Birkbeck College

Msc Computer Science Project Report

Learning and Video Games: Implementing an Evolutionary Agent

Author:
Monty West

Supervisor:
Dr. George Magoulas

MSc Computer Science project report, Department of Computer Science and Information Systems, Birkbeck College, University of London 2015

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

Abstract

Artificial intelligence in video games has long shunned the use of machine learning in favour of a handcrafted approach. However, the recent rise in the use of video games as a benchmark for academic AI research has demonstrated interesting and successful learning approaches. This project follows this research and explores the viability of a game-playing learning AI. Considering previous approaches, an evolutionary agent was created for a platform game based on Super Mario Bros.

The project builds on top of software developed for the Mario AI Competition, which provides the game-engine and agent interface, as well as several other pertinent features. The basic agent was constructed first and a learning framework was built to improve it, utilising a genetic algorithm. The project followed an agile methodology, revisiting design by analysing learning capability.

The aim was to produce an agent that shows meaningful improvement during learning and demonstrated unforeseen behaviours. Ultimately this was achieved. The final learnt agent is able to complete most medium and some high difficulty levels and demonstrated several advanced strategies.

Contents

1 Introduction

Artificial intelligence (AI) is a core tenant of video games, traditionally utilised as adversaries or opponents to human players. Likewise, game playing has long been a staple of AI research. However, academic research has traditionally focused mostly on board and card games and advances in game AI and academic AI have largely remained distinct.

The first video game opponents were simple discrete algorithms, such as the computer paddle in *Pong*. In the late 1970s, video game AIs became more advanced, utilising search algorithms and reacting to user input. In *Pacman*, the ghost displayed distinct personalities and worked together against the human player [2]. In the mid 1990s, approaches became more 'agent' based. Finite State Machines (FSMs) emerged as a dominant game AI technique, as seen in games like *Half-Life* [3]. Later, in the 2000s, Behaviour Trees gained pre-eminence, as seen in games such as *F.E.A.R.* [4] and *Halo 2* [5]. These later advances borrowed little from contemporary development in academic AI and remained localised to the gaming industry.

In the last ten years, with an increase in processing power and developments in the complexity of games, many academic techniques have been harnessed by developers. For example, Monte Carlo Tree Search techniques developed in Go AI research have been used in *Total War: Rome II* [6]. In 2008's *Left 4 Dead*, Player Modelling was used to alter play experience for different users [7, p. 10]. Furthermore, AI and related techniques are no longer only being used as adversaries. There has been a rise in intelligent Procedural Content Generation in games in recent years, in both a game-world sense (for example *MineCraft* and *Terraria*) and also a story sense (for example *Skyrim's* Radiant Quest System) [8].

However, machine learning has yet to find a meaningful contribution to the world of video gaming, despite being a staple of academic research into AI, especially in robotics and board games. High complexity, small datasets and time constraints greatly hinder the effective implementation of learning techniques in the industry.

Conversely, commercial games have recently enjoyed more consideration in academic research. Games such as *Ms. Pac Man*, *Starcraft*, *Unreal Tournament*, *Super Mario Bros.* and open-source counterparts *TORCS* [29] and *Cellz* [10] have been at the centre of recent competitions and papers [11] [12]. These competitions tend to focus on agent-based game-playing AI, with many entrants adopting an evolutionary learning approach. This research could have applications as AI 'competitors' to human players, which is especially relevant to the racing, FPS and RTS genres.

This project will combine the notions of game-playing agents, evolutionary learning and traditional video game AI. By considering similar existing work, an evolutionary agent will be produced that learns to effectively play a 2D platforming game.

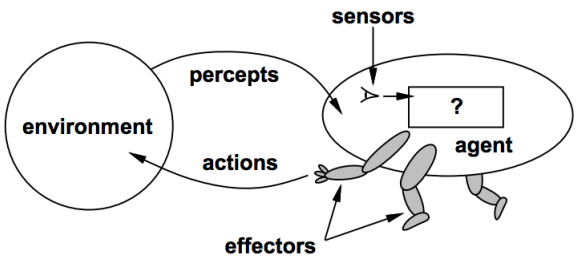


Figure 1: Illustration of an intelligent agent, taking from [16, p. 32]

1.1 Concept Definitions

At this point it is useful to introduce some high level descriptions/definitions of some key concepts that will be used in this report.

1.1.1 Intelligent Agents (IAs)

An intelligent agent is an entity that uses **sensors** to perceive its **environment** and acts based on that perception through **actuators** or **effectors**. In software, this is often realised as an autonomous program or module that takes its perception of the **environment** as input and returns **actions** as output. Figure 1 shows the basic structure of an intelligent agent. [15, p. 34]

1.1.2 Rule-based systems

A rule-based system decides **actions** from **inputs** as prescribed by a **ruleset** or **rule base**. A **semantic reasoner** is used to manage the relationship between input and the ruleset. This follows a **match-resolve-act** cycle, which first finds all rules matching an input, chooses one based on a conflict strategy and then uses the rule to act on the input, usually in the form of an output. [17, pp. 28-29]

1.1.3 Biologically Inspired Learning

Several computational learning approaches have derived their inspiration from learning in animals and humans. Two such approaches are relevant to this project: Reinforcement learning, strategy modelled on simplistic interpretation of how animals learn behaviour from their environment [13, s. 1.2]; and evolutionary computation, algorithms that apply Darwinian principles of evolution [14].

Reinforcement Learning

A reinforcement learning agent focuses on a learning problem, with its goal to maximise **reward**. Given a current **state**, the agent chooses an **action**

available to it, which is determined by a **policy**. This action maps the current **state** to a new **state**. The **transition** is then evaluated for its **reward** . This **reward** often affects the **policy** of future iterations, but **policies** may be stochastic to some level. [13, s. 1.3]

Evolutionary Computation

Evolutionary computation encompasses many learning algorithms, two of which are described in detail below. The process looks to optimise a data representation of the problem through random variation and selection in a **population**. It commonly employs techniques similar to survival, breeding and mutation found in biological evolutionary theory. [14]

Genetic Algorithms (GAs)

Genetic Algorithms are a subset of evolutionary computation. They model the solution as a **population** of **individuals**. Each **individual** has a set of **genes** (a **genome**), which can be thought of as simple pieces of analogous information (most often in the form of bit strings). Each **individual** is assessed by some **fitness function**. This assessment can be used to cull the **population**, akin to survival of the fittest, or to increase the individual's chance of influencing the next **population**. The new **population** is created by **breeding**, using a combination of the following: **crossover** of the **genome** from two (or more) **individuals** (akin to sexual reproduction), **mutation** of the **genes** of one **individual** (akin to asexual reproduction) and **re-ordering** of the **genes** of one **individual**. Each new **population** is called a **generation**. [18, p. 7]

Evolution Strategies (ESes)

Evolution Strategies are another example of evolutionary computation. They differ from standard Genetic Algorithms by using **truncation selection** before breeding. The top μ individuals of the population are chosen (usually by fitness) and bred to create λ children. ES notation has the following form: $\boldsymbol{(\mu/\rho \stackrel{+}{,} \lambda)}$. ρ denotes the number of individuals from μ used in the creation of a single λ, (i.e. number of parents of each child) this report will only consider the case $\rho = 1$. The + and , are explained below: [19, p. 6-10] [41, s. 4.1.2]

$(\mu\ ,\ \lambda)$ Denotes an ES that has a population size of lambda. The top μ individuals are taken from the λ in generation $g - 1$, which then produce λ children for generation g. This is done by creating λ/μ clones of each μ and then mutating them individually.

$(\mu + \lambda)$ Differs from the **$(\mu\ ,\ \lambda)$** variant by adding the μ individuals chosen from generation $g-1$ to the new generation g after the mutation phase. Hence the population size is $\lambda + \mu$.

1.1.4 Online/Offline Learning

Offline An offline (or batch) learner trains on an entire dataset before applying changes.

Online A online learner reacts/learns from data immediately after each datapoint.

1.2 Motivation

Ventures in utilising learning in commercial video games have been limited and largely ineffectual. Game development works on strict cycles and there are limited resources to invest into AI research, especially if the outcome is uncertain. Furthermore, one player playing one game produces a very small data set, making learning from the player challenging. [22]

However, there are many reasons why good execution of these techniques is desirable, especially biologically inspired learning. Humans must learn and react to environments and scenarios during games, based purely on their perception of the game (and not its inner working). Having non-playable characters do the same may produce a more believable, immersive and relatable AI. Secondly, such learning algorithms produce agents that can respond well in new situations (over say FSMs or discrete logic), making new content easy to produce or generate. Lastly, modern games have large and diverse player bases, having a game that can respond and personalise to a specific player can help cater to all. [11, p. 7, p. 13]

Despite the lack of commercial success, video games can act as great benchmark for learning agents. Playing games (as a human) naturally has some degree a learning curve. They are designed to challenge humans, and will therefore challenge learning methods, particularly those inspired by biological processes. Also, games require quick reactions to stimulus, something not true of traditional AI challenges such as board games. In addition, most games have some notion of scoring, suitable for a fitness function. Lastly, they are generally accessible to students, academics and the general public alike. [11, p. 9] [12, p. 1] [26, p. 2]

As such, games have now been utilised in several academic AI competitions. These competitions are at the forefront of research and development into learning techniques in video games, and will be explored in more detail in Section 2.1.

Exploring the use of learning techniques for use in video games is a challenging and eminent area of research, with interest from both the video

game and computation intelligence communities. This project is motivated by this fact and influenced by the variety of previous approaches taken in these competitions and the unexpected results they produced.

1.3 Aim and Objectives

The aim of the project is to explore the use of behavioural learning techniques in creating a game-playing agent-based AI. This will be achieved by producing an intelligent agent, developed by an evolutionary algorithm, that plays a 2D side-scrolling platform game.

Objectives

1. **Design**
   Influenced by previous approaches, design an agent and learning procedure that can demonstrate meaningful improvement during learning.

2. **Implementation**
   Implement agent and learning designs in a modular, customisable and test-driven manner, utilising external libraries and features of the game software.

3. **Testing**
   Provide significant test coverage to the functionality of the implementation.

4. **Learning**
   Grant the agent a large and diverse testbed from which to learn as well as ample time and resources to do so.

5. **Evaluation**
   Evaluate both the competence and learning capability of the agent and compare to alternative approaches.

1.4 Methodology

The two central components to this project, the agent and the learning process, will be designed, implemented and tested as two separate modules. This presents a clean separation of concerns. The agent module will be completed first, followed by the learning module. A third module will also be created, which will allow the agent to play generated levels and receive a score.

It is foreseeable that certain parts of the project will be exploratory and as such a redesign of the either the agent or learning module may be required. Therefore, it will be important to take an agile and iterative approach to production, revisiting past objectives if need be.

The open-source Mario benchmark software[1] (used in the Mario AI Competition) will form the initial codebase for this project. It is written in Java and contains the game-engine and useful features such as level generation and level playing classes.

I intend to use Scala as the main language of this project as I believe that its functional and concurrent features will suit the problem. Scala and Java are widely compatible (as they both compile JVM bytecode) so integration should not have a significant impact on project time. However, the codebase will be assessed for the suitability of this approach, and Java 8 will be used if Scala proves to be counterproductive.

Furthermore, the software will be assessed for the inclusion of build tools, such as *sbt* and *Maven*; testing frameworks, such as *ScalaMock* and *Mockito* and logging, such as *log4j* and Scala's inbuilt logging package.

Objective 1: Design

The design of both the agent and learning module will follow the approach adopted in previous work regarding the REALM agent, which is discussed in detail in Section 2.2.3.

As in previous work, the agent module developed for the project will be a rule-based system that maps environment based conditions to explicit key presses. However, the proposed approach will differ from the REALM agent by exploring several additional perceptions of the available sensory information. These perceptions will be limited to 'visible' environment (e.g. the direction Mario is moving, presence of enemies etc.). They will be measured and distilled into a collection of simple observations.

The learning module will utilise an offline genetic algorithm as seen in previous work, which will be described later in Section 2.2.3. Agents will be individuals, with rulesets as genomes and fitness will be determined by playing a series of generated levels. However, different approaches to mutation, crossover and reordering will be explored, as well as careful consideration of calculating fitness from level playing statistics.

Objective 2: Implementation

The agent module will conform to the *Agent* interface included in the benchmark. This will allow the agent to used in other areas of the benchmark software, such as the game-playing and evaluation classes. Each agent will be defined by its ruleset, the contents of which will determine its behaviour. The agent will implement a semantic reasoner for the ruleset, returning a action as prescribed by the chosen rule.

The agent will gather its sensory information from the game's *Environment* class, which reports visible properties of the level scene. If necessary

[1] Available at http://www.marioai.org/gameplay-track/getting-started

this class will be extended to include any missing 'visible' data. As discussed in 1.4, this will be simplified to a list of observations and compared against the agent's ruleset.

The learning module will utilise an external library alongside the level playing module. Many evolutionary computation libraries exist for Java (and therefore Scala), *ECJ* [40] and *JGAP* [43] will both be considered for use in this project.

The implementation will manage the integration of this library with both the agent and level playing modules. The algorithm will evolve a simple data structure representation of rulesets, inject them into agents and assess those agents by having them play several carefully parametrised levels. Statistics returned from these levels will form the base variables of the fitness function, with multipliers being configurable. To aid improvement of the learning process, these parameters will be held externally and fitness values will be logged.

Objective 3: Testing

Test coverage of the agent module will be handled by black-box testing. Unit tests will be written to test the semantic reasoner and the environment observation methods.

Due to the stochastic nature of genetic algorithms, testing of the learning module will be limited. However, the breeding can be tested by verifying that children stay within the boundaries. Some evolutionary libraries, such as JGAP, provide several unit tests. If available, these will become the primary source of test coverage for the module.

Objective 4: Learning

Optimising the learning parameters (including the parameters of the breeding phase, fitness function and level playing) will be an important stage of the project. Assessment of the parameters used in previous agents such as REALM (discussed in 2.2.3) D. Perez et al. [37] and Agent Smith [45] will inform the initial values. Learning module logs will subsequently be analysed for improvements. For example, if there is a lot of variation in fitness then perhaps mutation should be restricted, or if the average fitness does not eventually level out then the further generations should be created.

The design of the agent can also influence the effectiveness of the learning algorithm. The size of the search space is determined by the conditions and actions of the rulesets, the reduction of which could improve evolutionary capability. Hence, the learning phase of the project may inform a redesign of the agent, which is one of the main reasons why this project will take an agile approach.

The learning itself is likely to be a time consuming, computationally heavy procedure. To assist in providing ample resources to this process the project will have access to two 8 core servers, as well as a laptop with an Intel i7 processor.

Objective 5: Evaluation

On the conclusion of the learning process, the best/final agent will be extracted and evaluated. This will be done by using the level playing module. The agent will go through an extensive set of levels, based on the approach taken by the 2010 Mario AI Competition.

The primary comparison will be with a handcrafted ruleset, which will assess the significance of the agent evolution. Other comparisons can be made against agents that are included as examples in the benchmark, such as the *ForwardJumpingAgent* that was used for similar comparisons in the 2009 competition, as well as other entrants into the competition [27, p. 7].

The second part of this objective is to evaluate the learning procedure. Figures such as average and maximum generation fitness can provide an insight into the effectiveness of the genetic algorithm. Furthermore, a baseline for these values can be provided by having the handcrafted agent play the levels alongside the evolving agent. The final evaluation report will provide an analysis of these figures.

1.5 Report Structure

This report will cover previous approaches to the project's aim; the design, implementation of the agent and learning process; and evaluate both the results and the project as a whole.

Section 2 details existing work, focusing on relevant entrants in recent Game AI competitions. It will consider their approaches to learning and agent design for both effectiveness and significance to learning in video games. It will look in particular at the Mario AI Competition and the winner entrant in 2010, the REALM agent.

Section 3 will cover the project's specification, discussing functional and non-functional requirements. It will also include a description of the major dependencies that influenced project design.

Section 4 will explain the design, implementation and testing of the rule-based agent. It will demonstrate how the agent was built to enable evolution by a genetic algorithm, as well as how it perceives its environment and chooses an action. Reasons for the choice of project language and build tools are also included here.

Section 5 will detail the development of the level playing module. This contains an account of the modifications that had to be made to the game

engine software. It covers the extension to the game engine and how it was designed and implemented with a view to parametrisation and level variety.

Section 6 explains the choice of genetic algorithm and the basic parameters used. It also describes extensions made to the learning library to allow it to effectively evolve the agent in an easily customisable and observable fashion. Lastly, it details how specific mutation and fitness parameters were chosen in response to initial learning runs in order to improve the process.

Section 7 presents the data gathered during the learning run(s). Using this data, it studies the effectiveness of the learning algorithm by examining metrics such as fitness increase over generations and variance. It also analyses the learnt agent(s), highlighting interesting behaviour and drawing comparisons with handcrafted agents.

Section 8 evaluates the three major portions of the project. Firstly, it considers the agent framework, evaluating it on complexity, speed and capability, and offering possible improvements. It then discusses the positives and negatives of the level playing module, as well as issues that were left unaddressed. Subsequently, it examines the choice of learning algorithm and parameters, with a view to future revisions and additions. Lastly, it considers the effectiveness of the project methodology as a whole.

Genre	Game	Description
Racing	TORCS (Open-source) [29]	**The Simulated Car Racing Competition** Competitors enter agent drivers, that undergo races against other entrants which include qualifying and multi-car racing. The competition encourages the use of learning techniques. [30]
First Person Shooter (FPS)	Unreal Tournament 2004	**The 2K BotPrize** Competitors enter 'bots' that play a multi-player game against a mix of other bots and humans. Entrants are judged on Turing test basis, where a panel of judges attempt to identify the human players. [31]
Real Time Strategy (RTS)	Starcraft	**The Starcraft AI Competition** Agents play against each other in a 1 on 1 knock-out style tournament. Implementing an agent involves solving both micro objectives, such as path-planning, and macro objectives, such as base progression. [32]
Platformer	Infinite Mario Bros (Open-source)	**The Mario AI Competition** Competitors submit agents that attempt to play (as a human would) or create levels. The competition is split into 'tracks', including Gameplay, Learning, Turing and Level Generation. In Gameplay, each agent must play unseen levels, earning a score, which is compared to other entrants. [26]

Table 1: This table summarises some recent game AI competitions [28]

2 Existing Work

2.1 Learning Agents and Game AI Competitions

Over the last few years, several game based AI competitions have been run, over a variety of genres. These competitions challenge entrants to implement an agent that plays a game and is rated according to the competitions specification. They have attracted both academic [26, p. 2] and media interest [12, p. 2]. The competitions tend to encourage the use of learning techniques, hence the recent publication of several interesting papers concerning the application of biologically inspired learning agents in video games. Approaches tend to vary widely, modelling and tackling the problem differently and adapting traditional techniques in previously unseen ways. [26, p. 11]

Some brief details of the competitions which are of relevance to this project are compiled in to Table 1. The Mario AI Competition is also explored in more detail below.

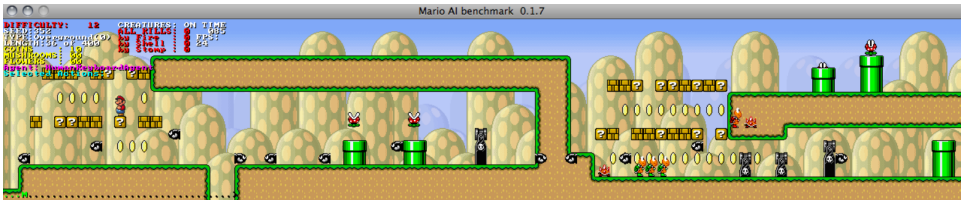


Figure 2: Example of part of a level in the Mario AI Competition's benchmark software. Taken from [26, p. 6]

2.1.1 The Mario AI Competition

The Mario AI Competition, organised by Sergey Karakovskiy and Julian Togelius, ran between 2009-2012 and used an adapted version of the open-source game Infinite Mario Bros. From 2010 onwards the competition was split into four distinct 'tracks'. We shall focus on the unseen Gameplay track, where agents play several unseen levels as Mario with the aim to finish the level (and score highly). [12] [26]

Infinite Mario Bros.

Infinite Mario Bros (IMB) [33] is an open-source clone of Super Mario Bros. 3, created by Markus Persson. The core gameplay is described as a *Platformer*. The game is viewed side-on with a 2D perspective. Players control Mario and travel left to right in an attempt to reach the end of the level (and maximise score). The screen shows a short section of the level, with Mario centred. Mario must navigate terrain and avoid enemies and pits. To do this, Mario can move left and right, jump, duck and speed up. Mario also exists in 3 different states, *small*, *big* and *fire* (the latter of which enables Mario to shoot fireballs), accessed by finding powerups. Touching an enemy (in most cases) reverts Mario to a previous state. Mario dies if he touches an enemy in the *small* state or falls into a pit, at which point the level ends. Score is affected by how many coins Mario has collected, how many enemies he has killed (by jumping on them or by using fireballs or shells) and how quickly he has completed the level. [26, p. 3]

An example of a typical level in IMB can be found in Figure 2.

Suitability to Learning

The competitions adaptation of IMB (known henceforth as the 'benchmark') incorporates a parameterised level generator and allows for the game to be sped-up by removing the reliance on the GUI and system clock. This makes it a great testbed for reinforcement learning. The ability to learn from large sets of diverse data makes learning a much more effective technique. [26, p. 3]

Besides that, the Mario benchmark presents an interesting challenge for learning algorithms. Despite only a limited view of the "world" at any one time the state and observable space is still of quite high-dimension. Though not to the same extent, so too is the action space. Any combination of five key presses per timestep gives an action space of 2^5 [26, p. 3]. Consequently, part of the problem when implementing a learning algorithm for the Mario benchmark is reducing these search spaces. This is the topic of papers by Handa and Ross and Bagnell [35], who separately addressed this issue in their papers [34] and [35] respectively.

Lastly, there is a considerable learning curve associated with Mario. The simplest levels could easily be solved by agents hard coded to jump when they reach an obstruction, whereas difficult levels require complex and varied behaviour. For example, traversing a series of pits may require a well placed series of jumps, or passing a group of enemies may require careful timing. Furthermore, considerations such as score, or the need to backtrack from a dead-end greatly increase the complexity of the problem. [26, p. 3, p. 12]

2.2 Previous Learning Agent Approaches

Agent-based AI approaches in commercial games tend to focus on finite state machines, behaviour trees and rulesets, with no learning component. Learning agents are more prevalent in AI competitions and academia, where it is not only encouraged, but viewed as an interesting research topic [12, p. 1]. Examples from both standpoints are compiled in Table 2.

2.2.1 Evolutionary Algorithms

In Section 1.1 we presented some basic concepts of evolutionary and genetic computing. This approach is a common choice of learning methods used in game-playing agents. D. Perez et al. note in their paper [37, p. 1] that evolutionary algorithms are particularly suitable for video game environments:

> *'Their stochastic nature, along with tunable high- or low-level representations, contribute to the discovery of non-obvious solutions, while their population-based nature can contribute to adaptability, particularly in dynamic environments.'*

The evolutionary approach has been used across several genres of video games. For example, *neuroevolution*, a technique that evolves neural networks, was used in both a racing game agent (by L. Cardamone [30, p. 137]) and a FPS agent (by the UT∧2 team [31]). Perhaps the most popular approach was to use genetic algorithms (GAs) to evolve a more traditional game AI agent. R. Small used a GA to evolve a ruleset for a FPS agent [45], T. Sandberg evolved parameters of a potential field in his Starcraft agent [39] , *City Conquest's* in-game AI used an agent-based GA-evolved build

plan [22] and D. Perez et al. used a grammatical evolution (a GA variant) to produce behaviour trees for a Mario AI Competition entry [37].

2.2.2 Multi-tiered Approaches

Several of the most successful learning agents take a multi-tiered approach. By splitting high-level behaviour from low-level actions agents can demonstrate a more interesting, and even human-like, performance. For example, COBOSTAR, an entrant in the 2009 Simulated Car Racing Competition, used offline learning to determine high-level parameters such as desired speed and angle alongside a low-level crash avoidance module [30, p. 136]. UT$\wedge$2 used learning to give their FPS bot broad human behaviours and a separate constraint system to limit aiming ability [31]. Overmind, the winner of the 2010 Starcraft Competition, planned resource use and technology progression at a macro level, but used A* search micro-controllers to coordinate units [9].

One learning agent that successfully utilised both an evolutionary algorithm and a multi-tiered approach is the Mario agent REALM, which is explored in more detail below.

2.2.3 REALM

The REALM agent, developed by Slawomir Bojarski and Clare Bates Congdon, was the winner of the 2010 Mario AI competition, in both the unseen and learning Gameplay tracks. REALM stands for **R**ule Based **E**volutionary Computation **A**gent that **L**earns to Play **M**ario. REALM went through two versions (V1 and V2), with the second being the agent submitted to the 2010 competition.

Rule-based

Each time step REALM creates a list of binary observations of the current scene, for example IS_ENEMY_CLOSE_LOWER_RIGHT and IS_PIT_AHEAD. Conditions on observations are mapped to actions in a simple ruleset. These conditions are ternary (either TRUE, FALSE or DONT_CARE) [36, p. 85]. A rule is chosen that best fits the current observations, with ties being settled by rule order, and an action is returned [36, p. 86].

Actions in V1 are explicit key-press combinations, whereas in V2 they are high-level plans. These plans are passed to a simulator, which reassesses the environment and uses A* to produce the key-press combination. This two-tier approach was designed in part to reduce the search space of the learning algorithm. [36, pp. 85-87]

Learning

REALM evolves ruleset using an ES for 1000 generations. The best performing rule set from the final generation was chosen to act as the agent for the competition. Hence, REALM is an agent focused on offline learning. [36, pp. 87-89]

Populations have a fixed size of 50 individuals, with each individual's genome being a ruleset. Each rule represents a gene and each individual has 20. Initially rules are randomised, with each condition having a 30%, 30%, 40% chance to be TRUE, FALSE or DONT_CARE respectively.

Individuals are evaluated by running through 12 different levels. The fitness of an individual is a modified score, averaged over the levels. Score focuses on distance, completion of level, Mario's state at the end and number of kills. Each level an individual plays increases in difficulty. Levels are predictably generated, with the seed being recalculated at the start of each generation. This is to avoid over-fitting and to encourage more general rules.

REALM used the $(\boldsymbol{\mu + \lambda})$ variant ES, with $\mu = 5$ and $\lambda = 45$ (i.e. the best 5 individuals are chosen and produce 9 clones each). Offspring are exposed to: **Mutation**, where rule conditions and actions may change value; **Crossover**, where a rule from one child may be swapped with a rule from another child[2] and **Reordering**, where rules are randomly reordered. These occur with probabilities of 10%, 10% and 20% respectively [36, p. 88]. Unfortunately, the method employed to perform these operations is not clearly explained in the REALM paper [36].

Performance

The REALM V1 agent saw a larger improvement over the evolution, but only achieved 65% of the V2 agent's score on average. It is noted that V1 struggled with high concentrations of enemies and large pits. The creators also assert that the V2 agent was more interesting to watch, exhibiting more advanced and human-like behaviours. [36, pp. 89-90]

The ruleset developed from REALM V2 was entered into the 2010 unseen Gameplay track. It not only scored the highest overall score, but also the highest number of kills and was never disqualified (by taking too long to decide on an action). Competition organisers note that REALM dealt with difficult levels better than other entrants. [26, p. 10]

[2]This is similar to a $(\boldsymbol{\mu/\rho + \lambda})$ ES approach with $\rho = 2$, but crossover occurs in the mutation phase and between all children, rather than specifically with children from another parent.

Name	Game/Competition	Approach
M. Erickson [26]	2009 Mario AI Competition	A crossover heavy GA to evolve an expression tree.
E. Speed [27]	2009 Mario AI Competition	GA to evolve grid-based rulesets. Ran out of memory during the competition.
S. Polikarpov [27, p. 7]	2009-10 The Mario AI Competition	Ontogenetic reinforcement learning to train a neural network with action sequences as neurons.
REALM [36]	2010 Mario AI Competition	GA to evolve rulesets mapping environment to high-level behaviour.
D. Perez et al [37]	2010 Mario AI Competition	Grammatical evolution with a GA to develop behaviour trees.
FEETSIES [38]	2010 Mario AI Competition	"Cuckoo Search via Lévy Flights" to develop a ruleset mapping an observation grid to actions.
COBOSTAR [30, p. 136]	2009 Simulated Car Racing Competition	Covariance matrix adaptation evolution strategy to map sensory information to target angle and speed. Online reinforcement learning to avoid repeating mistakes.
L. Cardamone [30, p. 137]	2009 Simulated Car Racing Competition	Neuroevolution to develop basic driving behaviour.
Agent Smith [45]	Unreal Tournament 3	GAs to evolve very simple rulesets, which determine basic bot behaviour.
UT∧2 [31]	2013 2K Botprize	Neuroevolution with a fitness function focused on being 'human-like'.
T. Sandberg [39]	Starcraft	Evolutionary algorithms to tune potential field parameters.
Berkeley Overmind [9]	The Starcraft AI Competition	Reinforcement learning to tune parameters for potential fields and A* search.
In-game Opponent AI [22]	City Conquest	GAs to evolve build plans with fitness measured in a 1-on-1 AI match.
In-game Creature AI[23]	Black & White	Reinforcement Learning applied to a neural network representing the creatures desires.
In-game Car AI [24]	Project Gotham Racing	Reinforcement learning to optomise racing lines.

Table 2: Biologically inspired learning, agent-based approaches to game playing AI

3 Project Specification

3.1 Functional Requirements

Functionally, the project can be split into three parts: the agent framework, which is responsible for gathering sensory information from the game and producing an action; level generation and playing, which is responsible for having an agent play generated levels with varying parameters; and the learning module, which will apply a genetic algorithm to a representation of an agent, with an aim to improving its level playing ability.

3.1.1 Agent

The agent framework will implement the interface supplied in the Mario AI benchmark; receive the game *Environment* and produce an *Action*. It must be able to encode individual agents into a simple format (e.g. a bit string or collection of numbers). Additionally, it should be able to encode and decode agents to and from external files.

The framework will be assessed in three ways. Firstly on its complexity, keeping the search space of the encoded agent small is important for the learning process. Secondly on its speed, the agent must be able to respond within one game tick. Thirdly on its capability, the framework must facilitate agents that can complete the easiest levels, and attempt the hardest ones. It is also required that human created agent encoding(s) be written (within the framework) to assist this assessment.

3.1.2 Level Playing

The level generation and playing module must be able to generate and play levels using an agent, producing a score on completion. It should extend the existing framework included in the Mario AI Benchmark software. Furthermore, it should be able to read parameters for generation and scoring from an external file.

The module will be evaluated on the level of variety in generated levels and the amount of information it can gather in order to score the agent.

3.1.3 Learning

The learning module should utilise a genetic algorithm to evolve an agent (in encoded form). It should also ensure that as many as possible of the parameters that govern the process can be held in external files, this includes overall strategy as well fine grained detail (e.g. mutation probabilities and evaluation multipliers). Where impossible or inappropriate to hold such parameters externally, it must be able to read them dynamically from their governing packages, for example boundaries on the agent encoding should

be loaded from the agent package, rather than held statically in the learning module. It must have the facility to report statistics from learning runs, as well as write out final evolved agents.

The learning module will also be assessed on three counts. Firstly, learning should not run for too long, grating the freedom to increase generation count or adjust parameters. Secondly, the learning process should demonstrate a meaning improvement to the agent over generations as this demonstrates an effective genetic algorithm. Thirdly, the final evolved agent will be assessed, using the level playing package. It will be tested against the handcrafted agents and analysed for behaviours and strategies not considered during their creation.

3.2 Non-functional requirements

Both the level playing module and the agent framework should not prevent or harm thread safety, allowing multi-threading in the learning module. Each part should be deterministic, i.e. if given the same parameter files will always produce the same results. Lastly, the entire project must have the ability to be packaged and run externally.

3.3 Major Dependencies

The project has two major dependencies: a game engine and a learning library. Their selection influenced the design of all aspects of the project and hence are included here.

As previously mentioned, the project will extend the Mario AI Benchmark. As previously discussed in Section 2.1.1 the Mario AI Benchmark is an open-source Java code-base, build around a clone of the game Super Mario Bros. 3. It was chosen for its aforementioned suitability to learning and for its other pertinent features, including an agent interface, level playing and generation package and other useful additions (e.g. the ability to turn off the reliance on the system clock).

The use of the Mario AI Benchmark restricts the choice of language (discussed further in Section 4.2.1) to those that can run on the JVM. Hence, ECJ was chosen as the learning library as it is written in Java and is available as both source-code and packaged jar. Furthermore, ECJ fit the project specification very well: it provides support for GAs and ESes, its classes and settings are decided at runtime from external parameter files, it is highly flexible and open to extension, and has facilities for logging run statistics.

Although the intention was to include these dependencies as packaged jars, modification of their source code was necessary (which was available on academic licences). This code was modified in Java, packaged and included as dependencies in the main project. Details of the modifications can found in Sections 5.3 and 6.3.1.

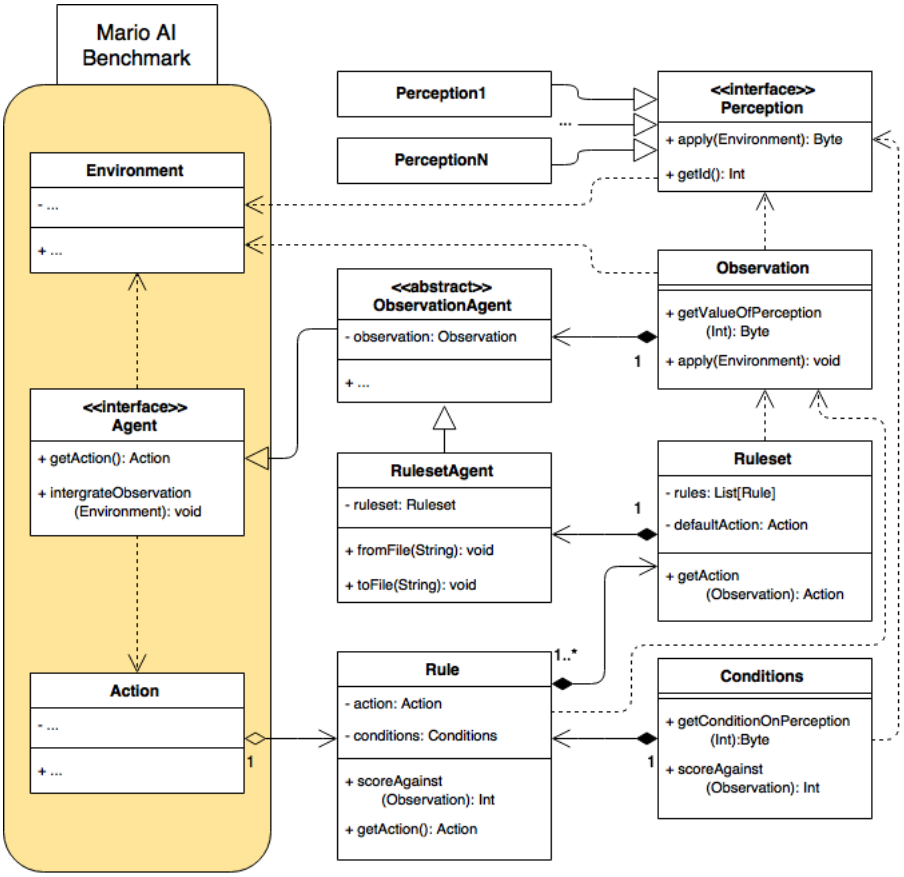


Figure 3: UML class diagram of the agent framework.

4 Agent Framework

4.1 Design

The benchmark offers an *Agent* interface, which when extended, can be passed into the game playing portion of the software. Every game tick the current *Environment* is passed to the agent, and then the agent is asked for an *Action*, which is a set of explicit key press for Mario to perform.

The agent framework's basis will extend this interface. Similar to REALM's agent, it will be a rule-based system. A UML class diagram of the system is given in Figure 3.

Each *Agent* instance is initialised with a ruleset, consisting of a list of rules, each containing an *Action*. On receiving the current game *Environment*, it is passed to a number of perception objects. Each one is responsible for a exactly one measurement (e.g. Enemy ahead, Mario moving to right etc.). These measurements are collected in an observation object and stored

(a) Obstacles *(b) Pit (with Stairs)* *(c) Enemies*

Figure 4: Example level features that agents must overcome.

in the agent.

When asked for an action, the *Agent* passes this observation to its ruleset, which tests it against each rule. Rules contain conditions on what perception measurements should be, which determine its score for a given observation. The highest scoring rule (with a conflict strategy for ties) is then asked for its action, which is returned from the agent.

Examples of the typical level features that the agent must detect and overcome can be seen in Figure 4.

Perceptions are hard coded; as such, all of an agents behaviour is determined by its ruleset. Thus, an implementation of the ruleset which allows persistence ensures that agents can be read and written to external files. Furthermore, rulesets are immutable and only information received during the current game tick is considered. Hence, agents have no state and are deterministic and thread safe (assuming the game engine is).

The capability of this framework is closely tied to the choice of perceptions. The more detail an agent is able to observe in its environment, the more potentially capable the agent can be. However, increasing the number of perceptions, and the detail they perceive at, increases the search space, making capable agents less likely to result from random mutation. Therefore, selection of perceptions must be careful considered, with a balance being struck between detail and brevity.

The perceptions chosen were based upon those used in the REALM v1 agent [36, p. 85]. The following is the final list of perceptions chosen:

MarioMode Measures whether Mario is **small**, **big** or **fire** (capable of throwing fireballs).

JumpAvailable Detects if Mario has the ability to jump (i.e. on the ground or gripping a wall with the jump key off).

OnGround Measures whether Mario is in the air or not.

EnemyLeft Detects if an enemy is to the left of Mario, as seen in Figure 4c.

EnemyUpperRight Detects if an enemy is to the upper right of Mario, as seen in Figure 4c.

EnemyLowerRight Detects if an enemy is to the lower right or directly to the right of Mario, as seen in Figure 4c.

ObstacleAhead Detects if there is a block or terrain directly to the right of Mario, as seen in Figure 4a.

PitAhead Detects if there is a pit (falling into pits will end the level) to the right of Mario, as seen in Figure 4b.

PitBelow Detects if there is a pit directly below Mario, as seen in Figure 4b.

MovingX Measures the direction Mario is moving horizontally.

MovingY Measures the direction Mario is moving vertically.

The set of actions Mario can take is any combination of 6 key presses: **left**, **right**, **up**, **down**, **jump** and **speed** (which also shoots a fireball if possible). It was decided that this set would be restricted for use in the ruleset agent, only allowing combinations of **left**, **right**, **jump** and **speed**. This was to reduce the size of the rules, which reduces the search space of any learning process. The **up** key performed no function and the **down** key made Mario crouch, which seemed too niche to be useful during the design process.

4.2 Language and Tools

4.2.1 Scala

The dependency on the Mario AI Benchmark restricts the language choice to those that can run on the JVM. Therefore, Scala was chosen to be the primary language of the project. Its functional programming elements (pattern matching, list processing etc.) are very applicable to the ruleset and its semantic reasoner. Scala's focus on immutability aids in maintaining the thread safety requirement. Furthermore, ECJ's structure necessitates the use of type casting, which Scala handles elegantly. Several other Scala features were used throughout the project, such as lambdas, singletons, type aliases and case classes.

4.2.2 Maven

With two major and several minor dependencies, their management is important to the project. Maven was chosen to maintain this, as well as package the main project and both the sub-projects. It was chosen for its

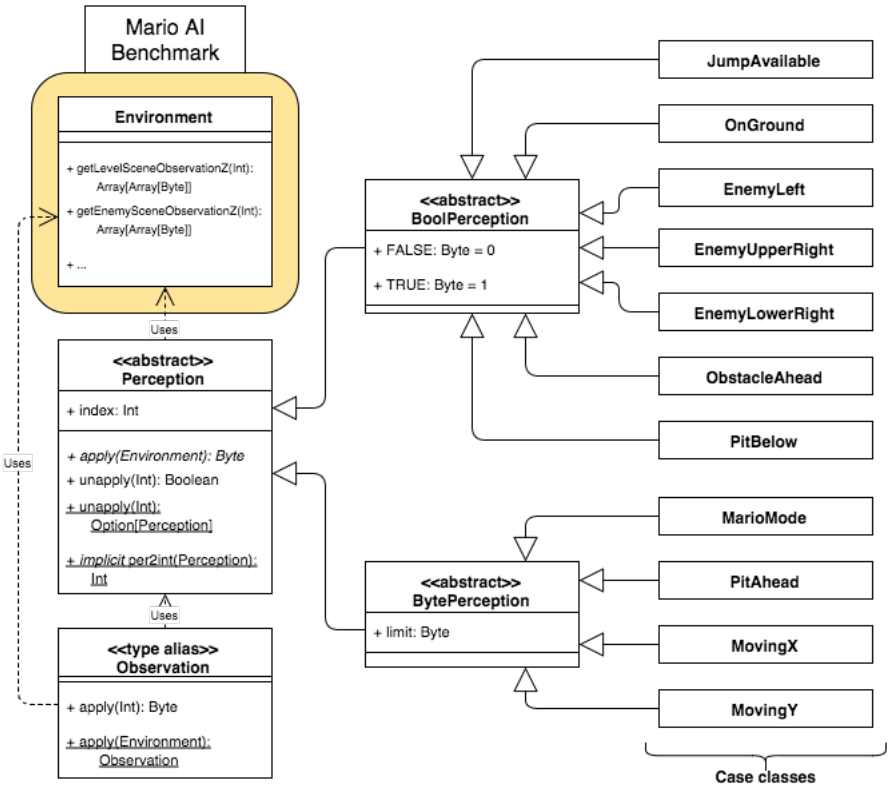


Figure 5: UML class diagram of the perception system.

ability to package both Java and Scala projects, keeping the tools consistent across the entire code base.

4.3 Implementation

The design of the agent framework allows for agents to be fully determined by their ruleset. In order for an agent to be used in an evolutionary algorithm, it must be represented by a simple data structure. Hence, implementation must allow for agent's rulesets to be unambiguously represented as, for example, a one dimensional collection. This section will describe the implementation steps taken to represent rulesets as a array of 8-bit integers, utilising Scala's built in types Vector and Byte.

4.3.1 Perceptions

The Perception interface was implemented as an abstract class, with an index integer field and an apply method. *apply* takes the game Environment and returns a Byte (an 8-bit integer) instance, representing the measurement. Perception was extended into two further abstract classes: BoolPer-

ception, which enforces apply's return be either 1 or 0, representing true and false respectively; and BytePerception, which has a limit field and enforces apply's return is between 0 and limit (inclusively). A full list of perceptions and what their byte values represent can be found in Table A1 in Appendix A.

Concrete perceptions were implemented as objects (singletons) using the *case* keyword, which allows for exhaustive pattern matching on the Perception class, as well as type safe extension by adding further case objects. Each one declares a unique index integer to the Perception superclass (starting at 0 and increasing by one for each). They implement the apply method to return their specific measurement. Figure 5 shows a class diagram of the this system.

For illustration, consider these examples. The perception PitBelow extends BoolPerception, with a unique index of 8 and implements *apply* to return 1 if there is a pit directly below Mario and 0 otherwise. MarioMode extends BytePerception, with a unique index of 0 and a limit of 2, with *apply* returning 0 if Mario is **small**, 1 for **big** and 2 for **fire**.

This approach keeps all the information about perceptions contained in one file. The number of perceptions is easily extended by adding more case objects. Furthermore, it allows the Observation and Conditions classes to be implemented as fixed length byte vectors, with the vector's index matching the perception's unique index field. With use of Scala's implicit and extractor functionality, building the Observation and validating the Conditions vectors is type safe and concise:

```
val observationVector =
    Vector.tabulate(Perception.NUMBER_OF_PERCEPTIONS) {
        n: Int => n match {
            // This retrieves the perception object with the index
            // that matches n.
            case Perception(perception) => perception(environment)
        }
    }

def validateConditions(conditionsVector: Vector[Byte]): Boolean = {
    conditionsVector.zipWithIndex.forall {
        case (b: Byte, Perception(perception)) => perception match {
            case boolP : BoolPerception =>
                (b == boolP.TRUE) || (b == boolP.FALSE)
                    || (b == DONT_CARE)
            case byteP : BytePerception =>
                ((0 <= b) && (b <= byteP.limit)) || (b == DONT_CARE)
        }
        case _ => false
    }
}
```

Notice that no information about specific concrete perceptions is required, enforcing the open-closed design principle and allowing perceptions

to be added without need to alter this code.

4.3.2 Perceiving the Environment

The *Environment* interface contains several individual methods that report Mario's situation. Therefore, implementing Perceptions that concerned Mario (e.g. MarioMode, MovingX etc.) was trivial.

Perceptions pertaining to enemies, obstacles or pits was more challenging. Environment provides two 19x19 arrays, one for enemies and one for terrain. Centred around Mario, each array element represents a 'square' (16 pixels) of the level scene. The value of an element marks the presence of an enemy or terrain square.

Enemy and obstacle perceptions pass the relevant array, a lambda test function, coordinates for a box segment of the array and a boolean to a helper function. This function uses Scala's for loop comprehensions to search through the box, applying the lambda to each element, returning the boolean parameter if the lambda returns true at least once. In this way it is easy to search for an enemy or obstacle in a box relative to Mario. Pits work in a similar way, but declare columns instead (see Listing A1 in Appendix A). If there is no terrain in the column below Mario's height, then it is considered a pit. Take EnemyLeft for example:

```
case object EnemyLeft extends BoolPerception(3) {
    // Minus = (Up,Left) | Plus = (Down,Right)
    val AREA_UL = (-2,-2); val AREA_BR = (1, -1);

    def apply(environment: Environment): Byte = {
        if(Perception.enemyInBoxRelativeToMario(environment,
            AREA_UL, AREA_BR)) 1 else 0
    }
}

def enemyInBoxRelativeToMario(environment: Environment,
          a: (Int, Int), b: (Int, Int)): Boolean = {
    val enemies = environment.getEnemiesObservationZ(2);
    val test = (grid: Array[Array[Byte]], tup: Tuple2[Int, Int]) =>{
         val x = grid(tup._1)(tup._2)
         x == 1
    }
    checkBox(enemies, test, getMarioPos(environment), a, b, true)
}
```

```
def checkBox(grid: Array[Array[Byte]],
             test: (Array[Array[Byte]], (Int, Int))=>Boolean,
             mario: (Int, Int),
             a: (Int, Int), b: (Int, Int),
             ret: Boolean): Boolean = {

    val relARow = min(grid.length-1, max(0, (a._1 + mario._1)))
    val relACol = min(grid(0).length-1, max(0, (a._2 + mario._2)))
    val relBRow = min(grid.length-1, max(0, (b._1 + mario._1)))
    val relBCol = min(grid(0).length-1, max(0, (b._2 + mario._2)))

    for {
      i <- min(relARow, relBRow) to max(relARow, relBRow)
      j <- min(relACol, relBCol) to max(relARow, relBCol)
      if (test(grid, (i, j)))
    }{ return ret }
    !ret
}
```

4.3.3 Observation, Conditions and Actions

For clarity an Action class was created for use in the ruleset, with an adapter method to convert it into the boolean array expected in the Agent interface. As the Observation and Conditions classes were to be implemented as fixed length byte vectors, so was the Action class.

Action vectors have a fixed length of 4, where elements represent **left**, **right**, **jump** and **speed** respectively. Observation vectors have a fixed length equal to the number of perceptions and hold the byte returned by each Perception's apply function. Conditions have the same length and hold data in the same range as Observation, the condition on each perception therefore being that the corresponding element in the observation be equal. They have one additional possible value, DONT_CARE (equal to byte -1), which represents that no condition be placed on that perception.

Instead of implementing these classes as wrappers for *Vector[Byte]*, which can be inefficient and overly verbose, type aliases were used. This allowed each class to be referred to explicitly (rather than just by variable name), which provides readability and type safety, whilst still having direct access to the list processing methods included in the Vector class. They were declared on the agent framework's package object, making them accessible package wide. An object with static data and factory methods was included for each.

For example, this allowed Observation to be used as such:

```
abstract class ObservationAgent extends Agent {
    // Using factory method for a blank observation
    var observation: Observation = Observation.BLANK
    ...
```

```
    def integrateObservation{env: Environment}: Unit = {
        // Using the Observation factory method
        // to build a new observation
        observation = Observation(env)
        if (printObservation) {
            // Using Vector method foreach directly
            observation.foreach {
                b:Byte => print(" ~ " + b)
            }
        }
        ...
    }
}
```

4.3.4 Rules and Rulesets

Having both Conditions and Action implemented as byte vectors allows rules to be represented in the same way. Each rule is simple the concatenation of the Conditions and Action vectors. The rule vector are fixed length as both Conditions and Action are. Moreover, as rulesets contain just a list of rules, rulesets can be unambiguously represented by a single dimension byte vector. This allows rulesets not only to be persisted easily (as say a csv) but also gives the data representation needed for the evolutionary process.

In this case both Rule and Ruleset were implemented as wrapper classes for *Vector[Byte]* and *Seq[Rule]* respectively. Ruleset also holds a default Action, which is used if no rule matches the environment.

The semantic reasoner of the rule system is split across both classes. In Rule, the *scoreAgainst* method is passed the observation and produces a score by looping through the conditions and adding 1 if the condition and observation match in value and 0 if the condition is DONT_CARE. If a mismatched condition is found, the method immediately returns with -1. It is implemented tail recursively to provide maximum efficiency.

```
def scoreAgainst(observation: Observation): Int = {
    val conditions = ruleVector.slice(0, Conditions.LENGTH)
    @tailrec
    def scoreRecu(i: Int, sum: Int = 0): Int = {
        if (i == Conditions.LENGTH) sum
        else conditions(i) match {
            case Conditions.DONT_CARE => scoreRecu(i+1, sum)
            case b if b == observation(i) => scoreRecu(i+1, sum+1)
            case _ => -1
        }
    }
    scoreRecu(0)
}
```

In Ruleset, the *getBestAction* is passed the observation and returns an action boolean array. Using tail recursion it performs a fold operation on

its rules, saving and returning the best scoring rule (preferring earlier rules when tied). If no rule gets a score of 0 or above then the default action is returned.

```
def getBestExAction(observation: Observation): Array[Boolean] = {

    @tailrec
    def getBestRuleRecu(rs: Seq[Rule], best: Option[Rule] = None,
        bestScore: Int = 0): Option[Rule] =
        rs match {
          case Nil => best
          case (r +: ts) => {
              val newScore = r.scoreAgainst(observation)
              if (newRuleBetter(bestScore, newScore))
                  getBestRuleRecu(ts, Some(r), newScore)
              else
                  getBestRuleRecu(ts, best, bestScore)
        }
    }

    getBestRuleRecu(rules) match {
        case None => defaultAction.toBooleanArray
        case Some(r) => r.getAction.toBooleanArray
    }
}
```

4.3.5 Persistence

Agent's are persisted by persisting their ruleset. Rulesets are persisted in single line csv files. An IO helper object is passed an agent, extracts its ruleset and requests its vector representation, writing each byte separated by a comma. On reading an agent file, it constructs a byte vector. This byte vector is passed to the Ruleset's factory method, which groups the vector by rule length to form the rule sequence.

4.4 Testing

Due to the agent modules heavy reliance of the Environment interface the use of a mocking facility was required. The ScalaMock testing library was adding to the project to provide this.

Perceptions were unit tested individually, using white-box approach (due to the inclusion of mocking). Each test stubbed the Environment interface, instructing it to return a specific value (or array) for the relevant call, and testing that the perception echoed or processed it correctly. This allowed Perceptions to be tested independently of the game engine and provided test coverage. However, as there was very little documentation Environment interface, expected return values had to be investigated manually and edge cases could have easily been missed.

Rulesets (and Rules) were tested with a largely black-box end-to-end style. This was required due to the reliance of type aliases. Fixed rulesets were constructed to verify that the *getAction* method returned the expected action based on a fixed observation. Mocking of individual rules was not used in case the Rule class was altered to be a type alias instead of a wrapper class.

These tests were added to Maven's build lifecycle, and hence run on every project build.

4.5 Handcrafted Agents

For the purpose of evaluation and comparison three handcrafted rulesets were created for the agent framework. Full rulesets for these agents can be found in Appendix B.

Forward Jumping This ruleset commands the agent to jump whenever it can, regardless of its surroundings. It contains a single rule and a default action. It is a blind agent that does not take advantage of the framework, however it is surprising effective. An analogous agent was used for comparisons in the 2009 Mario AI Competition and was found to score higher than many entrants [27, p. 7]. The learning process will aim to discourage this behaviour as it is neither interesting nor optimal.

Simple Reactive This agent only jumps when it detects an enemy, obstacle or pit in its way. It contains 5 rules and defaults to moving right at speed. This agent makes better use of the framework, but still does not use all of its perceptions. Its behaviour is more interesting, however it tends to score similarly to the forward jumping agent. Despite low attainment, a learnt agent that behaves similarly will be evaluating more favourably, because it is making use of the agent's perceptions.

Complex This agent is the most interesting and highest scoring of the three. It has a multitude of behaviours and builds off of the simple reactive agent. It contains 18 rules and makes use of all perceptions except MarioMode and EnemyLeft. Its behaviour was investigated at length and as such will form a good comparison to the final learnt agent. Effective evolved behaviours unconsidered in this agent's creation are a sign of the validity of the learning process.

Creation of the handcrafted rulesets also informed additions and alterations to the agent's perceptions. For example, it was originally difficult to create an effective pit strategy and so PitAhead was changed from a BoolPerception to a BytePerception, returning 3 values representing **none**, **far** and **close**.

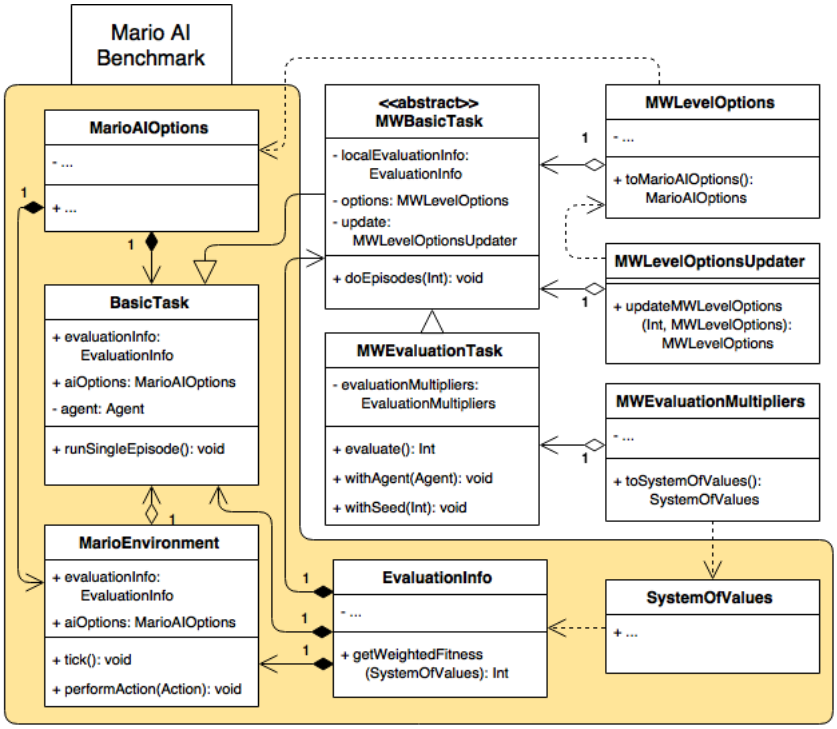


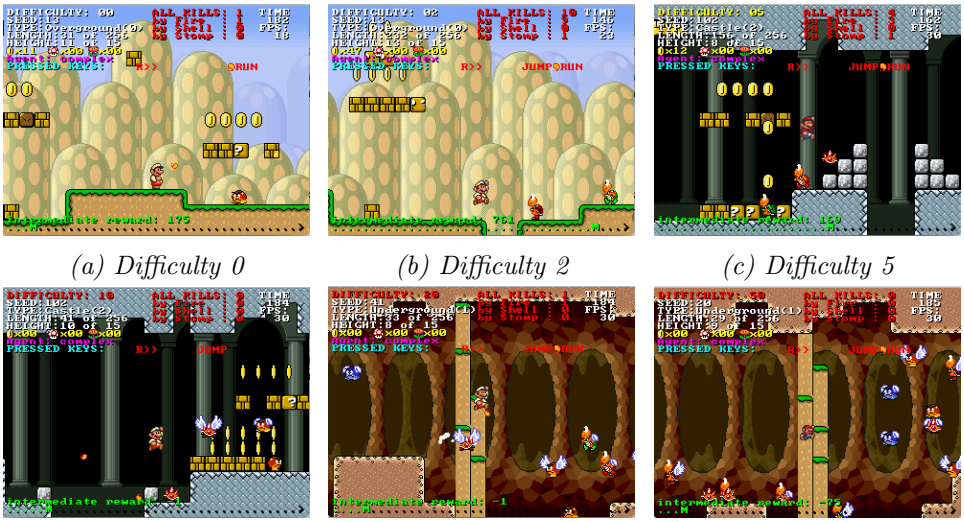
Figure 6: UML class diagram of the level playing module.

5 Level Playing Module

The main purpose of the level playing module is to evaluate the fitness of agents during the learning process. An effective learning process needs a diverse testbed, thus the level playing module must be deterministic, highly configurable and able to provide variety.

5.1 Design

The heart of the game engine is the benchmark's *MarioEnvironment* class. It is responsible for calling the *LevelGeneration* class, updating the scene each tick with an action, and reporting the scene through the Environment interface. Parameters controlling level generation and playing are contained in the *MarioAIOptions* class. The *BasicTask* class controls the game loop in conjunction with an agent. It initialises the MarioEnvironment class with options, then runs through the loop, commanding a tick, passing the environment to the agent and finally requesting an action and passing it to the MarioEnvironment instance. Statistics pertaining to the agent's performance are stored in MarioEnvironment in the *EvaluationInfo* class, which are cloned to BasicTask at the end of the level. Fitness can be requested from EvaluationInfo with the optional parameter, a *SystemOfValues* instance,



(a) Difficulty 0 *(b) Difficulty 2* *(c) Difficulty 5*

(d) Difficulty 10 *(e) Difficulty 20* *(f) Difficulty 50*

*Figure 7: Handcrafted **complex** agent playing on different difficulty levels.*

which contains the multipliers for various measurements.

The MarioAIOptions class contains several useful parameters. They include: an integer seed, which is passed to the level generator's RNGs; a boolean that toggles visualisation; integers that determine level length and time-limit; and booleans that toggle enemies, pits and blocks. One parameter of particular importance is level difficulty, which has an effect on nearly all factors of level generation. Examples of different difficulties can be found in Figure 7.

The SystemOfValues class contains multipliers for distance achieved, whether or not the level was completed, the amount of time left on completion as well as many others.

The level playing module will extend this system. Its primary objective will be to allow for multiple levels (episodes) to be played with options that update as prescribed by some parameter class. Upon completion it will produce a fitness based on all levels played. Furthermore, it will allow for the injection of different agents and level seeds, to allow different agents to play the same options without rebuilding the instance. A UML class diagram for the module can be found in Figure 6.

Level options are stored in the *MWLevelOptions* class, which acts as an adapter for the MarioAIOptions class. Each new level these will be updated by the dedicated *MWLevelOptionsUpdater* class. The *MWEvaluationMultipliers* class is an adapter for the SystemOfValues class, which is used to calculate the fitness at the end of a sequence of levels. These classes are separated from the game playing task and are included using constructor

dependency injection. This helps to decouple the system and improve the ability to modify and test.

Both MWLevelOptions and MWEvaluationMultipliers are designed to be data classes, which provides determinism and thread safety, as well as easy persistence. MWLevelOptionsUpdater's qualities in this regard are Implementation details and are discussed in a following section. A full list of field for both the MWLevelOptions and MWEvaluationMultipliers classes can be found in Appendix C.

5.2 Implementation

5.2.1 Parameter Classes

The MWLevelOptions and MWEvaluationMultipliers classes were implemented as data classes in an immutable builder pattern style. Each field has a *withField(field)* method that returns a cloned instance with that field changed. This affords a concise, declarative style, whilst maintaining immutability. MWEvaluationMultipliers has a implicit converter to a SystemOfValues instance, which is required for evaluation in the benchmark. However, a similar approach was not possible for converting MWLevelOptions to a MarioAIOptions instance (required for the game-engine). MarioAIOptions hold more parameters than MWLevelOptions (e.g. agent and level seed), therefore the adapter function takes both the current MarioAIOptions and a MWLevelOptions instance as parameters and updates the former by overriding the corresponding fields with values from the latter.

MWLevelOptions was not implemented as a class. Instead, MWBasicTask expects a lambda function. This lambda takes the episode number and the current MWLevelOptions, returning an updated set of options. MWLevelOptions builder structure ensures this is always a new instance, and hence maintains immutability. The inclusion of the episode number allows the function to remain deterministic. Moreover, with the ability to build closures in Scala, this lambda can be built from data (for example, a list of options, where indexes relate to episode number).

5.2.2 Persistence

The primary use of the level playing module is during the evaluation stage of the learning process. Hence, it was decided to use ECJ's parameter file system to persist level playing parameters, which allows them to be written in the same file as the rest of the learning parameters.

ECJ's parameter system builds upon Java's Properties file system. From a parameter file (formatted in the Java Properties format) a ParameterDatabase can be built, from which specific parameters can be requested

using the Parameter class. This system was used to persist the two parameter classes, MWLevelOptions and MWEvaluationMultipliers; the level options update lambda data; and other level playing data such as number of levels (episodes) and base level seed. For example, the following lines would set the number of levels to 10 and the base difficulty to 5:

```
          level.num-levels = 10
level.base.difficulty-num = 5
```

A static utility class EvaluationParamsUtil was created to handle the reading of the level playing data from these files. A ParameterDatabase is built and passed to utility class, which builds the required parameter class. For MWLevelOptions and MWEvaluationMultipliers it searches for the prefixes '`level.base`' and '`mult`' respectively and then looks for suffixes corresponding to specific fields. If a field's suffix is not found then it is initialised to a default value (which is always zero for MWEvaluationMutlipliers).

The update lambda is built as a closure on a collection of Map instances, one for each field in MWLevelOptions. For each `n` from 0 to the number of levels (episodes), the utility function looks for the prefix '`level.{n}`', which is used to hold the update for episode **n**. For each field a Map is built and using the same suffixes as for MWLevelOptions, key-value pairs are added mapping **n** to the value found. When the update lambda is called, it consults these maps and updates the MWLevelOptions with the new value if one is found for the current episode number.

For example, if the parameter file contained the following lines:

```
  level.num-levels = 4
level.base.enemies = false
   level.1.enemies = true
   level.3.enemies = false
```

Then enemies would be off for the first episode, on for the second and third and off again for the fourth and final episode.

5.2.3 The Episode Loop

The entry point of the level playing module is the MWEvaluationTask class, which extends the abstract MWBasicTask class. MWEvaluationTask is instantiates with a base set of options (as MWLevelOptions), a MWEvaluationMutlipliers instance and an update lambda, as well as the number of episodes (levels) to run. The agent and base level seed can be injected with the *withAgent(agent)* and *withLevelSeed(seed)* methods (which also reset of evaluation information).

```
class MWEvaluationTask(val numberOfLevels: Int,
                       val evalValues: MWEvaluationMultipliers,
                       override val baseLevelOptions: MWLevelOptions,
                       override val updateOptionsFunc: (Int,
                           MWLevelOptions) => MWLevelOptions)
                          extends MWBasicTask("MWMainPlayTask",
                              baseLevelOptions, updateOptionsFunc,
                              visualisation, args) with EvaluationTask
                               {

    private var baseLevelSeed: Int = 0;

    override def nextLevelSeed(episode: Int, lastSeed: Int) = {
        (3*episode) + lastSeed
    }

    override def evaluate: Int = {
        doEpisodes
        localEvaluationInfo.computeWeightedFitness(evalValues)
    }

    override def withAgent(agent: Agent): MWEvaluationTask = {
        super.injectAgent(agent, true)
        this
    }

    override def withLevelSeed(seed: Int): MWEvaluationTask = {
        baseLevelSeed = seed
        super.injectLevelSeed(seed, true)
        this
    }
}
```

On calling the *evaluate()* method, the number of levels is passed to the *doEpisodes(numberOfEpisode)* (a superclass method), which loops as follows:

```
def doEpisodes(amount: Int): Unit = {
    @tailrec
    def runSingle(iteration: Int, prevOptions: MWLevelOptions,
        disqualifications: Int): Int = {
        if (iteration == amount) {
            disqualifications
        } else {
            // Calls the update lambda to get episodes set of options
            val newOptions = updateOptions(iteration, prevOptions)

            // Converts options to class required for game-engine
            val marioAIOptions = MWLevelOptions.updateMarioAIOptions(
                super.options, newOptions)
```

```
            // Updates the level seed (which is set to increase each
                episode by MWEvaluationClass)
            marioAIOptions.setLevelRandSeed(nextLevelSeed(iteration,
                marioAIOptions.getLevelRandSeed))

            // Resets the evaluation information in the super class
            super.setOptionsAndReset(marioAIOptions)

            // Generates and runs the level using the super class
            // Returns true if the agent was not disqualified
            val notDisqualified: Boolean = runSingleEpisode(1)
            val disqualification = if (!notDisqualified) 1 else 0

            // Update the evaluation information for the entire run
            // (as the super classes evaluationInfo gets reset every
                level
            updateLocalEvaluationInfo(super.getEvaluationInfo)

            // Loop
            runSingle(iteration+1, newOptions, disqualifications +
                disqualified)
        }
    }

    // Sets the base options
    super.setOptionsAndReset(MWLevelOptions.updateMarioAIOptions(
        options, baseLevelOptions))
    disqualifications = runSingle(0, baseLevelOptions, 0)
}
```

Before every episode the options are updated by the lambda, updating the base options in the first episode. A new level seed is also requested from the MWEvaluationClass, which simply increases it each episode. These updates are converted and added to a MarioAIOptions instance and passed to the superclass in the benchmark. A single level is then generated and played using the superclass. Evaluation information is added to MWBasicTask's local evaluation information (as the former is reset every episode) and the function loops tail recursively.

The function exits when the it has looped for each level. At this point the *evaluate()* method requests a fitness from the local evaluation information, passing in the evaluation multipliers, which is then returned.

5.3 Modifying the Benchmark

Preliminary runs of the benchmark software revealed several issues and defects. As the learning algorithm would run over several generations, any errors could halt it prematurely, which could be costly in terms of project time keeping. In order to address this a Java project was created from the benchmark's source code and fixes were made. This code was packaged with Maven and included as a dependency in the main project.

Several minor exceptions were caught or addressed, however the two largest issues failed 'quietly'. They concerned the LevelGeneration class and surrounded enemy and pit generation with regards to level difficulty. Fixes were made to ensure level difficulty scaled more consistently.

5.3.1 Enemy Generation

In observing the benchmark generated levels it was apparent that enemy density was very high, even on lowest difficulties. Examining the LevelGeneration class revealed this to the result of what was probably unintended behaviour.

Levels are generated in zones of varying length. Quite often, a zero length zone is created, which has no effect on terrain. However, enemies were still being added to these zones, creating very high density columns of enemies during levels. This was addressed, with the addition of a more gentle enemy count curve and better spacing.

5.3.2 Pit Generation

Another apparent shortcoming was pit length. Pits were only of two sizes, small and very large, and after a certain level difficulty were always very large. Although this was intended behaviour, comments from the original developers suggest it was a placeholder for a more sophisticated system. An edit was made to scale maximum pit length on level difficulty. Each pit's length is chosen probabilistically on a bell curve, which is shifted by level difficulty.

5.4 Testing

The benchmark's BasicTask class is tightly coupled to the MarioEnvironment, which means that we were unable to mock the game-engine for testing. This problem extends to the MWEvaluationTask class.

However, the decoupling of the parameter classes from MWEvaluationTask means that the persistence section of the module can easily be tested. The EvaluationParamUtil class is white-box unit tested by stubbing the ParameterDatabase interface and verifying the contents of the parameter classes requested. For example:

```
"getEvaluationMutlipliers" should "return eval mults from database
    and zero otherwise" in {
    val base = blankParam.push(EvaluationParamsUtil.P_EVAL_BASE)
    (pdStub.exists _) when(base, *) returns(true)

    (pdStub.exists _) when(base.push(EvaluationParamsUtil.P_COINS),
        *) returns(true)
    (pdStub.getIntWithDefault _) when(base.push(EvaluationParamsUtil
        .P_COINS), *, *) returns(10)
```

Agent	Total Score	Levels Completed	Enemies Killed	Distance
Complex	1,817,195	171 (33%)	1498 (8%)	63,894 (48%)
Simple Reactive	1,095,287	88 (17%)	590 (3%)	43,286 (32%)
Forward Jumping	954,640	76 (15%)	677 (3%)	36,980 (28%)

Table 3: Competitive statistics from handcrafted agents playing the evaluation task with a seed of 10.

```
        (pdStub.exists _) when(base.push(EvaluationParamsUtil.
            P_KILLED_BY_SHELL), *) returns(true)
        (pdStub.getIntWithDefault _) when(base.push(EvaluationParamsUtil
            .P_KILLED_BY_SHELL), *, *) returns(200)

        (pdStub.exists _) when(base.push(EvaluationParamsUtil.P_DISTANCE
            ), *) returns(true)
        (pdStub.getIntWithDefault _) when(base.push(EvaluationParamsUtil
            .P_DISTANCE), *, *) returns(1)

        val evalMults = EvaluationParamsUtil.getEvaluationMutlipliers(
            pdStub, base);

        assert(evalMults.coins == 10)
        assert(evalMults.killedByShell == 200)
        assert(evalMults.distance == 1)
        assert(evalMults.win == 0)
        assert(evalMults.kills == 0)
        ...
    }
```

5.5 Comparator Task

In order to quantifiably compare agents, a competitive set of evaluation task options was created, modelled on those used during the final evaluation stage of the 2010 Mario AI Competition.

Agents play 512 levels, spread equally over 16 difficulty levels (0 to 15). Options such as enemies, pits, blocks etc. are periodically turned off for a level. Length is varies greatly from level to level, with the time-limit being adjusted accordingly. Evaluation multipliers reward all possible positive statistics, such as enemy killed, coins collected and distance travelled. A full view of the comparator task's parameter classes can be found in Appendix D.

The scores and other statistics attained by the three handcrafted agents playing the evaluation task, on seed 10, can be found in Table 3.

6 Learning

The learning module's goal is to utilise the ECJ library to evolve a population of byte vectors (representing agent rulesets). This is achieved through extending ECJ's classes, in conjunction with a parameter file. The ECJ library is an extensive collection of learning processes and, due to the limitations of this report, only the relevant sections, and the extensions thereof, are described. An important part of the project was adjusting the parameter file in reaction to previous learning runs, these revisions are discussed in Section 6.6. However, some parameters were deemed fundamental and stayed invariant throughout this process; these are described, alongside the general evolutionary approach, in the following section.

6.1 Evolutionary Approach

Similar to the approach taken in developing the REALM agent (described in Section 2.2.3), this project uses a $\boldsymbol{(\mu + \lambda)}$ evolutionary strategy. However, unlike REALM, it focuses purely on mutation, with no crossover between individuals (a technique not traditionally applied with ESes [42, p.]).

Evolution runs for 1000 generations with a population size of 50. The truncation selection size for our ES (μ) is set to 5. Hence, the top 5 fittest individuals are chosen each generation. These are cloned and mutated to form 45 (λ) new individuals, which are then joined by the original 5 for the next generation.

Each individual's genome is a byte vector of length 300, representing a ruleset containing 20 rules. The default action is fixed over all rulesets and is specified by the parameter file.

Each element of the genome (a gene) has several mutation parameters associated to it. During the mutation phase each gene will be mutated with the probability prescribed by its mutation probability. Each gene can be mutating in one of two ways. The first will simply choose a random byte between its minimum and maximum values (inclusive), known as **reset** mutation. The second will favour one particular byte value, with a certain probability, which is known as **favour** mutation. Which method of mutation a gene undergoes, as well as the favoured byte and its probability are all contained in the individual gene's parameters, allowing for very fine grained control over the mutation process.

The evaluation stage takes each individual's genome and constructs an agent. The individual's fitness is calculated by passing the agent to the MW-EvaluationTask, which has been initialised with a fixed set of level playing parameters decreed by the parameter file. These parameters have a custom set of evaluation multipliers and describe a task of 10 levels, varying level options for each level. Each generation builds this task with its own unique level seed to avoid over-fitting to one set of generated levels.

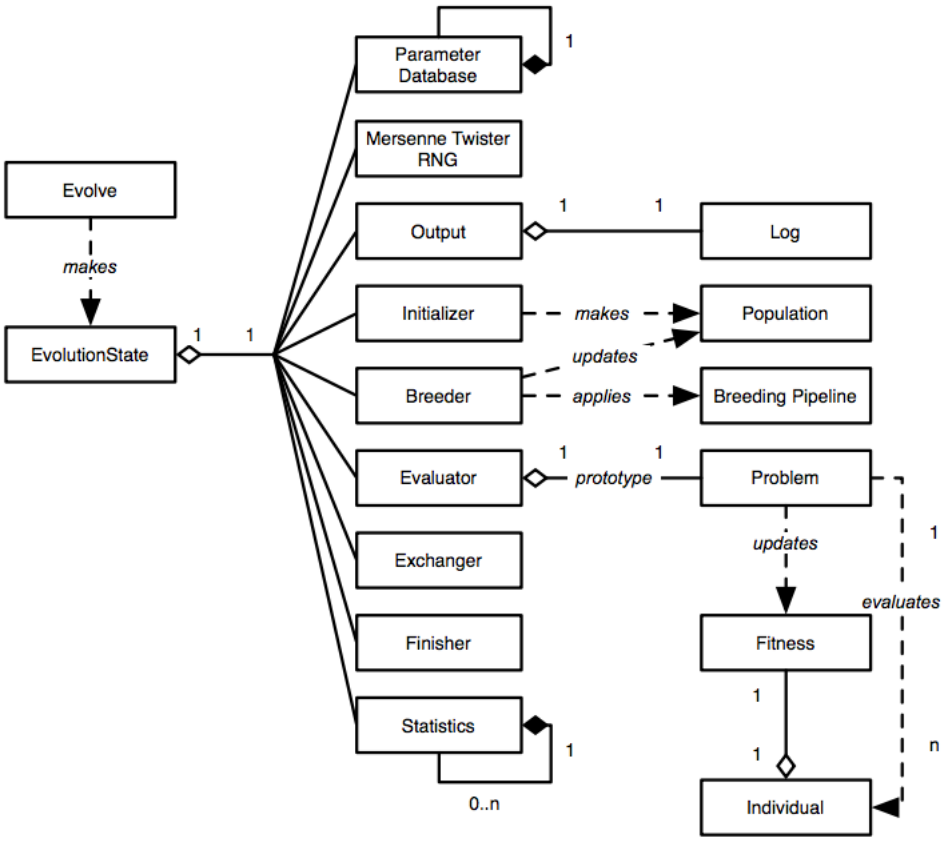


Figure 8: A top-level UML class diagram of the operation facilities in EvolutionState, taken from [41, p. 10].

Statistics for average and best fitness are logged each generation. Level seed is also reported, to allow direct comparison between the learning process and the hand-crafted agents. After the final generation a selection of agents are written to agent files, including the best individual of the final generation and the best individual over all generations.

6.2 The ECJ Library

The primary method for specialising ECJ for a particular use case is to extend the classes it provides, and specify these subclasses, alongside other pertinent values, in the parameter file. Nearly every method in ECJ's classes is passed the pseudo-global EvolutionState singleton, which holds instances of all other classes in use. A top-level view of ECJ's structure can be found in Figures 8 and 9.

The learning module will be built on top of the ECJ library, with a view to adhering to its style and ideology. For our approach, several of ECJ's

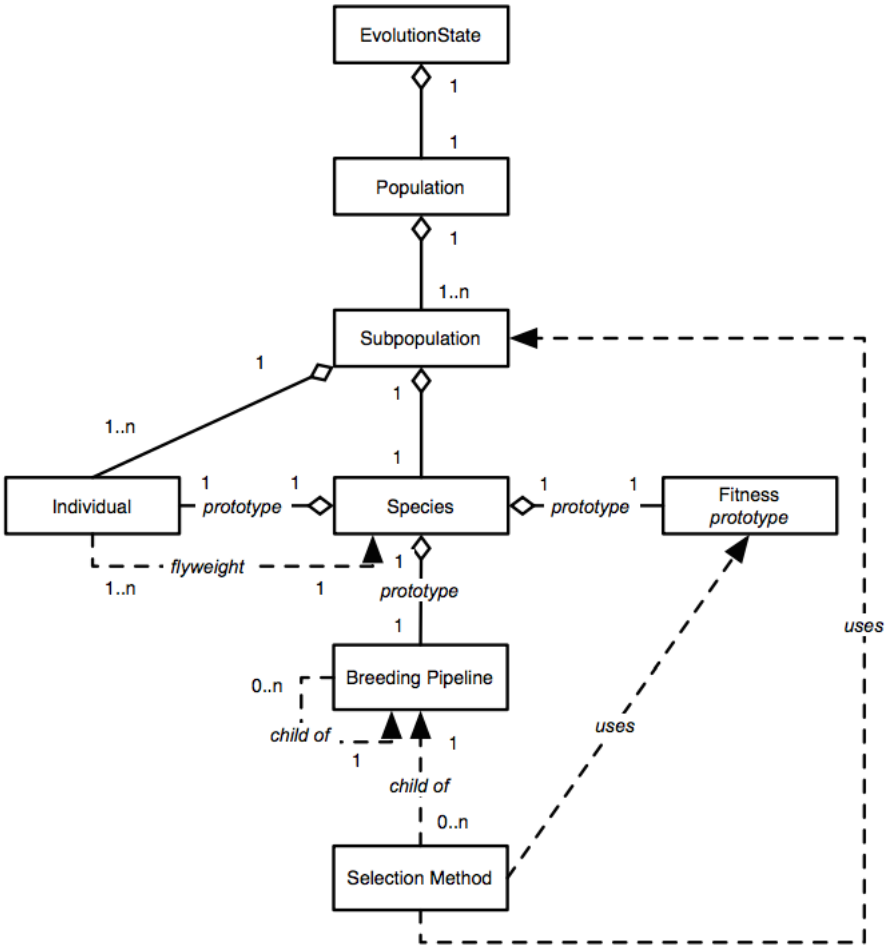


Figure 9: A top-level UML class diagram of ECJ's data objects, taken from [41, p. 11].

classes did not need to extended, and simply specifying their default implementations in the parameter file was enough. The classes of interest to this project are the following: *Species*, *Individual*, *SelectionMethod*, *Breeder*, *BreedingPipeline*, *Evaluator*, *Problem* and *Statistics*. A simplified class diagram of the learning module is included as Figure 10.

ECJ provides the *ByteVectorIndividual* class, which contains a Java array of *bytes*, this is used to hold individual byte vector genomes. It also decrees the use of the *IntegerVectorSpecies* class, which holds the gene parameters (e.g. minimum value, mutation probability). Our design first extends this class to the *DynamicParameterIntegerVectorSpecies* class, which holds a subclass instance of another new class, the *DynamicSpeciesParameters* class. The specific subclass to be used is specified in the parameter file. The purpose of this class is to provide a facility to override the min, max and

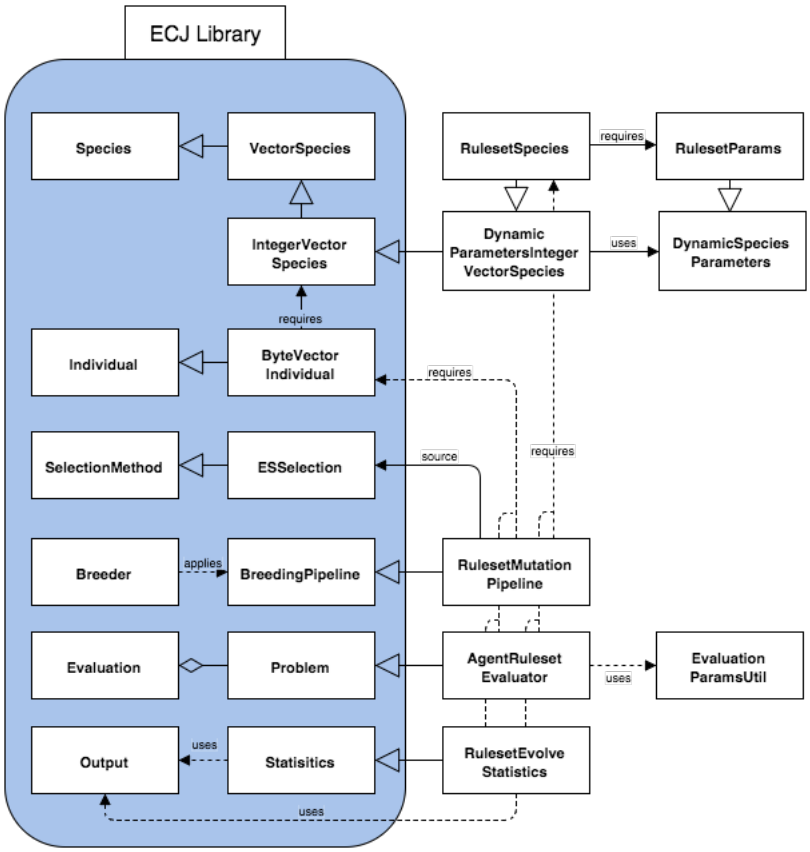


Figure 10: A simplified UML class diagram of the Learning module's extension to ECJ.

mutation probability properties of each gene directly from the agent framework (rather than the parameter file). This class is then further extended to the *RulesetSpecies* class, which holds gene parameters pertaining to favour mutation. This class also extends the parameter file's vocabulary to allow gene parameters to specified on a **condition** or **action** (the composite parts of a rule) basis. For example, the favour byte for all genes that represent conditions can be set as follows in the parameter file:

```
...species.condition.favour_byte  = -1
```

The *SelectionMethod* and *Breeder* classes are responsible for selecting individuals (based on fitness) to clone and pass to the breeding pipeline. ECJ natively supports evolution strategies and so it is enough to specify these classes to be ECJ's *ESSelection* and *MuPlusLambda* classes. The *BreedingPipeline* extension, *RulesetMutationPipeline*, performs the mutation of individuals. For each gene in a individual's genome it requests the corresponding mutation parameters from the species class. It then clones and, if required, mutates the gene with either favour or reset mutation.

The *Evaluator* class is responsible for threading and batching the individuals to have their fitness calculated by cloned instances of the *Problem* class. Hence, only the Problem class is extended, to the *AgentRulesetEvaluator* class. On initialisation this class reads the level options from the parameter file using the EvaluationParamsUtil class described in Section 5.2.2. Every generation, agents are constructed from the individuals of the population and passed, with the options and the level seed, to the MWEvaluationTask class. This returns a score, which is attached to the individual as its fitness.

Generational statistical reporting is handled by an extension of the *Statistics* class, which will output to log files specified in the parameter file.

6.3 Implementation

Extensions to the ECJ library were implemented in Scala, organised into a package and situated in the same project as the agent framework and level playing modules. This was to allow the level playing parameters to be read through ECJ's ParameterDatabase class (described in Section 5.2.2).

Nearly every class in the ECJ library implements a *setup(...)* method, which is used initialise an instance. This method is passed the EvolutionState singleton, from which it can access the ParameterDatabase, which holds the contents of the parameter file. Classes can poll the ParameterDatabase by constructing a *Parameter* instance (in a composite pattern) with string keys.

6.3.1 Modifying the ECJ Library

During implementation it became clear that the proposed species design was problematic due to the setup process of IntegerVectorSpecies and its superclass *VectorSpecies*. The setup method reads in values for gene minimum, maximum and mutation probability and then calls for the *Individual* prototype to be built. This means that any subclass of IntegerVectorSpecies cannot decide the values of the gene minimum, maximum and mutation probability. If they are written before calling setup on the superclass (which is required) they will be overridden, if they are written after the calling setup on the superclass they will not be built into the prototype. To address this the VectorSpecies class was modified to include the *prePrototypeSetup* method, which is called immediately before constructing the prototype in the setup method. Subclasses can extend this method to overwrite any properties set by IntegerVectorSpecies and VectorSpecies.

```
public void setup(final EvolutionState state, final Parameter base)
    {
    // Add min, max, mutation probability etc.
    ...

    // Allow subclasses to override values
    prePrototypeSetup(state, base, def);
    state.output.exitIfErrors();

    // NOW call super.setup(...), which will in turn sets up the
        prototypical individual
    super.setup(state,base);
}

// new method added
protected void prePrototypeSetup(final EvolutionState state, final
    Parameter base, final Parameter def) {
    //None by default, for subclasses
}
```

6.3.2 Species

IntegerVectorSpecies and VectorSpecies store gene parameters (e.g. min, max, mutation probability) as arrays of equal length to the genome, matching by index. These arrays are built during the setup method by polling the ParameterDatabase for the suffixes '`min-value`', '`max-value`' and '`mutation-prob`'. The parameter file can specify them globally, in segments or by individual gene (by index). For segments and single indexes the *loadParameterForGene* method is used, which takes the gene index and a prefix parameter (e.g. segment number or index). It builds a Parameter instance by appending the gene parameter suffixes to the passed prefix. It requests the Parameter's value from the ParameterDatabase. If a value is found, it sets it as the element of the corresponding gene parameter array at the given index..

Dynamic Parameters

The DynamicParametersIntegerVectorSpecies performs the same task as described above, but obtains the values from a specified class rather than the parameter file. This class must be a subclass of the DynamicSpeciesParameters class. DynamicSpeciesParameter define methods for global, segment and index gene parameters, each return an Option on the value. The default implementation returns None, allowing subclasses to override only the methods they want to be used in the learning run. For our learning run we used the RulesetParams class, which implemented the *minGene(index)* and *maxGene(index)* methods. These methods return the min and max value for a particular gene, with the values being obtained via the agent framework:

```
def getIndexType(index: Int): IndexType = (index % ruleLength) match
     {
    case n if n < conditionLength => Condition
    case _ => Action
}
override def maxGene(index: Int): Option[Int] = getIndexType(index)
    match {
    case Action => Some(Math.max(MWAction.ACTION_FALSE, MWAction.
        ACTION_TRUE))
    case Condition => (index % ruleLength) match {
        case Perception(perception) => perception match {
            case bp : BoolPerception => Some(Math.max(bp.TRUE, bp.
                FALSE))
            case ip : BytePerception => Some(ip.limit.toInt)
        }
        ...
    }
}
```

The setup method of DynamicParametersIntegerVectorSpecies reads the subclass name (in this case RulesetParams) from the parameter file and instantiates it. The prePrototypeSetup method is then used to write the values (overriding those set by IntegerVectorSpecies). It calls each of DynamicSpeciesParameters' methods, if None is returned no action is taken, but if a value is return it is written to the parameter arrays.

```
override def prePrototypeSetup(state: EvolutionState, base:
    Parameter, default: Parameter): Unit = {
    if (dynamicParamsClassOpt.isDefined) {
        val dynamicParamsClass: DynamicSpeciesParameters
            = dynamicParamsClassOpt.get
        if (dynamicParamsClass.minGene.isDefined) {
            fill(minGene, dynamicParamsClass.minGene.get)
        }
        ...
        for(i <- 0 until genomeSize) {
            if (dynamicParamsClass.maxGene(i).isDefined) {
                maxGene(i) = dynamicParamsClass.maxGene(i).get
            }
            ...
        }
    }
    super.prePrototypeSetup(state, base, default)
}
```

Ruleset Species

The prePrototypeSetup method of RulesetSpecies checks that the DynamicSpeciesParameter class used is RulesetParams, which contains several utility functions. Adding to globally, by segment and by index methods for declaring gene parameters, this setup also looks for Parameter prefixes '`condition`'

and '`action`'. If found, it uses a utility function to run loadParametersForGene on all indexes corresponding to the prefix. This method is also passed the prefix, in this way IntegerVectorSpecies specific parameters such as min and max can be set on a condition or action basis without needing to implement them directly. For example, take the following line in the parameter file:

```
...species.condition.mutation-prob = 0.1
```

The '`condition`' prefix is found by RulesetSpecies setup method, which indirectly loops through all indexes `x` that pertain to conditions:

```
override def prePrototypeSetup(state: EvolutionState, base:
    Parameter, default: Parameter): Unit = {
    super.prePrototypeSetup(state, base, default)
    ...
    if(state.parameters.exists(base.push(RulesetSpecies.P_CONDITION)
        , default.push(RulesetSpecies.P_CONDITION))) {
        dpc.runOnIndexes(Condition, genomeSize) {
            (x: Int, mod: Int) => {
                loadParametersForGene(state, x,
                  base.push(RulesetSpecies.P_CONDITION),
                  default.push(RulesetSpecies.P_CONDITION), "")
            }
        }
    }
}
```

It passes the condition prefix and the index to the loadParameterForGene method. In IntegerVectorSpecies, the suffix '`mutation-probability`' is added to the condition prefix and the value 0.1 is found, and set for all indexes `x` in the mutation probability array.

RulesetSpecies also extends the loadParametersForGene method to read in the values for favour mutation from the parameter file. This is controlled by three arrays: favourMutation, a boolean array which holds whether or not a gene should be favour mutated; favourByte, a byte array holding the favoured byte; and favourProbability, the probability with which this byte will be chosen. By extending loadParametersForGene these parameters can be read in on condition or action bases, as well as by segment or individual index.

6.3.3 Mutation

To implement the mutation strategy, the RulesetMutationPipeline was created. It extends the BreedingPipeline and overrides the *produce* method. It makes no use of the setup method as all mutation parameters are stored in the Species class. The breeding pipeline is threaded, and as such RulesetMutationPipeline is prototypes and cloned for each breed thread. The produce method is called to mutate batches of Individuals, which vary in

size.

RulesetMutationPipelines produce method first calls produce on its source (in this case the ESSelection class), which fills the array of individuals to be mutated. It verifies that these individuals are ByteVectorIndividuals and their Species is RulesetSpecies. It then clones each individual and resets its fitness. Lastly it loops through each individual and passes it to the *mutateIndividual* method.

The mutateIndividual method loops through the individual's genome by index. It uses this index to acquire each gene's corresponding parameters from the species class (e.g. minimum, maximum, mutation probability, favour mutated, favour byte etc.). Using the thread's random number generator stored in the EvolutionState object is decides whether to mutate by calling `nextBoolean(mutationProbability)`. If so it mutates the gene by favour or reset mutation (again decided by the gene parameters). For reset mutation it replaces the gene with a random byte between the gene's minimum and maximum. For favour mutation it makes another call to nextBoolean, with the favour probability parameter. If this returns true, the gene is replaced by the favour byte, otherwise a byte between the minimum and maximum is chosen, excluding the favour byte. The code for this method is given below (edited for brevity).

```
protected def mutateIndividual(state: EvolutionState, thread: Int,
    vecInd: ByteVectorIndividual, species: RulesetSpecies):
    ByteVectorIndividual = {
   for (n <- 0 until vecInd.genome.length) {
      if (state.random(thread).nextBoolean(species.
          mutationProbability(n))) {
         if (species.favourMutation(n)) {
            if (random.nextBoolean(favourProbability))
               vecInd.genome(n) = species.favourByte(n)
            else
               vecInd.genome(n) = getRandomByte(
                  species.minGene(n).toByte,
                  species.maxGene(n).toByte,
                  state.random(thread),
                  species.favourByte(n)
                )
         } else {
            vecInd.genome(n) = getRandomByte(
              species.minGene(n).toByte,
              species.maxGene(n).toByte,
              state.random(thread)
            )
         }
      }
   }
   vecInd.evaluated = false
   vecInd
}
```

6.3.4 Evaluation

The Evaluator class prototypes the AgentRulesetEvaluator class and holds one clone for each evaluation thread. The AgentRulesetEvaluator overrides three methods from the Problem class: *setup* which is called before creating the prototype, *prepareToEvaluate* which is called once per thread clone before evaluation, and *evaluate* which is called on single individuals multiple times per thread.

The setup method loads in the MWLevelOptions, MWEvaluationMultipliers, update lambda, number of levels, generational level seeds and the default ruleset action using the EvaluationParamUtil class. As these are built into the prototype they are shared across all evaluation threads, taking advantage of their immutability (explained in Section 5.2.1).

The MWEvaluationTask instance is built with these options once per thread, in the prepareToEvaluate method. As the game loop is mutating process it cannot be shared across threads, but as the task can be reset and new agents injected, it can be used multiple times in one thread.

The evaluate method is passed an individual, which is verified to be a ByteVectorIndividual. The individual's genome is built into a ruleset, which is used to initialise an MWRulesetAgent. This agent, along with the level seed for the current generation, is injecting into the task. The task then evaluates the agent, returning a score, which is attached to the individual as its fitness.

```
override def evaluate(state: EvolutionState, individual: Individual,
     subpop: Int, thread: Int): Unit = {
    individual match {
        case ind: ByteVectorIndividual => {
            if (task.isDefined) {
                val evalTask = task.get
                val name = this.buildIndAgentName(state, individual,
                    subpop, thread)
                val ruleset: Ruleset = Ruleset.buildFromArray(ind.
                    genome, defaultAction)
                val agent: Agent = MWRulesetAgent(name, ruleset)

                val iFitness = evalTask.withAgent(agent)
                                    .withLevelSeed(_taskSeeds(state.
                                        generation))
                                    .evaluate

                ind.fitness match {
                    case _: SimpleFitness => {
                        ind.fitness.asInstanceOf[SimpleFitness].
                            setFitness(state, iFitness.toDouble, false
                            )
                        ind.evaluated = true
                    }
```

```
                    case _ => {
                        state.output.fatal("This evaluator (
                            EvolvedAgentRulesetEvaluator) requires a
                            individuals to have SimpleFitness")
                    }
                }
            } else {
                state.output.fatal("Task was not defined when
                    evaluating individual, implying prepareToEvaluate
                     was not run on this instance.")
            }
        }
        case _ => {
          state.output.fatal("This evaluator (AgentRulesetEvaluator)
              requires a ByteVectorIndividual")
        }
    }
}
```

6.3.5 Statistics

The Statistics class and its subclasses are setup at the beginning of the learning run and then called as 'hook' at various points during the generational loop. RulesetEvolveStatistics implements *setup* and two of the hooks, *postEvaluationStatistics* and *finalStatistics*. The setup method reads the filenames for the generation log, final log and the final agent exports. It registers the log file with EvolutionState's *Output* instance to allow it to be called from the other methods.

The postEvaluationStatistics method is passed the EvolutionState, from which it can access the current population and generation number. It loops through the individuals of the current population, calculating the average fitness and determining the best individual. The average fitness, best fitness, current level seed, generation number and the best individual's genome are all written to the generation log using the Output class:

```
val all = "~all~ " + genNum + "," + levelSeed + "," + avScore + ","
    + bestScore
state.output.println(
  "--------- GENERATION " + genNum + " ---------\n", genLog)
state.output.println(all +
       "\nLevel Seed : " + levelSeed +
       "\nAverage Score : " + avScore +
       "\nBest Score : " + bestScore +
       "\nBest Agent :-" +
       "\n " + agentStr +
       "\n-----------------------------------\n\n", genLog)
```

The best individual in the generation is saved to the currentBestIndividual field. Further checks to see if it is the best overall individual or has the biggest difference to the average are also performed, resulting in the in-

dividual being saved to the bestOverallIndividual and biggestDiffIndividual fields respectively. These checks are also controlled by generation number, only being performed after a certain generation, specified in the parameter file.

The finalStatistics method retrieves the currentBestIndividual, overallBestIndividual and biggestDiffIndividual. From each it constructs a ruleset, initialises an agent with the correct default action and passes it to agent IO utility class to be persisted into the corresponding filenames captured during setup. The entire final generation and their fitness values are also written to the final log.

6.4 Testing

ECJ's structure makes unit testing extremely difficult, due mainly to its heavy reliance on the EvolutionState singleton. It is utilised by every class and passed to nearly every method. ScalaMock is unable to mock this object as it is not accessed through an interface and building a fixed instance would be overly time consuming (and tantamount to simply running the software).

Ultimately, besides the EvaluationParamsUtil class, no testing was performed. Instead, the implementation adopted a 'fail loudly' approach, where any discrepancy was logged and triggered a program shutdown. In this way errors were easily noticed and corrected.

The lack of testing is not desirable, and given more time, efforts would have been made to rectify this. However, as the software has no external 'users' (it being used as a tool solely by those that wrote it) it did not affect the final result.

6.5 Running

As learning could take several hours, or even days, the software was run on an external quad-core server. Maven was used to package the main project, with all its dependencies in to a jar file, which allowed it to easily transferred and run. As the project went through several revisions and performed many learning runs, shell scripts were written to facilitate this process. The push script called Maven to package the project, running all tests, which was then copied to the server (using ssh), along with parameter files and their runner scripts. These runner scripts were then run remotely to begin the learning process. On completion another script retrieved the generation log and the agent files.

Immutability and thread safety were a major focus of the project. This was done to allow the evaluation and breeding stages to be run over four threads (to take advantage of the server's quad-core). However, during the initial run it was quickly discovered that multithreading the evaluation process was impossible. Even though each thread had its own evaluation

Level	Difficulty	Enemies	Type	Length	Time	Other
1	2	N	1	200	100	
2	3	N	0	200	100	
3	5	N	2	200	100	
4	10	N	0	200	100	
5	2	N	0	200	100	Flat Level
6	7	N	0	200	100	Flat Level
7	2	Y	0	200	100	Frozen Enemies
8	2	Y	0	200	100	
9	3	Y	1	200	100	Tubes
10	5	Y	0	200	100	Cannons, no Blocks

Table 4: Level playing parameters of the LEMMEL learning run. Type 0 is outside, and type 1 and 2 are inside.

Statistic	Multiplier
Distance	1
Completion	3200
Mario Mode	200
Enemy Kills	100
Time Left	2

Table 5: Evaluation multipliers of the LEMMEL learning run.

Gene Parameter	Probability
Condition Mutation	0.05
Condition Favour	0.5
MarioMode Favour	0.95
EnemyLeft Favour	0.9
Action Mutation	0.09

Table 6: Mutation parameters of the LEMMEL learning run.

task instance, many of the game engine's assets were held statically and were mutable. Moreover, the MarioEnvironment was implemented as a singleton. This meant only one task could be performed at a time, resulting in the evaluation stage being set to run in only one thread. Despite this setback, learning runs took approximately 6 hours to complete.

6.6 Parameters and Revisions

Many learning runs were performed during the project, including numerous parameter revisions. For brevity, only the most significants changes will be presented.

Initial runs used parameter files that closely followed the approach used by the REALM agent team. Conditions favoured the DONT_CARE byte with a probability of 0.4, whereas actions were reset mutated. Every gene had a mutation probability of 0.2. Agents were evaluated over 10 levels in a similar style to REALM's method. The levels varied over a wide range of difficulties, but favoured easier levels. Enemies were enabled for all but 2 levels. The evaluation multipliers rewarded distance, level completion, Mario's final mode, kills and time left upon completion.

The agents produced with these parameters did not display any interesting behaviours and functioned similarly to the handcrafted Forward Jumping agent (4.5). the first revision aimed to address this.

Mutation probability of both conditions and actions was lowered and the DONT_CARE condition favour probability was increased to 0.5. The biggest change came with the evaluation task. Agents now played a narrower band of difficulties, generally more difficult than the first set. Enemies were only enabled for the last 4 levels and pits were enabled for all. These changes were made as they suited a reactive agent over the blind forward jumping approach. Lastly, the level completion was given more weight.

Agents evolved with these parameter demonstrated much more compelling behaviours, as well as higher overall attainment. However, it was observed that much of their ruleset's were unused.

A second revision was made with an aim to increasing the number of rules used and to reduce the search space. Favour probability for genes corresponding to conditions on MarioMode and EnemyLeft were greatly increased, making DONT_CARE likely in these conditions on all rules. For clarity, the learning run performed using these parameters will be referred to as the **LEMMEL** run (for **LE**ss **M**ario**M**ode and **E**nemy**L**eft)

The agents produced with this approach are evaluated in depth on Sections 7.2 and 7.3. Tables 4, 5 and 6 summarise the LEMMEL parameters, which can be found in full, in ECJ's file format, in Appendix E.

7 Results

7.1 Learning Data

Statistical output from each generation of the LEMMEL run (the most significant and successful learning run, described in Section 6.6) was collated and organised. The information is presented in graphical form in Figure 11. It includes the best (red) and average (blue) fitness for each generation, with polynomial trend lines. The **complex** handcrafted agent (see Section 4.5) was also passed into the evaluation task of each generation. Its fitness is also included on the graph in green.

7.1.1 Improvement over Time

The graph indicates that both best and average fitness of the learning agents improved over time. Average fitness increased gently over the first 150 generations, from below 10,000 to around 25,000, after which it levelled off, with some fluctuations. Best fitness increase rapidly over the first 150 generations, from around 15,000 up to just below 60,000. It then rose more gently until generation 500, reaching approximately the 62,000 mark. There was a slight dip in best fitness after generation 850, which is likely a result of high variance, which is explained in the the two proceeded sections.

The stagnation in improvement can be explained, in part, by the best individuals saturating the evaluation task. Scores above 60,000 are typical of very capable agents, who have completed 8 or 9 out of the 10 task levels. Scores over 70,000 represent a near perfect execution of the levels, completing them all in good time without many enemy collisions. We can see that, after generation 400, many populations contained individuals scoring in excess of 60,000, thus reducing the amount with which they could improve.

This is also reiterated by comparing the learnt agents' fitness with that of the complex agent. The trend line shows that the best performing agent of a generation often outperformed the handcrafted agent. Further analysis comparing the handcrafted agents with the those developed in the LEMMEL run is discussed in Section 7.2.

7.1.2 Generation Variance

A feature very evident in the LEMMEL graph is the amount of fitness variance between successive generations, which can be seen in both the best and average fitness. Analysis of the polynomial regressions reveals the standard error of regression for these datasets is 5,612 and 3,287 respectively. These figures are high for an optimisation algorithm, especially given that truncation selection usually helps to reduce deviation [46, s. 3.8.3]. Moreover, as

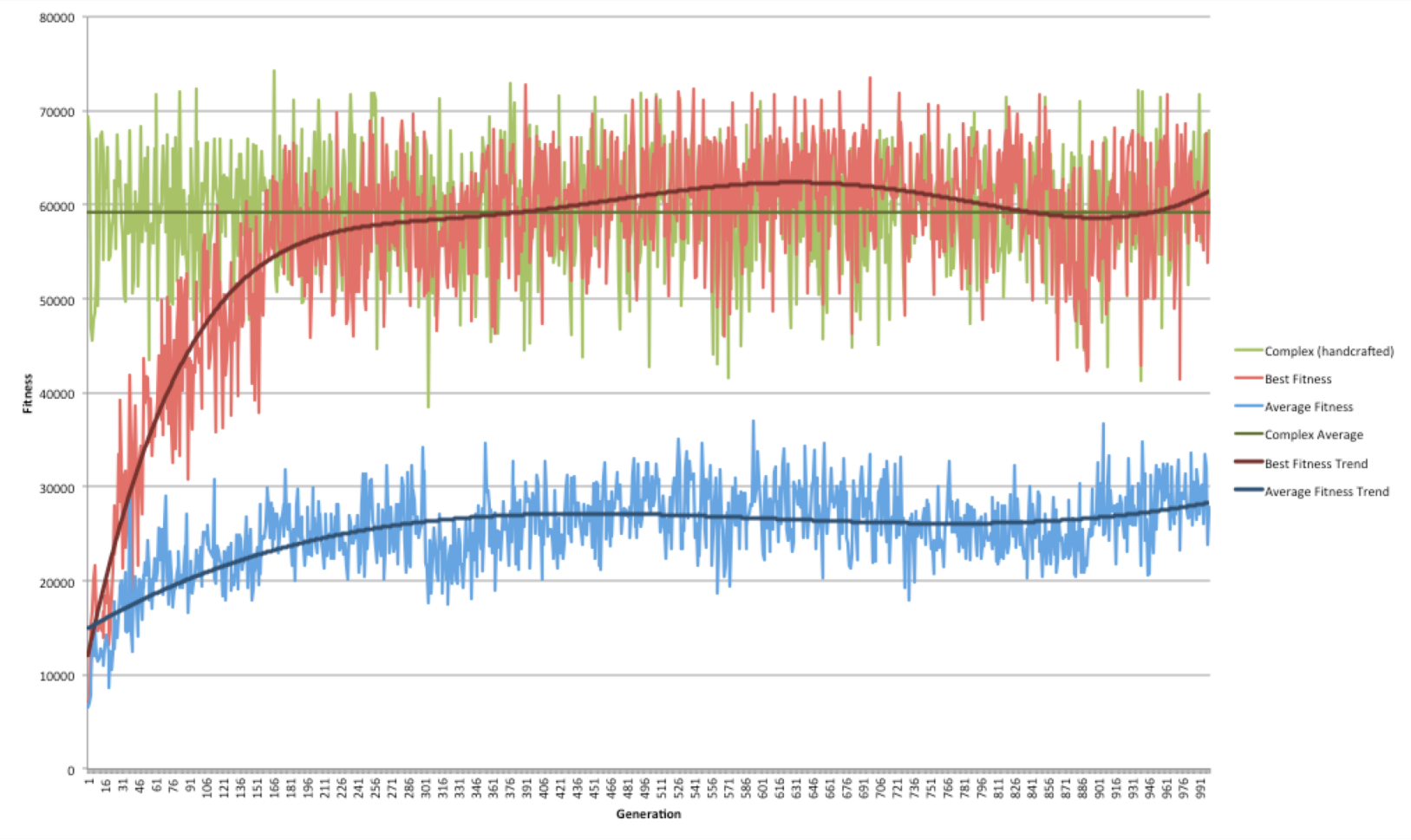


Figure 11: Graph charting the best and average fitness over the LEMMEL learning run. Handcrafted ***complex*** *agent fitness also included.*

Run	Average Fitness	Standard Deviation
LEMMEL	27,740	18,850
Fixed Seed	29,601	25,293

Table 7: Deviation over the fitness of the final population of the learning runs, demonstrating high population variance.

a $(\mu + \lambda)$ evolutionary strategy is being used (i.e. the best performing individuals are carried over to the next generation), one would not expect the best fitness to decrease often. However, the data shows this occurs regularly.

This trait is due to the use of different level seed per generation, which means that each generation has a slightly different evaluation task. It was unexpected that the level seed would affect task difficulty (and therefore best fitness) as drastically as the data suggests.

To investigate this further, and to ascertain if it had any effect on the capability of the learn agent(s), a supplementary learning run was performed. This run had a single level seed, which was used in every generation, thus every generation had an identical evaluation task. For clarity, this run will be referred to as the **fixed-seed** run. The data from the fixed-seed run is charted in Figure 12.

As expected, the best fitness in this run only ever increased[3]. Moreoever, the deviation in average fitness decreased, with a standard error of regression of 2,817. Furthermore, the highest fitness achieved was higher, which may mean either it is more capable than those evolved in the LEMMEL run or simply became very suited to the fixed set of levels. To determine this the agent will be included in the learnt agent analysis in Section 7.2, alongside those from the LEMMEL run.

7.1.3 Population Variance

Both the LEMMEL graph and the fixed seed graph show a large difference between best and average fitness. this suggests a high deviation over fitness within each population. This is confirmed by analysing the final generation of both runs, for which the full population statistics are available. The average fitness and standard deviation are shown in Table 7.

As each generation is constructed from the best individuals of the previous generation, such population variance shows that the ruleset genomes are highly sensitive to mutation. For example, we can consider the final population from the fixed seed run. The truncation selection of generation 999

[3]Except for after generation 882. This fitness evaluation is erroneous and an example of the occasional non-deterministic nature of the benchmark game-engine. Attempts to repeat the value (using that generations best agent) were unsuccessful and instead produced a value that was lower than the previous generations best fitness.

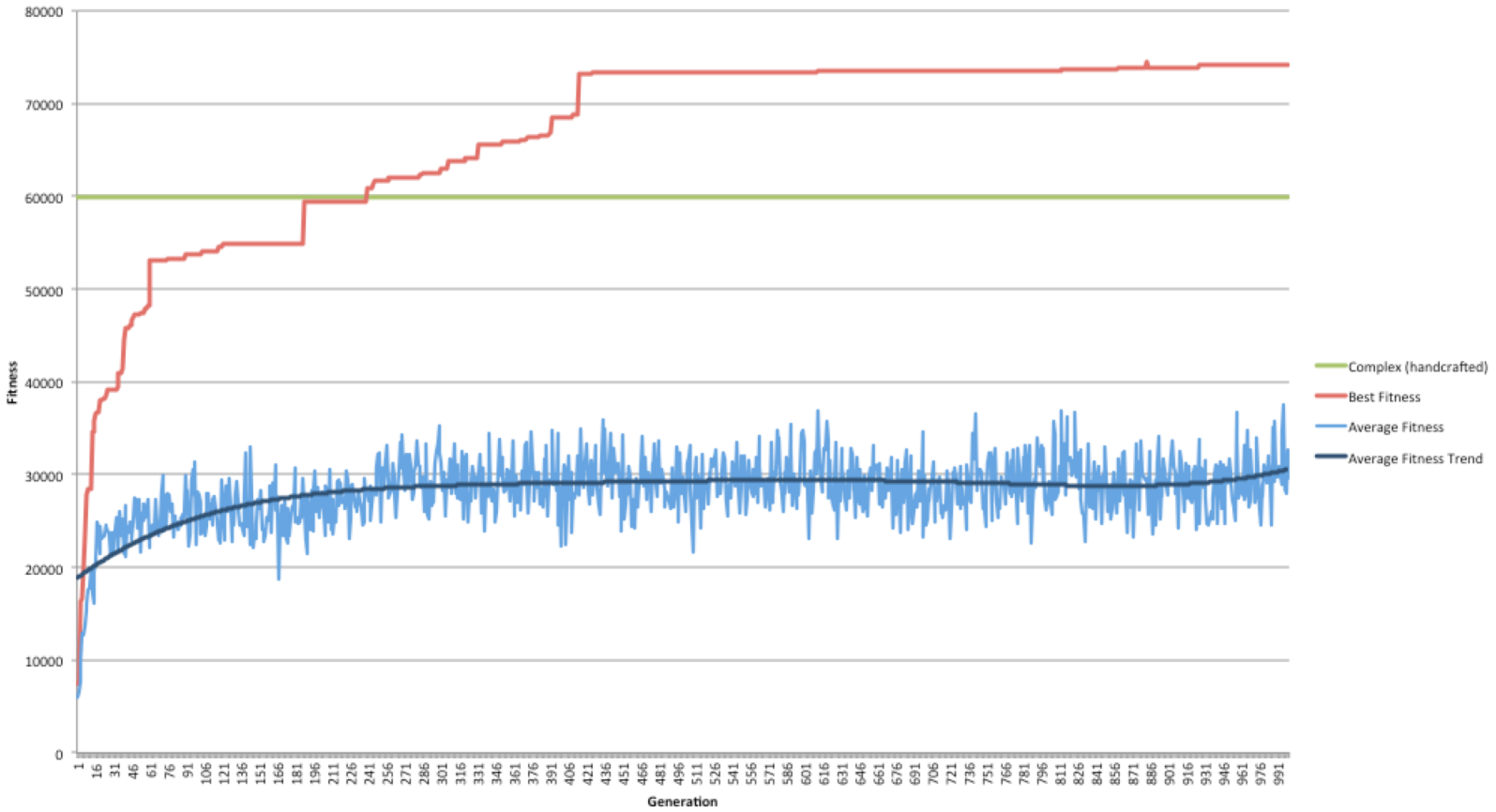


Figure 12: A learning run which had a fixed level seed, performed in response to high generational variation in the LEMMEL run.

restricted the parent pool to 5 individuals, each with a fitness of over 74,000. The generation 1000 population was made up entirely of their mutations, but still contained 23 (46%) individuals whose fitness was under 10,000.

More specifically, the best individual of generation 999 contained the following rule (a key for which can be found in Appendix A):

#	Conditions				Actions			
	ELR	OA	PB	MY	Left	Right	Jump	Speed
15	0	0	0	1	F	F	F	F

This rule states that when there are no enemies or obstacles in front and no pit below, Mario should do nothing. The was rarely used due to the condition that Mario must also be moving down: **MY** (MovingY) = 1. In one individual, a single gene mutation removed the condition on **MY**, this caused the rule to activate constantly at the beginning of each level, leaving Mario frozen. This caused the fitness to drop from 74,194 to 4,687, which shows that a single mutation can have a devastating affect on ruleset fitness.

7.2 Handcrafted vs. Learnt Agents

As explained in Section 6.3.5, the learning process was configured to extract three rulesets to be considered for evaluation. After generation 800, the algorithm saves the agent with the best overall fitness and the agent with the largest fitness compared to the average. The two such agents from the LEMMEL run (alongside the final agent from the fixed seed run) are compared to the handcrafted agents in this section. For reference, they will be named **learnt best**, **learnt difference** and **learnt fixed seed** agents respectively, and called the learnt agents collectively.

7.2.1 Learning Evaluation Task

The three learnt agents and three handcrafted agents were run through the each generation's evaluation task of the LEMMEL run. The agents perfrom 1000 tasks, with the options and level seed from LEMMEL parameter file (full details can be found in Section 6.6. Results are compiled in to Table8. We can see from the data that both the leant best and learnt difference agents performed at the same level as the complex agent and considerably better than the other handcrafted agents. This implies they became very proficient at the task they learnt over. This is especially significant as the LEMMEL evaluation task was designed to favour the complex and simple reactive agents.

We can also see that the learnt fixed seed agent did not perform quite as well, suggesting it became too suited to the single set of levels it was evolved over. The large difference between each agent's best and worst fitness reiterates that the use of different level seeds may be overly affecting

Agent	Average	Offset	Best	Worst
Complex	59,206	0	74,286	38,494
Simple Reactive	37,310	21,896	61,784	19,526
Forward Jumping	20,458	38,748	45,022	8,378
Learnt Difference	58,930	276	72,672	36,440
Learnt Best	59,079	127	73,058	43,740
Learnt Fixed Seed	54,115	5,091	74,194	38,494

Table 8: Statistics from handcrafted and learnt agents playing each generation's evaluation task from the LEMMEL run.

Agent	Total Score	Levels Completed	Enemies Killed	Distance
Complex	1,758,931	168 (33%)	1,440 (8%)	62,231 (45%)
Simple Reactive	1,102,733	87 (17%)	588 (3%)	44,038 (32%)
Forward Jumping	954,141	65 (13%)	743 (4%)	38,484 (28%)
Learnt Difference	1,521,200	144 (28%)	1,130 (6%)	55,710 (40%)
Learnt Best	1,510,697	133 (26%)	1,603 (8%)	54,341 (39%)
Learnt Fixed Seed	1,267,914	102 (20%)	1,173 (6%)	48,131 (35%)

Table 9: Competitive statistics from handcrafted and learnt agents playing the comparator task with a seed of 1000.

difficulty. However, as the learnt fixed seed agent did not score as highly as the other learnt agents, it suggests that although the LEMMEL run had high generational variance, it produced more capable agents.

7.2.2 Comparator Task

The six agents were also passed into the more extensive comparator task (described in Section 5.5) with a seed of 1000. Each agents results are presented in Table 9. We can see from the data that both the learnt best and learnt difference agents performed better than the two simpler handcrafted agents, but not as well as the complex agent. Considering the evaluation task result, this shows that the learning evaluation task may not adequately represent the comparator task and the game more generally.The data also shows that the learnt fixed seed agent did not perform quite as well, suggesting that agents that learn from many sets of levels are more adept at tackling a greater variety of challenges.

Figure 13 charts how the agents performed at different difficulties in the comparator task. None of the agents performed well on difficulties over 10. Further analysis showed that they all failed to clear the large jumps which are common to these levels, demonstrating a weakness in the agent framework. Elsewhere, the complex agent outperformed the learnt agents on

#	Conditions											Actions			
	MM	JA	OG	EL	EUR	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
1			1		1	1		0	1	2	0			T	T
2		0								2	2		T	T	T
3		1			1		1		0		0			T	T
4		0	0			0	1	0	1	0	1	T	T	T	T
5					0	0	1		0	0	0			T	
6		0			1	1	1		0	0				T	T
7		1	0			0		0	0		2				
8				0	0	0	0		1	2	1		T	T	T
9									0	0	2			T	
10		0	1				1	1		0					
11	0		1	1	0	1		1	0		1		T		T
12					1			2							T
13							0		0	2	1	T		T	T
14		1	0	1	0	1	1	0		0		T	T	T	T
15		1							1	0	2	T	T	T	T
16		1	1	0				1				T	T	T	
17		1	1		0				1		0		T	T	T
18		1	1			1		0						T	
19		1			1			2	0		0	T			T
20			0			0	0		0	0	0	T	T	T	T
-													T		T

Table 10: Ruleset for learnt difference agent. Blank entries denote DONT_CARE *for Conditions and* ***false*** *for Actions. A key for this table can be found in Appendix A*

nearly all difficulties. However, both the learnt best and the learnt difference agents outperformed the complex agent on difficulty 2. This was the most prevalent difficulty in the LEMMEL evaluation task and suggests that a longer or more representative task may produce a more capable agent. We can also see that the disparity between the learnt fixed seed agent and its peers come mainly from the lower difficulty levels. At these difficulties, the other learnt agents take greater advantage of the reduced presence of enemies and pits. This is especially true of difficulty 0, where there are no pits and very simple enemies, in which the fixed seed agent was outperformed by all the other agents.

7.3 Learnt Agent Analysis

Of all the learnt agents, the learnt difference agent performs the best on the comparator task. It also demonstrates the most interesting behaviour, which is analysed in this section. A full ruleset for this agent can be found in Table 10, a key for the table can be found in Appendix A.

7.3.1 Rule usage

The agent only regularly uses 7 of its 20 rules (35%), and relies heavily on the default action. On one hand, some of its rules contain conditions that are too specific, for example, Rule 11, which is never used. On the other,

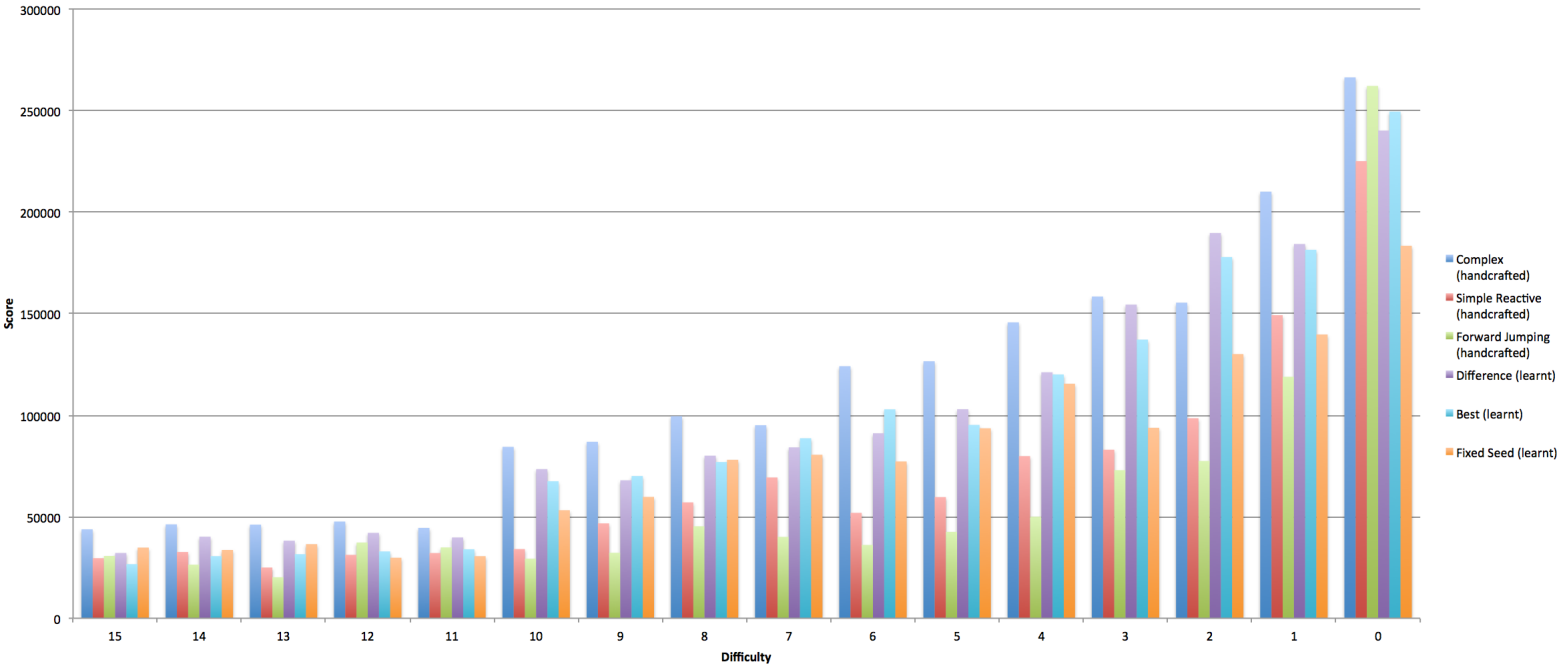


Figure 13: Graph charting the score achieved on different difficulty levels during the comparator task.

some rules are not specific enough and are eclipsed by others, for example, Rule 12, which is often forgone in favour of Rule 19. Rather than mutation or favour rate (see Section 6.1) being too high or too low, this suggests that the design of the genome mutation itself is too simplistic. Methods of addressing this are discussed in the Section 8.3.

7.3.2 Behaviours

The learnt difference agent acts similarly to the complex and simple reactive agents; it attempts to jump over obstacles, pits and enemies when it detects them and runs at speed otherwise. It struggles with the largest pits and avoiding collisions when enemy concentration is high. Moreover, much like the complex agent, it occasionally gets stuck in an action loop and runs out of time.

Four significant and interesting behaviours, and the rules that enable them, are presented subsequently.

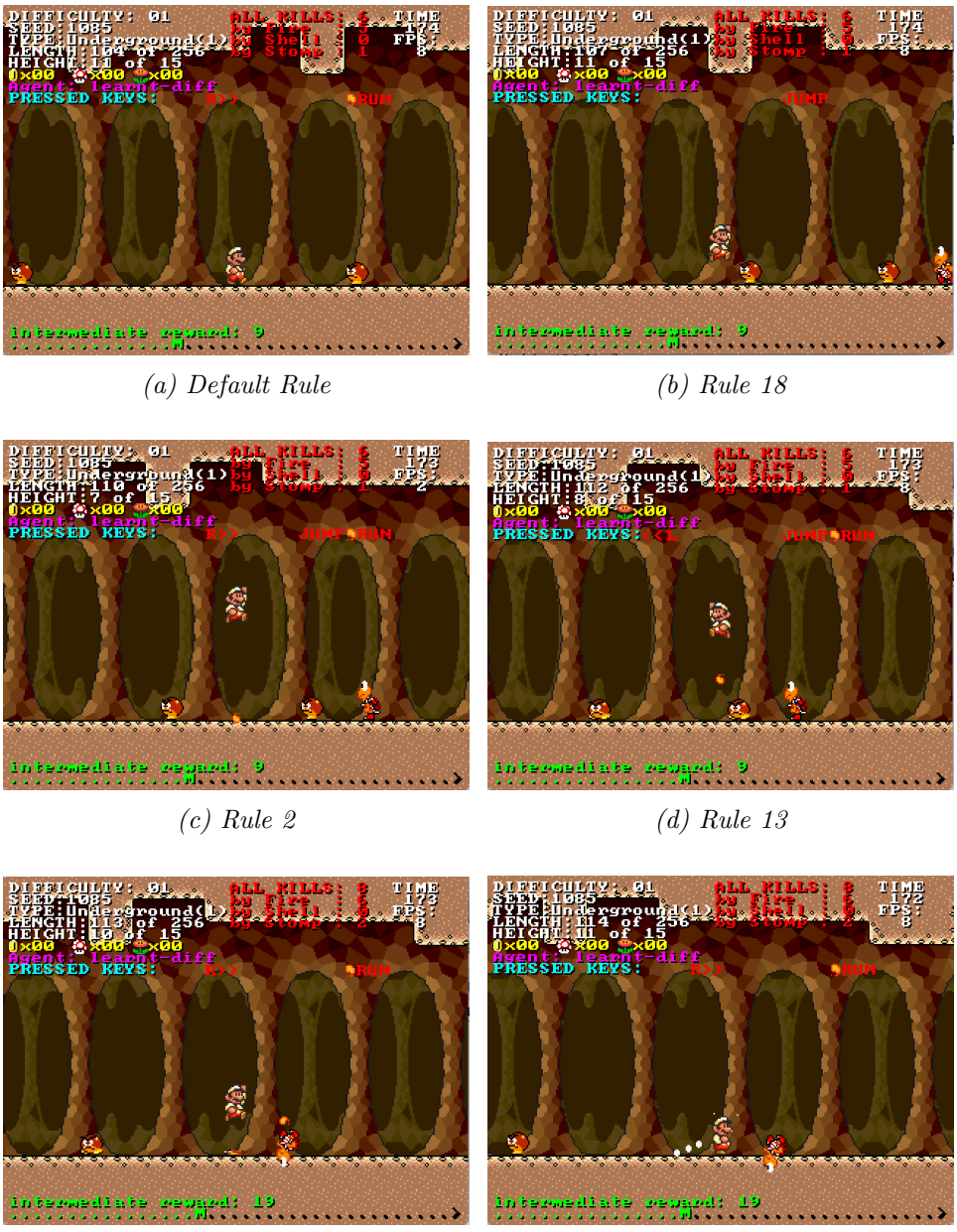
Enemies

#	**Conditions**								**Actions**			
	JA	OG	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
2	0						2	2	F	T	T	T
13				0		0	2	1	T	F	T	T
18	1	1	1		0				F	F	T	F
-									F	T	F	T

To avoid colliding with enemies the agent uses Rules 2, 13 and 18, together with the the default action. These rules are pictured above with the final entry denoting the default action/rule. A key for these rules can be found in Appendix A.

Figure 14 illustrates this behaviour. Mario approaches the enemy using the default action. On detecting the enemy (**ELR** (EnemyLowerRight) = 1) Rule 18 activates and Mario jumps. Mario is now travelling up and right (**MX** (MovingX) = 2, **MY** (MovingY) = 2), thus Rule 2 activates to continue the jump and move him quickly to the right. Rule 2 also has the side effect of shooting a fireball if Mario is in **fire** mode. Rule 13 activates as Mario starts to descend (**MY** = 1), turning Mario left and slowing the jump. This allows the fireball to beat Mario to the landing zone, killing any enemies that might pose an immediate threat.

Enemy avoidance is the weakness of several of the learnt agents. The difference agent's approach of jumping high over them and shooting a fireball is one of the most effective. Using Rule 13 to slow the jump once the enemy is cleared allows Mario to avoid jumping into subsequent enemies or pits.



(a) Default Rule *(b) Rule 18*

(c) Rule 2 *(d) Rule 13*

(e) Default Rule *(f) Default Rule*

Figure 14: Learnt Difference agent tackling enemies.

Obstacles

#	Conditions						Actions			
	EUR	ELR	OA	PB	MX	MY	Left	Right	Jump	Speed
5	0	0	1	0	0	0	F	F	T	F
9				0	0	2	F	F	T	F
13			0	0	2	1	T	F	T	T
-							F	T	F	T

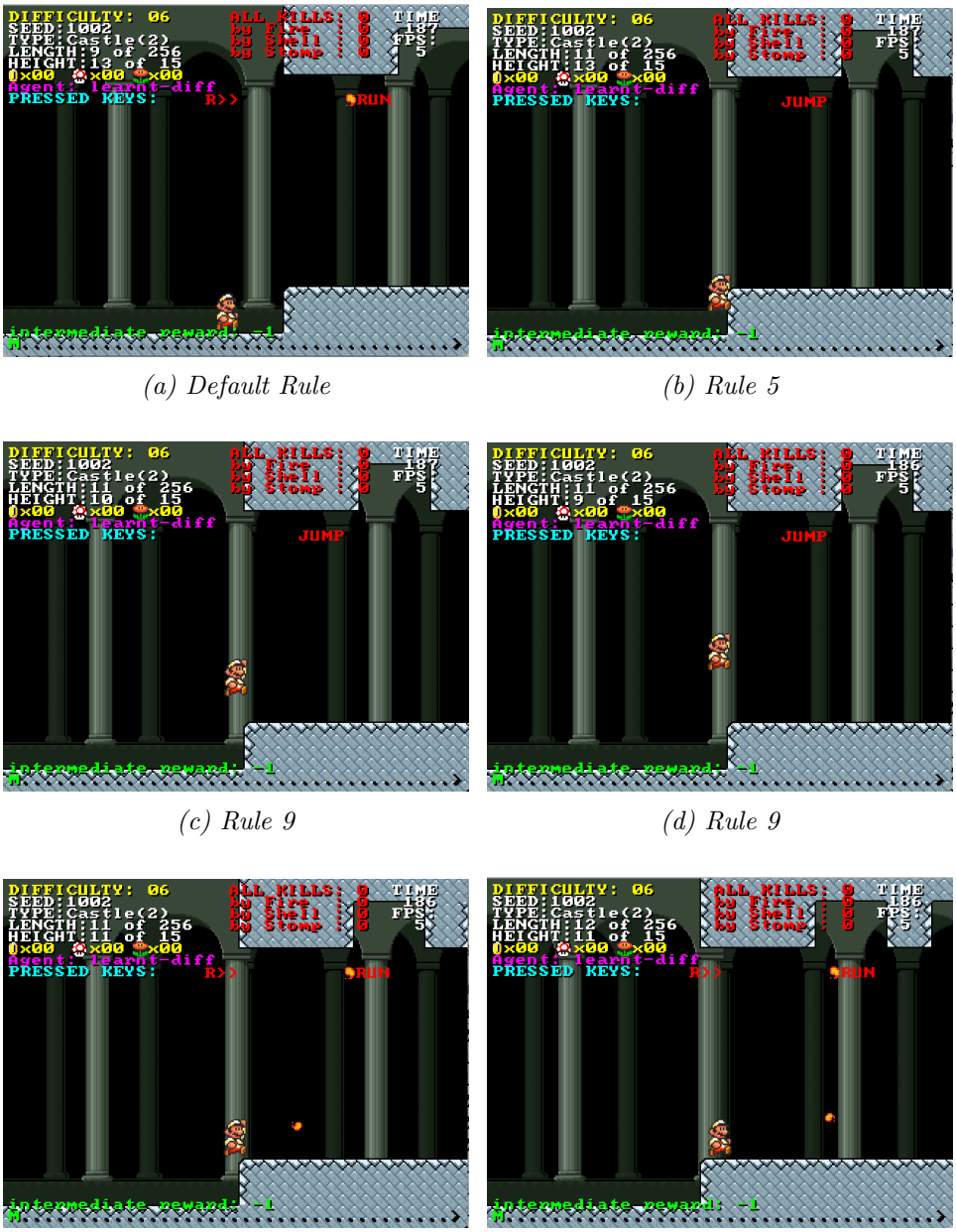
In order to overcome obstacles, such as level terrain and blocks, the agent uses Rules 5 and 9, in conjunciton with the default action.

The process is demonstrated in Figure 15. The agent approaches the wall using the default action. On hitting the obstruction, Mario stops moving (**MX** (MovingX) = 0, **MY** (MovingY) = 0) and detects the obstacle (**OA** (ObstacleAhead) = 1). This activates Rule 5, causing Mario to jump. As Mario is now moving straight upwards (**MX** = 0, **MY** = 2) Rule 9 is activate and ensures Mario continues the jump to its maximum height. When descending, Mario alternates between using Rule 13 and the default action. This results in Mario moving slightly to the right and landing on the obstacle, successfully clearing it.

The behaviour is particularly interesting due to the condition on Mario being stationary before jumping. This allows Mario to clear obstacles without over-jumping. Over-jumping could cause a him to fall into a proceeding pit, especially when climbing 'stairs' (see Figure 16).

This is an example of a behaviour not considered during the construction of the handcrafted agents. The complex agent handles the issue of over-jumping by detecting potential problems and moving left.

Many evolved rulesets, through several of the learning runs, demonstrate this behaviour, confirming its benefit to the agent. Furthermore, during the LEMMEL run, it is one of the earliest to develop, appearing first at around generation 50.



(a) Default Rule *(b) Rule 5*

(c) Rule 9 *(d) Rule 9*

(e) Default Rule *(f) Default Rule*

Figure 15: Learnt Difference agent tackling obstacles, without over-jumping.

Pits

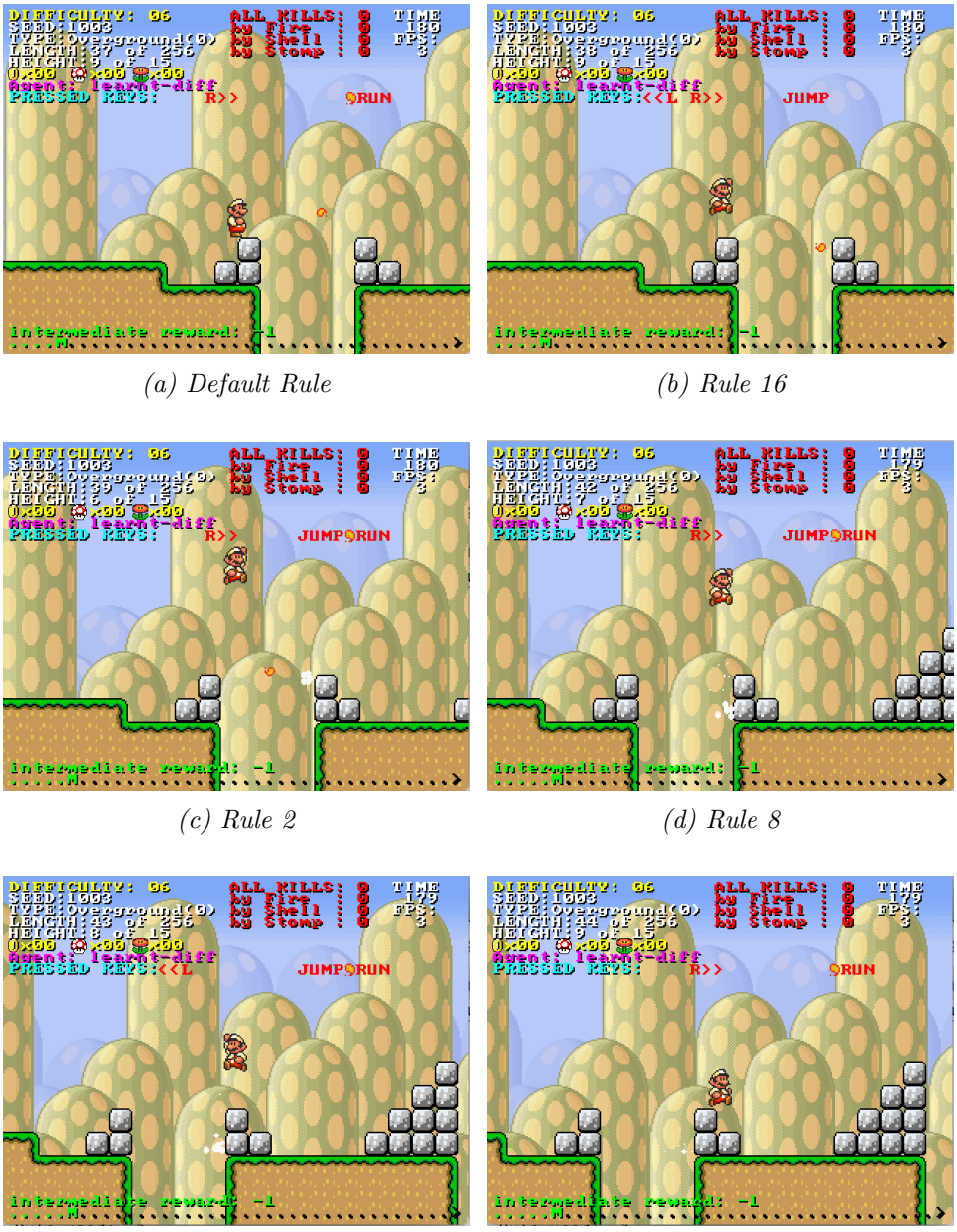
#	Conditions										Actions			
	JA	OG	EL	EUR	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
2	0								2	2	F	T	T	T
8			0	0	0	0		1	2	1	F	T	T	T
13						0		0	2	1	T	F	T	T
16	1	1	0				1				T	T	T	F
-											F	T	F	T

To jump over pits Mario uses Rules 2, 8, 13 and 16, along with the default action.

Mario can be seen jumping over a pit with 'stairs' in Figure 16. If the pit does not have stairs then Mario approaches using the default action, otherwise, he climbs them one by one using his obstacle behaviour. As soon as he lands on the top step (**JA** (JumpAvailable) = 1, **OG** (OnGround) =1) he detects that the pit is close (**PA** (PitAhead) = 1). This activates Rule 16, causing Mario to jump. Rule 2 then activates to maximise the jump (as we saw with enemies). When Mario reaches the top of the jump and begins to descend (**MY** (MovingY) = 1) Rule 2 deactivates. If Mario is still above the pit (**PB** (PitBelow) = 1) Rule 8 is used to ensure Mario reaches as far as possible. As soon as Mario is no longer over the pit (**PB** = 0) Rule 13 is used to slow the jump and avoid over-jumping. If Rule 13 is used to the point that Mario stops moving right (**MX** (MovingX) = 0 or 1) the default action is used instead, which helps Mario avoid turning back into the pit. Often, this means that Mario lands the very edge of the pit, at which point he uses the default action to continue.

Pits are the most dangerous part of any level, as they can cause an instant loss. Hence, clearing them is vitally important for the capability of an agent. The learnt agent's strategy of always maximising the jump whilst it is above a pit means that it clears the majority of them. Furthermore, its ability to land on the edge of the right-hand side of the pit allows Mario to successfully avoid over-jumping and tackle the next hurdle.

Every learnt agent displays some level of this behaviour. However, most of them are negatively affected by the proximity of enemies, whereas the learnt agent's pit strategy is less concerned with their presence. This is important as a pit can cause instant failure, whereas an enemy collision is only fatal when in **small** mode. In fact, the learnt agent also displays an additional positive behaviour in regards to the combination of enemies and pits, which is explained next.



(a) Default Rule *(b) Rule 16*

(c) Rule 2 *(d) Rule 8*

(e) Rule 13 *(f) Default Rule*

Figure 16: Learnt Difference agent tackling pits, without over-jumping.

Pits with Enemies

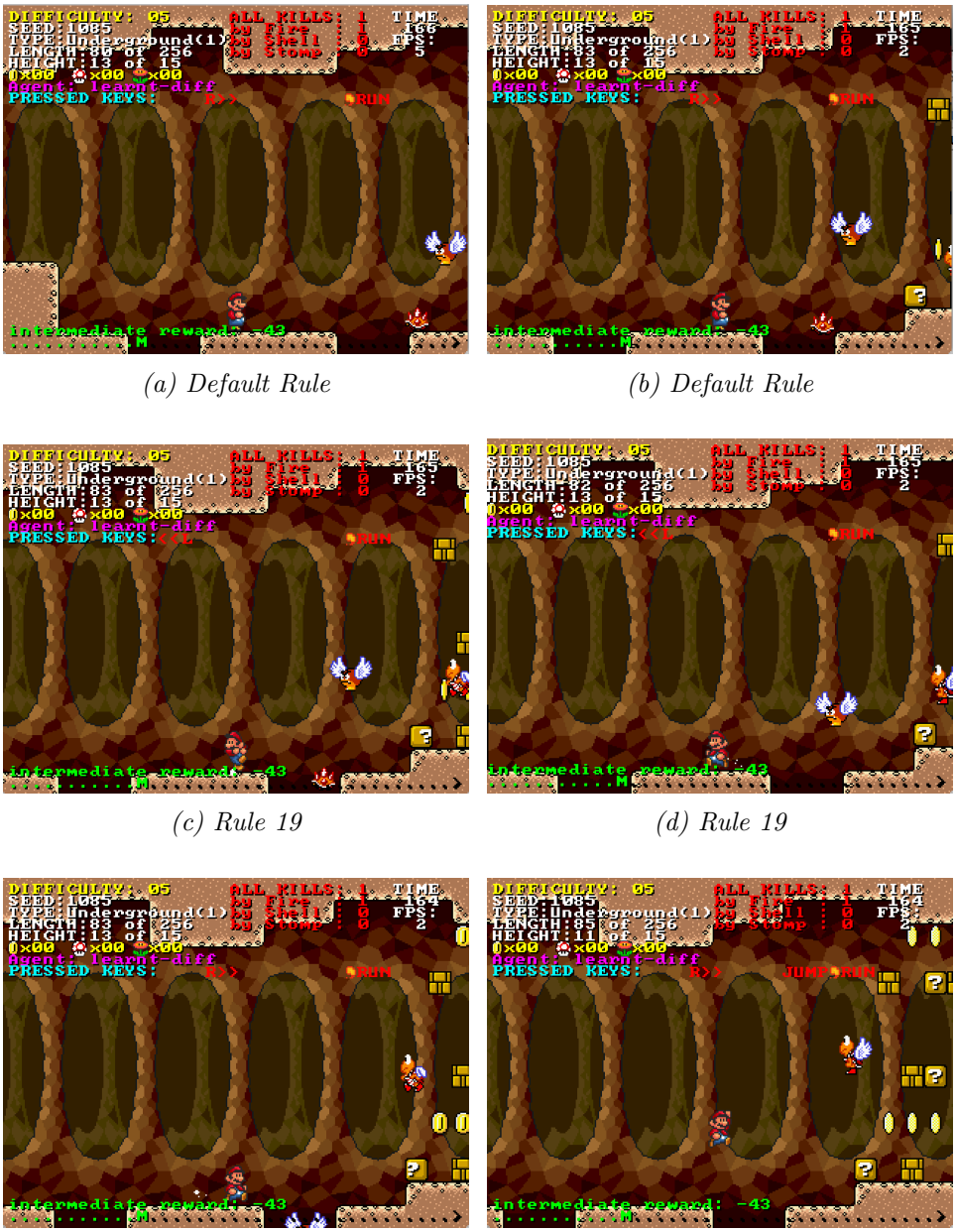
#	Conditions							Actions			
	JA	OG	EL	EUR	PA	PB	MY	Left	Right	Jump	Speed
16	1	1	0		1			T	T	T	F
19	1			1	2	0	0	T	F	F	T
-								F	T	F	T

In situations when the path over a pit is blocked by an enemy, the learnt difference agent displays an advanced strategy. It delays jumping over the pit until the enemy had moved away using Rule 19.

Figure 17 demonstrates the strategy. As Mario approaches a pit he first detects that one is **far** (**PA** (PitAhead) = 2), as he gets closer he detects it as **close** (**PA** = 1). If there is an enemy to his upper right (**EUR** (EnemyUpperRight) = 1) at the point he perceives the pit as **far** Rule 19 activates, which stops Mario a causes him to move left. The use of the **speed** action here, allows Mario to turn quick enough to avoid perceiving the pit as **close** and activating Rule 16. Turning and moving away from the pit allows the threat to either move away or fall into the pit (as can be seen in Figure 17). At which point Rule 19 deactivates a Mario once again approaches the pit. If no more threats are detected, he uses the pit strategy to clear the pit. The other conditions on Rule 19 also serve an important purpose: it will not occur if Mario has already started to jump over the pit. Turning left whilst already trying to clear a pit would more than likely result in Mario falling and failing the level.

The use of Rule 19 allows Mario to avoid collisions with enemies, maintaining the **fire** mode, allowing him to use fireballs. As we saw in his enemy strategy, fireballs are an important tool for clearing enemies, increasing the chances of completing a level.

This behaviour is unique to the learnt difference agent and is part of what makes it the most interesting a capable agent.



(a) Default Rule *(b) Default Rule*

(c) Rule 19 *(d) Rule 19*

(e) Default Rule *(f) Rule 16*

Figure 17: Learnt Difference agent tackling pits where enemies are blocking the jump.

8 Project Evaluation

With the results presented above, we can now critically consider the project in terms of the requirements set out in the project's aim, objectives (Section 1.3) and specification (Section 3).

8.1 Agent Framework

The primary objective of the agent framework was to enable and encourage meaningful improvement during learning. The focus on a rule based system allowed for agents to be represented with a simple encoding, which afforded a great deal of freedom to the learning module. It reduced development time as several features of the ECJ library could be used as is and any extensions did not have to process a complex data structure. Moreover, it allowed for greater control, which was crucial in honing and revising the learning parameters. Ultimately, we can see from the results that, given the right parameters, the agent framework did enable significant improvement over generation, with later agents achieving near maximum fitness.

However, the results show a high fitness deviation over each generation's population. This is likely due to the inherent sensitivity to mutation of a rule-based approach. One gene mutation can transform a once useful rule into one that is counter productive, with a great affect to the agent's capability.

In addition to the project's aims and objective, three criteria were laid out for the agent's use of a rule based system in Section 3: speed, capability and complexity. The framework design relies heavily on how the agent senses the environment; as such the selection of perceptions is a major factor in its evaluation.

The **complex** handcrafted ruleset shows that the framework's choice of perceptions allows for capable agents. It rarely fails to finish the easier levels and can successfully tackle the more difficult ones. Furthermore, the depth of the perceptions and the available actions allows for competent behaviour to be demonstrated by relatively small rulesets, which improves the agent's speed. In fact, agent response time never rises above a millisecond. This allows for levels to be run at practically any number of frames per second and has little effect on total learning time.

On the other hand, the framework is still limited. It is impossible for any ruleset agent to break itself out of a loop or to navigate a dead end, as they have no memory. Similarly, it can never develop a truly effective enemy killing behaviour without differentiating between types.

However, as discussed in Section 4.1, improving capability by embellishing the agent's perceptions comes at the penalty of complexity. A more complex design lends to a larger search space for the learning algorithm, which hinders the development of a successful agent through random mu-

tation. This was also a crucial consideration of the REALM team, whose V2 agent was designed, in part, to reduce the search space of the VI agent [36][p. 86].

In its current form, the agent framework balances these factors reasonably well. Our results show that a successful and interesting agent can be evolved despite the total number of possible rules

$$4^4 * 3^7 * 2^4 = 8,957,952$$

being higher than that of REALM's V1 agent $(7,558,272)$ [36][p. 86]. Thus, improvements to the capability of the agent should not increase the search space, but instead focus on making sure each perception is purposeful.

In the LEMMEL run parameters (and in the handcrafted agents) use of the MarioMode and EnemyLeft perceptions was discouraged. Our learning parameters allowed us to reduce the search space by increasing the probability of the DONT_CARE condition. This is clearly a counterintuitive approach and the agent design should be amended by removing these perceptions.

Such removals would allow for additional perceptions and/or design extensions without increasing agent complexity. Given more time, a redesign that allowed agent encoding control over its perceptions would be considered. Perceptions would remain fixed, but the parameters that control them would be variable. For example, the area in which an agent looks for an obstacle or enemy could be controlled by parameters on a ruleset wide basis. Moreover, perceptions that measure the last used action (a simplistic form of memory) and the identification of enemies in greater detail would also be investigated.

8.2 Level Playing

The primary motivation for the level playing module was to meet Objective 4: Provide a large and diverse test bed from which to learn. This was reiterated in the project specification (Section 3), which decrees that it must, in a parametrised manner, produce a variety of levels and ways to score an agent.

The former was achieved by providing access to the benchmark's level generation options through simple data classes. Additionally, as detailed in Section 5.3, modifications were made to the LevelGeneration class to provide greater diversity and to ensure parameters were being treated correctly. This was an unexpected workload, and as such could not be afforded ample attention. Without time constraints, further modifications would have been made, including improved scaling of enemy types and better block placement. Furthermore, these and previous changes, as well as other level generation features, would have been made parametrisable, allowing the MWLevelOptions class more power and precision.

The latter was achieved by the MWEvaluationMultipliers class, which parametrised a linear function over level playing statistics. The options mirrored the statistics produced by the benchmark software. Whilst this was effective in calculating agent fitness in the learning module, additional statistics would be beneficial. For example, it is currently impossible to punish an agent for the manner in which it failed (i.e. by falling into a pit, by colliding with an enemy, or by running out of time). If the project was to continue, the reliance of the benchmark software for these statistics would be reduced and a specialised EvaluationInfo class would be implemented, allowing for the level playing statistics to be extended.

Specifying and persisting these two elements was accomplished by extending ECJ's parameter file system. Adopting this existing system saved time and ensured it integrated well with the learning module. With the high number of revisions and the emergent importance of the evaluation task during the project, this decision proved vital. Whilst it provided great control when using a small number of levels, it was too verbose to be used for larger level tasks. For example, the comparator task (Section 5.5 and Appendix D) had to be hard-coded rather than specified in a level options file. With more time, the vocabulary of the parameter persistence would be extended to include controls for segments of episodes and values as a function of episode number (as seen in the comparator task).

The results demonstrated that the learning runs featured a high level of inter-generational variance. This is to be expected, especially of the average fitness, due partly to the agent framework's sensitivity to mutation. However, the variance is also displayed by the handcrafted agent, which suggests that it may in fact be caused by the difficulty of the evaluation task fluctuating between generations. As the only task parameter that changes between generations, the choice of level seed may be having too much of an effect on level generation. The next subsection will discuss ways tin which this issue can be tackled within the parameters of the learning algorithm. However, if given additional time, the influence of the level seed on level difficulty would be investigated.

Aside from level generation, the benchmark software also raised several other barriers to the project. Firstly, there was little to no documentation, which slowed the initial progress of the project. Secondly, it greatly hindered the testability of the project as a whole (as discussed in Section 5.4), making it very difficult to provide significant test coverage (as prescribed by Objective 3). Lastly, it impeded the project's ability to meet the non-functional requirement on thread safety. As mentioned in Section 6.5, the game-engine is implemented as a singleton, relying heavily on static assets. A continued project would look to refactor this and allow the evaluation stage of the learning process to be multithreaded.

8.3 Learning Process

Section 6.6 described the revision of the learning parameters during the project. The results show that honing these parameters was crucial to the success of the project. Due to time constraints, parameter adjustment had to be stopped to allow for results to be gathered and analysed. Without such a restriction more parameter revisions would have been made.

Firstly, a continued attempt to reduce generational variance (in conjunction with solutions mentioned in the previous two sections) would be made. We saw that fixing a single level seed for the entire run removed variance from the best fitness (as expected) and lowered variance in the average fitness. However, it produced a less capable agent, which was overly suited to the small number of levels it played (known as overfitting). Instead a compromise would be struck, perhaps taking seeds from a fixed pool, or have them run in generational segments. In this way agents are evaluated over a wider range of levels, but not at a detriment to the learning evolutionary process.

Secondly, reducing overfitting in general would be investigated. We saw in the results that the learnt agent performed better than the complex handcrafted agent at the learning evaluation task, but did not perform as well on the comparator task. We also saw that in both the fixed and variable seed run, the learnt agent achieved near perfect score relatively early, suggesting the task could be made more difficult. In future revisions, the learning task would be run on higher difficulties, with more focus on enemies. Also, with the ability to multithread the evaluation, a more extensive task could be run, without a time penalty. Increasing the number of levels would allow the task to be more representative of the comparator task.

Even without the above revisions, we have seen that the use of an Evolutionary Strategy has proved successful. As a result, both the aims and the specification of the project were widely met. Meaningful learning was observed and the final agent demonstrated interesting behaviours, including some that were unforeseen and not considered during the creation of handcrafted rulesets. Moreover, as the use of truncation selection is known to reduce variance in learning algorithms [46, s. 3.8.3], and due to the variance induced by both the agent framework's mutation sensitivity and the level playing's seed variation, the choice of an evolution strategy turned out to be very appropriate.

However, because of the scope of the project, its implementation is quite simplistic. With more time, several additional features would be considered. Firstly, the use of rule crossover would be investigate, as seen in the REALM team's approach (described in Section 2.2.3). Secondly, utilising a Tabu list, a common approach in optimisation search algorithms [47], would be studied. Rules that caused a significant drop in fitness would be added to a blacklist and would be avoided by future mutations. Lastly, adaptive

mutation rates, a common feature of ESes [19, s. 4], would be examined. The results showed that the learnt agent only used a small percentage of their ruleset. In order to combat this, mutation rates could change during a learning run on a rule by rule basis. If a rule goes unused in the evaluation stage, then its mutation rate is increased. However, if it is used often, then its mutation rate is decreased.

The results that were gathered from the statistical output of the learning module have proved a good analysis of the learnt agent and the process in general. However, it was impossible to properly ascertain key metrics such as population variance and mutation sensitivity. With more time, the fitness and genome of each individual in the learning run would be made available, allowing for a more in depth analysis.

8.4 Methodology

Overall, the project achieved the majority of its objectives and realised most of the project specification. Its failure to attain all of its aims was centred around two aspects: testing and testability and issues with the Mario benchmark software.

Testing proved more difficult and time consuming than anticipated, due in part to the inclusion of ECJ and the benchmark (see Sections 4.4, 5.4 and 6.4). Testing was infeasible for several parts of the level playing and learning modules. This precipitated the abandoning of test driven development (Objective 2) and reduced the test coverage (Objective 3) of the entire project. If the project were to be attempted again, more time and effort would be spent making the code more testable.

The Mario benchmark software included several problems (see Section 5.3). Those that were fixed, were time-consuming and caused a schedule overrun. Whereas those that could not be addressed, lead to the project specification not being fully adhered to (e.g. multithreading the learning process as discussed in Section 6.5). More research should have gone into the source code at an earlier date, which would have helped identify these issues sooner and allow for preparations to be made.

Due to the modifications of the benchmark software, comparisons to other approaches were limited. Framing the learnt agent in terms of the Mario AI Competition would have been misleading and uninformative as it used an engine that produced much more difficult levels. However, greater effort should have gone into making these comparisons, especially to the REALM agents. Given more time, the learnt agent would have performed tests in the original engine, allowing score comparisons to the REALM agent and other competition entrants.

Despite these issues, the project met its overarching objective. It successfully created an interesting and capable game-playing agent, developed by evolutionary computation.

Appendices

A Perceptions

	0	1	2
MarioMode (MM)	small	big	fire
JumpAvailable (JA)	false	true	-
OnGround (OG)	false	true	-
EnemyLeft (EL)	false	true	-
EnemyUpperRight (EUR)	false	true	-
EnemyLowerRight (ELR)	false	true	-
ObstacleAhead (OA)	false	true	-
PitAhead (PA)	none	close	far
PitBelow (PB)	false	true	-
MovingX (MX)	none	left	right
MovingY (MY)	none	down	up

Table A1: Displays the different perceptions and what their byte value represents.

```
// Returns NONE if there are no pits ahead, FAR and CLOSE if there is one in
      a certain number of columns relative to mario.
case object PitAhead extends BytePerception(7, 2) {
    val NONE: Byte = 0; val CLOSE: Byte = 1; val FAR: Byte = 2;
    val COL_CLOSE_L = 1; val COL_CLOSE_R = 1; val COL_FAR_L = 2; val
         COL_FAR_R = 2
    def apply(environment: Environment): Byte = {
        val pitOp: Option[Int] =
          Perception.getOpens(environment, COL_CLOSE_L, COL_FAR_R).headOption
        pitOp match {
            case Some(x) if (x >= COL_CLOSE_L && x <= COL_CLOSE_R) => CLOSE
            case Some(x) if (x >= COL_FAR_L && x <= COL_FAR_R) => FAR
            case _ => NONE
       }
    }
}

def getOpens(environment: Environment, a: Int, b: Int): List[Int] = {
    val level = environment.getLevelSceneObservationZ(2);
    val test = (x: Byte) => x == 0 || x == GeneralizerLevelScene.COIN_ANIM;
    val bottomRow = level.length - 1
    val mario = getMarioPos(environment)
    val left = max(0, min(a, b) + mario._2)
    val right = min(level(0).length, max(a, b) + mario._2)
    var opens: List[Int] = left to right toList

    for {
        i <- mario._1 + 1 to bottomRow
        if (!opens.isEmpty)
    }{
        opens = opens.filter { j => level(i)(j) == 0 || level(i)(j) ==
             GeneralizerLevelScene.COIN_ANIM }
    }
    opens.map { x => x - mario._2 }
}
```

Listing A1: Pit detection in the perceptions classes.

B Handcrafted Agent Rulesets

Included below are the full rulesets for the project's handcrafted agents. Blank entries denote a DONT_CARE condition or a FALSE action.

#	Conditions											Actions			
	MM	JA	OG	EL	EUR	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
1		0	1										T		
-													T	T	T

Table B1: Ruleset for handcrafted Forward Jumping Agent.

#	Conditions											Actions			
	MM	JA	OG	EL	EUR	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
1		0	1										T		
2						1							T	T	
3							1						T	T	
4								1					T	T	T
5									1				T	T	T
-													T		

Table B2: Ruleset for handcrafted Simple Reactive Agent.

#	Conditions											Actions			
	MM	JA	OG	EL	EUR	ELR	OA	PA	PB	MX	MY	Left	Right	Jump	Speed
1									1				T	T	T
2		1	1				1	2	0				T	T	
3		0	0				1	2	0				T		
4		1	1				0	2	0				T		T
5		1	1				0	1	0	2			T	T	T
6		0	1					1	0	2			T		T
7		0	0					2	0	2		T			
8		0	0					1	0	2	1	T			T
9		0	0					1	0				T	T	T
10			0			1		0	0	2	1	T			T
11		0	1			1		0	0				T		
12						1		0	0				T	T	
13			0			1		0	0				T	T	T
14					1		1	0	0			T		T	
15		0	1				1	0	0				T		
16							1	0	0				T	T	T
17							1	0	0	2			T	T	
18			1										T		T
-													T		

Table B3: Ruleset for handcrafted Complex Agent.

C Full Level and Evaluation Options

```
/***
 * Options that control level generation
 */
class MWLevelOptions(
    val blocks: Boolean, // Blocks appear
    val cannons: Boolean, // Cannons appear
    val coins: Boolean, // Coins appear
    val deadEnds: Boolean, // Dead ends appear in terrain forcing Mario to
        turn back
    val enemies: Boolean, // Enemies/Creatures appear
    val flatLevel: Boolean, // Level is flat, no change in elevation
    val frozenCreatures: Boolean, // All creatures don't move
    val pits: Boolean, // Pits appear
    val hiddenBlocks: Boolean, // Hidden blocks appear
    val tubes: Boolean, // Tubes/Pipes appear
    val ladders: Boolean, // Ladders appear
    val levelDifficulty: Int, // Difficulty of level, effective range 0-25,
        0 easiest
    val levelLength: Int, // Length of level in blocks
    val levelType: Int, // Type of level, 0-Outside 1-Cave, 2-Castle
    val startingMarioMode: Int, // Mode Mario starts as 0-small, 1-big, 2-
        fire
    val timeLimit: Int // Number of Mario seconds allowed to complete level
)

/***
 * Multipliers for several level playing statistics.
 * Comments describe the statistic. For example, if Mario
 * completes the level It will be win * 1, otherwise it will win * 0
 */
class MWEvaluationMultipliers(
    val distance: Int, // Distance travelled by Mario in pixels (16 pixels
        to a block)
    val win: Int, // 1 for level complete, 0 otherwise
    val mode: Int, // Mario's final mode on completion or death, 2-fire, 1-
        big, 0-small
    val coins: Int, // Number of coins collected
    val flowerFire: Int, // Number fire flowers collected
    val kills: Int, // Number of enemy kills
    val killedByFire: Int, // Number of kills by fireball
    val killedByShell: Int, // Number of kills by shell
    val killedByStomp: Int, // Number of kills by stomp
    val mushroom: Int, // Number of mushrooms collected
    val timeLeft: Int, // Mario seconds left on completion, 0 if level not
        completed
    val hiddenBlock: Int, // Number of hidden blocks hit
    val greenMushroom: Int, // Number of green mushrooms collected
    val stomp: Int // Unused
)
```

Listing C1: Full field definitions for MWLevelOptions and MWEvaluation-Multipliers described in Section 5.2.1

D Comparator Task Options

```
val defaultEvaluationMultipliers = new MWEvaluationMultipliers(
              1, //Distance
              2048, //Win
              16, //Mode
              16, //Coins
              64, //FlowerFire
              58, //Mushroom
              42, //Kills
              4, //KilledByFire
              17, //KilledByShell
              12, //KilledByStomp
              8, //TimeLeft
              24, //HiddenBlock
              58, //GreenMushroom
              10) //Stomp

val compBaseOptions: MWLevelOptions = new MWLevelOptions(
              true, //blocks
              true, //cannons
              true, //coins
              false, //deadEnds
              true, //enemies
              false, //flatLevel
              false, //frozenCreatures
              true, //gaps
              false, //hiddenBlocks
              false, //tubes
              false, //ladders
              0, //levelDifficulty
              256, //levelLength
              0, //levelType
              2, //startingMarioMode
              200) //timeLimit

def compUpdate(levelSeed: Int): (Int, MWLevelOptions) => MWLevelOptions = (i
     : Int, options: MWLevelOptions) => {
    options.withLevelLength(
            ((((i+levelSeed) * 431) % (501+levelSeed) ) % 462) + 50)
          .withTimeLimit((options.levelLength * 0.7).toInt)
          .withLevelType(i % 3)
          .withLevelDifficulty((compNumberOfLevels - i)/32)
          .withPits(i % 4 != 2)
          .withCannons(i % 6 == 2)
          .withTubes(i % 5 == 1)
          .withCoins(i % 5 != 0)
          .withBlocks(i % 6 != 2)
          .withLadders(i % 10 == 2)
          .withFrozenCreatures(i % 3 == 1)
          .withEnemies(!(i % 4 == 1))
          .withStartingMarioMode(
             if (i % 7 == 5 || i % 7 == 1) {
               if (i % 2 == 0) 0 else 1
             } else 2)
}
```

Listing D1: Parameter classes for the comparator task described in Section 5.5

E LEMMEL Learning Parameter File

```
parent.0 = @ec.es.ESDefaults es.params

#General
breedthreads  = 4
evalthreads  = 1
seed.0 = 1
seed.1 = 909
seed.2 = 499311
seed.3 = 90032

# +++ ES +++
breed = ec.es.MuPlusLambdaBreeder
es.mu.0   = 5
es.lambda.0  = 45
generations  = 1000

# +++ POP +++
pop.subpops       = 1
pop.subpop.0.size      = 50
pop.subpop.0.species     = com.montywest.marioai.learning.ec.vector.
    RulesetSpecies
pop.subpop.0.species.fitness  = ec.simple.SimpleFitness

# Rulelength 15, So 20 rules gives 300
pop.subpop.0.species.genome-size  = 300
pop.subpop.0.species.ind    = ec.vector.ByteVectorIndividual

# These will be ignored, but warnings otherwise
pop.subpop.0.species.min-gene  = -1
pop.subpop.0.species.max-gene  = 2
pop.subpop.0.species.mutation-type = reset
pop.subpop.0.species.mutation-prob = 0.0
pop.subpop.0.species.crossover-type = one

# Handles min and max
pop.subpop.0.species.dynamic-param-class = com.montywest.marioai.
    learning.ec.params.RulesetParams

# Condition params
pop.subpop.0.species.condition       = true
pop.subpop.0.species.condition.mutation-prob    = 0.05
pop.subpop.0.species.condition.favour_byte     = -1
pop.subpop.0.species.condition.favour_probability   = 0.5

# MarioMode Condition
pop.subpop.0.species.condition.0.favour_probability  = 0.95

# EnemyLeft Condition
pop.subpop.0.species.condition.3.favour_probability  = 0.9
```

```
# Action params
pop.subpop.0.species.action        = true
pop.subpop.0.species.action.mutation-prob     = 0.09


# +++ STATS +++
stat.num-children       = 1
stat.child.0        = com.montywest.marioai.learning.ec.stats.
    RulesetEvolveStatistics
stat.child.0.gen-file     = ../lmm-lel-gen.stat
stat.child.0.final-file     = ../lmm-lel-final.stat
stat.child.0.final-agent-file   = lmm-lel-final
stat.child.0.best-agent-file   = lmm-lel-best
stat.child.0.diff-agent-file   = lmm-lel-diff
stat.child.0.best-agent-limit   = 800
stat.child.0.diff-agent-limit   = 800


# +++ MUTATION +++
pop.subpop.0.species.pipe.source.0   = ec.es.ESSelection
pop.subpop.0.species.pipe     = com.montywest.marioai.learning.ec.
    vector.breed.RulesetMutationPipeline


# +++ EVAL +++
eval.problem       = com.montywest.marioai.learning.ec.eval.
    AgentRulesetEvaluator

# Seeds for generating levels
# each generation g, seed used is:
#  prev_seed + add + g*mult
# where prev_seed is seed_start on g = 0
eval.problem.seed      = true
eval.problem.seed.start     = 3348
eval.problem.seed.add     = 284839
eval.problem.seed.mult     = 2568849

# Evaluation Multiplier
eval.problem.mults      = true
eval.problem.mults.distance    = 1
eval.problem.mults.win     = 3200
eval.problem.mults.mode     = 200
eval.problem.mults.kills    = 100
eval.problem.mults.time-left   = 2

# Fallback Action
eval.problem.fallback-action   = true
eval.problem.fallback-action.right  = true
eval.problem.fallback-action.speed  = true
```

```
# Levels
eval.problem.level      = true
eval.problem.level.num-levels   = 10

eval.problem.level.base.dead-ends   = false
eval.problem.level.base.enemies   = false
eval.problem.level.base.cannons   = false
eval.problem.level.base.pipes   = false
eval.problem.level.base.start-mode-num = 2
eval.problem.level.base.length-num  = 200
eval.problem.level.base.time-limit  = 100
eval.problem.level.0.difficulty-num  = 2
eval.problem.level.0.type-num   = 1
eval.problem.level.1.difficulty-num  = 3
eval.problem.level.1.type-num   = 0
eval.problem.level.2.difficulty-num  = 5
eval.problem.level.2.type-num   = 2
eval.problem.level.3.difficulty-num  = 10
eval.problem.level.3.type-num   = 0
eval.problem.level.4.difficulty-num  = 2
eval.problem.level.4.flat    = true
eval.problem.level.5.difficulty-num  = 7
eval.problem.level.6.difficulty-num  = 2
eval.problem.level.6.flat    = false
eval.problem.level.6.enemies   = true
eval.problem.level.6.frozen-enemies  = true
eval.problem.level.7.difficulty-num  = 2
eval.problem.level.7.frozen-enemies  = false
eval.problem.level.8.difficulty-num  = 3
eval.problem.level.8.type-num   = 1
eval.problem.level.8.tubes    = true
eval.problem.level.9.difficulty-num  = 5
eval.problem.level.9.type-num   = 0
eval.problem.level.9.cannons   = true
eval.problem.level.9.blocks    = false
eval.problem.level.9.tubes    = false
```

F Full Source Code

References

[1] Siyuan Xu, *History of AI design in video games and its development in RTS games*, Department of Interactive Media & Game development, Worcester Polytechnic Institute, USA, https://sites.google.com/site/myangelcafe/articles/history_ai.

[2] Chad Birch, *Understanding Pac-Man Ghost Behaviour*, http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior, 2010.

[3] Alex J. Champandard, *The AI From Half-Like's SDK in Retrospective*, http://aigamedev.com/open/article/halflife-sdk/, 2008.

[4] Tommy Thompson, *Facing Your Fear*, http://t2thompson.com/2014/03/02/facing-your-fear/, 2014.

[5] Damian Isla, *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*, http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php, 2005.

[6] Alex J. Champandard, *Monte-Carlo Tree Search in TOTAL WAR: Rome II Campaign AI*, http://aigamedev.com/open/coverage/mcts-rome-ii/, 2014.

[7] Georgios N. Yannakakis, Pieter Spronck, Daniele Loiacono and Elisabeth Andre, *Player Modelling*, http://yannakakis.net/wp-content/uploads/2013/08/pm_submitted_final.pdf, 2013.

[8] Matt Bertz, *The Technology Behind The Elder Scrolls V: Skyrim*, http://www.gameinformer.com/games/the_elder_scrolls_v_skyrim/b/xbox360/archive/2011/01/17/the-technology-behind-elder-scrolls-v-skyrim.aspx, 2011.

[9] *The Berkeley Overmind Project*, University of Berkeley, California. http://overmind.cs.berkeley.edu/.

[10] Simon M. Lucas, *Cellz: A simple dynamical game for testing evolutionary algorithms*, Department of Computer Science, University of Essex, Colchester, Essex, UK, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.96.5068&rep=rep1&type=pdf.

[11] G. N. Yannakakis and J. Togelius, *A Panorama of Artificial and Computational Intelligence in Games*, IEEE Transactions on Computational Intelligence and AI in Games, http://yannakakis.net/wp-content/uploads/2014/07/panorama_submitted.pdf, 2014.

[12] Julian Togelius, Noor Shaker, Sergey Karakovskiy and Georgios N. Yannakakis, *The Mario AI Championship 2009-2012*, AI Magazine 34 (3), pp. 89-92, `http://noorshaker.com/docs/theMarioAI.pdf`, 2013.

[13] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction* The MIT Press, Cambridge, Massachusetts, London, England, Available: `http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html`, 1998.

[14] Gary B. Fogel et al., *Evolutionary Programming* Scholarpedia, 6(4):1818., `http://www.scholarpedia.org/article/Evolutionary_programming`, 2011.

[15] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Moden Approach (3rd ed.*, Upper Saddle River, New Jersey, 2009.

[16] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Moden Approach (1st ed.*, Upper Saddle River, New Jersey, 1995.

[17] Anoop Gupta, Charles Forgy, Allen Newell, and Robert Wedig, *Parallel Algorithms and Architectures for Rule-Based Systems*, Carnegie-Mellon University Pittsburgh, Pennsylvania, ISCA '86 Proceedings of the 13th annual international symposium on Computer architecture, pp. 28-37, 1986.

[18] Melanie Mitchell, *An Introduction to Genetic Algorithms*, Cambridge, MA: MIT Press, 1996.

[19] Hans-Georg Beyer and Hans-Paul Schwefel, *Evolution Strategies: A Comprehensive Introduction*, Natural Computing 1: 3?52, 2002.

[20] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger and Eric Liang, *Inverted autonomous helicopter flight via reinforcement learning*, International Symposium on Experimental Robotics, `http://www.robotics.stanford.edu/~ang/papers/iser04-invertedflight.pdf`, 2004.

[21] S. Singh, D. Litman, M. Kearns and M. Walker, *Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System*, Journal of Artificial Intelligence Research (JAIR), Volume 16, pages 105-133, 2002, `http://web.eecs.umich.edu/~baveja/Papers/RLDSjair.pdf`.

[22] Alex J. Champandard, *Making Designers Obsolete? Evolution in Game Design*, `http://aigamedev.com/open/interview/evolution-in-cityconquest/`, 2012.

[23] James Wexler, *A look at the smarts behind Lionhead Studio's 'Black and White' and where it can and will go in the future*, University of Rochester, Rochester, NY 14627, http://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf, 2002.

[24] Thore Graepal (Ralf Herbrich, Mykel Kockenderfer, David Stern, Phil Trelford), *Learning to Play: Machine Learning in Games*, Applied Games Group, Microsoft Research Cambridge, http://www.admin.cam.ac.uk/offices/research/documents/local/events/downloads/tm/06_ThoreGraepel.pdf.

[25] *Drivatar^{TM} in Forza Motorsport*, http://research.microsoft.com/en-us/projects/drivatar/forza.aspx.

[26] Sergey Karakovskiy and Julian Togelius, *Mario AI Benchmark and Competitions*, http://julian.togelius.com/Karakovskiy2012The.pdf, 2012.

[27] Julian Togelius, Sergey Karakovskiy and Robin Baumgarten, *The 2009 Mario AI Competition*, http://julian.togelius.com/Togelius2010The.pdf, 2010.

[28] Julian Togelius, *How to run a successful game-based AI competition*, http://julian.togelius.com/Togelius2014How.pdf, 2014.

[29] *TORCS: The Open Racing Car Simulation*, http://torcs.sourceforge.net/.

[30] Daniele Loiacono, Pier Luca Lanzi, Julian Togelius, Enrique Onieva, David A. Pelta, Martin V. Butz, Thies D. Lnneker, Luigi Cardamone, Diego Perez, Yago Sez, Mike Preuss, and Jan Quadflieg, *The 2009 Simulated Car Racing Championship*, IEEE Transactions on Computational Intelligence and AI in Games, VOL. 2, NO. 2, 2010.

[31] *The 2K BotPrize*, http://botprize.org/

[32] Michael Buro, David Churchill, *Real-Time Strategy Game Competitions*, Association for the Advancement of Artificial Intelligence, AI Magazine, pp. 106-108, https://skatgame.net/mburo/ps/aaai-competition-report-2012.pdf, 2012.

[33] *Infinite Mario Bros.*, Created by Markus Perrson, http://www.pcmariogames.com/infinite-mario.php.

[34] H. Handa, *Dimensionality reduction of scene and enemy information in mario*, Proceedings of the IEEE Congress on Evolutionary Computation, 2011.

[35] S. Ross and J. A. Bagnell, *Efficient reductions for imitation learning*, International Conference on Artificial Intelligence and Statistics (AISTATS), 2010.

[36] Slawomir Bojarski and Clare Bates Congdon, *REALM: A Rule-Based Evolutionary Computation Agent that Learns to Play Mario*, 2010 IEEE Conference on Computational Intelligence and Games (CIG '10) pp. 83-90.

[37] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, *Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution*, Proceedings of EvoApps, 2010, pp. 123-132.

[38] E. R. Speed, *Evolving a mario agent using cuckoo search and softmax heuristics*, Proceedings of the IEEE Consumer Electronics Society's Games Innovations Conference (ICE-GIC), 2010, pp. 1-7.

[39] Thomas Willer Sandberg, *Evolutionary Multi-Agent Potential Field based AI approach for SSC scenarios in RTS games*, University of Copenhagen, 2011.

[40] *ECJ: A Java-based Evolutionary Computation Research System*, `https://cs.gmu.edu/~eclab/projects/ecj/`.

[41] Sean Luke, *The ECJ Owner's Manual, v23*, `https://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf` George Mason University, 2015.

[42] Sean Luke, *ECJ Tutorial 3: Build a Floating-Point Evolution Strategies Problem*, `https://cs.gmu.edu/~eclab/projects/ecj/docs/tutorials/tutorial3/index.html` George Mason University.

[43] *JGAP: Java Genetic Algorithms Package*, `http://jgap.sourceforge.net/`.

[44] Robin Baumgarten, *A* Mario Agent*, `https://github.com/jumoel/mario-astar-robinbaumgarten`.

[45] Ryan Small, *Agent Smith: a Real-Time Game-Playing Agent for Interactive Dynamic Games*, GECCO '08, July 12-16, 2008, Atlanta, Georgia, USA. `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2008/docs/p1839.pdf`.

[46] Hartmut Pohlheim, *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation*, version 3.80, 2006, `http://www.geatbx.com/docu/index.html`.

[47] Fred Glover and Kenneth Srensen, *Metaheuristics*, Scholarpedia, 10(4):6532., `http://www.scholarpedia.org/article/Metaheuristics`, 2015.