Abstract

Marvin is a Robocode controller that intelligently combines online and offline search. The online search considers all possible movements, then weights them by a heuristic function. Heuristic weights and other parameters are trained using offline hill climbing in one of several scenarios. The end result is the best of all worlds: adaptability to varied opponents and situations, yet intelligent incorporation of domain knowledge through a physics model, high-level movement heuristics, and time-tested radar and gun strategies.

1 Architecture

The Robocode controller Marvin’s namesake is a paranoid android with “a brain the size of a planet,” as described in D. Adam’s *Hitchhiker’s Guide to the Galaxy*. Our robot, while lacking a planet-sized brain, attempts to incorporate as many intelligent heuristics and strategies as possible while optimizing them for specific opponents.

In Marvin’s overall architecture, the radar, gun, and robot movement operate independently of each other, interacting only with a model of the world and various Robocode events. Each subsystem is responsible for controlling its corresponding aspect of robot behavior. While this design prohibits, for example, explicit collusion between the movement and gun systems, this seemed like an acceptable limitation given the time constraints of this project. We therefore put most of our time into developing a novel movement planning system and adopted mostly standard, hand-coded strategies for the radar and gun.

1.1 Movement Control

**Developmental Goals:** dodge hostile bullets, avoid accidental collisions, ram weak enemies.

Most of our efforts and ideas went into developing Marvin’s movement strategy. Since we wanted to incorporate search into our robot, we formulated movement control as a search problem and solved it with depth-bounded depth-first-search, using a heuristic at each leaf node. We restricted the robot movements to a small finite set...
of allowed turns and a small finite set of allowed travel distances to make the search problem tractable\(^1\).

A movement plan can now be regarded as a sequence of combinations of these values, i.e. a finite sequence of tuples \((\alpha_i, d_i)\) and we can enumerate all possible movement plans of a maximum depth by a tree.

Figure B depicts Marvin’s movement search tree. Each state is a partial plan, and each operator adds a step to the plan \((d_i, \alpha_i)\), consisting of some distance \(d_i\) to move and some angle \(\alpha_i\) to turn. In place of explicit goal states, we build the complete tree up to a certain depth and evaluate all leaf plans using a heuristic function. Finally, we choose a particular plan with probability proportional to its heuristic value.

Unfortunately, trees of depths greater than one or two are not computable in a Robocode time-slice. If we were to use heuristic search rather than exhaustive search, greater depths might be possible. Of course, since we don’t know our enemy’s future actions, a tree of large depth is of limited use. Our experiments showed that shallow trees are sufficient for reasonable behavior.

1.1.1 Heuristic Components and Combination

The heuristic function that evaluates movement plans is composed of several independent measures of plan quality. By weighting these features, it can determine an overall score score for each candidate movement plan.

1. **Alternate directions:** To confuse the targeting system of the enemy, we should change the direction frequently.

2. **Alternate velocity:** To confuse the targeting system of the enemy, we should change the velocity frequently.

3. **Value of heading/speed relative to position:** Avoid situations where we are heading towards a wall.

4. **Maximize angular velocity:** The enemy has to move its gun quickly if it wants to focus on us.

5. **Preserve defined distance to enemy:** To avoid being rammed by the enemy and to be able to shoot the enemy more precisely, we try to keep a predefined distance. This optimal \(\text{desiredDistance}\) is another trainable parameter.

6. **Maintain a good position on the battlefield:** We work to maintain a rectangle of constant area between Marvin and the nearest corner.

The overall heuristic value is simply a weighted sum of the individual heuristic components.

\(^1\)Certainly, these parameters are adjustable and can be trained, though our hill climbing code is not configured to automatically tune them.
1.2 Gun Control

**Developmental Goal:** Shoot the enemy frequently, but keep misses low.

Right from the beginning of the project, we decided that our main focus was the design and development of a sophisticated robot movement control. However, the Robocode environment demands also a strategy for operating the gun. Prior experiments of learning an optimal gun control from scratch using AI techniques, e.g. Jacob Eisenstein, were rather dissatisfactory. There are several reasons:

- There is no direct feedback available which tells us if the enemy has been hit.
- The function that is being maximized is unlikely to be continuous in many parameters. Modifying some parameters a little typically results in either hitting the target or not, which is a big jump in the function value.
- A successful gun control needs to have a relatively large amount of complexity, and usually requires trigonometric functions. Learning such a function is often not feasible in a reasonable amount of time, because Robocode battles can take several seconds.

Consequently, we agreed on using a preimplemented standard gun control which is available from IBM’s “Secrets of the Robocode Masters” series. It includes simple predictive targeting, i.e. it points the gun not directly at the enemy, but expects the enemy to continue its current movement. Several new features, some of which can be trained using hill climbing, were added.

**Fire Power Selection** Since each shot reduces our own energy according to the fire power, it is wise to select a high fire power only if it is likely that a bullet will hit the target. Otherwise it is preferable to use lower fire power or not shoot at all. It is certainly not clear how to assess the likelihood that a bullet will hit its target, but it is clear that it depends on the distance, the velocity of the enemy and the direction into which the enemy is going. We have chosen a formula which has proved to be successful for many robots.

\[
firePower = 3 - \left( \frac{\max(distance_{enemy} - d, 0)}{d} \right)
\]

where \(d\) is a trainable parameter.

**When to shoot** Certainly, one can shoot whenever the gun is at the correct position according to the predictive targeting calculation. However, as long as the opponent keeps moving, there is always a small angle of deviation between the expected enemy position and our own gun bearing. A threshold on this size determines if we should shoot or not. Furthermore, we can only shoot if the gun has cooled down (Robocode property) and if our own energy is higher than the required amount for shooting. If the distance to the enemy is very high, it could be advantageous not to shoot.
1.3 Radar Control

Developmental Goal: Observe enemy movement as frequently as possible.

Both our gun and movement control depend on an accurate world model, which can only be maintained by receiving frequent information about our enemies. Marvin was designed to perform well in single opponent environments, where optimal radar control is a simple task: It follows from the Robocode physics model that the radar can always move faster than an enemy. Since a radar scan also gives us the velocity and heading of the opponent, the position of the enemy at the next time-step is known a priori. Our strategy is simple: at the beginning, we find the enemy by doing a 360 degree radar turn. Once the enemy has been sighted, we predict its position at the next time step and move our radar accordingly. Thus, we are guaranteed to “see” the enemy every time-step.

2 Parameter Training with Hill Climbing

Developmental Goal: Learn optimal parameters quickly for fighting a given opponent, or set of opponents.

Marvin depends not only on its architecture and component controllers, but also on the many parameters that define how those controllers work. Adjusting these parameters allows for a great degree of flexibility in Marvin’s behavior, ranging from a ramming-focused robot (set $desiredDistance$ to 0) to a sporadic dodger (set the angular velocity weight, $weight_4$, relatively high) to a corner-dweller (set a low $desiredPosition$ and a high battlefield position weight, $weight_6$). Different parameter combinations thus define different strategies and will work better against different opponents. We suspect that there may be commonalities among optimal strategies for different opponents as well. For example, setting all heuristic weights to zero results in a robot that does not move at all. Such a robot is unlikely to be optimal against any opponent.

We adopted a hill-climbing approach for parameter training because of its simplicity and because we have a very large search space: 11 continuous parameters, each discretized into only 10 regions, would still require testing $10^{11}$ (100 billion) different combinations, each one requiring several Robocode battles to effectively test.

Hill climbing is a local search technique. This makes it faster, since it doesn’t consider all possibilities, but it may get stuck in local maxima or plateaus, missing the global maximum altogether.

The general hill climbing algorithm we used is as follows. With each iteration, we modify one of the parameters to be trained (selected in a round-robin fashion) by some small, random value. If the change results in an improvement, we keep it; otherwise, we revert to the previous parameters. After a certain number of iterations with no improvement, hill climbing terminates. Which parameters to train, the size of the random change, the number of battles per test, and several other options may
be specified by the user in a properties file.

2.1 Marvin vs. Marvin

We theorized that training Marvin against a duplicate with the best known parameters would be a good challenge for several reasons. First, since the two contestants would be roughly equal, small changes in effectiveness could be seen to have greater effect. Furthermore, by training against our previous best self, we have an opponent that continues to become more difficult as our robot improves.

To train Marvin against itself, we created a duplicate class with a different package name, so that Robocode would treat the two separately. Since both the current parameters and the best known parameters are always written to disk during hill climbing, we set up Marvin to use the current parameters and the clone to use the best known parameters. With each modification, we tested to see if Marvin achieved a higher score than his clone in battle, and if so, kept the modification.

2.2 Marvin vs. Others

When training Marvin against other robots, we keep track of the best score achieved as well as the best parameters. Whenever a modification to our parameters yields a better score, we keep that modification and update the best score. Unfortunately, since scores depend on luck as well as parameters, a lucky series of battles can sometimes lead to a best score that is ridiculously high. After a set number of iterations with no improvement, we retest the best parameters to get a new score for them (which is usually much lower). If the new score remains unbeaten for some additional number of iterations, then hill climbing terminates. However, this is unlikely to happen, since some random modification can easily get lucky and beat the best parameters once we retest them. Therefore, we sometimes stop training before it has gone to completion when we believe it has likely converged, e.g. after 6 hours.

2.3 Scoring Functions

We defined four different score functions to determine if a parameter modification constituted an improvement or not. TotalScore is the overall score given by Robocode. RelativeScore is the difference between Marvin’s score and all its opponents. DamageDealtScore and DamageTakenScore represent the amount of damage we or our opponents dealt. These different functions allowed us to train Marvin with different goals in mind, or to train specific parameters based on their function. See the appendices for additional details.
3 Experimental Evaluation

3.1 Total Improvement From Training

In our first experiment, we tried modifying only the desired distance to keep from the enemy, while using distance from enemy as the only heuristic. The goal was to determine if hill climbing actually makes a difference. As shown in the adjacent figure, our total score slowly improved, then seemed to flatten out after hill climbing had converged. While this was reasonably successful, other parameters were harder to train independently and we did not get such clear results for them.

3.2 Training Against Different Opponents

In the following experiment, we set all parameters to arbitrary values and trained a single one, \( weight_6 \) i.e. the weight for the position heuristic, and \( targetDistance \) i.e. the optimal distance from the opponent. In each case, the TotalScore scoring function was applied. As for all heuristic weight parameters, the results had a large variation which made hill climbing for this parameter infeasible. However, we plotted the results for setting arbitrary parameter values.

As Figure 1 depict, the optimal values for the parameters can depend largely on the enemy. While in battles against the sample robot Fire \( weight_6 \) should be set to 0, the same heuristic is important when playing against itself. On the contrary, the graphs for the \( targetDistance \) parameter are similar across different enemies.

3.3 Importance of Search Depth

In our next experiment we tried to assess the impact of the search depth on the performance of Marvin. We played three series of battles of Marvin against itself, using search depths 1, 2 and 3. However, using search depth 3 turned out to be infeasible since the Robocode engine would prohibit Marvin from finishing the calculation in a single tick, i.e. time slice. That is understandable, since the number of paths is growing exponentially, with a branching factor of 35 (7 turns * 5 distances); at depth 3 more than 40000 paths would be considered. The performance of Marvin for depths 1 and 2 remained equal.
Figure 1: The diagrams above show Marvin’s total score depending on the position heuristic weight $w^a$, when fighting against itself (left) and against the sample robot Fire (right). The diagrams below show Marvin’s total score depending on $targetDistance$, when fighting against itself (left) and against the sample robot Fire (right).

4 Discussion

4.1 Architecture Successes

We found Marvin to be quite successful overall, both against simple robots and against other robots in the AI class. We credit much of our success to the use of an effective targetting mechanism, a complicated movement mechanism that incorporates randomness, and the inclusion of effective heuristics such as maintaining a constant distance or maximizing angular velocity to dodge.

4.2 Architectural Limitations

From watching the in-class battles with other robots and testing Marvin against some of the best robots out there, we became aware of certain limitations inherent in our robot’s hard-coded strategies. The predictive targeting algorithm, while effective, is not state-of-the-art: other designers of Robocode controllers have found a wave-based firing pattern to perform better against actual opponents. We suspect that this modification would improve the overall performance of our robot as well, hopefully making it more competitive with some of the better robots.

Another limitation is our mechanism for predicting future robot movements. While our physics model can perfectly model the consequence of any movement, it cannot determine what movements a robot is likely to make. We instead assume their dis-
tance travelled in \( t \) time-steps will equal \( v \times t \), where \( v \) was their last observed velocity. This unreasonable assumption affects the accuracy of our plan evaluation, since several of the component heuristics rely on knowing where the enemies will be at the end of our movement.

4.3 Hill Climbing Success

We found that we were successfully able to train parameters, and that this training did make a difference in terms of performance. The most interesting result was when we trained all of Marvin’s parameters against itself and ended up with a robot that was custom-designed to defeat itself. By using the speed change and heading change, two heuristics that were useless against most robots, Marvin was able to learn a jerky movement strategy that perfectly exploited the weaknesses of its own targeting mechanism.

Overall, we found that the most important heuristics were to keep a specified distance from the nearest enemy and to maximize our angular velocity.

4.4 Hill Climbing Limitations

Hill climbing turns out to be very slow and prone to accepting lucky modifications, at least when training against other robots. Once a very high score has been found by chance, only a luckier set of battles will beat it. We can decrease this effect somewhat by averaging over more battles, but that slows down hill climbing considerably. The effect of this is to favor parameters that produce scores with high variation, since those are more likely to sometimes beat the best score, even if their average score is actually lower than another set of parameters’. The best way to counteract this is probably to retest the best parameters more often. This destroys the advantage of high variance, since the high variance parameters will likely get a lower score when retested, and will thus soon be replaced by other parameters. Unfortunately, we must leave such experiments to future work.

We also found that many parameters interdepend, which means that they may not be effectively trained independently. This had the sad effect of invalidating many of our experiments, which attempted to show the pairwise interactions between two heuristic weights. We had better luck with training the distance we wish to maintain between Marvin and the nearest enemy. This is probably because it works as a standalone value, while all the weights function relative to each other.

A final weakness is that optimal parameters may be opponent-specific. We certainly found this to be the case with parameters such as \( \text{desiredDistance} \). While it was easily trained and produced clear improvements in performance, the optimal value depended on the opponent. In order to train Marvin to succeed against all opponents, one could try to play several battles against different opponents in each step of the training. Unfortunately, due to the long time it takes to execute a battle and the fact that the variation in the results is very high, this training would be extremely computationally expensive.
A Division of Labor

The initial design and ideas for the experiments were developed collaboratively. For the most implementation, Raphael focused on the Marvin’s architecture while Daniel focused on the hill climbing. Each of us also made a few modifications or extensions to the other’s code. We especially collaborated on the movement heuristics and parameters, since they represent the interface between the robot and its trainer.

B Path Evaluation Heuristics in Detail

Figure 2: Marvin’s online search for the best movement plan considers all discretized combinations of distance travelled and turning distance executed in sequence, up to a user-specified depth. At each node, we keep track of our expected future position and the number of ticks required to get there.

We extend notation from section 1.1 with $\text{epos}$ (position of enemy) and $\text{field}_\text{width}$, $\text{field}_\text{height}$ (battle field dimensions).

Given a path $((t_0, p_0, h_0, v_0), ..., (t_n, p_n, h_n, v_n))$ we defined the following the following features.

1. Alternate directions

To confuse the targeting system of the enemy, we should change the direction frequently.

$\text{heuristic}_1 = \min(|\frac{h_{i+1} - h_i}{90}|, 1)$

2. Alternate velocity
To confuse the targeting system of the enemy, we should change the velocity frequently.

\[
heuristic_2 = \min\left(\frac{|v_{i} - v_{i+1}|}{2}, 1\right)
\]

3. Value of heading/speed relative to position

Avoid situations where we are heading towards a wall.

\[
\begin{align*}
\gamma &= \text{angleBetweenHeadingAndClosestWall} \\
n &= \text{distanceToClosestWall}
\end{align*}
\]

\[
heuristic_3 = K_3 \cdot \frac{s}{v_{i+1}}
\]

where \(K_3\) normalizes the value to the interval \([0, 1]\).

4. Maximize angular velocity

The enemy has to move its gun quickly if it wants to focus on us.

\[
\begin{align*}
\gamma_1 &= \arctan(\text{pos}_{i+1}(x) - \text{epos}_{i+1}(x)), \\
\gamma_2 &= \arctan(\text{pos}_{i}(x) - \text{epos}_{i}(x)), \\
n_{i+1} &= \text{normalize}(\gamma_1 - \gamma_2)
\end{align*}
\]

\[
heuristic_4 = K_4 \cdot \frac{v}{\text{normalize}(\gamma_1 - \gamma_2)}
\]

where \(K_4\) normalizes the value to the interval \([0, 1]\).

5. Preserve defined distance to enemy

To avoid being rammed by the enemy and to be able to shoot the enemy more precisely, we try to keep a predefined distance. This optimal distance is another trainable parameter.

\[
v = \frac{|\text{desiredDistance} - \sqrt{\sum_{j \in \{x,y\}} (\text{pos}_{i+1}(j) - \text{epos}_{i+1}(j))^2}|}{\text{normalize}(\gamma_1 - \gamma_2)}
\]

\[
heuristic_5 = K_5 \cdot (1 - v)^2
\]

where \(K_5\) normalizes the value to the interval \([0, 1]\).

6. Maintain a good position on the battlefield
In corners the ability to move is limited to very few directions. Therefore, it is hard to escape and evade bullets. The center position is bad in multi-opponent battles, since we might get shot from both directions and we require the largest gun and radar turns. We work to maintain a rectangle of constant area between Marvin and the nearest corner. This optimal area is another trainable parameter.

\[
\begin{align*}
  v_1 &= pos_{i+1}(x) \cdot pos_{i+1}(y) \\
  v_2 &= (field_{width} - pos_{i+1}(x)) \cdot pos_{i+1}(y) \\
  v_3 &= (field_{width} - pos_{i+1}(x)) \cdot (field_{height} - pos_{i+1}(y)) \\
  v_4 &= pos_{i+1}(x) \cdot (field_{height} - pos_{i+1}(y)) \\
  \delta &= |desiredPosition - \frac{\min(v_1, v_2, v_3, v_4)}{\max(v_1, v_2, v_3, v_4)}| \\
  heuristic_6 &= (1 - \delta)^2
\end{align*}
\]

C Hill Climbing Scoring Functions

Since the trainable parameters are used in different calculations that have different goals, we need different scoring functions. For instance, \textit{weight}_4 represents the weight of the angular velocity heuristic whose goal is to dodge from hostile bullets. A scoring function that considers the amount of damage we did to the enemy is irrelevant. Altogether, we defined four scoring functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Trainable Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>TotalScore</td>
<td>Returns the total score as delivered by the Robocode engine.</td>
<td>\textit{weight}_5, \textit{weight}_6, desiredDistance</td>
</tr>
<tr>
<td>RelativeScore</td>
<td>Sums the scores of all enemies and returns our own score subtracted by that sum.</td>
<td>\textit{weight}_5, \textit{weight}_6, desiredDistance</td>
</tr>
<tr>
<td>DamageTakenScore</td>
<td>Sums our damage from hostile bullets and our damage from hostile ramming and returns its negative.</td>
<td>weight__i, i = 1.6</td>
</tr>
<tr>
<td>DamageDealtScore</td>
<td>Returns the sum of our bullet score and our ramming score.</td>
<td>\textit{firePowerDistance}</td>
</tr>
</tbody>
</table>