This is an obvious choice as *in theory* , if your gun is always pointed at the enemy, you should never miss the target (of course, this does not account for the movement of the other bots as your bullet is travelling towards them - we know that bullets take a strictly greater than zero value of time to arrive at its destination). In practice, it is unlikely you will find many high-level bots, that implement artificial intelligence, that just sit still; so a mechanism that copes with bots that move as well as those that sit still is ideal.

An `alignGun()` method is included so that in moving the body of the bot, the gun also moves with it (i.e. the Heading and GunHeading would be the same). Turning the bot around is often slow however so an alternative way of targeting another bot would be to move only the gun itself to point at the actual location of the enemy robot. The way the true location of the enemy bot is computed is shown as follows. Please note the name of the method `easyAngle()` has been modified for formatting purposes only. This method is put in place to ensure `NishiMiki` does not over-rotate itself when trying to find another bot - a common problem that was encountered in the early stages of development.

```
rotateGun = easyAngle((getHeading() + e.getBearing()) - getGunHeading());
```

We shall now consider the reasoning behind the choices made in the Strategy Design & Implementation of `NishiMiki`.

4 Justification

When deciding how to implement `NishiMiki`, a number of factors had to be considered and weighed up before implementation could actually begin.

- What are the advantages of this particular strategy?

[3] A sample bot that does not attack or move.

- What are the disadvantages?
- How could you potentially implement this strategy?

A host of different concepts were available to base a potential robot from; among the strategies that were explored *Simulated Annealing*, *Genetic Algorithms* and *Rule-Based Systems[2]* were reviewed as the better concepts in terms of outputting a seemingly "intelligent" design.

4.1 Simulated Annealing

Let us first talk about *hill-climbing*. This is a type of *Iterative Improvement Algorithm*, where it displays greedy search qualities which will try to only ever improve upon its previous state. This form of search is flawed in that if it reaches a local maximum point in a 3-dimensional plane it will try to improve (i.e. move higher than where it currently is), but by the definition of a local maximum (if X is a neighbourhood of x_0 and x_0 is a local maximum, $\quad x \leq x_0, \quad \forall x \in X$) we can see that there does not exist a point that is of a higher value, hence hill-climbing simply cannot know if it has reached a global maximum.

Simulated annealing is a type of *hill-climbing* algorithm but it has a random element which may force the search to get worse before it gets better; this property is advantageous as it enables a global maximum point to eventually be found rather than the algorithm get stuck at a local maximum. There are certain drawbacks in this kind of strategy however, when thinking of how to implement this strategy into `NishiMiki`, it would be relatively easy to create an array or linked list which would store values for the metric chosen. The only disadvantage that was prevalent was scalability, so the maintenance of an array if there were over 1000 rounds each with an average of several thousand ticks would be a massive issue in terms of local storage (this would not likely be a problem for a bot that would be allowed to read text files into its program). In future the idea of stochastic processes, specifically Markov Chains, may be of interest - where the next point/node chosen to visit next only depends on the current state (this may help the storage problem).

4.2 Genetic Algorithms

This concept takes two *parent* states and produces a successor based from those parents. The way it constructs a successor from the parents is as follows:

1. A group of many potential parent states are evaluated by a *fitness function*. Those that fall below a threshold may be culled from reproducing. The

fitness function also decides on the probability that a parent state will be chosen to reproduce.

2. Pairs of parents are chosen and a crossover point is chosen a random.
3. Two children are produced. The first of which inherits elements before the crossover from the first parent and the elements after the crossover from the second parent, and vice versa for the second child.
4. There is some small probability that one element in each child will *mutate* and become an entirely new element not inherited from either parent.

The advantages of using this strategy would be that the offspring produced has the potential to show a huge diversity and hence allows a wide search to be completed very quickly. One property is that Genetic Algorithms take large steps early on and smaller ones as children converge to an optimum value - this may not be a global optimum as already discussed in *Simulated Annealing* and can be quite heavily focussed on the initial choice of parent. If the fitness function is not sufficient then it may be the case that the Genetic Algorithm reaches a local optimum/maximum too soon and is unable to improve based on the population it has created. If this strategy were to be implemented into `NishiMiki`, a possibility would be to try to record every possible combination of actions, starting positions and opponents within a fixed battle (so the set of opponents would not change) and to output these combinations as `String` results and to evolve the robot's behaviour according to each of its performances under certain constraints. The algorithm would then try to find an optimal way to conduct itself in less-than-optimal conditions (e.g. if the bot were to start facing away from a single opponent and chose to fire, this would be noted down as ineffective and an offspring would hopefully have the bot turn around or move before firing and eventually converge to a state where it behaves the most effectively).

As it has probably been made apparent, the time and space that this approach would take up is even larger than that of *Simulated Annealing* and thus would be highly inefficient to test, let alone run in a real-time battle!

4.3 Rule-Based Systems

As there has already been lengthy discussion on what a Rule-Based System is, the advantages and disadvantages will be the main topic of analysis in this section.

The main advantage of using a Rule-Based System as opposed to either of the above strategies is the minimal space taken up by recording past data.

In `NishiMiki`, the only permanent data it holds is its rules; which, on the one hand, it very space-efficient, but from the converse perspective, it does not learn from past mistakes or experiences. The only true way it can "learn" is through the author writing up more and more ways to react in different scenarios so that it will eventually have an extensive enough knowledge base that it imitates learning as battles go on (note how similar this is to the Genetic Algorithm approach but much more prone to human error as well as the fact this method will not cover all scenarios within a better time frame - it is a simple fact that computer processing far outranks the average human). This system of writing more code would also not work within a single battle because the author cannot make edits to the code between rounds.

The main reason why this was the strategy of choice was in spite of its 'inability' to learn on its own. It was the simplicity of the maintenance and ease of understanding the data logged by the program. It is far easier to understand the output in this way than it is in either of the above strategies; and that is such a crucial factor that it managed to outweigh the benefits of the more autonomous Simulated Annealing and Genetic Algorithms approaches.

5 Testing & Evaluation

In order to test `NishiMiki` to its full extent a series of battles were carried out with different attributes and behaviour was observed and changes were made to improve the performance. Most of these tests were carried out against the `sample` bots provided by the Robocode software as they exhibit a range of behaviours.

5.1 Unit & Component Testing

All units of the code were identified as anything within a method such as a calculation and they were subject to analysis of relevance and functionality (i.e. Did it compute the expected result? Is this result used appropriately?). Unit Testing was relatively straightforward to carry out as a result.

5.1.1 Movement & Targeting

The movement of `NishiMiki` was tested against the `Crazy`, `Fire` and `Target` `sample` bots as they all exhibit a varying element of random movement and so the results that ensued reflected `NishiMiki`'s attempt to dodge bullets. Further-

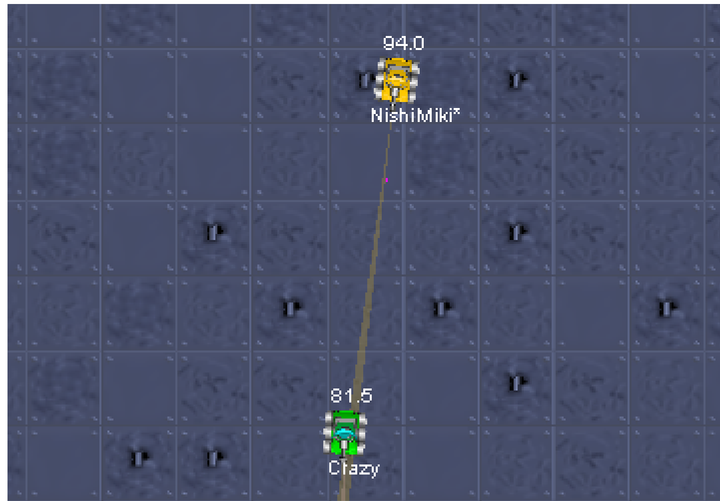

Figure 1: `NishiMiki` spots `Crazy` and fires directly at it.

more it was clear, especially in a one-on-one battle with `Target` that `NishiMiki` was able to halve the distance between itself and its enemy, as expected.

The targeting of `NishiMiki` was tested by using static bots at first to show that it could aim and fire (and hit) the enemy bots with little interference. This was assumed to be sufficient to prove that the component on an isolated level was working correctly. The tests that followed were aimed to improve on the targeting rules that were already in place. `NishiMiki` was tested one-on-one against all the `sample` bots and the observation that was expected is shown in *Figure 1*.

5.2 Integration & System Testing

The approach to how `NishiMiki` was tested at handling moving at similar times to targeting was originally through testing against multiple bots at once (i.e. Melee battles) which meant the bot had to effectively choose what to do first, which bot to hone in on or which bot to dodge out of the way from. This stage of testing was more difficult to observe as there was a much larger number of influences on `NishiMiki`'s surroundings. Testing was then reverted back to one-on-one battles with the key observations being:

- Successful "decision making" - `NishiMiki` was able to execute an appropriate behaviour in the correct situation.
- More effective against bots that sit completely still and those that move in

much more random directions than bots that have set behavioural paths.

  - To improve in the future, more elements of randomness would be incorporated as well as an improved movement "predictor" (a simple implementation of simulated annealing could work quite well here).

6 Conclusion

The findings that have been explored and analysed in this report provide sufficient evidence to suggest that the aims of the task have been achieved. Despite the fact that `NishiMiki` performs consistently just below average at this point in time, the argument for the implementation of a Rule-Based System lets us draw the conclusion that expanding the knowledge base would significantly improve its performance and that the task in hand would not be difficult to put into effect nor would it be hard to maintain upon the inevitable evolution of `NishiMiki`.

References

[1] Nathan Griffiths. *CS255 Artificial Intelligence Coursework: Robocode robot tank*. http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs255/coursework/cs255coursework.pdf, 2016.

[2] Nathan Griffiths. *CS255 Artificial Intelligence Lecture 4*. http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs255/slides/section4-4up.pdf, 2016.