UNIVERSITY OF ESSEX

DOCTORAL THESIS

Automating Game-design and Game-agent Balancing through Computational Intelligence

Author:
Mihail MOROŞAN

Supervisor:
Prof. Riccardo POLI

A thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science and Electronic Engineering

March 20, 2019

Declaration of Authorship

I, Mihail MOROŞAN, declare that this thesis titled, "Automating Game-design and Game-agent Balancing through Computational Intelligence" and work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where parts of the thesis are based on work done by myself jointly with others, I have made clear what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

Mihail MOROŞAN

Automating Game-design and Game-agent Balancing through Computational Intelligence

Game design has been a staple of human ingenuity and innovation for as long as games have been around. From sports, such as football, to applying game mechanics to the real world, such as reward schemes in shops, games have impacted the world in surprising ways. The process of developing games can, and should, be aided by automated systems, as machines have proven capable of finding innovative ways of complementing human intuition and inventiveness. When man and machine cooperate, better products are created and the world has only to benefit. This research seeks to find, test and assess methods of using genetic algorithms to human-led game balancing tasks. From tweaking difficulty to optimising pacing, to directing an intelligent agent's behaviour, all these can benefit from an evolutionary approach and save a game designer many hours, if not days, of work based on trial and error. Furthermore, to improve the speed of any developed GAs, predictive models have been designed to aid the evolutionary process in finding better solutions faster. While these techniques could be applied on a wider variety of tasks, they have been tested almost exclusively on game balance problems. The major contributions are in defining the main challenges of game balance from an academic perspective, proposing solutions for better cooperation between the academic and the industrial side of games, as well as technical improvements to genetic algorithms applied to these tasks. Results have been positive, with success found in both academic publications and industrial cooperation.

Acknowledgements

This work would not be possible without the help and support of many wonderful people.

Without a doubt, the sage advice, constant push for better experiments, valuable insight and many well-timed solutions of my supervisor, Prof. Riccardo Poli, facilitated the success of most of my research and a great deal of this thesis. Also, having seen how others receive feedback on their drafts, I consider myself the luckiest student in the world. Half the page may be in red, but it's red that makes the whole black on white statistically significantly better (citation needed).

The valuable advice and feedback during supervisory boards, as well as supportive words during the early stages of the PhD, when the research topic was significantly different, of Dr. Daniel Kudenko from the University of York, were also greatly welcome.

Jo, Marisa, all the amazing students in the programme, Prof. Jeremy Gow, Dr. Paul Cairns, Prof. Udo Kruschwitz, Dr. Richard Bartle, are all people that made the IGGI experience a good one. A wild ride it was.

The friends I made during courses, social events, online games, activity societies and other miscellaneous places, were invaluable in making the PhD years enjoyable and relaxing. Also, their patience whenever I would start talking about "the thesis" or "that conference review" was worth a lot more than they think.

Finally, but by far the most important, none of this would be possible without the support of my parents. They pushed me towards higher achievements, facilitated me going further distances, sacrificed their health and time to see me succeed. I hope to be able to repay even a tiny fragment of all that, or to make you proud wherever you're cheering me on from. Thank you!

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Introduction 1
1.1 Motivations . . . 2
1.2 Objectives . . . 3
1.3 Achievements . . . 4
1.4 List of Publications . . . 5
1.5 Thesis Outline . . . 6

2 Literature Review 9
2.1 Game Design . . . 10
2.1.1 Research on Game Genres and Mechanics . . . 10
2.1.2 Definitions of Balance . . . 10
2.2 Genetic Algorithms . . . 12
2.2.1 Genetic Programming . . . 13
2.2.2 Synthetic Problems . . . 14
OneMax . . . 15
Trap Function . . . 15
2.2.3 Multi-objective Optimisation . . . 16
2.2.4 Surrogate Models and Fitness Prediction . . . 17
2.3 Machine Learning Algorithms . . . 18
2.3.1 Neural Networks . . . 18
2.3.2 Decision Trees . . . 19
2.3.3 k-Nearest Neigbours . . . 20
2.4 Games and Computational Intelligence . . . 20
2.4.1 Monte-Carlo Tree Search . . . 21
2.4.2 Goal Oriented Action Planning . . . 21
2.4.3 Automated Game Design . . . 22
2.4.4 Solving Games . . . 23
2.4.5 Believable Agents . . . 23
2.5 Balance Through the Use of Computational Intelligence . . . 24
2.6 e-Sports . . . 26
2.7 Code . . . 27

3 Games Targeted During the Research 29
3.1 *Ms. Pac-Man* . . . 29
3.1.1 Introduction . . . 29
3.1.2 Interfacing with *Ms. Pac-Man* . . . 30
3.1.3 Research on *Ms. Pac-Man* . . . 30

3.2 *StarCraft* . . . 31
3.2.1 Introduction . . . 31
3.2.2 Interfacing with *StarCraft* . . . 32
3.2.3 Research on *StarCraft* . . . 32
3.3 *TORCS* . . . 34
3.3.1 Introduction . . . 34
3.3.2 Interfacing with *TORCS* . . . 34
3.4 *ComPet* . . . 35
3.4.1 Introduction . . . 35
3.4.2 Interfacing with *ComPet* . . . 36
3.5 Genesis Dei . . . 37

4 Balance Specification Language 39
4.1 Introduction . . . 39
4.2 Describing Elements to be Changed . . . 40
4.3 Evaluating Success . . . 41
4.3.1 Available Evaluators . . . 42
4.3.2 Final Score . . . 44
4.4 Communicating with the Games . . . 45
4.5 Extended Backus-Naur Notation . . . 46

5 Genetic Algorithms for Video Game Parameter Balance 49
5.1 Introduction . . . 49
5.2 *Ms. Pac-Man* Experiments . . . 50
5.2.1 Environment . . . 50
Fitness Evaluation . . . 52
Genetic Algorithm . . . 53
Choosing Weights . . . 54
Selecting the Number of Games Played . . . 56
5.2.2 Experiments . . . 57
5.2.3 Results . . . 58
Exploratory Run . . . 58
Testing GA Parameter Configurations . . . 61
Testing Different Values for Weights for Fitness Components . . . 63
5.3 *StarCraft* Experiments . . . 66
5.3.1 Environment . . . 66
Fitness Evaluation . . . 67
5.3.2 Experiments . . . 69
5.3.3 Results . . . 70
Completely Nullifying the ZZZKBot . . . 70
Preliminary Balancing of the ZZZKBot Strategy . . . 71
Balancing ZZZKBot Using Optimised GA Parameters . . . 74
5.4 *TORCS* Experiments . . . 76
5.4.1 Environment . . . 76
Fitness Evaluation . . . 77
Genetic Algorithm . . . 78
5.4.2 Experiment . . . 79
5.4.3 Results . . . 79
5.5 Discussion . . . 83
5.6 Summary . . . 84

6 Fitness Approximation for Faster GA-Based Game Balancing . . . 85
6.1 Introduction . . . 85
6.2 Pipeline . . . 87
6.2.1 Approximator Integration . . . 87
6.2.2 Neural Network . . . 89
6.2.3 C4.5 Decision Trees . . . 91
6.2.4 k-Nearest Neighbours . . . 92
6.3 Standard Fitness Function Experiments . . . 93
6.3.1 OneMax . . . 93
6.3.2 Trap . . . 93
6.3.3 Genetic Algorithm . . . 93
6.3.4 Neural Network . . . 94
6.3.5 Experiments . . . 94
6.3.6 OneMax Results . . . 95
6.3.7 Trap Results . . . 96
6.4 *Ms. Pac-Man* Experiments . . . 100
6.4.1 Environment . . . 100
Fitness Evaluation . . . 100
Genetic Algorithm . . . 101
Choosing the Approximator's Accuracy Threshold . . . 102
Choosing the Approximator's Prediction Acceptance Threshold 102
6.4.2 Experiments . . . 103
6.4.3 Results . . . 103
Data . . . 103
Overview of Runs on Unoptimised GA Configuration . . . 104
Performance of Various Approximators . . . 107
Performance of Using Various Accuracy Thresholds . . . 110
Performance of Using Various Prediction Thresholds . . . 111
Overview of Runs on the Optimised GA Configuration . . . 112
Comparing Approximators on Unoptimised GA Parameter Set to Optimised GA Parameter Set . . . 113
6.5 *TORCS* Experiments . . . 115
6.5.1 Environment . . . 115
6.5.2 Experiments . . . 115
6.5.3 Results . . . 115
Data . . . 115
Overview of Runs on Unoptimised GA Configuration . . . 115
Performance of Various Approximators . . . 116
Performance of Using Various Accuracy Thresholds . . . 119
Performance of Using Various Prediction Thresholds . . . 120
Overview of Runs on the Optimised GA Configuration . . . 121
Comparing Approximators on Unoptimised GA Parameter Set to Optimised GA Parameter Set . . . 121
6.6 *StarCraft* Experiment . . . 123
6.6.1 Introduction . . . 123
6.6.2 Results . . . 123
6.7 Discussion and Conclusion . . . 125
6.8 Summary . . . 127

7 Other Applications of Automated Balancing **129**
7.1 Introduction . . . 129
7.2 Commercial Application of Automated Game Balance with *ComPet* . . 130
7.2.1 Introduction . . . 130
7.2.2 Environment . . . 130
Fitness Evaluation . . . 132
Genetic Algorithm . . . 133
7.2.3 Experiment . . . 134
7.2.4 Results . . . 134
7.3 Evolving Game Agents with Diverse Behaviours . . . 138
7.3.1 Introduction . . . 138
Motivation . . . 138
Existing *Ms. Pac-Man* Agents . . . 138
7.3.2 Methodology . . . 140
Pipeline . . . 140
Neural Network Agent . . . 140
Genetic Algorithm . . . 142
Fitness Evaluation . . . 143
Experiments . . . 143
7.3.3 Results . . . 144
Experiment 1: The Strongest Neural Network Evolved . . . 144
Experiment 2: A Balanced Neural Network with High Variance 145
Experiment 3: A Balanced Neural Network with Low Variance 146
7.4 Discussion . . . 148
7.4.1 Industrial Applications . . . 148
7.4.2 Evolving Game Agents . . . 148
7.5 Summary . . . 150

8 Conclusions and Future Work **151**

A *ComPet* Example Gauntlet **153**

B Diplomatic Turn-Based Strategy Games **157**
B.1 Introduction . . . 157
B.2 A Description of DTBG . . . 157
B.3 Conflicts as a Gameplay Mechanic . . . 158
B.4 Discussing Potential Player Types . . . 160
B.4.1 Leaders (Socializers / Achievers) . . . 160
B.4.2 Followers (Explorers) . . . 161
B.4.3 Diplomats (Socializers / Explorers) . . . 161
B.4.4 Aggressors (Socializers / Killers) . . . 161
B.4.5 Warriors (Killers) . . . 162
B.4.6 Strategists (Explorers / Achievers) . . . 162
B.5 Game Design Space . . . 162
B.6 Discussion . . . 163

C Aggregate Data for Approximator Experiments **165**
C.1 Introduction . . . 165

Bibliography **179**

List of Figures

2.1 Fitness landscape for OneMax. Lower values represent better fitness. $u(X)$ represents the number of 1 bits present in the individual X 15
2.2 Fitness landscape of Trap for $Z = 4000$. Lower values are better 16
2.3 Example of a simple decision tree 20

3.1 Screenshot from a level in a game of *Ms. Pac-Man* 29
3.2 Screenshot from a game of *StarCraft* with most UI elements hidden . . 31
3.3 Pipeline of integration with *StarCraft* 33
3.4 Screenshot from a game of *TORCS* 34
3.5 Screenshot from *ComPet* presenting one of the available player pets . . 35
3.6 Screenshot from *ComPet*'s battle mode 36

4.1 Sample of the specification language presenting a list with a single parameter for *TORCS* 40
4.2 Sample of the specification language presenting metrics for *TORCS* . . 42
4.3 Sample of the specification language presenting a list with a single evaluator for *TORCS* 42
4.4 Sample of JSON data sent by a game describing the requested metrics after completing play 45

5.1 Distribution of scores achieved by a rule-based agent in *Ms. Pac-Man* (see text). The area highlighted in red represents scores above 1500. . . 51
5.2 A mapping of fitness values depending on the win-rate metric (*WR*), which is dependent on the actual simulation, and the *Ratio*. This shows how much the ratio impacts the gap between good individuals (with *WR* close to 0.5) and bad individuals 56
5.3 Comparison between the standard error of the mean of scores, as calculated mathematically, and the empirical standard error of scores when looking at actual *Ms. Pac-Man* scores, relative to the number of games sampled each time 57
5.4 Boxplot of best individual fitnesses in each of the 10 *Ms. Pac-Man* runs 58
5.5 Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing GA configurations with various rates of mutation 62
5.6 Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing GA configurations with and without reinitialisation 63
5.7 Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing different population sizes 64

5.8 Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing various objective weights. Dashed green lines represent the baseline fitness for each respective value of the weight ratio. Orange ranges represent the confidence thresholds 65
5.9 Boxplot of Δ_P fitness component for all 10 *StarCraft* runs 73
5.10 Win-rate % after human play-testing 74
5.11 Average best fitness values achieved after a given number of evaluations by the GA runs in the *StarCraft* experiment, when comparing the results of the exploratory GA parameters to the results of the optimised GA parameters. Lower values are better. 75
5.12 Visual representation of the times achieved at various checkpoints by the unchanged car (Car 1) compared to the best performing car, according to the requirements, (Car 2, from run 19) on the dirt track . . . 82

6.1 Pseudo-code for the integration of an approximator in a GA run 88
6.2 Diagram of the predictor logic within the GA individual evaluation . . 90
6.3 Boxplots for number of fitness evaluations required to achieve perfect fitness in OneMax w/o and w/ an approximator 96
6.4 Total evaluations required to achieve a perfect fitness in OneMax, w/o and w/ the neural network approximator 97
6.5 Boxplots for number of iterations required to achieve perfect fitness in Trap without and with a approximator. 98
6.6 Average best fitness values achieved after a given number of generations by the GAs without and with the C4.5 decision trees approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment 105
6.7 Average best fitness values achieved after a given number of evaluations by the GAs without and with the neural network approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment. Significance of difference between the paired results is also plotted. Lower values are better 107
6.8 Average best fitness values achieved after a given number of evaluations by the GAs without and with each of the tested approximators, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment with the optimised GA parameter set. Lower values are better. 113
6.9 Average best fitness values achieved after a given number of evaluations by the GAs without and with the neural network approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *TORCS* experiment with the optimised GA parameter set. Lower values are better. 122
6.10 Average best fitness values achieved after a given number of evaluations by the GAs without and with the C4.5 approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *StarCraft* experiment. Significance of difference between the paired results is also plotted. Lower values are better 124

7.1 Similarity map representing the Euclidean distance between the 10 *ComPet* run parameter change suggestions, with colour-coded clusters highlighting the 2 main balancing strategies proposed 136

7.2 Structure of the neural network used to score each node 141

B.1 Genesis planet view. Every little tower is a different player 159

List of Tables

4.1 Extended Backus-Naur Notation for the specification language used in defining balance tasks. 46

5.1 *Ms. Pac-Man* parameters to be changed, their displacement ranges and their decimal accuracy 51
5.2 *Ms. Pac-Man* metrics used in evaluation alongside their desired values and weights 52
5.3 Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing mutation rates. As a result of crossover, mutation and elitism percentages adding up to 100%, reinitialisation is not used 53
5.4 Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing impact of reinitialisation 54
5.5 Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing performance variation of population size 54
5.6 Amount by which metrics have to change to increase their respective fitness objective by 100 in the *Ms. Pac-Man* game balancing experiment, with respect to several configurations of C_W and C_Δ, as well as what the fitness of the parameter set of only 0s would be 55
5.7 *Ms. Pac-Man* experiment results, with the best solution in bold and the default, unchanged, version at the bottom. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes 59
5.8 Wilcoxon Signed Rank test when comparing various mutation rates . . 61
5.9 Original *StarCraft* parameters 67
5.10 Parameters chosen and their ranges for *StarCraft* 68
5.11 *StarCraft* metrics to be used in evaluation alongside their desired values and weights 68
5.12 Suggested changes to *StarCraft* parameters to minimise ZZZKBot's win-rate 70
5.13 Main *StarCraft* experiment results, with best individual(s) in bold. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes 72
5.14 Parameters chosen and their original values for *TORCS* 76
5.15 Parameters chosen and their ranges for *TORCS* 77
5.16 Times achieved by the vanilla version of the car on each track, alongside the desired times 77
5.17 *TORCS* metrics to be used in evaluation alongside their desired values and weights 78

5.18 Experiment results, with the best solution in bold and the default, unchanged, version at the bottom. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes 81

6.1 The mapping of fitness values to classifier classes 91
6.2 Neural network neuron count per layer 94
6.3 Number of runs that failed to get a perfect solution (out of 30) for Trap 97
6.4 *Ms. Pac-Man* parameters to be changed, their displacement ranges and their decimal accuracy 100
6.5 *Ms. Pac-Man* metrics to be used in evaluation alongside their desired values and weights for the approximator experiments 101
6.6 Genetic algorithm parameter sets tested for *Ms. Pac-Man* approximator experiments 102
6.7 Average run times when using each of the 3 machine learning algorithms as approximators, compared to using no approximator, in the *Ms. Pac-Man* experiments. 104
6.8 Average number of predictions made by each approximator configuration tested in the *Ms. Pac-Man* experiments, as well as the average number of false negatives generated as a result. 106
6.9 Percentage of time that the neural network approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best. 108
6.10 Percentage of time that the C4.5 decision tree approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best. 108
6.11 Percentage of time that the k-nearest neighbour approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best. 109
6.12 Average proportion of time in which each combination of approximator and accuracy threshold was statistically significant, regardless of prediction threshold, in the *Ms. Pac-Man* experiments. 110
6.13 Average proportion of time in which each combination of approximator and prediction threshold was statistically significant, regardless of prediction threshold, in the *Ms. Pac-Man* experiments, as well as average predictions computed. 111
6.14 In the *Ms. Pac-Man* approximator experiments using the optimised GA parameters: percentage of time that using the approximator resulted in significantly better results; the average number of predictions generated by the approximator each run and the average percentage of which were false negatives. 112
6.15 Average run times when using each of the 3 machine learning algorithms as approximators, compared to using no approximator, in the *TORCS* experiments. 116
6.16 Average number of predictions made by each approximator configuration tested in the *TORCS* experiments, as well as the average number of false negatives generated as a result. 117

6.17 Percentage of time that the neural network approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best. 117
6.18 Percentage of time that the C45 decision tree approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best. 118
6.19 Percentage of time that the k-nearest neighbour approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best. 119
6.20 Average proportion of time in which each combination of approximator and accuracy threshold was statistically significant, regardless of prediction threshold, in the *TORCS* experiments. 120
6.21 Average proportion of time in which each combination of approximator and prediction threshold was statistically significant, regardless of prediction threshold, in the *TORCS* experiments, as well as average predictions computed. 120
6.22 In the *TORCS* approximator experiments using the optimised GA parameters: percentage of time that using the approximator resulted in significantly better results; the average number of predictions generated by the approximator each run and the average percentage of which were false negatives 121

7.1 *ComPet* beasts in the experiment gauntlet 131
7.2 *ComPet* metrics collected on the gauntlet playing the unchanged version of the game 132
7.3 *ComPet* metrics to be used in evaluation alongside their desired values and weights 132
7.4 *ComPet* parameters to be changed, their displacement ranges, their decimal accuracy and their weight in the fitness evaluation 133
7.5 Main *ComPet* experiment results, highlighting the changes recommended by each run. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes. Each column (except Run) represents the displacement to one of the evolved parameters 135
7.6 Main *ComPet* experiment results, highlighting the individual fitness objectives and scores achieved by each run. Columns, except Run, represent the fitness objectives described in Table 7.3. The bolded row represents the run with the best fitness achieved out of all runs 135
7.7 Average scores, and their standard deviation, achieved by a rule-based agent and MCTS over 1000 games, as well as the desired values for the evolved agents 140
7.8 Features chosen for evaluating each node 142
7.9 Experiment results for the experiment attempting to generate as good a player as possible given the architecture evolved, with the best performing run in bold 145

7.10 Experiment results for the second experiment, where an agent is evolved to fit given the designer requirements of $DS = 1750$ and $DD = 1000$, with the best performing run in bold 146
7.11 Experiment results for the third experiment, where an agent is evolved to fit a different set of designer requirements of $DS = 1750$ and $DD = 100$, with the best performing run in bold 147

C.1 Results when comparing between approximator runs with the unoptimal GA parameter set and both optimal and unoptimal GA runs, in the *Ms. Pac-Man* experiments. Better and worse values represent the percentage of time that the second configuration proved significantly better, or worse respectively, than the first configuration 166
C.2 Results when comparing between approximator runs with the unoptimal GA parameter set and both optimal and unoptimal GA runs, in the *TORCS* experiments. Better and worse values represent the percentage of time that the second configuration proved significantly better, or worse respectively, than the first configuration 172

List of Abbreviations

GA	**G**enetic **A**lgorithm
PvP	**P**layer **v**ersus **P**layer
MCTS	**M**onte-**C**arlo **T**ree **S**earch
MOO	**M**ulti-**o**bjective **o**ptimisation
ML	**M**achine **L**earning
KNN	**K**-**N**earest **N**eighbours
GOAP	**G**oal **O**riented **A**ction **P**lanning
PvP	**P**layer-**v**ersus-**P**layer
BCI	**B**rain-**C**omputer **I**nterface

Chapter 1

Introduction

There is a lot of value to be gained from games. Their entertainment value, their ability to make people forget about their day to day problems, their hidden property of teaching people various skills, both physical and emotional, their community building traits, all of these can come from games. Video games are a subset of games available only in digital form. Available for games consoles or for personal computers, in virtual reality experiences or on a phone, they all share many traits.

A more recent development has found yet another bit of value for video games: scientific research. Understanding why people are attracted to video games is one facet of this. Figuring out how to use games to improve other fields is another. This thesis focuses on a third facet of research: creating tools to improve games themselves, or, at least simplify the process of developing games.

Game designers and developers spend days, weeks, months, even years on testing and tuning how their games are actually played. The common problem of dream not matching reality is all too true in video game development as well. Intent and delivery are two very different elements. While a designer might hope that a new feature has a certain result, real world usage might prove them wrong.

These designers and their studios must spend time and money on testing the game for each of the changes they decide to go through with. This, often times, requires dedicated testing groups, which cost money and take time to return with feedback. This has the potential to slow down the development process, or, if not done properly, result in undesirable behaviour in the game.

Unsurprisingly, there has been significant research in understanding game design, as well as in automating elements of it. It is, however, a field far from tapped

of potential.

1.1 Motivations

Game balance, or the art of finding just the right mechanics and numbers to achieve the dream mentioned previously, is currently still an almost entirely player-driven area. Teams of game testers are employed to find both bugs in their games, but also offer feedback on the game design. The research in this thesis focused on finding ways of optimising at least some elements of game testing and game balance.

The original motivation was the complicated process of balancing a strategy game developed in the early stages of research. Manual changes proved both cumbersome and frustrating to test, as results would only be obvious weeks after their implementation. Some of the changes, while small, proved outright disastrous. This prompted a better understanding of game balance, as well as the question of whether parts of the process could be optimised.

The obvious solution is more money. Hiring a team of game testers is an extra cost on top of any existing development costs. The longer they have to work on a single game, the more they cost the company and the later the date before that game can generate revenue. That is not always a possible avenue. The alternative solution, which might help with optimising the process of game balance, is the use of computational intelligence.

A major issue in that regard is the slight disconnect between academia and industry. A lot of academic research in games is done on old or synthetic games, with little interest given to its further application in real games. Alternatively, the fast pace of industry is not always willing to wait for academic research to develop the solutions they might or might not use. This gap benefits nobody in the long run. Some compromise and cooperation would very likely immensely benefit both.

In short, the main motivations were:

- There is still much to understand about game balance, from an academic perspective
- Game testing is slow and, at times, expensive

- There is a gap between academic research in games and industry adoption of said work

1.2 Objectives

This thesis has the main objective of exploring the problems described previously and attempts to find solutions for as many of them as possible. If computational intelligence can aid in any way, it must be explored and documented.

We need a better way of defining game balance in academia. By improving the way the problem is described, research can bring better results and future researchers can improve the field further.

We need to find ways of automating parts of the testing process. Automated algorithms could reduce the time and workload required to test various changes to games. If they simply identify the really bad changes before a tester has to spend time with them, that is already a great step forward.

Once methods and models for automating game balance are found, these must take in considerations performance. If they take just as much time as manual testing, for little benefit, adoption could be much harder. Any resulting algorithms should be optimised as much as possible.

Finally, industry expectations must be considered. An amazing algorithm used by nobody outside of the research would, in this case, be wasted time. Game studios are to offer direct feedback on the use of any resulting algorithms and it should be reflected in any reported work. Alongside that, the algorithms should already be applied to real games instead of synthetic models, to prove viability.

In summary the objectives are:

- Define game balance as well as possible, for future academic work in the field
- Explore options of automating the process of game testing
- Develop and test any resulting algorithms on a number of real games
- Optimise the algorithms as much as possible

- Develop any other optimisations or improvements that would help in having them be used outside academia

1.3 Achievements

Genetic algorithms for game balance The biggest focus of this thesis, and its core contribution, is the use of genetic algorithms for automating the process of game balance testing.

By defining the task of game balance as the tuning of game parameters until a set of game metrics fit designer requirements, one can use various algorithms to explore the solution space. This solution space, based on the game parameters, can be massive and brute force exploration is not sensible most of the time.

Our approach, using evolutionary algorithms, proved highly successful in a variety of games, with a lot of potential.

Balance specification language One of the immediate consequences of using genetic algorithms for game balance was the need for a better interface between designers and algorithms. A method of defining both parameters and relevant metrics was developed.

By translating designer goals to a form that is easy to understand by both machine and game designer, the processes of manual and automated game balance are greatly aided. The door is then opened for new algorithms to be used in conjunction with games already using this new specification language, with minimal effort.

This work was the immediate result of cooperating with a commercial games studio. Their feedback was critical in better understanding industrial requirements and expectations.

Machine learning for speeding up genetic algorithms One of the games tested, *StarCraft*, is extremely slow to balance using the methodology developed. This is due to us having had no access to the game's source code, thus having to rely on several unconventional methods of running the game in place.

The result was a search for methods that could speed up the genetic algorithm. Given the mapping between parameters and metrics, we made use of online machine learning algorithms to offer approximations of the desirability of some parameter changes, potentially resulting in a faster search.

Tests were done with neural networks, decision trees and k-nearest neighbour algorithms. Results proved positive, with immense potential for improving unoptimised genetic algorithm runs, something very likely to be present when used outside research. Additionally, these improvements are not, at least theoretically, limited to game balance tasks.

Genetic algorithms for agent balance Finally, one of the fairly accidental achievements during the writing of this thesis was the realisation that the exact same balance algorithms described earlier can be used to evolve game agents instead of game changes.

If the game agent can be represented as an array of numbers, in the way that, for example, a neural network can be, then the balance algorithms can evolve new agents that behave in ways defined by the game metrics. This can result in not only really strong agents that aim to beat the game or get as many points as possible, but also agents that behave in interesting ways.

We only test this on one scenario, but believe this could be possible for any game that allows a similar method of representing their agents.

1.4 List of Publications

This thesis is based on a number of publications, as well as on some unpublished work. The list of publications is presented below.

- M. Morosan and R. Poli, "Automated Game Balancing in Ms PacMan and StarCraft using Evolutionary Algorithms", in *Applications of Evolutionary Computation*, G. Squillero and K. Sim, Eds., Springer, Cham, 2017, pp. 377–392, ISBN: 978-3-319-55849-3. DOI: 10.1007/978-3-319-55849-3_25. [Online]. Available: http://link.springer.com/10.1007/978-3-319-55849-3_25

- M. Morosan and R. Poli, "Speeding Up Genetic Algorithm-based Game Balancing using Fitness Predictors", in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17, New York, NY, USA: ACM, 2017, pp. 91–92, ISBN: 978-1-4503-4939-0. DOI: 10.1145/3067695.3076011. [Online]. Available: http://doi.acm.org/10.1145/3067695.3076011

- M. Morosan and R. Poli, "Evolving a Designer-Balanced Neural Network for Ms PacMan", in *2017 9th Computer Science and Electronic Engineering (CEEC)*, Sep. 2017, pp. 100–105. DOI: 10.1109/CEEC.2017.8101607

- M. Morosan and R. Poli, "Online-Trained Fitness Approximators for Real-World Game Balancing", in *Applications of Evolutionary Computation*, K. Sim and P. Kaufmann, Eds., vol. 10784 LNCS, Cham: Springer International Publishing, 2018, pp. 292–307, ISBN: 978-3-319-77538-8. DOI: 10.1007/978-3-319-77538-8_21

- A. Iacob, M. Morosan, F. Sepulveda, *et al.*, "Genetic Optimisation of BCI Systems for Identifying Games Related Cognitive States", in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ACM, 2018, pp. 237–238

- M. Morosan and R. Poli, "Lessons from Testing an Evolutionary Automated Game Balancer in Industry", in *2018 IEEE Games, Entertainment, Media Conference (GEM) (2018 IEEE GEM)*, Galway, Ireland, Aug. 2018

1.5 Thesis Outline

Chapter 2 presents a comprehensive literature review. It starts by describing a bit of research on game genres and their mechanics, then goes into detail on the topic of game balance. It is followed by descriptions of various machine learning algorithms, such as genetic algorithms, neural networks and a few others. Next it provides a shallow description of the current state of game research in general, mostly focusing on topics tangentially related to game design and game artificial intelligence. Finally, a few more related topics are described, such as agent believability and e-Sports.

Chapter 3 goes into detail about the several games used during this research. It offers a brief presentation of the following games: *Ms. Pac-Man*, *StarCraft*, *TORCS*, *ComPet* and *Genesis Dei*. The goal is to introduce readers to these games and also present examples of research already done on them.

Chapter 4 introduces the specification language used throughout the entirety of this research. The reason behind the existence of this language is elaborated on, while also giving an exhaustive description of its functionality. It is completed by the presence of the language's Extended Backus-Naur Notation.

Chapter 5 describes the core of this thesis, the use of genetic algorithms for the balancing of video games. It explores the problem in depth using *Ms. Pac-Man* as the main game, then applies the findings to *TORCS* and *StarCraft*. Among the topics covered are how genetic algorithm parameters affect performance, how game metrics could be weighted and the impact that has on the methodology, as well as other considerations relevant to each game.

Chapter 6 explores and elaborates on the methods used to speed up the genetic algorithms developed in the previous chapter. These methods, called "approximators", are described in detail, alongside the pipeline of seamlessly introducing them into the genetic algorithm. The impact of various variables within this pipeline is explored in depth, with final recommendations given as to how this methodology is best used in general.

Chapter 7 goes on to present alternate ways of using the algorithms described in Chapter 5. One use case involves their use in an industrial setting, for a commercial video game. A second use case involves evolving the representation of a game agent itself, instead of the game parameters. As a result the balancing algorithms presented in previous chapters can be used to generate new game agents with designed behaviour. We apply this to *Ms. Pac-Man* agents and find that this works very well and there is great potential for further improvements.

Chapter 8 presents a short series of conclusions, discussing the achievements of the thesis and potential future work that could be done.

Chapter 2

Literature Review

A great amount of research is done in games, however there is a disconnect between what is valuable to researchers and what is valuable to the industry [7]. After many conversations with game developers and game designers, one of the main reasons is the diverging goals of the two parties. Algorithms developed are often only applied on synthetic problems, with little time from the researchers available for investing in real games. Similarly, most game professionals do not have the time or resources to invest in understanding the research and applying it to their games. The exceptions are usually the large game corporations [8].

We argue that research on video game balancing could greatly benefit the real world. This is an area where any improvement in the tools created by research can be relevant to many games, with developers potentially saving tens or hundreds of hours and a lot of money on design and testing time. By focusing on how to define balance problems in a simple way for both academics and designers, we believe more can be achieved by both parties.

Leaving the task of defining the success of an algorithm in the hands of the experiment designer brings both advantages and disadvantages. The immediate advantage is preserving creative freedom, something very valuable to an artistic medium such as games. Not all games are created equal, and not all games have the same definitions of “fun” or “engaging”. However, two immediate disadvantages are the requirement that the algorithm be able to adapt to any and all goals the designer has put in place, alongside the challenge of presenting the system’s limitations in an easy to understand manner. Obtuse systems are less likely to be adopted by game developers.

2.1 Game Design

2.1.1 Research on Game Genres and Mechanics

The video game world has been in constant evolution for the past 42 years, with a massive number of titles on many different platforms, all over the world. Genres have come and gone over these many years, each offering its own benefits to the world.

Cohen [9] described genres as open categories where every new game *"alters the genre by adding, contradicting, or changing constituents, especially those of members most closely related to it"*. Clearwater [10] writes *"since genres are historically situated, they should be understood as cultural processes. Genres evolve, morph and transform, sometimes go dormant and may even enjoy renewed interest from audiences"*.

Tychsen [11] does mention a lack of games in the massively multiplayer strategy genre as one of the reasons behind a lack of research behind this genre, alongside further complexities brought by the scale. By creating new games and researching their features, player behaviour within them, as well as possible future expansions to the concepts, this blank can be filled.

Game mechanics define an interaction between users and the system, to encourage exploration and learning of the simulation's possibilities. Core mechanics, as defined by Sicart [12], are elements that many, if not most, games share when discussing how players interact with them.

Apperley [13] argues that strategy requires a manipulation of the simulation *"as it progresses through time, in order to get the result with the most utility"*. Tychsen [11] describes two methods of time resolution: simultaneous and sequential. When considering sequential time resolution, everyone takes turns one by one. With simultaneous time resolution, turns, or time, flow in parallel for all players.

2.1.2 Definitions of Balance

The literature presents different notions of "balance", perhaps due to how different games, or even genres, naturally lead to (or need) slightly different versions of the concept. The most generic definition, supplied by Schreiber [14], is that game balance "is mostly about figuring out what numbers to use in a game". Some have

argued that a measure of balance can be derived from how interesting, while also uncertain, a game is during its play [15] [16].

Players will reliably find their own state of balance after investing enough time in the game. This can involve learning what the best strategies are and what game elements are strongest. This gives experienced players a clear advantage over novices, while the best players will have a fair battleground to fight on. Although this is definitely a state of balance, it is not the same concept as a balanced game, as there might exist game elements that are not used at all, resulting in wasted design space and game assets, as well as a potential lack of any strategic breadth.

Game designers can directly impact the strengths and weaknesses of their game's entities, but they cannot control player behaviour. By optimising the parameters of a game, they force players to adapt to the new environment and rediscover the state of balance described previously.

Defining what is and is not balanced is not an easy task, especially when a mathematical representation of the game's mechanics is unavailable and one has to rely on statistics and word of mouth [17]. In popular multiplayer games, both designers and players analyse how often a game entity (e.g., a champion in *League of Legends* or a gun in *Counter-Strike*) has impacted a player's chance of winning the game.

Sirlin [18] considers game balancing as the iterative task of bringing a game to a state where the options presented to a player are not only reasonably many, but also viable.

One way of looking at balance is to see it as a function of many variables, each representing a different facet of the game, that needs to be optimised [14]. However, the greater the complexity of the game, the more difficult it is to derive an explicit formulation for such a function or that the solution currently used in a game is optimal. Indeed, optimising game balance is difficult and game designers tend to use a sort of hill-climbing approach based on small steps where some of their games' parameters are changed in tiny increments or decrements, hoping the new values will bring the game closer to their own understanding and definition of balance.

Complex games allow for much variation in play style, decisions to be taken and different start configurations. Balance is almost impossible and often highly dependent on player opinion [17]. While some might consider an element of a game

to be imbalanced, there will often be many bringing good arguments towards the contrary. What is almost consistently true is that, in a competitive game, players will constantly look for the strongest options and abuse them. This way, they are a lot more likely to win.

However, people's understanding of games is highly subjective and designers have different understandings of what makes their game balanced or not. For the purpose of this research, the definition used by Beyer *et al.* is the one that is followed, as it is the most general available and is reported below [19].

Game balancing is the process of systematically modifying parameters of game components and operational rules in order to determine satisfactory configurations regarding predefined goals.

Often balance can be dependent on player enjoyment, with a requirement for "interesting" games. Cincotti *et al.* [15] argue that uncertainty can create a much more enjoyable experience. This can be interpreted as the requirement to not have any dominant strategy and, as a result, no way to crown a winner early in the game, assuming equally skilled players. By assessing the viability and power of the various strategies discovered by players, a designer could also aim to keep any and all in check.

2.2 Genetic Algorithms

Genetic algorithms (GA) are algorithms inspired by nature. They take the phrase "survival of the fittest" and adapt it to computational tasks. By allowing solutions to various tasks to be ranked and compared, the most fit among them are able to rise to the top. Then, similarly to nature, the fit solutions "procreate" and generate new, potentially even better, offspring.

They were first proposed by Alan Turing [20], with the first practical applications soon after [21]. The earliest attempt at applying genetic algorithms to games was done by Barricelli [22]. Popularity of this approach, however, came as a result of Holland's book *Adaptation in Natural and Artificial Systems* [23].

Genetic algorithms usually rely on the ability to assess how good a given solution is to solving a given problem. By then having an entire population of solutions,

usually generated randomly, one can rank them all and, at any given time, have access to the "best" solution so far. Between the various individuals in the population, operators are applied in order to generate likely brand new solutions. These operators can include crossover, where the new individual shares traits with two different parents from the original population; mutation, where an individual from the original population is changed in slight ways, resulting in a new one; or elitism, where one or more of the very best individuals are kept as they are.

A usual method or representing individuals is as an array of bits or floats [24]. This allows for very simple methods of crossover or mutation. Crossover can be achieved by recombining the two parents, in this case elements in each array. Mutation is even simpler, as one or more elements in the array can be modified to achieve the desired result.

For arrays as described previously, there are several types of crossover [25]: single-point, where parents are split by a single randomly chosen point; two-point, where two random points are chosen and any values between those points are swapped to generate the new individuals; uniform, where new bits are chosen independently from the parent's ones, based on given distributions; and more for more specialised representation models.

Genetic algorithms have evolved a lot over time, with many different variations appearing to solve different types of problems. They have been used to solve many problems, in many domains, both academic and industrial [26]. Many uses in games and other media have emerged, with tracks in major conferences covering their use in those fields, such as EvoGAMES in EvoStar [27] [28].

2.2.1 Genetic Programming

Genetic programming (GP) makes use of evolutionary algorithms to build solutions for problems with complex definitions [29]. Simple tasks can include function regression [30], while complex tasks could include entire behaviour trees for survival strategies [31]. By using methods found in nature, evolutionary algorithms take many generations of solutions, test them, then discard those deemed not to be good

enough. This results in ever-improving solutions. A naive mathematical simplification would be that they are searching an n-dimensional space for local and global optima.

Due to its stochastic nature, given a different random seed, GP is extremely likely to offer completely different solutions, as the search space for many tasks is immense. Many have attempted to improve the algorithm over time [32] [33], to minimise the generation of weak members from the get-go, but a general solution is not possible, as each task that GP tries to tackle brings its own complexities and requirements to the table. Brandy utilized genetic evolution to search for the best opening *StarCraft 2* strategies (similar to chess openings) to great success [34].

One interesting application is in the evolution of entire neural networks [35][36]. Rather than collecting immense amounts of data for the training of a neural network, evolution can just search for one that is close enough to the game's requirements. This allows for agents that are potentially just as good, but with a lot less effort in data collection.

Overall, the area of genetic programming is very well researched and documented, with much written and experimented. Many have attempted to optimise GA and GP behaviour, for example through controlling bloat [32] or through an empirical evaluation of selection models [37], with varying levels of success. There are few gaps one could find in literature surrounding this topic.

Evolved behaviour trees are common in games research [38] [39] and offer an interesting level of flexibility to an agent's intelligence. There have also been papers on using GA to optimise various real-world elements [40], which leads me to believe there must be ways of translating many of their techniques to games themselves.

2.2.2 Synthetic Problems

To test various implementations of GA, synthetic problems have been developed over time. A couple of simple, but notable ones, are OneMax and Trap.

OneMax

OneMax is a staple of the genetic algorithm test toolkit [41]. It involves the evolution of a vector of bits of length N. The fitness of an individual is equal to the number of times 1 is found in that vector. Alternatively, considering that for this experiment we count lower fitnesses as better, the fitness of an individual is equal to the number of times 0 is found in that vector (see Equation 2.1). The highest fitness is achieved when the vector is completely populated by values of 1.

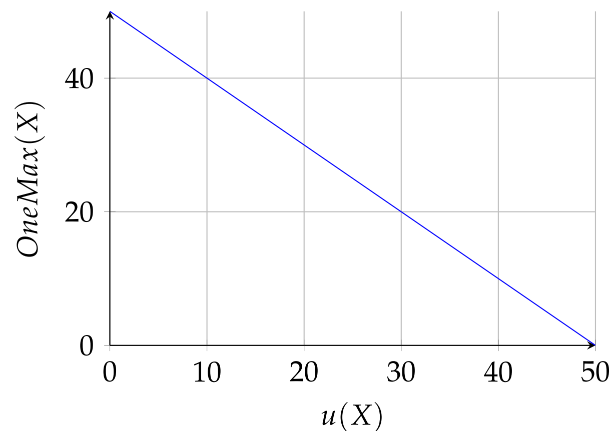

FIGURE 2.1: Fitness landscape for OneMax. Lower values represent better fitness. $u(X)$ represents the number of 1 bits present in the individual X

It is a simple task for a GA to solve [42]. It should also be very easy for a neural network to learn the fitness landscape of OneMax, given its linearity and lack of epistasis between the inputs.

$$OneMax(\vec{x}) = N - \sum_{i=1}^{N} x_i \tag{2.1}$$

The fitness evaluation of OneMax is extremely cheap computationally, thus might be valuable as a quick benchmark for other algorithms attempting to solve it.

Trap Function

Trap functions are commonly used in testing genetic algorithms due to their deceptive fitness landscape. We have made use of the one defined by Deb and Goldberg [43] as DECTRAP.

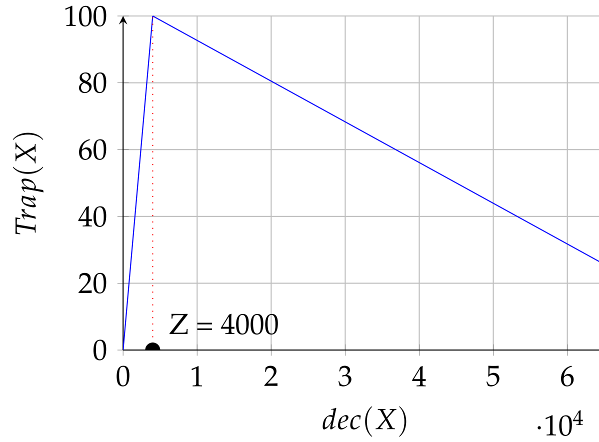

FIGURE 2.2: Fitness landscape of Trap for $Z = 4000$. Lower values are better

The evolved vector is N bits in length and represents a binary number. For the fitness evaluation of Trap, we convert the binary number to decimal (X) and compare it to a value Z. As a result, each individual represents a decimal number from 0 to l, where $l = 2^N - 1$.

$$Trap(\vec{x}) = \begin{cases} a - ((a/Z) * (Z - dec(\vec{x}))), & \text{if } dec(\vec{x}) < Z \\ a - ((b/(l - Z)) * (dec(\vec{x}) - Z)), & \text{otherwise} \end{cases} \tag{2.2}$$

In the Trap fitness equation 2.2, $dec(\vec{x})$ is the decimal representation of the binary vector x.

Concerning the fitness landscape of Trap, the fittest result is when $dec(\vec{x}) = 0$, located in the peak between 0 and Z. The narrower the range between 0 and Z is, the harder it is to correctly estimate individuals in that range.

Similarly to OneMax, Trap is a very cheap evaluation and can be considered a good initial test when checking an algorithm's ability to adapt to deceptive environments.

2.2.3 Multi-objective Optimisation

Very often problems have to look at solutions with different objectives, each important in its own way, where none of them is obviously better than others. Optimising one of the objectives can result in worse results for others and vice-versa, creating a search space where, very likely, there is no objectively best solution. This has led

to research in multi-objective optimisation (MOO) problems [44], resulting in such algorithms as the bat algorithm developed by Yang [45] and using unconstrained elite archives by Fieldsend *et al* [46].

Some examples of interesting multi-objective problems include optimising the process of laser beam cutting [47], generating meal plans that fit multiple dietary and cost requirements [48], as well as better design of biorefineries [49]. All these problems have the same challenge of presenting multiple suggestions to the experimenter, all different, all minimising one or more objectives.

There is no general consensus on the best method to approach optimising such problems, with Marler *et al* arguing that most algorithms have advantages and disadvantages when applied to engineering tasks [50]. Often, many researchers rely on generating a Pareto front of solutions and then focusing on better presenting it to the interested party. One example of such work is Urquhart *et al* and their work on EvoFilter [51].

Filtering through the results of a run, while extremely valuable, still presents what could be a lot of solutions to an interested party. This could prove unintuitive and, at times, overwhelming.

It can easily be argued that many game balance problems, given their reliance on predefined goals and expected "best behaviours", are multi-objective problems. Without a doubt, algorithms designed to tackle those tasks such could be used to find solutions that fit a designer's requirements.

However, after a look over the current state of multi-objective optimisation, as well as weighing in the willingness of game designers to spend time analysing various solutions, we have decided to not focus on how various MOO algorithms would help game balance. That is not to say that this area is not worth exploring in the future.

2.2.4 Surrogate Models and Fitness Prediction

Fitness approximation is an ongoing research area being actively explored [52][53]. Many GA tasks are bottlenecked by expensive fitness evaluations and greatly benefit from accurate estimations.

Neural networks have been used in the past to simulate actual fitness evaluation. It was applied by Johanson and Poli [54] in their evolution of music using genetic programming. Their need for neural network predictions came from the very expensive task of having humans evaluate each individual personally, with no accurate automated method available and a lot of subjectivity involved.

Later work had researchers employ neural networks to estimate the fitness of entire clusters of individuals [55]. The clusters were generated by machine learning techniques, particularly the k-Nearest-neighbour algorithm. Their approach is tested on several synthetic experiments, but no real-world scenarios.

Decision trees have also been paired with GAs to improve the data mining classification task in work by Carvalho and Freitas [56]. They used GAs to evolve rules for a decision tree system, presenting their work on many real-world data sets.

A lot of work has been previously done by researchers on methods to approximate fitness functions during GA runs [57], or even using GA runs to generate new approximation models to then be used in subsequent runs [58].

Recently, researchers studying techniques to treat cancer made use of neural networks as a surrogate model for fitness evaluations in their goal to optimise Intensity Modulated Radiotherapy Treatment beam angles [59]. The genetic algorithm they employ, with the help of the neural network, proves to be successful in finding better solutions compared to traditional methods used in practice. Their implementation uses a pre-trained neural network as a surrogate model. Limitations of this approach is the requirement for expert knowledge and prior data on the subject. Many tasks do not have the privilege of having access to that.

2.3 Machine Learning Algorithms

2.3.1 Neural Networks

Neural networks are a very popular approach to problem solving at the time of writing. They are algorithms meant to emulate, to a certain degree, the structure and functionality of the human brain. Their ability to learn from data and find patterns that would have been missed otherwise is well documented.

First introduced by Rosenblatt in 1959 with the "perceptron" [60], many variations have come to exist now, from convolutional neural networks [61] to long short-term memory neural networks [62] and, now very desirable in both industry and academia, deep neural networks [63]. Each different neural network structure has advantages and disadvantages.

One of the disadvantages of most neural networks is the need for significant amounts of data to train them. This is not always easy, or even possible. Some fields do not have access to much, if any, data that could be used to train a network. In those scenarios, the use of neural networks must be well reasoned, as alternate methods might prove more desirable.

An interesting area of research is the use of genetic algorithms to evolve the parameters or structure of neural networks. This has been done in the past to great success and could be valuable for consideration [64] [65].

Alternatively, instead of evolving just the structure of the network, one could evolve the entire network itself. This has the advantage that it does not require previous data for training, but it does result in a need for an alternate way of assessing the accuracy or fitness of any resulting network. Examples of work in this field include evolving robotic agents [66] and NeuroEvolution of Augmenting Topologies, or NEAT for short [67], a much more advanced variation of this methodology.

2.3.2 Decision Trees

Decision trees, particularly C4.5 decision trees [68], are a classification algorithm often used in machine learning due to their transparency. They analyse already classified training data and build an effective model, in the shape of a tree. Given a new bit of data, one can traverse the tree, following the logical questions posed by it, eventually arriving at a classification.

These algorithms are great because they are fairly fast, even with high amounts of complicated data, but also easy to interpret by both humans and machines. Compared to neural networks, for example, any decisions made on new data by a decision tree are clearly reasoned and one can backtrack through the tree to see exactly the questions asked of the input data to reach that result.

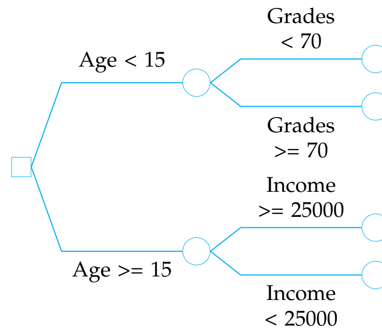

FIGURE 2.3: Example of a simple decision tree

An example of a very simple decision tree can be seen in Figure 2.3.

2.3.3 k-Nearest Neigbours

Similarly to decision trees mentioned earlier, k-Nearest neighbours is an algorithm used for classification and regression [69]. By using already correctly classified examples, the model can approximate any new samples given.

This is a very simple algorithm that, given a value for k, aims to find the closest individuals in the training set to the new sample. Once that is done, the new sample is classified as the majority class among the closest k.

2.4 Games and Computational Intelligence

Using AI technologies to improve games at the design stage is not something completely new, but it is an area that is far from tapped out. Yannakakis and Togelius named this "AI-assisted game design" [70] and it is potentially the best research area for the development of better games [71]. The most common applications are in game level generation [72] [73], with many uses in tools that ease design, such as SpeedTree [74], an automated procedural tree (and, as a result, forest) generator. Another, less common, use case is the generation of entire game rulesets [75] [76].

Currently, research on and using games is still in its relative infancy. Many more sub-fields are being opened for analysis and many techniques from other domains are being adapted to this new area.

As a result of its lack of maturity, much of the research done has not been given many chances to shine in industrial applications, however that is rapidly changing.

Many of the experiments done by researchers are done on synthetic problems, or very old games, as that has allowed them to prototype and test at much higher rates. This is a problem, as it does not give the medium much credibility.

2.4.1 Monte-Carlo Tree Search

MCTS is a relatively recent contender in the AI world, taking the lead in many AI competitions out of seemingly nowhere [77]. It attempts to find good “moves” for a given game state by simulating as many random steps in the future as possible, over many different paths, then choosing what it believes to be the best course of action. It began simply enough by solving board games such as Go [78], but soon enough it established a foothold [79] and got employed for video games as well [80] [81].

One of the algorithm’s main strengths is its flexibility in how much time it needs. It can end its search after 1 second or after 30 milliseconds. The longer it is left to calculate, however, the better the solution it can offer. It does however require, at this time, that there exists a relatively accurate and fast way of simulating what the player’s various actions would do to the game state, as MCTS requires hundreds and thousands of moves to be planned in the future, sometimes within 33 ms, or even less than that. Usually, simulating how a human player would play the game mechanics is not particularly complicated [82]. Simulating how a human player would interact diplomatically with other players is [83] [84].

One area where MCTS is still struggling with is games with much unknown information [85]. This can prove problematic, as many games hide a lot of information from their players.

2.4.2 Goal Oriented Action Planning

GOAP attempts to use multiple behaviour building blocks (such as the action to move, the action to attack, the action to build something, etc), each with its own requirements and potential benefits, to construct an ever-adapting plan that an agent might follow to achieve success. Given a current world state and a desired world state, a GOAP-led player will seek a plan of action that changes the given world state such that it matches its desired one once all actions in the sequence are completed.

GOAP was in use in robotics [86] long before it was considered for commercial games in the 2005 first-person shooter *F.E.A.R.* [87]. Given the fact that the plan is not scripted, but adapts to the ever-changing environment, *F.E.A.R.* enemies show signs of intelligence and clear cooperation with one another. GOAP has since been used in many game-related environments to solve tasks with many agents involved [88] and tasks where information changes rapidly [89].

When it comes to games that have both short-term plan requirements as well as long-term plan requirements, GOAP, potentially combined with other techniques, such as Markov chains [90] or genetic programming, is definitely a leading option for creating intelligent agents.

2.4.3 Automated Game Design

As mentioned previously, there is a great desire for creating completely new games through automated means. In a perfect environment, this means generating mechanics and rules, combining them to form a coherent ruleset, creating some levels that fit the given ruleset, then assessing the game's playability and difficulty. At this time, there is no system that combines all of these elements together, but bits and pieces of each one of them exist. Some of these are briefly reviewed below.

For automatic ruleset generation, Nelson *et al* have defined a formal logic representation of mechanics that can be recombined to generate new games, while offering a good method of assessing how possible it is to win the new game [91]. Schaul, for the purpose of generating a plethora of games for use in general video game research as a result of a common effort from multiple researchers [92], created a description language that can define a wide variety of 2-dimensional games [93]. This general video game language is actively used to test new entries in the General Video Game AI competition and is quite successful.

Automated level generation requires, for quick processing, a better understanding of the mechanics behind the game. Maze-like games, for example, would need the start and exit points to be quite far from each other, yet still accessible through the mechanics the game presents. Once this understanding is put in place, techniques to generate levels exist in plenty. The most popular one, also used in-depth in the

industry, is procedural level generation and evolution. Examples include generating Mario levels [94], content for general platformers [95] and evolving content for the Galactic Arms Race game [96].

2.4.4 Solving Games

Game AI is an area that is bustling with innovation month after month. One of the most efficient catalysts of this progress has been the desire to have AI beat the greatest players of various games, from Checkers, to Chess and *StarCraft*. Most recently, AlphaGo, a Go-playing AI, was able to defeat some of the greatest human players several years before it was expected to be done [97]. Many researchers have made use of various techniques to make their intelligent agents the best at some games, or even really good at many games, such as agents highlighted in general video game AI competitions [98].

Evolutionary algorithms have been applied to game AI as secondary techniques to optimise a different algorithm, such as neural networks, over a longer period of time. Fogel made use of evolution in finding a good set of approximately 1700 weights for its checkers-playing neural network Blondie24 [36]. Similar techniques were later employed by Olesen et al for evolving real-time strategy agents [99]. Genetic programming has also been used thoroughly to generate successful intelligent agents to beat various games [100].

2.4.5 Believable Agents

Currently, most games AI or decision systems are straightforward and quite predictable, due to their heavily scripted nature. Games such as *Europa Universalis* and *Civilization*, while showcasing great AI, make use of transparent number crunching algorithms that add the positives and negatives defined by the programmers together, then say “yes” or “no”. There is little humanity behind these decisions. As an example, if an AI agent wants one of your territories, it will, usually, tunnel vision towards it. More social games, such as Diplomacy, have had bots created for them [101] to varying success, but limitations do exist.

Creating more "human" artificial agents will create more dynamic and unpredictable (yet still smart) opponents, while opening the way for application in other research areas, such as real world politics or economics. It is hard enough for humans to understand how humans should / would behave in various scenarios of conflict [102], let alone a piece of computer code.

Creating believable agents is something that many researchers, as well as game developers, strive for. Tests have been created to assess an intelligent agent's ability to imitate humans [103], agents have been created to negotiate [83] [84], to assess threat [89], to communicate with people [104] and much more. Creating one that would combine all these in a meaningful way is not a simple task, as even the simplest mistake could create confusion not only for the agent, but for the players it is interacting with as well, not to mention the sheer complexity of assuring clean information transmission between the various control modules.

2.5 Balance Through the Use of Computational Intelligence

Computational intelligence can be used to aid in many areas, from medical diagnosis [105], to improving search results or more efficient antenna designs [106]. This sparked the question of whether there is potential for use in helping game designers as well.

"Balance" is a very broad umbrella term for many aspects of game design. The most generic definition, supplied by Schreiber [14], is that game balance "is mostly about figuring out what numbers to use in a game". This can cover procedural content generation [107], parameter optimisation in games, generating new games entirely [108], and potentially more.

Balance is extremely important for multiplayer games, as a feeling of fairness is critical in not only keeping players interested, but also keeping the game entertaining [109]. Some of the greatest multiplayer strategy games in history, such as *StarCraft* and *WarCraft*, have benefited from great balancing [110]. Finding ways to understand balance in the context of players is also interesting. Juul discusses the concept of zero-player games [111] and finds that games and players are intrinsically related.

There has been significant research done on using a designer's requirements for a balanced game and achieving those goals through the use of computational intelligence.

Mahlmann *et al.* [16] worked on matching game elements from Dominion, a popular board game, to create interesting variations of it for players. They successfully used GAs and AI agents to essentially create new games in the Dominion design space. There was no information on whether this approach was applied in any commercial manner.

Work has also been done on balancing the mechanics available in a game of *Top Trumps* in regards to current literature understanding of fairness and excitement [112]. The approach can be useful when designers are uncertain as to what they desire and are willing to accept an academic understanding of "fun". However it might fall flat should that understanding of a game's requirements differ from the designer's vision. Similarly to the previous research presented, no information was given on whether this was adopted by the designers of *Top Trumps*.

Beyer *et al.* present an integrated process for game balancing [19], describing challenges and potential avenues for success, but it comes from the angle of academic research. It does not go in depth on how important a designer's input is and how much can change in the experimentation phase, as well as the structure of both small and large game studios and their internal pipeline for development. The work offers great insight on the challenges presented by automated balancing and offers advice on how such algorithms can be used to facilitate the work of games designers. One very important conclusion from this work is that, at this time, most artificial agents do not play like humans and so the suggestions offered by an automated system based on such agents may or may not work for human players. Thus designers must always double check what they are doing to their games.

Chen *et al.* [113] applied GAs to find balanced character skills for role-playing games. It was done on a very small scale, with a minimal custom game model used and a simple rules-based agent controlling behaviour. The approach does not consider how their methodology would apply to real games or the presence of noise generated by humans or AI agents. As a result, it is unclear how it would behave in a real-world scenario.

Lucas *et al.* used a simple approach to defining parameters a game can have by applying hill-climber algorithms to the General Video Game AI framework [114]. They made strong assumptions as to what makes a set of parameters successful, based on the breadth of agents that can successfully play any given game. This work, while similar to own work published earlier, does not follow the same goals of being designer-focused and applicable to real games, instead aiming to automate the testing of new games. This could be valuable in the process of generating new games.

GAs are capable of finding interesting, often innovative [115][116][117], ways of solving given problems. They do not always generate perfect solutions, but not all tasks require perfect optimality in the first place. Given that games can have many parameters, each with its own limits as to the values it can have, as well as a wide variety of relationships between them, the search space is immense. GAs thrive in these scenarios.

2.6 e-Sports

Competitive video games, also known as e-Sports, are a relatively new endeavour in the digital world. Games, such as *League of Legends*, *DotA 2*, *Counter-Strike: Global Offensive* and more have become mainstream entertainment forms, with people not only playing them, but also becoming spectators for what is the professional scene. Similar to traditional sports (Football, Tennis, etc), these video game competitions attract a massive amount of spectators and, as a result, sponsors.

There has been little in-depth research in this area, with most of it focusing on generating competitive agents [118], a trend shared by a lot of agent research in games. Some of the works that can be found are purely descriptive [119] [120] [121], while authors such as Rambusch et al are trying to describe the cultural and economical implications of this new world [122]. Some economics researchers are also attempting to better understand e-Sports as a market [123].

There is virtually no research on considering e-Sports as a potential new grounds for AI research. This is an area where the best AI technologies would have the hardest competition to face, similar to how Chess masters used to be the final hurdle for

AI in the past. There are cases where even tool-assisted players (scripters, players that cheat to gain an advantage) lose because of their weak overall decision making, which is an interesting topic for AI research, as players usually have very little time to decide how to act in the heat of the moment.

Automated game balancing can greatly benefit e-Sports, as an ever shifting state of balance is what they need to remain both interesting and fair.

2.7 Code

There are many libraries available for implementing the various machine learning algorithms used throughout this thesis. After some exploration, and accounting for personal preference, the language used was almost exclusively C#. The Accord Framework [124] was used for most machine learning requirements (all except GAs) and for all statistical calculations.

Chapter 3

Games Targeted During the Research

3.1 *Ms. Pac-Man*

3.1.1 Introduction

Ms. Pac-Man is a single player game played on a 2-dimensional board, where the player controls the "PacMan" and has the goal of collecting as many points as possible while navigating a maze-like map. There are four ghosts opposing the player. An example of the game as it is being started can be seen in Figure 3.1.

Points are received whenever the PacMan eats a pellet or power pill, or when a "scared" ghost is eaten. Ghosts become scared temporarily after the PacMan eats the aforementioned power pills.

The ghosts follow predictable strategies, but their movements can be random

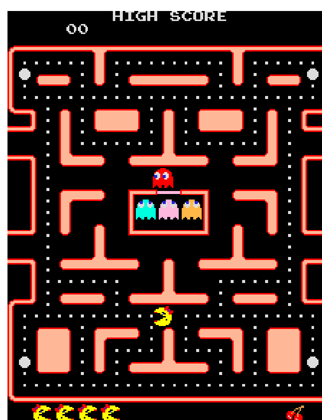

FIGURE 3.1: Screenshot from a level in a game of *Ms. Pac-Man*

at times, making for an unpredictable game every time it is played. This means deterministic AI agents will not always achieve the same score in consecutive games.

The goal when playing *Ms. Pac-Man* is to survive for as long as possible while collecting as many points as are available. If all pills from a level are collected, the game changes levels to a new one, with new geometry, but the same enemies and goals.

3.1.2 Interfacing with *Ms. Pac-Man*

To simulate the game, an adaptation of existing code by Shelton was used [125] which is available on GitHub [126]. This is a faithful recreation of the original game, with the added ability to be run without any graphical rendering or pause between frames. As a result, hundreds of games can be simulated every second when quick agents control it. By having access to the source code, we were able to directly alter the game's parameters, bypassing any requirement for editing running memory or binary files. All agents used had direct access to the game's state at any point.

3.1.3 Research on *Ms. Pac-Man*

There are many AI agents available for *Ms. Pac-Man*. Most of these agents were developed to 'beat' the game, where the most important aim was to get as many points as possible. The strongest one at the time of writing is based on a hybrid reward architecture [127], with strong showings from Q-learning [128], Deep Q-Networks [129] and Monte-Carlo Tree Search [130]. Success has also been had with neural networks [131] [132] [133].

There has also been some research on generating new levels, of varying difficulty, for the game through procedural content generation [134]. Given the game's simplicity, *Ms. Pac-Man* is a good practice case for algorithms before being attempted on more complex ones.

FIGURE 3.2: Screenshot from a game of *StarCraft* with most UI elements hidden

3.2 *StarCraft*

3.2.1 Introduction

StarCraft is a real-time strategy game from Blizzard Entertainment, known for its e-sports environment, but more importantly, its AI development community [135]. The game involves three distinct races, the Terran, the Protoss and the Zerg, each with its own available units, strategies and strengths. It is played from a top-down perspective, with the player managing both an economy and an army. An example of this can be seen in Figure 3.2.

The game is played in real-time, meaning that both players must act with some degree of urgency in issuing their commands, while paying attention to multiple areas of the game. These areas can include the workers that gather resources, the military unit production facilities, the environment and the natural defences offered by it, such as cliffs or bridges, and access areas through which the enemy might make an attack. Each player can only see the parts of the map where it has units or buildings, thus, for most of the game, does not see what their opponents are doing. This is a very important element of the game, as a player will not know what strategy their opponent is employing for quite some time and will have minimal time to adapt to it should it be strong against them.

3.2.2 Interfacing with *StarCraft*

BWAPI [136] is an open-source piece of software that allows third-party applications to access and interact with *StarCraft* in real time, resulting in the possibility of creating AI agents for the game. These agents can either play with game information supplied by BWAPI, or assess screen information on their own directly. Most agents developed take advantage of the data supplied to them by BWAPI.

BWAPI required a couple of patches that would allow multiple independent instances of the game to run in parallel, a critical requirement for the computationally expensive studies described in later chapters.

ChaosLauncher [137] is a tool written in Pascal that injects third-party libraries, such as BWAPI, into *StarCraft*'s executable, to facilitate their functionality. Similarly to BWAPI, changes to the original code were required to allow multiple instances of *StarCraft* to run at the same time.

With the changes mentioned above, multiple games of *StarCraft* can be played in parallel, each with its own AI agents and different maps.

If changing the game's parameters is needed, these are sent to a custom C++ application that alters the corresponding variables in the game by adding the displacement to the original value. This is done by replacing several binary values in the targeted *StarCraft* map file and repacking it properly, to allow it to be read by the game. To achieve this, the open source StormLib library [138] was used to unpack and repack maps, modified with own code for replacing the game's parameters. The game itself remains unchanged, but the map file tells it to use the new set of parameters rather than the default ones.

A graph of the entire pipeline is presented in Figure 3.3.

3.2.3 Research on *StarCraft*

Again, the vast majority of research on the game is to generate the best possible agents for playing the game. However, as opposed to *Ms. Pac-Man*, there is no one algorithm that has "solved" the game. Right now, even the best agents are unable

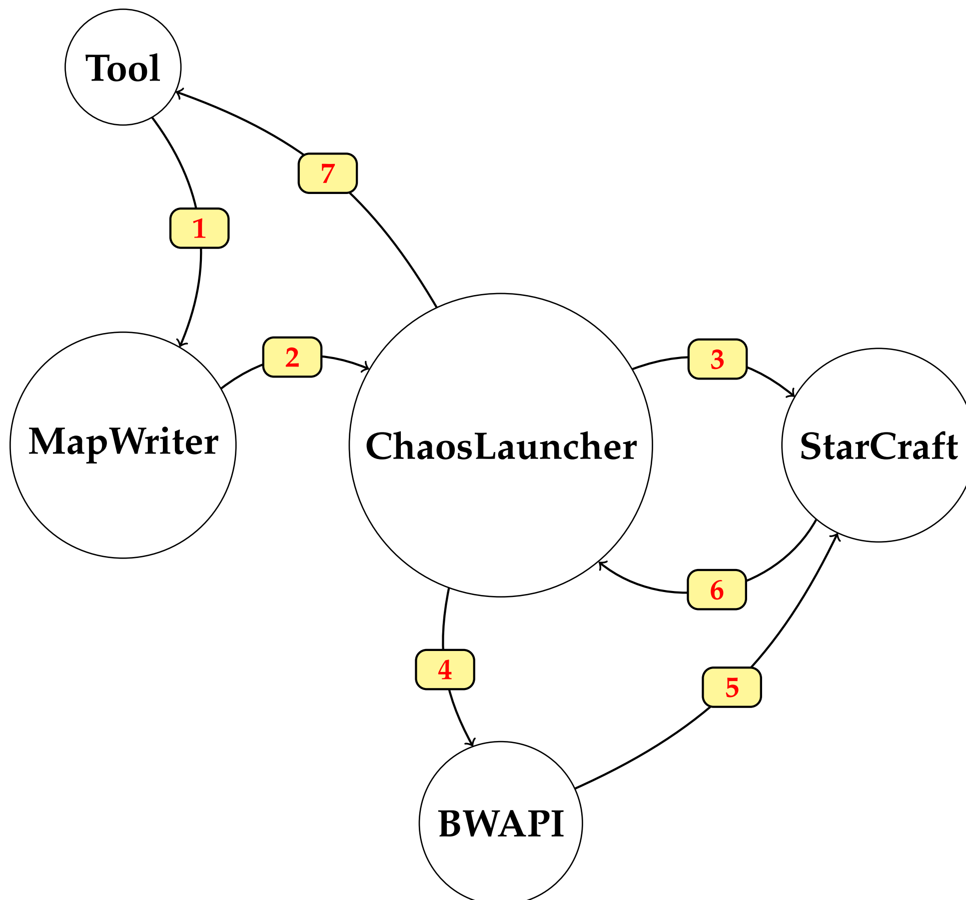

FIGURE 3.3: Pipeline of integration with *StarCraft*

FIGURE 3.4: Screenshot from a game of *TORCS*

to compete with most of the high-tier human players [139] [140] [141] [142]. García-Sánchez *et al.* [143] were successful at evolving strategies for an AI to play the game at a higher level than many other artificial agents using GAs.

3.3 *TORCS*

3.3.1 Introduction

TORCS is an open source racing game simulator [144]. It is extremely configurable, with almost every element of the game being modifiable without recompiling the source code. Another interesting element is its lack of stochastic elements. Unless the driver has random decisions, a race with the same car on the same track will yield the same lap performances every time. A screenshot of the game can be seen in Figure 3.4.

Many elements of the game could be considered for changes, from the layout of the tracks, to the technical specifications of the cars, to way artificial agents drive in the game.

3.3.2 Interfacing with *TORCS*

To send and receive data between the game and any algorithms developed, we wrote a small bridge application to act as an interface. This is later described in Section 4.4.

FIGURE 3.5: Screenshot from *ComPet* presenting one of the available player pets

All this application does is connect to a message queue server and wait for commands. These commands, queued by the algorithm, contain a specification file complete with values for the enabled parameters. This specification file is one built using the systems later described in chapter 4. For each of those parameters, the bridge locates them in the appropriate XML file within the *TORCS* installation (based on their name), modifies their values, plays the games as needed, then reports back to the message queue server (and, by extension, the algorithm) the resulting metrics.

Beyond making temporary changes to the values in the game's XML files, no changes are made to the game's mechanics or binary structure.

3.4 *ComPet*

3.4.1 Introduction

ComPet is a commercial turn-based strategy game developed by MindArk [145]. It gives the player control over a roster of characters called pets, each with its own attributes (such as initiative or endurance), its own experience level and various combat abilities. An example of such a pet can be seen in Figure 3.5.

The game has two main gameplay loops. The default one is where the player battles the various beasts in the game's campaign mode in one-on-one combat with one of their own pets (see Figure 3.6). This campaign mode is increasingly more

FIGURE 3.6: Screenshot from *ComPet*'s battle mode

difficult and requires the player to adapt to different strategies or even wait until their pets are strong enough to handle the challenge.

Battling other players in 1-on-1 combat is the second gameplay loop. This mode has two goals: gathering experience points to strengthen the player's pets, and becoming a better ranked player on the leaderboards.

The combat is turn based, with a statistic called 'initiative' deciding which pet goes first. Each pet is allowed to use an ability each turn, ranging from direct damage abilities, to healing and skills that improve your statistics or decrease the opponent's statistics. Once a pet has reached 0 or fewer health points, the other pet is declared victorious.

The other interesting element of the game is its campaign mode. Given that this is a player-versus-player (PvP) experience at its core, the developers want the players to spend as much time there compared to time spent in the campaign. As a result, an interesting target of balance is to optimise the difficulty of the game such that players have to further improve their pets in the PvP arenas before they can continue through the campaign.

3.4.2 Interfacing with *ComPet*

For easier experiments, the game's campaign mode was abstracted into the concept of a 'gauntlet'. A gauntlet represents a series of beasts to be fought, as well as a single

pet the player starts with. The role of the player is taken by an AI agent developed by MindArk. An example of a file describing a gauntlet can be found in Appendix A.

This AI agent is rule-based with rules derived from real-world play tactics. It behaves admirably given the environment and is an acceptable approximation of human players for the purposes of any experiments.

To simulate actual gameplay, one would start a fight with the first beast in the series. If victorious, the player's pet is rewarded with experience points, levelling them up if enough points were collected. Then, the pet would attempt to fight the next beast in the series. If unsuccessful in a fight, the player would focus on the most recently defeated pet (or the first one if that is impossible) for several fights, hoping to gather enough new abilities and attribute points to try again.

Once the pet successfully defeats the final beast in the series (or a maximum number of fights has been reached), all the relevant metrics from every single fight are collected. These metrics can be used either for manual analysis by a designer, or for automated design methods, such as those presented in future chapters.

3.5 Genesis Dei

During the first year of research, one area of interest was the development of intelligent agents for playing diplomatic strategy games. This work resulted in the development of a game, *Genesis Dei*.

While the game was under development for several months, the scope of the research changed and the project was put on indefinite hiatus. The findings from the final state of that research can be seen in Appendix B.

Chapter 4

Balance Specification Language

4.1 Introduction

The literature review (Chapter 2) has highlighted one of the main issues present in game balance research at the time of writing: there is a gap between industry and academia in the sense that industry does not use many academic algorithms, while researchers are often unable to use real world games for their experiments.

In an effort to make a first step towards bridging this gap, in this chapter I describe a language for game balance specifications, the development of which was triggered by a cooperation with a commercial games studio, MindArk Sweden. The language allows designers to define what can be changed in their games by game-balancing aglorithms, as well as how to evaluate the degree of success of any changes in their game that an algorithm proposes. This decouples the algorithms to balance the games from the games themselves, which has the advantage that once developers have written a specification file for their game, they open it up to a plethora of search and optimisation algorithms, including any new developments in academia.

This does require a small amount of extra work from the developers of a game, as said game needs to understand how to actually apply any changes it is given by an algorithm. This can be done by having an extra application dedicated to receiving the proposed changes, applying them to the game, then running the new version to collect results.

The specification language is, essentially, a structured JSON file, making it really

```
{
  "name": "F_cars.F_car1-ow1.f_car1-ow1.S_Car.A_mass.T_val
 ",
  "rangeMin": -300,
  "rangeMax": 300,
  "rangeAccuracy": 1,
  "minimise": "minimise",
  "weight": 1,
  "enabled": true
}
```

FIGURE 4.1: Sample of the specification language presenting a list with a single parameter for *TORCS*

easy to read by both humans and software programs, while being easy to manipulate.

This language was also partly inspired by work done by groups such as Metaheuristics in the Large [146] and work by Swan *et al* [147].

4.2 Describing Elements to be Changed

A "parameter" is a variable in a game that has been deemed valuable for balancing. An example of a parameter can be seen in Figure 4.1.

For the designer a parameter's name is valuable, as it allows easy tracking and understanding of any changes proposed by the underlying algorithms. The name could be purely cosmetic, or it could be used to identify deeper elements in a game's design. In the given example, a file defining the elements to change for an experiment using *TORCS*, the name was used to describe the path to the appropriate XML file in the game's installation folder, as well as section within said file, where the value to be changed is located. This is similar to a specialised interpretation of an XML XPath.

From the point of view of an algorithm, what is being changed in the game is not important at all. The specification makes the assumption that the order in which parameters are defined in a parameter list is also the order they will be passed back to the game. What is important is knowing what values the parameters are allowed to have. This includes the range of values, "rangeMin" and "rangeMax", as well as their arithmetic precision, "rangeAccuracy".

An extra element available to the designer is the "enabled" flag. If the flag is set to "false", the parameter is not changed at all and is ignored by the algorithm. If it is set to 'true', the parameter gets changed as normal.

Another option is the possibility of defining a parameter as not just a single value, but a list of values with a given length, all within the same ranges defined earlier. Should it be important, the values can be forced to be distinct, as this can be used to generate permutations. This option is available by setting the "listsize" property of the parameter.

When making changes to a game, a designer might be interested in making changes as small as possible. This is possible by setting the "minimise" flag to "minimise" and the property "weight" to a non-zero value. This tells the underlying algorithm to consider the magnitude of parameter values when calculating a final score for the change set, with an importance value weighted by the weight option ($W_{\Delta i}$).

$$Score_{Params} = \sum_i |\Delta_i| \times W_{\Delta i} \tag{4.1}$$

In the function above, Δ_i represents the i_{th} parameter. Also the summation is only over parameters that were flagged as enabled.

4.3 Evaluating Success

Making changes to a game and then playing the altered game is not enough. Once the game (or games, should one be not enough to assess success) has ended, the designer should have access to various values defining that play session. These could be anything from how often each player won, to how many times a weapon was used, to how long it took someone to solve a puzzle. We call these values 'metrics'. Metrics are how the game communicates with the algorithms. An example of a list of metrics (with one element) can be seen in Figure 4.2.

To assess the success of a set of changes, a method of quantifying how close they are to optimal is needed. This is done by comparing the metrics generated after play to what the designer has deemed as optimal metrics. These comparisons are what the specification language defines as "evaluators". Each evaluator assesses a

```
"metrics": [
    {
        "name": "S_E-Track 6.S_Results.S_Qualifications.
    S_Rank.S_1.A_best lap time.T_val",
        "type": "Double"
    }
]
```

FIGURE 4.2: Sample of the specification language presenting metrics for *TORCS*

```
"evaluators": [
    {
        "name": "LapTimeDirtTrack",
        "type": "AverageEvaluator",
        "metric": "S_E-Track 6.S_Results.S_Qualifications.
    S_Rank.S_1.A_best lap time.T_val",
        "target": 70,
        "weight": "100",
        "enabled": true
    }
]
```

FIGURE 4.3: Sample of the specification language presenting a list with a single evaluator for *TORCS*

single metric, but a metric can be assessed by multiple evaluators. An example of an evaluator can be seen in Figure 4.3.

Each evaluator has access to three values: a target optimal (*Target*) value, an optional extra parameter, used by some evaluators (*OptionalParam*), and a weight (*Weight*), used to define relative importance. They are passed a list of values (*Values*) that can be as small as a single element.

Similarly to parameters, evaluators can be marked as enabled or disabled.

4.3.1 Available Evaluators

- The simplest evaluator is the "average" evaluator. It receives the list of values of a metric, gathered in one or multiple games, computes the average of those values, then compares the result to the given desired value. The resulting score is the absolute difference between the optimal value and the average of the

metric, multiplied by the weight. Formally:

$$Eval_{Avg} = |Mean(Values) - Target| \times Weight \quad (4.2)$$

An example of such an evaluator could be that, after making changes to a car's engine, the designer wants it to achieve, on average, a 5 second faster lap when driving a given race track 20 times in a row.

- Another evaluator is the "median" evaluator. It is used identically to the previous one, except the mean is replaced by the median of the metric being evaluated. Formally:

$$Eval_{Median} = |Median(Values) - Target| \times Weight \quad (4.3)$$

This evaluator could be used, for example, when a designer wants an agent to achieve a given median score, in situations where the edge cases are very high or very low, skewing the average.

- The "standard deviation" evaluator computes the standard deviation of a list of values, then compares it to the desired value. The resulting score is the absolute difference between the optimal value and that standard deviation, multiplied by the weight, that is:

$$Eval_{StdDev} = |StandardDeviation(Values) - Target| \times Weight \quad (4.4)$$

This can be used in scenarios where a certain consistency of results is wanted, such as wanting scores achieved by an AI agent to be as close as chosen by the designer.

- The "at least proportion" evaluator compares a list of values to see how many of them are above a given value (passed via the *OptionalParam* argument), then compares that number to the desired value. The number of elements above the threshold is stored as a proportion of the total numbers in the list in *Ratio*. Should the ratio be higher than or equal to the target value, the score is 0 (a perfect result). Otherwise, the resulting score is the absolute difference between

the target value and the average of the metric, multiplied by the weight. The function can be represented as:

$$Eval_{AtLeast} = \begin{cases} 0 & \text{if } Ratio \geq Target \\ (Target - Ratio) \times Weight & \text{otherwise} \end{cases} \tag{4.5}$$

$$Ratio = \frac{1}{n} \times \sum_{i} \delta(S_i \geq OptionalParam) \tag{4.6}$$

An example of this evaluator could be that a designer would want players to collect at least 50 coins each level, with no upper limit. Score would be penalised should fewer coins be found.

- The "at most proportion" evaluator is identical to the previous one, except the desired value represents the upper limit, instead of the lower limit.

$$Eval_{AtMost} = \begin{cases} (Ratio - Target) \times Weight & \text{if } Ratio \geq Target \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

4.3.2 Final Score

After having calculated the result for each enabled evaluator, the final score is simply the sum of all the results. Similarly to other work in the area, lower values are better, with a total score of 0 representing a perfect solution.

The reason behind using a simple weighted sum instead of more complex multi-objective approaches, such as those utilised by Gravina and Loiacono [148], is its simplicity. While the sum could allow for some objectives to dominate others and stop the underlying algorithm from exploring the search space more efficiently, it is a lot easier to explain and visualise. A straightforward system where someone untrained in machine learning optimisation can easily change the numbers and understand their impact is much more likely to be used outside of research.

```
{
  "metrics": {
    "S_E-Track 6.S_Results.S_Qualifications.S_Rank.S_1.A_best
    lap time.T_val": "78.33",
    "S_Dirt 3.S_Results.S_Qualifications.S_Rank.S_1.A_lap
   times.T_val": ["63.96", "64.12", "65.51", "63.99",
   "64.99"]
  }
}
```


FIGURE 4.4: Sample of JSON data sent by a game describing the requested metrics after completing play

4.4 Communicating with the Games

While parameters and metrics are how the algorithms and games trade information from a logical point of view, a bridge is required to actually pass the digital data. Our framework currently defines a method that invokes an external command (such as running an executable or a script), as well as a distributed solution using message queues.

When invoking the external command, the game designer has access to several variables, such as: a comma separated list of the parameter values, a path to the JSON specification file and a random seed (a value that can be used to make it such that random number generators return the same sequence of numbers, valuable when attempting to replicate results). The external command responsible of applying the relevant changes to the game's parameters, then launch the game (or games) and record the required metrics.

The message queue solution simply sets the 'value' property of each enabled parameter, then sends the entire new JSON file to the message queue server. It is then distributed to a bridge script or application, as described in the previous paragraph, for playing. Once done, the bridge returns a JSON file with the appropriate metrics and their values. This assumes the bridge can read the JSON file and apply the appropriate changes. An example of the metrics a game could return can be seen in Figure 4.4.

These are not the only ways one could balance their game. Should the default options not suffice, a "custom" section is available for a designer to define extra

parameters that are relevant to them. These could include such elements as how many games to run for each set of changes, or which maps to be used for a scenario.

4.5 Extended Backus-Naur Notation

This notation is an extension of the one originally created by Crockford for the application/json media type [149].

TABLE 4.1: Extended Backus-Naur Notation for the specification language used in defining balance tasks.

1	JSON-text	::==	begin-object cat-name value-separator cat-params value-separator cat-metrics value-separator cat-evals value-separator cat-ga value-separator cat-bridge value-separator cat-custom end-object
2	begin-array	::==	ws ‘[’ ws
3	begin-object	::==	ws ‘{’ ws
4	end-array	::==	ws ‘]’ ws
5	end-object	::==	ws ‘}’ ws
6	name-separator	::==	ws ‘:’ ws
7	value-separator	::==	ws ‘,’ ws
8	ws	::==	*(%x20 / %x09 / %x0A / %x0D) (Whitespace)
9	value	::==	false / null / true / object / array / number / string
10	false	::==	‘false’
11	null	::==	‘null’
12	true	::==	‘true’
13	object	::==	begin-object [member *(value-separator member)] end-object
14	member	::==	string name-separator value
15	array	::==	begin-array [value *(value-separator value)] end-array
16	number	::==	[minus] int [frac] [exp]

```
17   decimal-point   ::==  ‘.’
18   digit1-9        ::==  ‘1’-’9’
19   e               ::==  ‘e’ / ‘E’
20   exp             ::==  e [ minus / plus ] 1*DIGIT
21   frac            ::==  decimal-point 1*DIGIT
22   int             ::==  zero / ( digit1-9 *DIGIT )
23   minus           ::==  ‘-’
24   plus            ::==  ‘+’
25   zero            ::==  ‘0’
26   string          ::==  quotation-mark *char quotation-mark
27   char            ::==  unescaped / escape (%x22 / %x5C / %x2F /
                           %x62 / %x66 / %x6E / %x72 / %x74 / %x75
                           4HEXDIG )
28   escape          ::==  ‘\’
29   quotation-mark  ::==  ‘”’
30   unescaped       ::==  %x20-21 / %x23-5B / %x5D-10FFFF
31   cat-name        ::==  “name” name-separator value
32   cat-params      ::==  “parameters” name-separator begin-array [
                           param *( value-separator param ) ] end-array
33   param           ::==  begin-object [ param-property name-separator
                           value *( value-separator param-property name-
                           separator value ) ] end-object
34   param-property  ::==  “name” | “rangeMin” | “rangeMax” |
                           “rangeAccuracy” | “minimise” | “weight” |
                           “custom” | “enabled” | “listsize”
35   cat-metrics     ::==  “metrics” name-separator begin-array [ metric
                           *( value-separator metric ) ] end-array
36   metric          ::==  begin-object “name” name-separator value
                           value-separator “type” name-separator metric-
                           type end-object
37   metric-type     ::==  “Double” | “List”
```

38	cat-evals	::==	“evaluators” name-separator begin-array [evaluator *(value-separator evaluator)] end-array
39	evaluator	::==	begin-object [eval-property name-separator value *(value-separator eval-property name-separator value)] end-object
40	eval-property	::==	“name” | “metric” | “target” | “type” | “weight” | “enabled” | “optionalparam”
41	cat-ga	::==	“gaparams” name-separator begin-object [member *(value-separator member)] end-object
42	cat-bridge	::==	“bridge” name-separator begin-object [bridge-property name-separator value *(value-separator bridge-property name-separator value)]
43	bridge-property	::==	“executable” | “password” | “port” | “queue-name” | “server” | “type” | “username” | “amqpurl”
44	cat-custom	::==	“custom” name-separator begin-object [member *(value-separator member)] end-object

Chapter 5

Genetic Algorithms for Video Game Parameter Balance

5.1 Introduction

As described in Chapter 2, there is untapped potential in exploring automated game balancing. This research, if successful, could prove valuable for both academia and game studios.

As a result of surveying the existing few approaches, as well as achieving a better understanding of various machine learning algorithms, the first methodology considered is to use genetic algorithms to search the balance space. This requires several elements: one or more games to test this on, a method of representing balance tasks, a measure of assessing success and an appropriate adaptation of genetic algorithms.

For this work we made use of 3 games: *Ms. Pac-Man*, *StarCraft* and *TORCS*. Most of the preliminary work was done on *Ms. Pac-Man*, with the other 2 acting as significantly different games to test the versatility of our approach.

This chapter describes the environments and methodologies adopted for each one of the 3 games used in the experiments. All were done using the specification language described in chapter 4.

5.2 *Ms. Pac-Man* Experiments

5.2.1 Environment

For the *Ms. Pac-Man* experiments, the game's rules were altered slightly as follows. Instead of aiming for a high score, the game is "won" once the player achieves 1500 points. As a result, players can have a win-rate associated to their performance alongside their average score and performance can be evaluated based on that.

Even with a deterministic agent, the stochastic nature of the game leads to the scores to be very varied. As a result a number of games have to be played for each individual, storing all the scores the agent achieved. Any scores above or equal to 1500 are counted as a win, while any scores below that are considered losses. That represents the individual's win-rate. The method of choosing the number of games to be played for each individual is described later on in this section.

For these experiments, to see if this method is applicable to the game, a quick rules-based agent was used, as it can represent a lower-level human player and many more games can be simulated in a short amount of time. This rules-based agent is a greedy-random agent based on work done by Thompson *et al.* [150], with modifications by Shelton [151].

As shown in Figure 5.1 that reports the result of 10,000 game simulations, given this change to the game's mechanics, the rule-based agent "wins" the game only 19.6% of the time. The distribution of these scores presented in the figure was generated using the Parzen window method, using a Gaussian kernel estimator.

Given this win-rate of 19.6%, which would mean the agent wins the game approximately 1 in 5 times, we decided, as game designers, to aim at making the game easier. An easier game would allow the agent to achieve a higher win-rate. For the purposes of these experiments we decided that a win-rate of 50% (or 0.5) would be the target, as that would make the game a lot easier for the agent, but still challenging to some degree.

The elements of the game to be changed are the speeds at which the entities in the game move. For the PacMan, there is only one speed to change, as there are no elements in the game that modify it. However, when the PacMan eats a power pill, the ghosts have their speed temporarily changed. In the base version of the game,

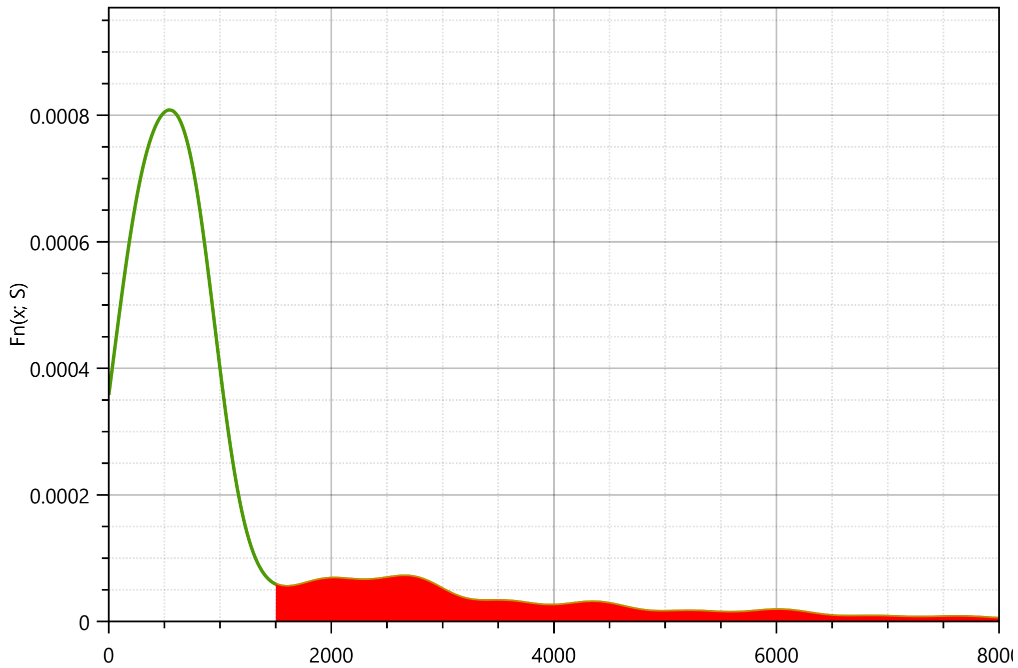

FIGURE 5.1: Distribution of scores achieved by a rule-based agent in *Ms. Pac-Man* (see text). The area highlighted in red represents scores above 1500.

that new speed is slower, but we removed this constraint from the balancing process. As a result, there are 2 speeds to alter for each of the 4 ghosts. All parameters can be seen in Table 5.1. For simplicity, all parameters were considered equal in importance, thus given the same weight C_Δ (see Section 4.2). The reason a numeric value is not stated in the above mentioned table is due to the fact that we experiment with different values of C_Δ.

TABLE 5.1: *Ms. Pac-Man* parameters to be changed, their displacement ranges and their decimal accuracy

	Parameter	Min	Max	Accuracy	Weight
Δ_1	PacMan's speed	-3	+5	10^0	C_Δ
Δ_2	First Ghost's chase speed	-3	+5	10^0	C_Δ
Δ_3	Second Ghost's chase speed	-3	+5	10^0	C_Δ
Δ_4	Third Ghost's chase speed	-3	+5	10^0	C_Δ
Δ_5	Fourth Ghost's chase speed	-3	+5	10^0	C_Δ
Δ_6	First Ghost's flee speed	-3	+5	10^0	C_Δ
Δ_7	Second Ghost's flee speed	-3	+5	10^0	C_Δ
Δ_8	Third Ghost's flee speed	-3	+5	10^0	C_Δ
Δ_9	Fourth Ghost's flee speed	-3	+5	10^0	C_Δ

TABLE 5.2: *Ms. Pac-Man* metrics used in evaluation alongside their desired values and weights

	Name	Desired	Evaluator	Weight
W	Win-rate achieved by the agent	0.5	Average	C_W
Δ_P	Sum of parameter absolute displacements	0	Minimise	C_Δ

Fitness Evaluation

From a game designer's standpoint, the end goal is to control two features: how often the main agent wins and how big the changes are to the original game parameters. Smaller changes to the original game parameters are required due to the desire to improve the existing version of the game as opposed to generating wildly different variations. Both metrics can be seen in Table 5.2.

So, fitness is a multi-objective function of the win-rate and the absolute difference between the default game parameters and the newly evolved ones. The closer the win-rate is to the desired one, the better the fitness. Also, the smaller the difference between the original parameters and the evolved parameters, the better the fitness, as it is preferred to have incremental changes rather than big ones. This is the immediate result of wanting to optimise an existing game rather than creating a new one from existing mechanics.

Formally, the fitness function can be written as:

$$Fitness = W + \Delta_P \tag{5.1}$$

$$W = |WR - DWR| \times C_W \tag{5.2}$$

$$\Delta_P = \sum_{i=0}^{n} |\Delta_i| \times C_\Delta \tag{5.3}$$

In the win-rate optimisation function W, WR = Win-Rate (from 0 to 1), DWR = desired win-rate and C_W = win-rate bias factor. In the parameter optimisation function Δ_P, Δ_i = difference between the original ith parameter and the evolved ith parameter and C_Δ = parameter weight.

For the remainder of this chapter, $DWR = 0.5$, as the desired win rate is 50%. Our approach considers smaller fitness values to be better, with 0 representing a perfect

TABLE 5.3: Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing mutation rates. As a result of crossover, mutation and elitism percentages adding up to 100%, reinitialisation is not used

Pop. Size	Crossover	Mutation	Mutation Rate	Elitism
50	40%	40%	5%	20%
50	40%	40%	10%	20%
50	40%	40%	20%	20%
50	40%	40%	30%	20%
50	40%	40%	40%	20%
50	40%	40%	50%	20%

solution.

The reason Δ_P can be simplified to a sum multiplied by C_Δ in Equation 3 that all parameters have the same weight for these experiments. Should the designer consider some parameters more valuable to minimise than others, then that could be reflected in the fitness function.

Genetic Algorithm

The evolutionary algorithm employed is a variant of a generational GA with two-point crossover, a specialised mutation operator and elitism. If the crossover percentage, mutation percentage and elitism do not add up to 100%, the remaining amount is used to generate new individuals through reinitialisation [152].

The mutation operator was applied with a (per allele) mutation rate of 50% (meaning that on average 50% of the elements of an individual would be mutated). At each application of the operator, a displacement is randomly generated within the range of acceptable values for that allele (as presented in Figure 5.1) and added to the corresponding parameter value.

Experiments used tournament selection with a tournament size of 6.

Each evolved vector was constrained to only contain values between those presented in Table 5.1. Values outside of these ranges are not realistic for the tested scenario.

Several batches of experiments were undertaken. The first batch explored the impact of the mutation rate on performance and the various configurations of parameters can be seen in Table 5.3. The second batch tested whether reinitialisation

TABLE 5.4: Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing impact of reinitialisation

Pop. Size	Crossover	Mutation	Elitism	Reinitialisation
50	40%	40%	20%	0%
50	30%	30%	20%	20%
100	40%	40%	20%	0%
100	30%	30%	20%	20%

TABLE 5.5: Genetic algorithm parameter sets tested for *Ms. Pac-Man* when testing performance variation of population size

Pop. Size	Crossover	Mutation	Elitism
10	40%	40%	20%
20	40%	40%	20%
50	40%	40%	20%
100	40%	40%	20%
200	40%	40%	20%

was beneficial, using the best configuration found in the previous batch as a starting point. Values tested can be seen in Table 5.4. The final batch looked at how different population sizes altered GA performance. Each one of those is presented in Table 5.5.

Each GA run would evolve until a total of 2000 individuals were evaluated. This was the budget allocated for each run.

Choosing Weights

Selecting the values for the weights is very important, as they will define how the solution space is explored.

For these particular experiments, where there are only 2 weights (C_Δ and C_W), it is obvious there is only one degree of freedom in the fitness function. As a result we can define $Ratio = \frac{C_W}{C_\Delta}$, give C_Δ a constant value, then write $C_W = Ratio * C_\Delta$.

By changing *Ratio*, we can observe how the fitness landscape would change relative to the different values the metrics can have. A way of looking at weight selection is to explore how much of a change in each collected metric would be needed to alter the fitness of an individual by a certain amount. Given values for *Ratio*, one can easily glance at the relative value of each evaluator.

TABLE 5.6: Amount by which metrics have to change to increase their respective fitness objective by 100 in the *Ms. Pac-Man* game balancing experiment, with respect to several configurations of C_W and C_Δ, as well as what the fitness of the parameter set of only 0s would be

Ratio	C_W	C_Δ	*WR* change	Δ_P change	Baseline fitness
1	100	100	±100%	±1	30.4
2	200	100	±50%	±1	60.8
5	500	100	±20%	±1	152.0
10	1000	100	±10%	±1	304.0
20	2000	100	±5%	±1	608.0
50	5000	100	±2%	±1	1520.0

Looking at the changes in metrics required to increase fitness by a same amount, available in Table 5.6, it can help a designer decide how much importance they want to give each of the 2 objectives. The baseline fitness described is the fitness an individual proposing no changes to the game (thus having all zeroes for its parameters) would achieve.

Visual graphics can further elaborate on the impact the weights have on the fitness score. Figure 5.2 shows the fitness spread when changing the *Ratio*, over all possible values of the winrate (*WR*), when $\Delta_P = 7$.

At a glance, it looks like values for *Ratio* below 5 are not entirely realistic, as they put very little relative value on the win-rate. It is reasonable to assume that most solutions will try to minimise displacements to parameters instead of minimising the win-rate objective almost every run, not actually solving the problem presented.

The flipside would seem true for very high values of *Ratio*. One of the experiments presented later in this section attempts to explore these cases.

Once a designer has chosen the weights for the various objectives, a baseline fitness value can be computed. This is the fitness the unchanged version of the game achieves. For these experiments, it can be seen that the vanilla version of the game has a Δ_P objective fitness score of 0, while the W objective is dependent on the ratio and the $WR = 0.196$ described earlier in the chapter , as well as $DWR = 0.5$. The value of the total baseline fitness score, for each of the tested configurations, can be found in Table 5.6.

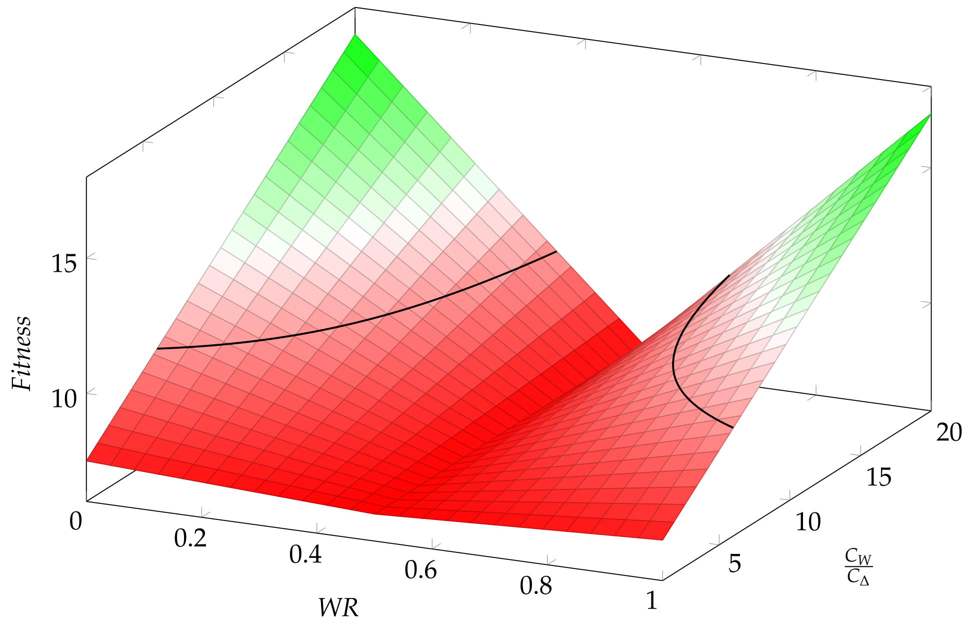

FIGURE 5.2: A mapping of fitness values depending on the win-rate metric (*WR*), which is dependent on the actual simulation, and the *Ratio*. This shows how much the ratio impacts the gap between good individuals (with *WR* close to 0.5) and bad individuals

Selecting the Number of Games Played

When choosing the number of games to play several factors must be considered. The immediate problem with selecting a high value for this is the proportional increase in computation time. Twice as many games results in twice as much time required to achieve the same number of evaluations.

On the flipside, choosing a small number can result in unreliable results. The variance between scores able to be achieved, especially by automated agents, is high (see Figure 5.1). A balance must be found between computation time and variance.

The more scores are sampled, the lower the standard error of the mean is. Figure 5.3 shows both the formulaic standard deviation of expected means, as well as an empirical representation of the standard deviations of *Ms. Pac-Man* scores sampled at different game counts.

The immediate and obvious observation from Figure 5.3 is that the more games are sampled, the smaller the variance in scores is. In a perfect scenario, a variance of 0 means that even a smaller sample is equivalent to something close to the 'true' result represented by the 10000 original scores and the fitness value they achieve.

Choosing the number of games to be played for an experiment is yet another

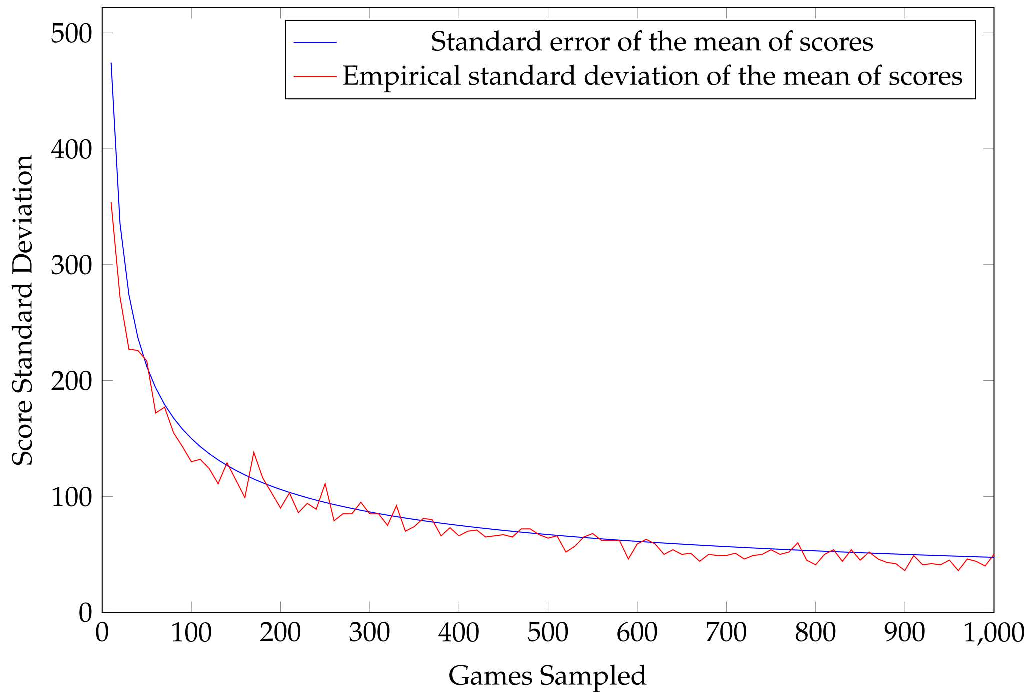

FIGURE 5.3: Comparison between the standard error of the mean of scores, as calculated mathematically, and the empirical standard error of scores when looking at actual *Ms. Pac-Man* scores, relative to the number of games sampled each time

variable the designer can then alter. If time is less valuable than accuracy, a high value is advised. Otherwise, a smaller value should be assigned.

For the experiments presented further in this section, the number of games chosen was 100. The standard error of the mean is 148, an overall acceptable value given our tasks. Of course, at this time, this is a subjective choice between accuracy and computation time and this one gives a good balance.

5.2.2 Experiments

The designer requirement for all experiments was to make the game easier for a novice player, but not too much easier than the original version. Considering the original win-rate of 19.67%, we wanted the game to be won around 50% of the time. This meant $DWR = 0.5$.

Three experimental studies were performed:

- An exploratory study used the following configuration of parameters: Population size of 100, a crossover proportion of 40%, a mutation proportion of 40%, a mutation rate of 50%, elitism of 20%, $C_W = 1000$ and $C_\Delta = 100$.

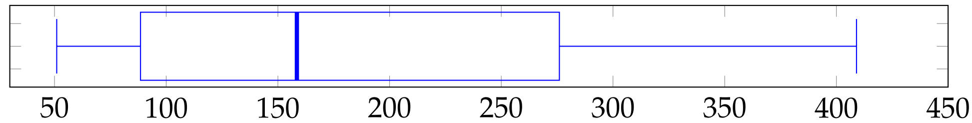

FIGURE 5.4: Boxplot of best individual fitnesses in each of the 10 *Ms. Pac-Man* runs

- The second study sought to find the relationship between mutation rate, population size, reinitialisation ratio and fitness over time. Configurations were chosen from Table 5.3. Weights were set to $C_W = 1000$ and $C_\Delta = 100$, as preliminary analysis highlighted them as good enough.

- Finally, a study on the impact of various values for the objective weights was done. This used the optimal configuration of GA parameters from the second experiment. Configurations were chosen from Table 5.6.

All experiments except for the exploratory study were the result of 50 runs for each configuration tested. Each of the 50 runs had the same starting seed between experiments.

5.2.3 Results

Exploratory Run

A total of 10 runs were done. The best parameters generated by each run, as well as the fitness components, can be seen in Table 5.7. Each run had other good solutions, but here we will focus on only the ones with the best fitness.

The Δ_P values (see Equation 5.1) of the best individuals in each run averaged 188.2 ($\pm$117.77), with a median of 158.5 (see also box-plot in Figure 5.4). The win-rate component W (see Equation 5.1) of the fitness function was optimal in 6 of 10 best-of-run individuals.

It is worth noting that 8 out of the 10 runs managed to find suggestions that have better fitness than the baseline configuration's fitness. This means these are suggestions that fit the requirements better than leaving parameters as they were.

TABLE 5.7: *Ms. Pac-Man* experiment results, with the best solution in bold and the default, unchanged, version at the bottom. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes

Run	S_{PacMan}	S_{G1}	S_{G2}	S_{G3}	S_{G4}	F_{G1}	F_{G2}	F_{G3}	F_{G4}	Δ_P	Win-rate
1	+0.58	+0.52	−0.16	−0.34	−0.43	+0.01	+0.04	+0.01	−0.09	218	50/100
2	+0.83	+0.76	−0.43	+0	+0.03	−0.09	−0.01	−0.03	+0.37	255	50/100
3	−0.43	−0.27	−0.11	−0.27	−0.02	−0.02	+0	−0.03	−0.04	119	51/100
4	**+0.07**	**+0.01**	**+0.02**	**−0.16**	**+0.06**	**−0.05**	**+0**	**+0.06**	**−0.09**	**51**	**50/100**
5	+1.64	+1.23	+0.23	−0.16	+0.06	−0.02	+0	−0.06	+0	339	50/100
6	+1.6	+1.21	−0.3	−0.2	−0.47	−0.03	−0.25	+0.04	+0	409	50/100
7	+0.05	−0.01	+0.41	−0.22	+0.02	−0.01	+0.06	+0.01	+0.04	84	47/100
8	+0.74	+0.65	−0.23	+0.02	−0.01	+0	+0.04	−0.16	−0.06	191	50/100
9	+0.31	+0.27	+0.27	+0.02	−0.03	+0.03	+0.2	−0.06	−0.06	126	40/100
10	−0.04	−0.03	+0.06	+0.22	−0.1	+0.03	+0.35	+0.01	−0.07	90	48/100

Some of the best individuals made minor changes to the game's mechanics, but these changes resulted in a much easier game for the rules-based agent. In general, the GA found 4 different methods of balancing the game towards the required designer metric.

The first strategy (runs 1, 2, 5, 6 and 8) is to make the main player and one ghost faster (up to 30% faster), while barely altering the other ghosts. This is a worthy avenue to take by a designer, but might not be the best one available due to physical reaction time differences between AI agents and humans.

The second strategy (runs 3 and 10) is to slow everyone down slightly. The ghosts are slowed down less than the player, which seems to allow for more points to be gathered, thus winning the game more often. It is very likely the average score would decrease, should we return to the original version of the game, but that is a side effect of the new goal.

The third strategy, found in run 4, is very mild tweaks to all values. Surprisingly, this fits the requirements very well, while changing the game the least out of all the presented options. This would likely be the solution a designer would be most interested in. However, some might find the changes too small and consider the run lucky, which it very well might have been. Another 100 games were run with this configuration, with a new random seed, to double check the results. The new winrate was, this time, only 42% (with 42 wins out of 100 games), however that is still much higher than the original 19.67% reported with the original parameters, and quite close to the 50% desired. This shows that even small changes can impact the difficulty of a game in unexpected ways.

Lastly, in runs 7 and 9, various variations of speeding up both the player and some of the ghosts were used, although there was no obvious pattern that would warrant further exploration. Their fitness is not great either and can thus be considered unsuccessful runs.

Given this array of suggestions, a designer can run diagnostics with different AI agents or human testers and assess the value of the ones they found most interesting. These can be further adjusted manually or the fitness function can be altered to reflect a better understanding of the task.

The results were very good and resulted in a conference publication [1].

Testing GA Parameter Configurations

Once the exploratory study yielded good results, further optimisations to the GA parameters were sought in a second set of experiments. All the configurations tested can be found in Table 5.3, Table 5.4 and Table 5.5.

The main goal of these runs was to observe how quickly and effectively good solutions are found, as opposed to observing and analysing the suggestions given. These runs would allow us to choose potentially better default values for experiments in other games.

The parameters of the GA that were varied included the mutation rate, reinitialisation and population size.

Mutation Rate Results can be seen in Figure 5.5. Through the first 500 evaluations, results are very similar. Differences appear beyond those first 500, with some rates proving objectively worse than others.

The best performing configurations were the ones with mutation rates of 10% and 20%. All other configurations had much worse results overall, despite all other parameters being identical.

The worst result, by far, was the one where mutation was very aggressive, at a rate of 50%, followed closely by 40%. Interesting is the fact that the next worst configuration was the one where the mutation rate was too low, at 5%.

When comparing results using a Wilcoxon Signed Rank test, the configurations with mutation rates of 10% and 20% were significantly better than the others, with results available in Table 5.8. While neither was statistically better than the other, the results using the mutation rate of 20% were, on average, slightly better. The *p*-value was also closer to significance when comparing to the results with a mutation rate of 10%.

TABLE 5.8: Wilcoxon Signed Rank test when comparing various mutation rates

Rate compared to	5%	10%	20%	30%	40%	50%
10%	4.32E-06	-	0.905	0.027	3.40E-11	2.11E-15
20%	1.05E-08	0.095	-	0.004	5.38E-10	0

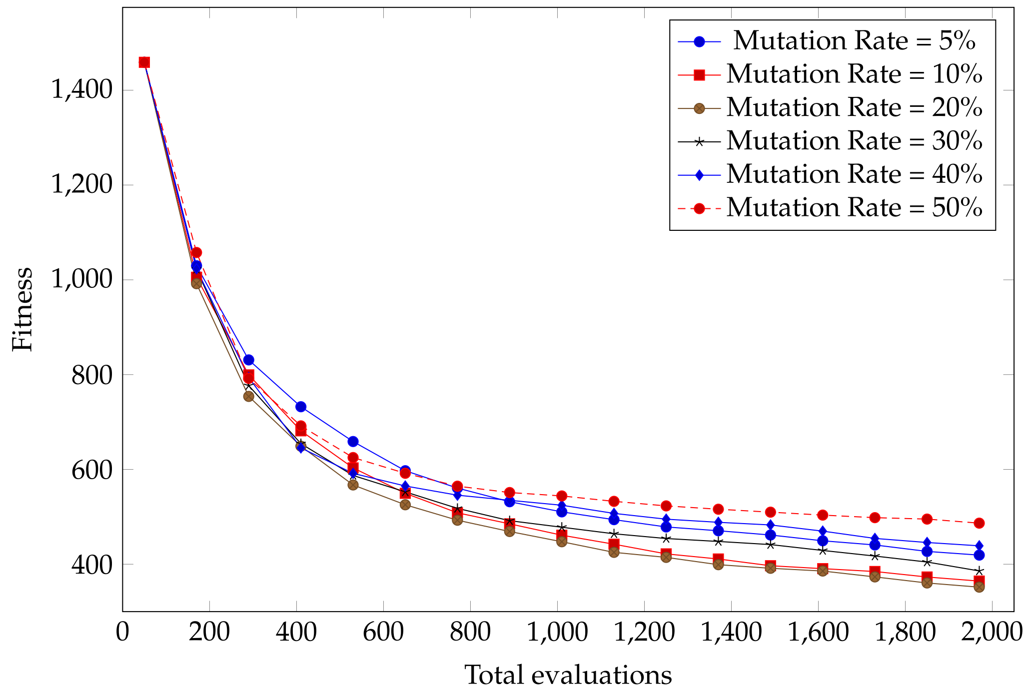

FIGURE 5.5: Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing GA configurations with various rates of mutation

Based on these results, for the remainder of the experiments, a value of 20% was chosen for the mutation rate.

Reinitialisation Afterwards we considered whether reinitialisation was valuable or not. This is the operation where random new individuals are generated and added to the population, with the goal of maintaining diversity. Figure 5.6 shows that there are no benefits to having it present for this experiment. Both with a population size of 50 and 100, using a reinitialisation proportion of 20% did not improve the results in relevant ways. Quite the opposite, with reinitialisation bringing worse fitnesses at almost every data point.

It is possible that, if applied with a smaller rate, reinitialisation could be better. We did not explore this possibility because differences were relatively small in any case. Following this, the decision to remove it from future experiments was taken.

Population Size The configurations presented in Table 5.5 were used, with a mutation rate of 20%, as resulting from the previous experiments.

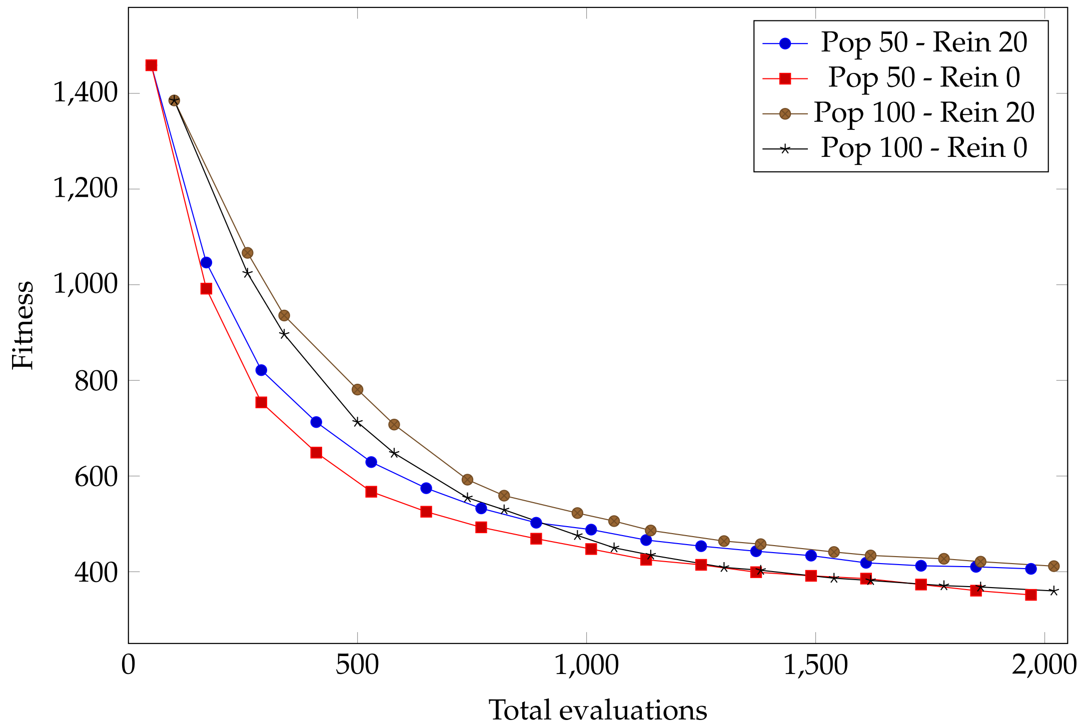

FIGURE 5.6: Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing GA configurations with and without reinitialisation

Looking at the results in Figure 5.7 it can be seen that the bigger the population, the longer it takes to reach good results. However, the bigger the population, the better the results given the budget of 2000 evaluations, with one exception. The runs where the population size was set to 200 barely managed to achieve equivalent results to the runs with population size 100 within budget, presenting worse results, on average, before the final few generations. It is likely, with a higher budget, that they would catch up and possibly further improve results.

Overall, the best results were those with population sizes 50 and 100. These gave good results at every stage of their runs.

Testing Different Values for Weights for Fitness Components

Looking at the results from the previous studies, the best configuration proved to be the one where the population size was set to 50, with no reevaluation and a mutation rate of 20%. This was the chosen configuration to then test the values of different weights for fitness components. The configurations tested were those presented in Table 5.6. For each configuration, 50 runs were done.

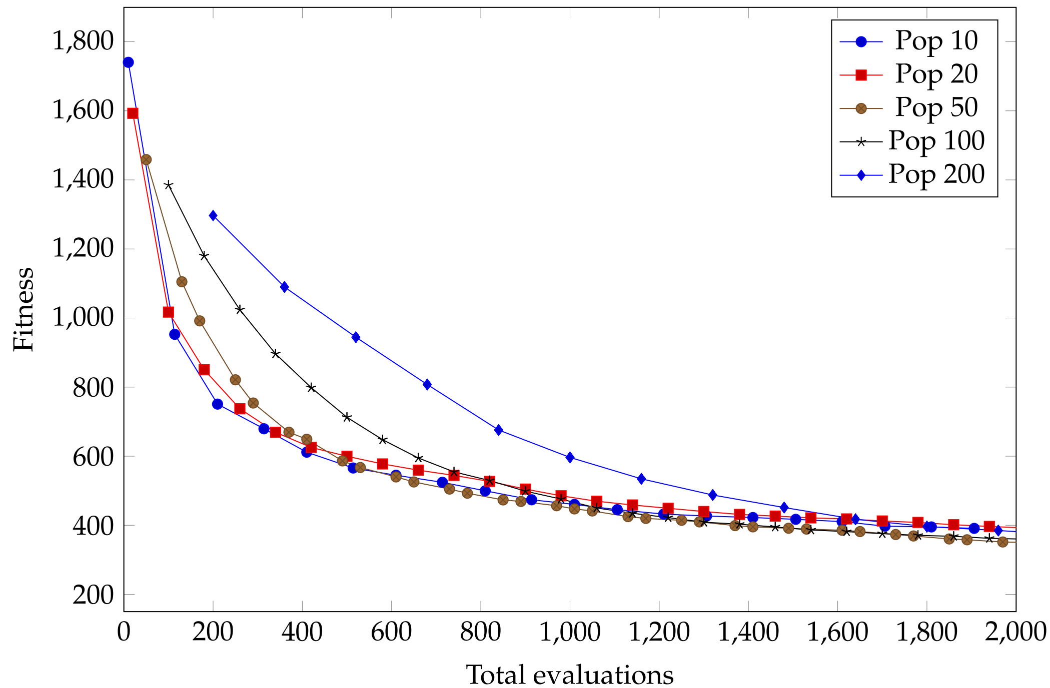

FIGURE 5.7: Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing different population sizes

One of the marks of success is whether each configuration manages to find suggestions that have a better fitness than the baseline fitness, as well as how fast those suggestions are found. Failure to find a better individual means that, for that pair of weights, there is no solution better than no changes.

All results are presented in Figure 5.8. As expected, when minimising changes to the parameters is extremely important (i.e., ratio values of 5 and less), parameter sets that fit the requirements better than making no changes are virtually impossible to find.

It is interesting to note how the ratio value of 10, chosen for the preliminary experiment, behaves. While in that study most results managed to be better than the status quo, with more runs the average is actually worse. This is likely a sign that, still, the parameter weight is too high compared to the win-rate weight. On the flip-side, a value of 50 for the ratio results in pretty much any move towards the desired win-rate to be considered significantly better than the baseline.

These results highlight an interesting conundrum for multi-objective optimisation. On one hand, investing a lot of time running multiple GA configurations is required to allow one to better understand their problem and what parameters are

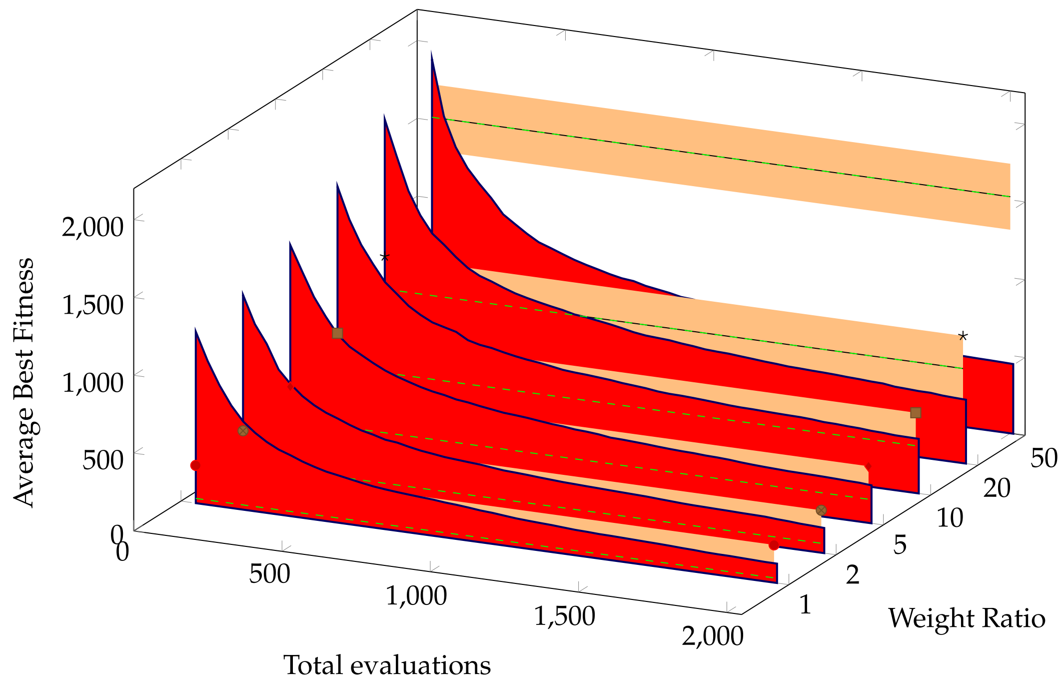

FIGURE 5.8: Average best fitness values achieved after a given number of evaluations by the GA runs in the *Ms. Pac-Man* experiment, when comparing various objective weights. Dashed green lines represent the baseline fitness for each respective value of the weight ratio. Orange ranges represent the confidence thresholds

better for various weights. However, industrial use cases might not have the time or desire to deeply analyse so many abstract concepts relating to their game or product.

Some of the analysis presented in Section 5.2.1 did hint at this and could help in pre-emptively selecting good values. Some trial and error will, however, always be required when looking for fast suggestions to existing problems.

5.3 *StarCraft* Experiments

5.3.1 Environment

The *StarCraft* environment, being immensely more computationally taxing, was explored in much less detail than the *Ms. Pac-Man* one. Very similar algorithms and considerations were used as with the *Ms. Pac-Man* experiments. This use case is a much more realistic scenario, as it highlights how a designer would be pushed to go with the results of a first, or at most second, batch of runs, due to the time considerations.

Beyond the tools used to interact with the game, described in Section 3.2, no changes were made to the game mechanics or rules.

There is one Zerg (one of the playable factions in the game) AI agent, called ZZZKBot, written by Chris Coxe [153], that has a very high win-rate against most other AI agents and intermediate or novice human players [154]. Prior knowledge of ZZZKBot's strategy, or an experienced player, are required to successfully defend against it. The designer aim was to change the game to make this AI bot no longer dominant.

ZZZKBot employs the 4-pool zergling rush strategy. What this strategy entails is not critical to the understanding of this chapter, but is given for those curious. The Zerg player makes their cheapest offensive units (the zergling) in bulk as quickly as possible, sending them to attack the opponent as soon as they are complete. This is a very aggressive strategy that, if not explicitly defended against, will result in success. If, however, the initial onslaught is pushed back, the Zerg player has greatly diminished chances of victory, as they have ignored any economic growth. In most games, in the standard version of the game, the Zerg AI rushes to the first military building (the Spawning Pool) which then allows them to train zerglings en-masse. By the time the first wave of zerglings arrives at the Terran opponent, there should be minimal resistance. The Terran SCVs have to defend, while the first marine is being trained. A failed defence will mean the Zerg player overwhelms the base.

As a result, the important units in *StarCraft* for this study are:

TABLE 5.9: Original *StarCraft* parameters

	Health	Attack	Minerals Cost	Time to Train
Marine	40	6	50	24s
SCV	60	5	50	20s
Zergling	35	5	50	28s

- the Terran SCV, the basic worker of the Terran faction. They are weak in combat, but are the only resource-gathering unit.
- the Terran Marine, the cheapest military unit available to the Terran faction. They are efficient in multiple numbers, attack from range and move at a medium speed.
- the Zerg Zergling, the cheapest military unit available to the Zerg faction. They are efficient in multiple numbers, attack from melee range and are fast.

The game's relevant parameters, for this study, are how strong the marine, SCV and zergling units are, in regards to their available hit points (how hard they are to kill), their attack damage (how strong their attacks are), the amount of time it takes to train one of each, and how many minerals they cost to train (Table 5.9). Of course, there are many more units whose defence and attack could be evolved, but were not valuable for this niche scenario. The ranges the parameters can have, as well as the accuracies and chosen weights can be seen in Table 5.10.

From a design perspective, one extra point of damage is more important than one extra point of health, as health is a finite resource, while damage is dealt repeatedly for as long as the unit is alive. Similarly, build time and mineral cost are more relevant than health. As a result, the health changes of the 3 units have a lower weight compared to the other parameters.

Fitness Evaluation

For each parameter set, the map was created, then a number of games were run in sequence, storing the results (win or loss) of the Zerg AI and the duration of each game. For these experiments the number of games chosen was 10. The Zerg AI played against the game's highest difficulty Terran player.

TABLE 5.10: Parameters chosen and their ranges for *StarCraft*

	Description	Min Change	Max Change	Accuracy	$C_{\Delta i}$
A_1	Terran Marine Attack	-4	+26	10^0	100
A_2	Terran SCV Attack	-4	+26	10^0	100
A_3	Zerg Zergling Attack	-4	+26	10^0	100
H_1	Terran Marine Health	-30	+30	10^0	50
H_2	Terran SCV Health	-30	+30	10^0	50
H_3	Zerg Zergling Health	-30	+30	10^0	50
M_1	Terran Marine Mineral Cost	-25	+25	10^0	100
M_2	Terran SCV Mineral Cost	-25	+25	10^0	100
M_3	Zerg Zergling Mineral Cost	-25	+25	10^0	100
B_1	Terran Marine Build Time	-20	+20	10^0	100
B_2	Terran SCV Build Time	-20	+20	10^0	100
B_3	Zerg Zergling Build Time	-20	+20	10^0	100

TABLE 5.11: *StarCraft* metrics to be used in evaluation alongside their desired values and weights

	Name	Desired	Evaluator	Weight
W	Win-rate achieved by ZZZKBot	DWR	Average	$C_W = 10000$
Δ_P	Sum of parameter absolute displacements	0	Minimise	$C_{\Delta i}$

Similar to the *Ms. Pac-Man* experiments, the end goal was to control two features: how often the main agent wins and how big the changes are to the original game parameters. The smaller the changes are to the original game parameters, the less likely it is that the game's players would become dissatisfied with the changes. These metrics can be seen in Table 5.11. The desired win-rate is not given a value yet as we have sought a couple of experiments, each with a different value for it.

As a result, fitness is a multi-objective function of the win-rate and the absolute difference between the default game parameters and the newly evolved ones. The closer the win-rate is to the desired one, the better the fitness. Also, the smaller the difference between the original parameters and the evolved parameters, the better the fitness, as it is preferred to have incremental changes rather than big ones. This is the immediate result of wanting to optimise an existing game rather than creating a new one from existing mechanics.

The objective of optimising the win-rate could be further split into multiple game

scenarios, each one dealing with different AI agents or maps, but we decided to keep it straightforward and focus on methodology by using only two AI agents (ZZZKBot and the default *StarCraft* Terran AI).

Formally, the fitness function can be written as:

$$Fitness = W + \Delta_P \tag{5.4}$$

$$W = |WR - DWR| \times C_W \tag{5.5}$$

$$\Delta_P = \sum_{i=0}^{n} (|\Delta_i| \times C_{\Delta i}) \tag{5.6}$$

In the win-rate optimisation function W, WR = Win-Rate (from 0 to 1) and DWR = desired win-rate, C_W = win-rate bias factor. In the parameter optimisation function Δ_P, Δ_i = difference between the original ith parameter and the evolved ith parameter and $C_{\Delta i}$ = parameter weight.

We consider smaller fitness values to be better, with 0 representing a perfect solution.

5.3.2 Experiments

Two experiments were exploratory studies done in parallel with the *Ms. Pac-Man* one.

The first one had the purpose of assessing whether a GA can completely nullify the ZZZKBot AI's strategy, thus achieve 0% win-rate. Intuition says that pushing the Zergling's attack damage as low as possible, or, alternatively, increasing the SCVs attack high enough, would be sufficient to neutralize the Zerg AI strategy. If the GA arrived at a similar conclusion, it would be a sanity check for its ability to alter the game appropriately. The baseline fitness (where all game parameters are set to 0, thus proposing no changes to the existing game) is 10000 and will be used to compare results.

The second exploratory experiment required the search for changes to the game that made the ZZZKBot AI's strategy only successful 50% of the time. This is a very complex task, and there is no immediately obvious solution to it. It is also very likely that there might be multiple ways of achieving this goal, and these could be

TABLE 5.12: Suggested changes to *StarCraft* parameters to minimise ZZZKBot's win-rate

	Health	Attack	Minerals Cost	Time to Train
Marine	+2	+2	+0	-1s
SCV	+0	-1	+0	+0s
Zergling	-29	+0	-1	+0s

found by different GA runs. The reason behind choosing 50% as the target win-rate is due to a desire to remove a dominant strategy from the pool of available tactics. By removing the dominant strategy, the game should once again allow for multiple strategies to be viable. The baseline fitness (where all parameters are set to 0) was 5000 and was used to assess results.

Finally, once a good set of parameters was computed in the second exploratory experiment, the corresponding version of *StarCraft* was played by people of varying skills, to assess its viability with respect to the original version. Players had to play as the Terran race and face ZZZKBot in two best of 3 series, one with the original parameters, and one with a parameter set identified by the GA, then describe which parameter set they considered the fairest and most entertaining to play with, as well as any strategic considerations that they made as a result of the changes.

Finally, the second exploratory experiment was ran again with the optimised GA parameter configuration found in the *Ms. Pac-Man* experiments. The assumption was that the results would be on par, if not better, than the exploratory ones.

5.3.3 Results

Completely Nullifying the ZZZKBot

For *StarCraft* a single run was done, with the objective of completely disabling the ZZZKBot's strategy. This was done just as a test of the much more complex version of our system required by the game, with the parameters described in the previous section, $DWR = 0.0$ and $C_W = 400$.

The best set of parameters computed is presented in Table 5.12. This individual had a W of 0, which means ZZZKBot lost all 10 of the games it played, and a Δ_P

of 2050. With these parameters, the game's default Terran AI agents consistently defeat ZZZKBot, while also diverging from the game's default parameters by a small amount, as was required.

This result is in line with one of the intuitive solutions a game designer would come up with to solve the task. While, for this example, the Marine and SCV parameters did not require many, if any, changes, it highlights the GA's ability to come to the same conclusion and focus on changing the relevant unit parameters only, in this case the zergling's health.

Preliminary Balancing of the ZZZKBot Strategy

In this experiment 10 runs were done, each with a different seed, with $DWR = 0.5$, $C_{\Delta} = 1$ and $C_W = 400$.

Given the complex setup, each generation required approximately 20 minutes to complete on a 16-core computing cluster. Consequently, each run took approximately 17 hours. Results can be seen in Table 5.13.

The Δ_P values of the best individuals in each run averaged 3985 ($\pm$722.7), with a median of 4025 (see also box-plot in Figure 5.9). The win-rate component of the fitness function consistently achieved the optimal value, as a result of having won 5 out of 10 games.

With all of the evolved parameter sets, the game's default Terran AI agent has a chance to defend against ZZZKBot in approximately 50% of their encounters. The difference between the default parameters and the computed ones is generally substantial in most runs. Each run consistently managed to optimise the win-rate objective W of the fitness function.

Looking at the results, there is an immediate observation to be made: every single run considered at least one (usually two) of the parameters H_1 A_1 (the Terran Marine's health and attack damage) and H_3 (the Zerg Zergling's health) as requiring major changes. The Terran unit's attack is increased by a generous amount, while the Zerg unit's health requires some decrease. Sometimes these changes are partnered with significant increases to the Marine's health. The GA also considers parameters A_2, A_2, M_3 and B_3 as requiring virtually no change.

TABLE 5.13: Main *StarCraft* experiment results, with best individual(s) in bold. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes

Run	H_1	A_1	M_1	B_1	H_2	A_2	M_2	B_2	H_3	A_3	M_3	B_3	Δ_P
1	+17	+12	+10	+5	+0	−1	−3	+0	−3	+0	+2	+0	4200
2	+4	+3	+7	+0	−2	−3	−2	+2	−19	−3	−1	−1	3450
3	+12	+22	−3	+4	−18	+0	+3	+0	−4	+0	+0	+1	5000
4	+12	+8	−1	+1	−5	+4	+1	−4	−9	−1	+5	+3	4100
5	+13	+9	−2	−1	+2	+0	+13	−4	−12	+2	+3	−3	5050
6	+4	+20	−1	−3	+1	−1	−1	+0	+0	−3	−4	+3	3850
7	**+3**	**+9**	**+4**	**−2**	**+1**	**+2**	**+0**	**+1**	**−9**	**+1**	**−1**	**+0**	**2650**
8	−1	+10	+3	−5	+0	+2	−1	+3	−11	−3	+0	+1	3400
9	+3	+8	+4	+1	+8	+4	+0	+0	−14	+0	−7	+3	3950
10	+2	+20	−3	−7	+6	+1	+4	−7	+0	−1	+0	+0	4200

Interestingly, a couple of runs (1 and 3) also suggested increasing the build time for the Terran Marines (B_1). They were also the runs that had the biggest improvements to that unit's health and attack proposed.

The results of each run are similar to each other, all presenting the same general suggestions to the designer. This could prompt a look at the deeper mechanics of the game should the magnitude of the changes be too high for implementation.

Overall these results show that our approach was successful, as the different GA runs did not just give us different avenues to solving the task that fit the requirements but also made the strategies designers can choose when balancing their games clear.

Testing Results with Human Players Five people, one with thousands of hours of experience in the game, 2 with intermediate experience and 2 novices, were asked to play between 2 and 6 games each of *StarCraft* against ZZZKBot, half of those games using the vanilla parameters and the other half using the evolved parameters from run 7. Using ChaosLauncher and BWAPI, the whole setup was automated. Players were asked to play to the best of their abilities, advising them that the games would be short and that one map had some parameters modified compared to the other, as well as that they would play against a fairly aggressive Zerg player. Results can be seen in Figure 5.10.

The novice players consistently lost in both versions of the game, but, after realising they could defend with the workers, often found it easier to survive slightly longer on the evolved map that gave their SCV workers a slight increase in attack power. This does highlight the fact that balancing the game against the insane difficulty computer opponent means weaker players do not benefit as much from minor changes in their favour. This is worth considering for the future, as a multi-objective fitness evaluation can be used to balance against multiple difficulties at once, each potentially having its own requirements and target optimal values. Of course, it is

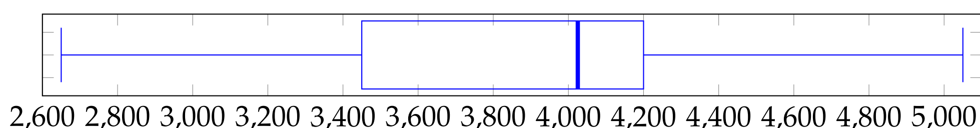

FIGURE 5.9: Boxplot of Δ_P fitness component for all 10 *StarCraft* runs

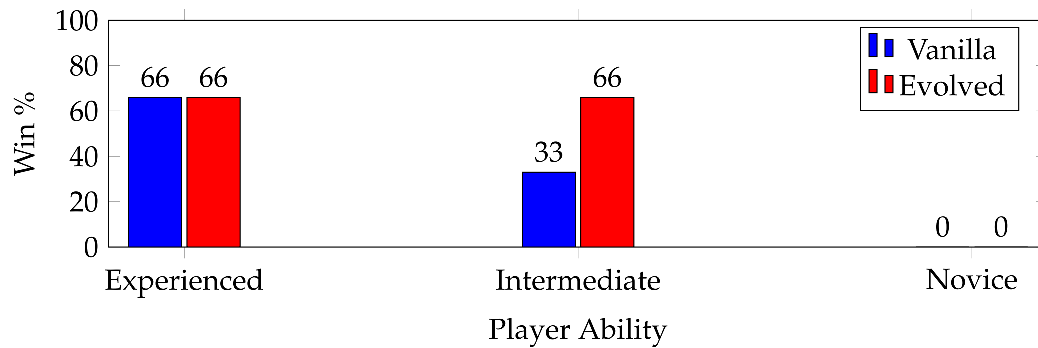

FIGURE 5.10: Win-rate % after human play-testing

also representative of how humans do attempt to adapt to changes and make use of them to their advantage through trial and error.

The intermediate players took better advantage of the changes, successfully defending using both workers and an alternate strategy on the modified map, but finding it more challenging on the vanilla version of the game. They reported that the stronger workers were something beneficial, but also stated that they could have won more reliably on the unchanged map were they able to micro-manage their units more effectively, an ability that requires a lot of practice.

The experienced player did not make any use of the stronger workers, opting to always use an optimal strategy that involved marines and a bunker (a building that allows marines to shoot from safety until it is destroyed) in all scenarios. They still lost a game on both versions of the map due to positioning mistakes, but reliably showcased how one could beat the Zerg rush strategy and keep a healthy economy to eventually win the game. This is as expected, as the employed Zerg strategy, while very strong, is not a dominant one when playing against high-level opponents.

This is purely exploratory qualitative data, as the sample was very low and the experiment was not designed to allow for control. It can be useful for preparing future experiments concerning this sort of work.

Balancing ZZZKBot Using Optimised GA Parameters

The previous experiment was rerun using the updated parameters, as found in Section 5.2.3. A total of 10 runs were done, each one paired to one from the exploratory study.

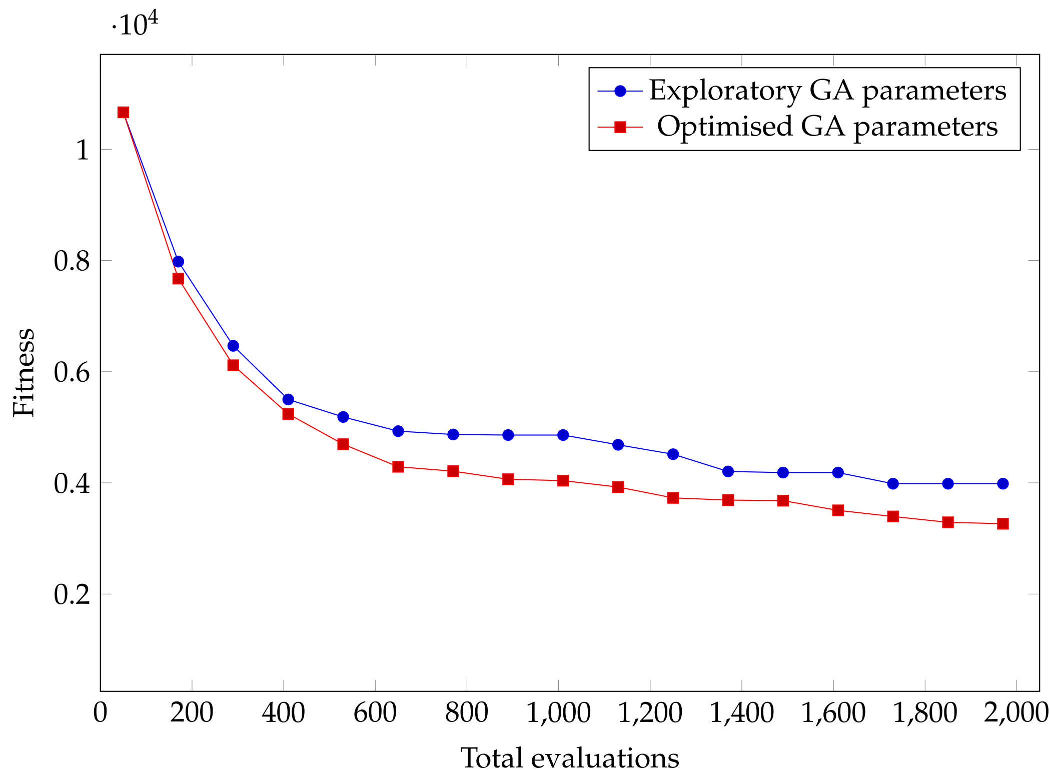

FIGURE 5.11: Average best fitness values achieved after a given number of evaluations by the GA runs in the *StarCraft* experiment, when comparing the results of the exploratory GA parameters to the results of the optimised GA parameters. Lower values are better.

As expected as a result of the *Ms. Pac-Man* experiments, the different GA parameters resulted in better results faster. These can be observed in Figure 5.11. The Δ_P values of the best individuals in each run averaged 3245 (±564.9), with a median of 3175. To confirm this, a Wilcoxon Signed Rank test was applied to the best results of each paired run. The result was $p = 0.0078$, a highly significant value.

The solution space, however, was not particularly different from the one presented in Section 5.3.3. Most of the proposed changes were optimised versions of the results already presented in Table 5.13. The biggest variation is when some runs figured out they can decrease the build time of Terran Marines and, as a result, require smaller improvements to their health and attack. Overall, however, the same strategies were suggested.

TABLE 5.14: Parameters chosen and their original values for *TORCS*

	Description	Original
P_M	Car Mass	600 kg
P_D	Drag Coefficient	0.6
P_C	Clutch Inertia	0.115 $kg.m^2$
P_L	Steering Lock	21 deg
P_S	Steering Speed	360 deg/s
P_R	Rear Differential Ratio	5.5
P_B	Brake Pressure	24000 kPa
P_{A1}	Front Anti-Roll Bar Spring	0 lbs/in
P_{A2}	Rear Anti-Roll Bar Spring	0 lbs/in

5.4 *TORCS* Experiments

5.4.1 Environment

Many elements of the game could be changed for balancing purposes, from the layout of the tracks, to the technical specifications of the cars, to the way artificial agents drive in the game. For this experiment we decided to look at changing the performance of a single vehicle in the game, the *car1-ow1*.

After discussing this with an experienced engineer from the car racing industry, the elements of the car chosen for tweaking (the parameters) were: the total mass of the car (P_M), the drag coefficient (P_D), the clutch inertia value (P_C), the steering speed (P_S), the steering lock (P_L), the rear differential ratio (P_R), the maximum pressure applicable by the breaking system (P_B), the front anti-roll bar spring value (P_{A1}), and the rear anti-roll bar spring value (P_{A2}). The original values for these parameter can be seen in Table 5.14. As before, displacements from the default values to these car parameters are what the GA evolves. The ranges these parameter displacements could take, as well as the decimal accuracy they can have, can be seen in Table 5.15.

The car is driven by one of the game's default driver agents, *berniw*. To collect metrics, the car was driven on 3 different tracks, one for each distinct type available in the game (dirt, oval and road). The values gathered were the lap times achieved on each one. As designers seeking to balance the car itself, the goal was to improve the car's performance on the road track (achieve a better lap time), decrease its performance on the oval track (achieve a worse lap time), but maintain its performance

TABLE 5.15: Parameters chosen and their ranges for *TORCS*

	Description	Min Change	Max Change	Acc	$C_{\Delta i}$
P_M	Car Mass	-300 kg	300 kg	10^0	0.5
P_D	Drag Coefficient	-0.15	0.15	10^{-2}	1000
P_C	Clutch Inertia	-0.05 $kg.m^2$	0.05 $kg.m^2$	10^{-4}	3000
P_L	Steering Lock	-15 deg	20 deg	10^0	7.5
P_S	Steering Speed	-300 deg/s	0 deg/s	10^0	0.5
P_R	Rear Differential Ratio	-5	5	10^{-1}	30
P_B	Brake Pressure	-19000 kPa	19000 kPa	10^0	0.005
P_{A1}	Front Anti-Roll Bar Spring	0 lbs/in	5000 lbs/in	10^0	0.025
P_{A2}	Rear Anti-Roll Bar Spring	0 lbs/in	5000 lbs/in	10^0	0.025

TABLE 5.16: Times achieved by the vanilla version of the car on each track, alongside the desired times

Track	Original Time	Desired Time
Road	78.33s	70.00s
Oval	26.95s	32.00s
Dirt	63.93s	64.00s

on the dirt track (maintain its previous lap time). The performance of the original version of the car, as well as the desired new values, can be seen in Table 5.16.

Fitness Evaluation

The first objective is minimisation of all 9 parameters, or applying the least displacement possible to the car properties. The second objective is achieving the desired performance on the road track, for which we used an average evaluator comparing the appropriate metric to the desired value, as seen in Table 5.16. The third objective is the same as before, except looking at the times achieved on the oval track. The last objective is treated identically, except the relevant metric is the time to complete the dirt track. All objectives can be seen listed in Table 5.17.

Formally, the fitness function can be written as:

$$Fitness = T_R + T_O + T_D + \Delta_P \tag{5.7}$$

TABLE 5.17: *TORCS* metrics to be used in evaluation alongside their desired values and weights

	Name	Desired	Evaluator	Weight
T_R	Time on Road map	$D_R = 70$	Average	$W_R = 70$
T_O	Time on Oval map	$D_O = 70$	Average	$W_O = 70$
T_D	Time on Dirt map	$D_D = 70$	Average	$W_D = 70$
Δ_P	Sum of parameter absolute displacements	0	Minimise	Various $Weight_{\Delta i}$

$$T_R = |Time_R - D_R| \times W_R \tag{5.8}$$

$$T_O = |Time_O - D_O| \times W_O \tag{5.9}$$

$$T_D = |Time_D - D_D| \times W_D \tag{5.10}$$

$$\Delta_P = \sum_{i=0}^{n} |\Delta_i| \times C_{\Delta i} \tag{5.11}$$

In these functions, $Time_R$ is the time achieved on the road map, $Time_O$ is the time achieved on the oval map, $Time_D$ is the time achieved on the dirt map. D_R, D_O and D_D are the desired values for each of the road, oval and dirt map, respectively. Δ_i is the displacement of the ith parameter, while $C_{\Delta i}$ is that parameter's weight.

For these experiments, each value for *Target* was taken from Table 5.16, all the evaluator *Weight* values were set to 70, while all parameter $Weight_{\Delta i}$ were set to the values in Table 5.15. As a result, all of the evaluators have similar importance. This results in all four main objectives having similar relevance to the final fitness score.

Running the game with unchanged parameters gives a fitness value of 941.5. This will be considered the baseline for comparing suggestions from the GA.

TORCS is not stochastic and, as a result, no more than 1 game needed to be played for each individual. Each game consisted of running a single race of one lap on each of the 3 tracks and reporting the times achieved on each one.

Genetic Algorithm

The best GA configuration from the *Ms. Pac-Man* experiments in Section 5.2.3 were used to generate suggestions for balancing *TORCS*. This was the one where elitism

was applied to 20% of the population, mutation was applied to 40% of the population, at a rate of 20%, and crossover was applied to 40% of the population.

Population size was 100, with a tournament size of 6. Experiments ran until 2000 evaluations of the fitness function were used.

5.4.2 Experiment

The main experiment involved changing the parameters of the targeted car until its performance on all 3 tracks was as desired, as described previously.

A total of 20 runs were done, each with a different seed for the random number generator.

5.4.3 Results

Looking at the results in Table 5.18 we see quite a few valuable pieces of information. The most important one is that each run manages to produce suggestions that improve the fitness of the reference (no changes) configuration. This was despite the penalty brought on by having a non-zero parameter fitness objective.

Beyond that, the thing that immediately jumps out is the fact that the car's mass (P_M) is always greatly reduced. This makes sense, as that would give it a significant speed boost on most tracks, which is valuable for one of the objectives.

The GA decided not to change most of the other parameters much, as it would have, most likely, resulted in worse fitnesses. It is very likely that higher displacements would have had a negative effects on the fitness as a result of the minimisation requirement. Any minor improvements to times would be offset by the penalty of changing the value.

The one interesting exception is the steering lock (P_L). This parameter was mostly ignored in all runs except the 3 best performing ones. In those 3 runs, the biggest change towards decreasing it proved best. This resulted in the car driving much slower on the oval track, as needed, while impacting the other two tracks much less than expected.

This suggests that we are dealing with a multi-modal fitness landscape with a narrow global optimum and a much wider local optimum, as the other 17 runs did not hone onto this area of the search space.

The majority of runs managed to successfully improve the car's times on the dirt track from 78 seconds to approximately 70 seconds, with an example visible in Figure 5.12. Maintaining performance on the road track seemed slightly harder to achieve, with that particular fitness objective proving harder to minimise. However, even harder to minimise was decreasing the car's performance on the oval track. While all runs managed to slow the car down on that track, the magnitude was not as high as required.

These are all results that designers will find useful, as they offer valuable insight into the relationships between the various parameters. They could then start a new series of game balancing runs, where some of the changes suggested by a few of the runs are implemented, while new parameters and evaluators are added to the mix.

TABLE 5.18: Experiment results, with the best solution in bold and the default, unchanged, version at the bottom. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes

Run	P_M	P_D	P_C	P_L	P_S	P_R	P_B	P_{A1}	P_{A2}	Fitness
1	−213	+0	+0.0034	−14	−24	+0	+1,318	+249	+129	735.12
2	−192	−0.01	+0.0006	+0	−22	+0.5	+2,206	+748	+162	827.96
3	−207	+0	−0.0023	+1	−35	+0.6	−741	+23	+503	800.67
4	−188	−0.03	+0.0009	+0	−50	+0.7	−1,519	+332	+192	837.25
5	−271	+0	+0.0041	−1	−64	+1	+6,013	+107	+283	853.79
6	−223	−0.01	−0.0086	+0	−20	+0.8	−4,342	+163	+209	826.77
7	−190	+0	+0.0035	+0	−27	+0.5	+104	+115	+826	826.4
8	−190	+0.01	+0.0002	+1	−4	+0.9	+6,841	+1,091	+36	847.08
9	−239	+0.02	−0.0087	+0	−11	+0.8	+4,784	+466	+121	834.57
10	−223	+0	+0.0098	−1	−25	+0.7	+15	+334	+73	811.23
11	−180	−0.01	+0.0022	+1	−12	+0.7	−3	+200	+374	823.4
12	−176	−0.01	+0.0017	+1	−78	+0.6	−103	+144	+153	835.5
13	−208	−0.01	+0.0024	−15	−26	+0.6	−2,826	+367	+800	695.52
14	−219	+0.01	+0.0077	+1	−13	+0.8	−417	+42	+355	807.07
15	−198	−0.03	−0.0028	+0	−14	−2.7	+5,581	+73	+47	854.53
16	−211	+0.01	−0.0018	+0	−13	+0.8	−228	+842	+198	806.02
17	−184	+0	−0.0014	+0	−1	+0.7	+6,492	+13	+537	816.45
18	−219	−0.01	−0.0002	+3	−58	+0.6	−652	+228	+245	847.14
19	**−230**	**+0.03**	**+0.0093**	**−14**	**−9**	**+0.9**	**+940**	**+362**	**+110**	**635.44**
20	−233	+0.01	−0.0096	−1	−3	+0.8	−1,767	+667	+171	815.88
Ref	+0	+0	+0	+0	+0	+0	+0	+0	+0	941.5

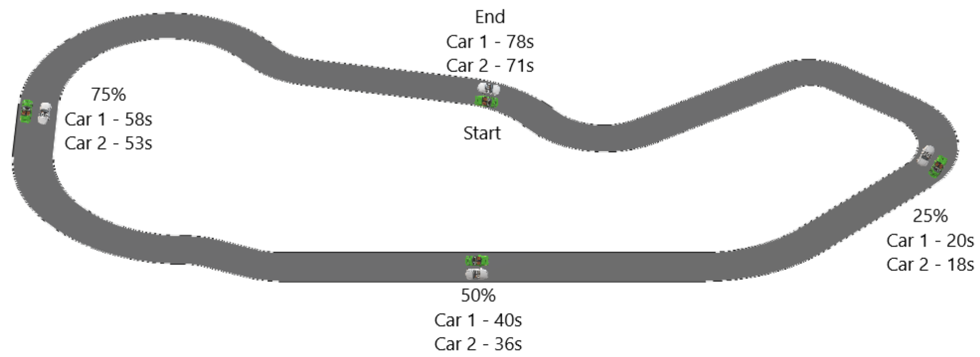

FIGURE 5.12: Visual representation of the times achieved at various checkpoints by the unchanged car (Car 1) compared to the best performing car, according to the requirements, (Car 2, from run 19) on the dirt track

5.5 Discussion

For the purpose of the *StarCraft* experiments, only a subset of the game's parameters was taken in consideration. Other available variables, such as the build times and costs of the Terran Barracks or the Zerg Spawning Pool (the buildings required to actively train the units considered), as well as the attributes of other beasts in the campaign, could have also been evolved. GAs have no major issues optimising problems with many more than 12 variables [155][156] and would not find the extra parameters overly problematic. Better hardware or access to a game's source code would also greatly increase the speed of fitness evaluation, allowing for more individuals in a population, more games to be played, or more generations to be run.

The results do show a lot of promise, as the methodology proposed could not only be useful in balancing a game itself, but, as seen after testing with human players, could aid in calibrating the performance of other intelligent agents to displaying a desired level of skill and expertise. Instead of evolving the game parameters, one could evolve the AI parameters and run a virtually unchanged set of fitness tests, optimising it to behave at a required level.

For performance, the bottleneck was in no way related to the length of the arrays, but in playing out the games themselves for the fitness evaluation. Better hardware or access to a game's source code would also greatly increase the speed of an evaluation, allowing for more individuals in a population, more games to be played, or more generations to be run. This was made obvious by the difference between the *StarCraft* experiment and the other two. While a single *StarCraft* individual in the population took up to a minute to evaluate, a *TORCS* individual would need around 6 seconds, while the evaluation of a *Ms. Pac-Man* individual would complete in under a second.

The main challenges are still the fact that a good understanding of genetic algorithms is required to optimise its performance, as different configurations yield different results. However, except in the case of very bad configurations, the GA will converge towards desirable solutions eventually.

5.6 Summary

By using genetic algorithms we were able to successfully present suggestions for game changes in three vastly different games. These suggestions were automatically developed, with human input only during the designing phase of the experiments. As a result, supervision was not needed during the various trials attempted by the algorithm.

Each game (*Ms. Pac-Man*, *StarCraft* and *TORCS*) proved to have its own challenges, but the same methodology was successful every time. This is reassuring, as it lends credibility to the claim that this approach could be valuable in other games as well, or even in other fields.

This chapter also tackled the challenge of multiple fitness objectives. While for the benefit of less technically experienced users our approach was to simplify this by using a simple weighted sum, there is much that could be improved in the future.

Overall, this work has further proven the validity of using machine learning approaches to game design to complement game developers.

Chapter 6

Fitness Approximation for Faster GA-Based Game Balancing

6.1 Introduction

The previous chapter highlighted one big problem with balancing games through the use of GAs: the high number of games the GA needs to play to achieve desirable results. While in some cases this is not a big problem, for some games, such as *StarCraft*, this can result in experiments that run for days, or even weeks. Reducing the time it takes to achieve acceptable suggestions is, as a result, extremely important.

For the task of balancing games, when utilising computationally cheap agents to play the game, such as a trained neural network, a finite state machine or a random agent, evaluating a game variation is mostly dependent on how quickly the game itself takes to run. A lightweight agent results in fasters runs of the game. However, expensive agents, such as those utilising Monte-Carlo Tree Search, can take anywhere between several seconds or even minutes to complete even a single game. This is due to a common practice with such agents where they utilise as much as possible of a given maximum time limit they have, such as 40ms per frame in games attempting to run at 25 frames per second [130]. This results in code that needs to run in real time instead of being able to be sped up during simulation, which results in very long simulation times. When using these agents, the game mechanics are no longer the most demanding aspect of the entire simulation.

One method of achieving good solutions within a given budget that we already covered was to find "optimal" parameters for the GA. However, there is no promise

that the ones found of *Ms. Pac-Man*, that proved to be better than ones chosen arbitrarily for *TORCS* and *StarCraft* as well, would work in other games. A more task-agnostic method is required to potentially work on multiple games.

As described in Section 2.2.4, a lot of work has already been done on surrogate models and fitness prediction. Our approach for this work takes inspiration from those methodologies and brings forward a novel way of combining various machine learning technologies to speed up GA-based game balance.

We set out a pipeline that involves developing a surrogate model from online data generated by the GA as it is running, testing the model every generation, then applying it to new GA individuals before they are evaluated using the expensive fitness function. These models can save the GA from evaluating some individuals, achieving better results faster.

We test this pipeline using a neural network on a couple of standard problems: the OneMax problem and the Trap problem, as defined in Section 2.2.2. Afterwards, we add two more machine learning algorithms to the models available and test this approach on *Ms. Pac-Man*, *TORCS* and *StarCraft*.

6.2 Pipeline

6.2.1 Approximator Integration

Normally, a GA generation has all individuals that were newly generated evaluated and their fitnesses saved for future operations. We mine this valuable data, as it could help map the parameters that we are evolving to the fitness values, assuming there is any relationship between them.

At the end of each GA generation, all the individuals that were evaluated by running games are passed to the approximator. These individuals are stored in the approximator's data set, for use during model generation.

The data set acts as a queue with a maximum capacity. Whenever a new individual is added to the data set, if there are more individuals than a given maximum (N_{Max}) in it, the oldest one is removed.

During a generation, after the population goes through elitism, crossover and mutation, the new population is passed to the approximator. This population has individuals that have not been evaluated yet and, as a result, do not have a fitness value.

Using a subset of the data set described earlier (80% for all these experiments), the approximator generates (or trains) its model of the fitness landscape, mapping the parameters to either the fitness objectives or a fitness class (more details on the fitness class in Section 6.2.3). The model generation is only done if there is a minimum number of individuals in the data set (N_{Min}). For simplicity, in all the experiments in this chapter, both N_{Min} and N_{Max} are set to 200.

The reason a minimum number of individuals (N_{Min}) is present is to allow the model generation to actually have a fair amount of data to use. The maximum number of individuals (N_{Max}) is valuable to stop the algorithm from having an infinite upper limit in theoretical memory usage.

The remaining subset of the data set (20% for all the experiments presented) is used to validate the approximator's model, resulting in an accuracy value. This accuracy is computed differently depending on the underlying model used, but the result is expected to be a value less than, or equal, to 100%. Should the accuracy be above a given threshold (Acc_{Min}), the approximator is allowed to predict the fitness

```
function GA_Generation(population, approximator):
    approximator.updateExamples(population)
    approximator.learnModelFromExamples()

    median <- medianFitness(population)

    population.elitism()
    population.crossover()
    population.mutation()
    population.reinitialisation()

    if approximator.accuracy >= AccMin then:
        foreach individual in population:
            f <- approximator.predict(individual)
            if f >= PredMin then:
                individual.fitness <- f

    population.evaluateIndividualsWithoutFitness()
```

FIGURE 6.1: Pseudo-code for the integration of an approximator in a GA run

values of any new individuals in the population. Otherwise, the approximator does not get used at all for that generation. Acc_{Min} can be any value between 0% (the predictor gets used all the time) and 100% (the predictor only gets used when it achieves perfect accuracy on the validation set). Choosing the value of Acc_{Min} is one of the considerations presented later on in the chapter.

When applied to an individual, the approximator receives that individual's parameter vector and returns a prediction on the fitness. If the approximator is a classifier, the expected output is mapped from the resulting class and will be covered in a later section. If the predicted fitness is worse than a given threshold based on the previous generation's fitnesses ($Pred_{Min}$), it is accepted and no time consuming evaluation is done for that individual. [1] Values $Pred_{Min}$ could have include the median of the previous generation, the first quartile, the average or an arbitrary hard-coded value. Good values for $Pred_{Min}$ are explored in more depth later on in the chapter.

However, if the predicted fitness is better than the prediction threshold ($Pred_{Min}$), a normal evaluation of that individual is done regardless. The reason we evaluate these individuals is to better assess incremental changes to already good individuals

[1] For the purpose of these experiments, we evaluate each individual regardless of it having been predicted or not, but do not replace an approximator's prediction with the actual simulation results. Instead, we use this evaluation to calculate the number of false negatives generated by the system.

and to, potentially, further improve on the generation's average fitness. Integration of an approximator into the normal run of a GA is presented in Figure 6.1 and Figure 6.2.

Because the approximator is only considered during evaluation, it does not interfere with any other GA optimisations that an experiment designer would want to implement.

It is also worth mentioning that individuals whose fitness was predicted are not added to the approximator's data set.

For these experiments we implemented 3 different approximators: a neural network (NN), a C4.5 decision tree classifier (C4.5), and a *k*-nearest neighbours (k-NN) classifier. These are described in the next three sub-sections.

6.2.2 Neural Network

The neural network employed for these experiments is a feed-forward neural network with a single hidden layer, using the sigmoid activation function. The training is done via backpropagation [157]. This is a simple, yet powerful and fast, neural network that, while rarely the best option for every task, is a safe choice for many situations [158][159][160].

Choosing the network's topology is important when using neural networks [161]. The number of input neurons is equal to the number of game parameters the GA is evolving. The number of output neurons is equal to how many fitness objectives are being tracked, with any parameter minimisation fitness objective being independently tracked.

When training the neural network with the data described in the previous section, some manipulation has to be done. The inputs and outputs are normalized to be within the 0 to 1 range. For input data, this is straightforward, as the ranges parameters could have is already known, as they are defined by the experiment designer.

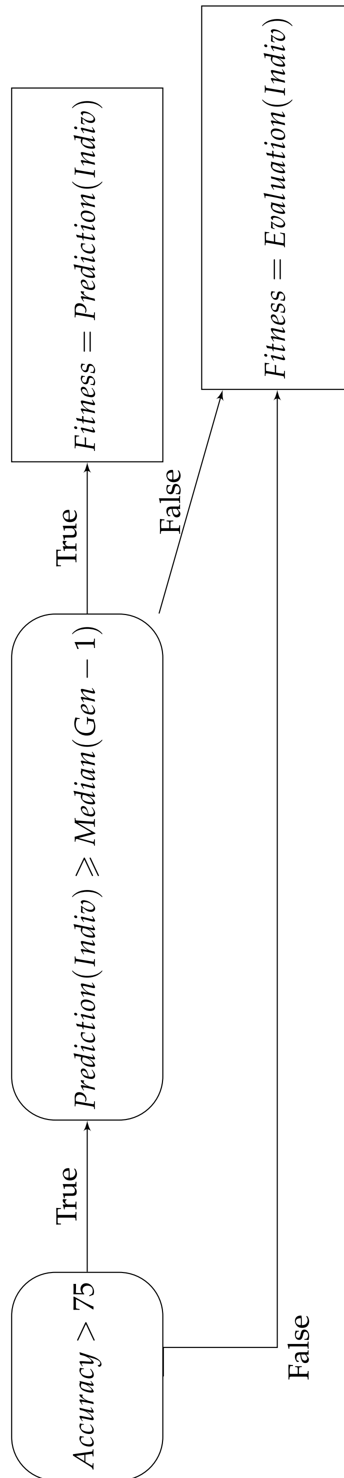

FIGURE 6.2: Diagram of the predictor logic within the GA individual evaluation

TABLE 6.1: The mapping of fitness values to classifier classes

Fitness Class	Fitness Min	Fitness Max
Class 0	0	FirstQuartile(Fitnesses)-1
Class 1	FirstQuartile(Fitnesses)	Median(Fitnesses)-1
Class 2	Median(Fitnesses)	ThirdQuartile(Fitnesses)-1
Class 3	ThirdQuartile(Fitnesses)	Infinity

This normalisation is defined as follows:

$$Normalise(Val) = (Val - Range_{Min}) / (Range_{Max} - Range_{Min}) \tag{6.1}$$

Outputs are maintained in a different manner, as only the minimum values are known beforehand ($Range_{Min} = 0$). This is due to lower fitness values being considered better, with 0 representing a perfect result. The maximum values are tracked by comparing the current maximum values for each output to the ones received from the GA every generation. Should there be output values higher than the ones on record, the ones on record are replaced with the new values. While not a perfect solution due to its stochasticity, this allows the algorithm to adapt to various tasks without prior knowledge.

The accuracy of the neural network is computed by applying the predictor to the validation data, then comparing the outputs to the known correct values. The approach used is the Square Loss function, also known as the Euclidean loss.

6.2.3 C4.5 Decision Trees

C4.5 decision trees [68] is a classification algorithm often used in machine learning due to its transparency. It analyses training data and builds an effective decision tree. It described in slightly more detail in Section 2.3.2.

Because this is a classification algorithm, we had to be able to assign classes to GA individuals, regardless of the game being optimised. The sum of all fitness objectives is taken, which represents the final fitness of an individual, then compared to the first quartile, median, and third quartile of the previous generation's fitness values. Depending on these comparisons, a fitness class from 0 to 3 is assigned to that individual. This mapping can be seen in Table 6.1.

Once the data set has all the fitnesses mapped to fitness classes, the C4.5 model can be generated using the training set, then validated on the validation set. The model is generated anew every generation, as the relevant statistics used to map the fitness to a class change over time.

Validation is done by using the model to predict the class of individuals in the validation set, then comparing the result to the actual class of that individual. The accuracy value is, just as with the neural network approximator, calculated by using the Square Loss function.

Should the resulting model have a high enough accuracy (defined during the start of the experiment), it is then used to predict that generation's unevaluated individuals as described previously. However, the prediction is a class instead of a fitness value. As a result, a second mapping, from fitness class to fitness value, has to be done. For simplicity, for each class except class 3, the fitness value assigned is the value of Fitness Max in Table 6.1. For individuals classified as class 3, they receive a fitness value equal to $ThirdQuartile(Fitnesses) + 1$. This may be very close to an individual of class 2, however the alternative, infinity, seemed less desirable.

6.2.4 k-Nearest Neighbours

The k-nearest neighbours algorithm [69] is a non-parametric method commonly applied for classification and regression. It is described in more detail in Section 2.3.3. The reason behind choosing it for this task is due to the likelihood of similar solutions belonging to the same fitness class. Very close parameter sets have a higher chance to result in the same gameplay changes, thus also have very close fitness.

Apart from the model being generated and used for classification, everything is done in the exact same way as with the C4.5 algorithm. Mapping the fitness values of the data set to the classes (see Table 6.1) and then back to fitness values, as well as calculating the validation error, are done identically.

For these experiments, the value of k was chosen to be 3, while the number of classes we used to split fitnesses in was the same as the one chosen for C4.5 decision trees (4).

6.3 Standard Fitness Function Experiments

6.3.1 OneMax

We made use of the OneMax implementation described in the literature review in Section 2.2.2.

This experiment is here to highlight how many evaluations are required to achieve a perfect result rather than focusing on the actual time spent running the computations, making OneMax a good choice due to its extremely low computational footprint. It is also a sanity test for our algorithm to assess whether it can handle a linear fitness environment.

6.3.2 Trap

The second problem employed was the Trap implementation described in the literature review in Section 2.2.2. This is a significantly more challenging problem than OneMax due to its deceptive landscape. For this experiment, $N = 16$, $a = 100$, $b = 75$ and $l = 65535$.

Training an approximator on randomly sampled data is very likely to result in a network that does not know of the global optimum peak at location $X = 0$. It is expected that the best result would almost always be $Fitness = 25$ at $X = 65535$.

We will run several several experiments with various values for Z. This is done to assess how our algorithm handles increasingly more deceptive fitness landscapes. We do not expect our optimisations to be particularly successful in this scenario.

Similarly to OneMax, Trap is a very cheap evaluation and we only consider how many evaluations were required to achieve the best fitness possible.

6.3.3 Genetic Algorithm

The evolutionary algorithm employed is very similar to the one employed in Chapter 5. It is a variant of a generational GA with two-point crossover (applied with a rate of 35%), a specialised mutation operator (applied with a per-individual rate of 35% and described later) and elitism (applied to the top 15% of the population). The

TABLE 6.2: Neural network neuron count per layer

Experiment	Input	Hidden	Output
OneMax	50	18	1
Trap	16	18	1

final 15% of the each generation is randomly sampled from the search space through reinitialisation [152].

The mutation operator for both OneMax and Trap involves a bit flip for each allele mutated. It was applied with a (per allele) mutation rate of 5% for OneMax and Trap (meaning that on average 5% of the elements of an individual would be mutated).

The population size was 20 for both experiments. All experiments used tournament selection, with a tournament size of 3. Experiments ran until a perfect solution was evolved or 500 generations passed, whichever happened first.

6.3.4 Neural Network

For these experiments, the number of hidden layer neurons was chosen arbitrarily to be 18 respectively. The exact structures can be seen in Table 6.2.

The network is trained using data generated by the GA. Every time the GA does the evaluation of an individual, the parameters and fitness values are given to the predictor for storage in the data set.

Training of the neural network is only done if there is a minimum number of individuals in the data set (N_{Min}), as described in Section 6.2.1.

6.3.5 Experiments

For each function (OneMax and Trap), we ran 30 runs normally, without the approximator, and then the same number of runs with the approximator. To better compare results, we paired each run without a approximator to one with a approximator. Each pair had the same starting populations and random seeds.

For the OneMax experiments we used a value of $N = 50$.

During the trap experiment, we looked at 3 different values of Z: $Z = 10000$, $Z = 4000$ and $Z = 1000$.

From each experiment we collected several metrics:

- average fitness per generation
- best fitness per generation
- fitness evaluations done per generation, as this value when using the approximator can be less than the population size

6.3.6 OneMax Results

We kept track of how many fitness calculations the GA was doing. In the experiment without an approximator, it was always 20 calculations in the first generation, plus 17 for every generation onwards (out of 20 individuals, 3 persist through elitism and don't need evaluation). Each run ended when it reached a best fitness value of 0 or 500 generations passed, whichever happened first.

Every run, with and without an approximator, managed to achieve a perfect solution within 100 generations. As a result, the important metric is how many evaluations of the fitness function were required to achieve that perfect solution.

Figure 6.3 shows a boxplot of the number of evaluations required by the GA to achieve the perfect solutions. Without an approximator, the mean number of evaluations required was 953.9, with a median of 904 and a standard deviation of 270.5. With the approximator, the mean number of evaluations required was 487.1, with a median of 485.5 and a standard deviation of 167.7.

Using the approximator proved to be the better choice in almost every single run, with markedly fewer evaluations required. The average number of evaluations to achieve each fitness can be seen in Figure 6.4.

Doing a two sample Wilcoxon Signed Rank test on the resulting iterations, with the null hypothesis that the second sample was worse than the first gives us a p value of $p = 8.2 \times 10^{-7}$, which is a highly significant result and confirms our assumption that using the approximator would result in fewer evaluations of the OneMax function.

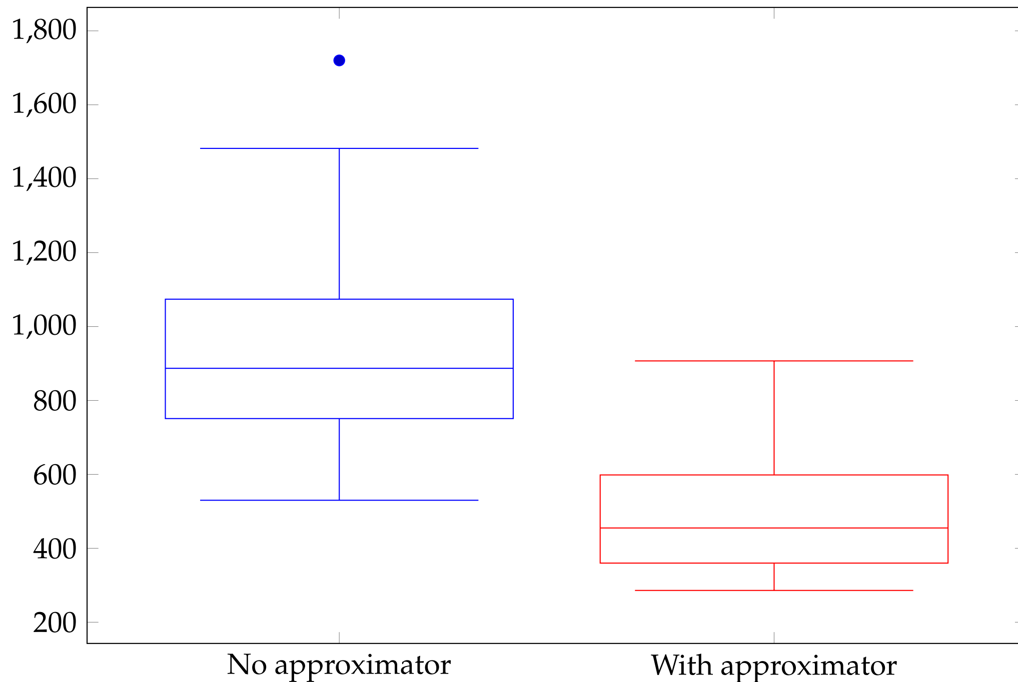

FIGURE 6.3: Boxplots for number of fitness evaluations required to achieve perfect fitness in OneMax w/o and w/ an approximator

The number of fitness calculations is, on average, 47.7% smaller with the approximator than the version without it.

6.3.7 Trap Results

For the trap experiment, we expected that the results would depend on the initial sampling and that, especially for the hardest scenario ($Z = 1000$), the approximator would be prone to overfitting the deceptive peak, as it is the one most likely to provide the most samples for training.

All runs were paired, starting with the same random seed and populations, as in the OneMax experiment. The GA was allowed to run for up to 500 generations before it was terminated.

For all values of Z, the GA without a approximator achieved a perfect fitness of 0.

As expected, the quality of the approximator proved to heavily depend on the initial population, as well as the randomly generated individuals in the first few generations. The number of runs that failed to achieve a perfect solution can be seen in Table 6.3.

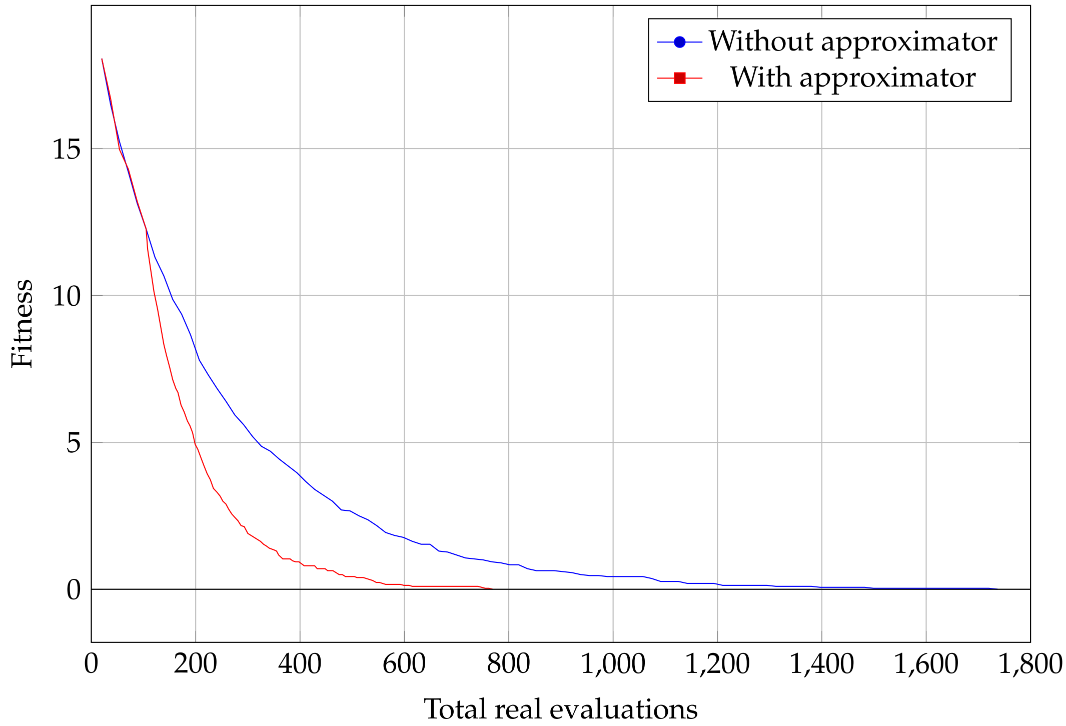

FIGURE 6.4: Total evaluations required to achieve a perfect fitness in OneMax, w/o and w/ the neural network approximator

TABLE 6.3: Number of runs that failed to get a perfect solution (out of 30) for Trap

	$Z = 10000$	$Z = 4000$	$Z = 1000$
None	0	0	0
Approximator	7	14	28

For $Z = 10000$, to achieve success, it took the GA without an approximator 389.5 evaluations on average. In the 23 runs in which the GA with the approximator successfully achieved a perfect fitness ($Runs_s$), the average number of evaluations required was 227.7 (N_s). Doing a two sample Wilcoxon Signed Rank test on these results, with the null hypothesis that the second sample was worse than the first, gives us value of $p = 0.006$, a significant value. The 7 runs that failed to achieve a perfect fitness ($Runs_f$) required an average of 567.4 evaluations (N_f).

For $Z = 4000$, the GA without a approximator achieved a perfect solution, on average, in 487.5 evaluations. In the 16 runs in which the GA with the approximator successfully achieved a perfect fitness ($Runs_s$), the average number of evaluations required was 212.2 (N_s). Doing a two sample Wilcoxon Signed Rank test on the successful runs, with the null hypothesis that the second sample was worse than the

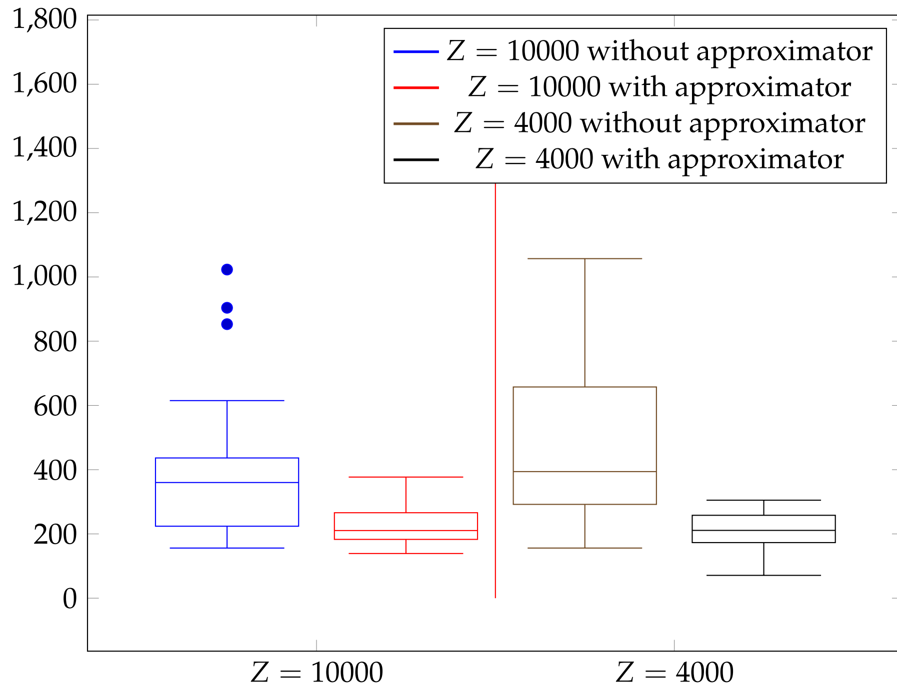

FIGURE 6.5: Boxplots for number of iterations required to achieve perfect fitness in Trap without and with a approximator.

first, gives us a p value of $p = 0.01$, a significant value. The 14 runs that failed to achieve a perfect fitness required an average of 394 evaluations (N_f).

For the hardest task, where $Z = 1000$, it is interesting to note that, with the approximator, the number of evaluations never went above 1000 and averaged at 346.5 with a median of 265. The GA without an approximator GA needed, on average, 1997 evaluations to achieve a perfect solution. Unfortunately, the GA with the approximator failed to find the global peak in almost every run. In the 2 successful runs, the average number of evaluations was $N_s = 257.5$. In the 28 unsuccessful runs, the average number of evaluations was $N_f = 352.9$.

Let us assume that p is the probability of solving the problem in one run, which we can estimate from actual runs as $p = Runs_s / Total_{Runs}$, where $Runs_s$ is the number of successful runs. Let us further assume that N_s is the average number of fitness evaluations spent when the problem got solved, and N_f is the number of fitness evaluations spent when the problem was not solved. We can, as a result, calculate [162] the cost (in terms of fitness evaluations) of solving the problem as seen here:

$$E = \frac{pN_s + (1-p)N_f}{p} \tag{6.2}$$

E represents the expected number of fitness evaluations one needs to spend to achieve one 100% correct solution. So, for $Z = 10000$, we have $E = 400.4$. For $Z = 4000$ we have $E = 557.0$. For $Z = 1000$ we have $E = 5198.1$, which indicates that the hardest problem is approximately 10 times harder than the other two.

Boxplots of the number of evaluations required to achieve the perfect fitness for $Z = 10000$ and $Z = 4000$ can be seen in Figure 6.5.

6.4 *Ms. Pac-Man* Experiments

6.4.1 Environment

For the *Ms. Pac-Man* experiments, the game's rules were altered in a similar fashion to the studies done in Section 5.2 of Chapter 5, with an additional fitness metric. Not only is the number of "wins" (scores over 1500 points) counted, but also what the average score was. This is another measure of difficulty and can allow a designer to further optimise their goal.

The requirement to change the parameters by as little as possible was also kept, with the same value ranges as in Table 5.1.

The same deterministic agent described in Section 5.2.1 was used to simulate gameplay. The number of games played per simulation was also kept at 50.

Fitness Evaluation

The same pipeline was used as the one presented in Chapter 5, with changes to the evaluators as described previously.

Weights were chosen based on the results of experiments done in the previous chapter and the updated parameter list reported in Table 6.4. The updated metrics list, including the new element to the fitness function, can be seen in Table 6.5.

Formally, the fitness function for this set of experiments can be written as:

$$Fitness = W + M + \Delta_P \tag{6.3}$$

TABLE 6.4: *Ms. Pac-Man* parameters to be changed, their displacement ranges and their decimal accuracy

	Parameter	Min	Max	Accuracy	Weight
Δ_1	PacMan's speed	-3	+5	10^0	100
Δ_2	First Ghost's chase speed	-3	+5	10^0	100
Δ_3	Second Ghost's chase speed	-3	+5	10^0	100
Δ_4	Third Ghost's chase speed	-3	+5	10^0	100
Δ_5	Fourth Ghost's chase speed	-3	+5	10^0	100
Δ_6	First Ghost's flee speed	-3	+5	10^0	100
Δ_7	Second Ghost's flee speed	-3	+5	10^0	100
Δ_8	Third Ghost's flee speed	-3	+5	10^0	100
Δ_9	Fourth Ghost's flee speed	-3	+5	10^0	100

TABLE 6.5: *Ms. Pac-Man* metrics to be used in evaluation alongside their desired values and weights for the approximator experiments

	Name	Desired	Evaluator	Weight
W	Win-rate achieved by the agent	0.5	Average	$C_W = 5000$
M	Average score achieved by the agent	1500	Average	$C_M = 1$
Δ_P	Sum of parameter absolute displacements	0	Minimise	$W_\Delta = 100$

$$W = |WR - DWR| \times C_W \tag{6.4}$$

$$M = |\overline{Scores} - DMean| \times C_M \tag{6.5}$$

$$\Delta_P = \sum_{i=0}^{n} |\Delta_i| \times W_\Delta \tag{6.6}$$

In the functions W and Δ_P, everything is identical to the experiments presented in Section 5.2. In the average score optimisation function M, *Scores* is a list of point scores achieved by the agent, $\overline{Scores}$ = the average value of *Scores*, *DMean* = desired mean and C_M = average score bias factor.

Genetic Algorithm

The same genetic algorithm as the one described in Section 5.2.1 was used.

To test the value of the approximators in various scenarios, we made use of two parameter sets for the GA. One set was among the ones performing less than optimally, as that is a likely scenario for someone using these algorithms without much GA experience. The second set was the best performing one resulting from experiments in Section 5.2.3 from the previous chapter and was used as a benchmark for comparing between runs with unoptimised GA parameters and approximators, and what could be considered the optimised GA parameters without approximators. Both sets used a population size of 100, a tournament size of 6 and elitism of 20%. The differing parameters, with regards to what settings are used in each configuration, can be seen in Table 6.6.

TABLE 6.6: Genetic algorithm parameter sets tested for *Ms. Pac-Man* approximator experiments

Config	Crossover	Mutation	Mutation Rate	Reinit
Unoptimised	30%	30%	50%	20%
Optimised	40%	40%	20%	0%

Choosing the Approximator's Accuracy Threshold

As described in Section 6.2.1, the approximator's accuracy threshold Acc_{Min} can be any value between 0% and 100%. Instinct and experience warrant a high value, as a low value can result in many false predictions. However, too high and the system rarely, if ever, gets used, as it might have problems achieving that accuracy given the search space.

For each combination of GA parameter set and game, we also tried three values for Acc_{Min}. The values tested were 0.60, 0.75 and 0.85. This was to assess the impact of this variable on the algorithms. We believe that any higher, or lower, value would not bring much benefit to the algorithm.

Choosing the Approximator's Prediction Acceptance Threshold

Similarly to the accuracy threshold, the approximator also makes use of a prediction threshold ($Pred_{Min}$), where a prediction is only kept if the fitness predicted is worse than a given value. This value could be a constant chosen at the beginning of the experiment, but that would require a lot of prior knowledge on both the fitness landscape of the problem and the trajectory of the GA. Instead, we have chosen to have the approximator set this value based on the previous generation's fitnesses.

Increasing this threshold should result in fewer predictions being accepted, while decreasing it would result in a much more aggressive approximator and fewer individuals having their real fitness evaluated. To test this, we also tried three different values for $Pred_{Min}$: the first quartile of the previous generation, the median of the previous generation, as well as the third quartile of the previous generation.

6.4.2 Experiments

For each combination of value tested for approximator accuracy threshold and prediction acceptance threshold, we ran 20 GA runs without an approximator, and then the same number of runs with each of the 3 approximators. All of these runs were using the unoptimised GA parameter set and were paired, such that the *i*th run of each experiment had the same random seed and starting populations.

For all the experiments both N_{Min} and N_{Max} (the minimum number of individuals stored in the approximator's data set before being applied and the maximum number stored respectively) were set to 200.

From each experiment we collected several metrics:

- average fitness per generation
- best fitness per generation
- fitness evaluations done per generation, as this value can be less than the population size when using the approximator
- false negatives generated by the approximator. These are instances where the approximator assigned an individual a fitness worse than the prediction threshold, however the real result would have been better than it (and should have been evaluated instead of keeping the prediction)

Afterwards we took one of the better performing approximator configurations and ran 20 runs for each of the 3 approximators on the optimal GA parameter set to see how performance changed in this scenario.

Finally, we compared results from approximator runs on the unoptimised GA to the vanilla optimised GA ones. This was to see if using the approximator could possibly counteract the effects of suboptimal GA parameters.

6.4.3 Results

Data

Data on which the results described are based on is available in Appendix C.

TABLE 6.7: Average run times when using each of the 3 machine learning algorithms as approximators, compared to using no approximator, in the *Ms. Pac-Man* experiments.

Experiment	Average run time
Ms. Pac-Man Base	15s / run
Ms. Pac-Man C45	17s / run
Ms. Pac-Man Neural Network	19s / run
Ms. Pac-Man KNN	17s / run

Overview of Runs on Unoptimised GA Configuration

The immediate thing to notice is that the runtimes were very similar between runs without an approximator and those with one. The runs using approximators took marginally more time, in absolute terms, to use the same evaluation budget as the runs without approximators. This can be seen in Table 6.7. This highlights that the approximators themselves add very little to the computation, which is dominated by the fitness evaluation.

These algorithms for fitness approximation can have various use cases. Some might benefit from better results faster, while others will be willing to wait a long time for more accurate results.

Something that required immediate analysis was to see how runs behaved when comparing their best fitnesses generation by generation, ignoring any time saved by skipping evaluations. Most of the times, comparing the same two generations between runs with and without predictors showed the ones using predictors behaving slightly worse, as can be seen in Figure 6.6. However this does not tell the entire story regarding performance.

To better assess the quality of the evolved results, we kept track of the best fitnesses achieved at every time point, by every run. Snapshots were taken every 20 evaluations, comparing the best fitnesses for each run with, and without, the relevant approximator. This allowed for statistical significance to be calculated, using the two sample Wilcoxon Signed Ranked test, with the null hypothesis that the approximator yielded worse results, for each of those snapshots. The final result is a plot of statistical significance (p value) over total evaluations.

An example of the statistical significance graph can be seen in Figure 6.7, where

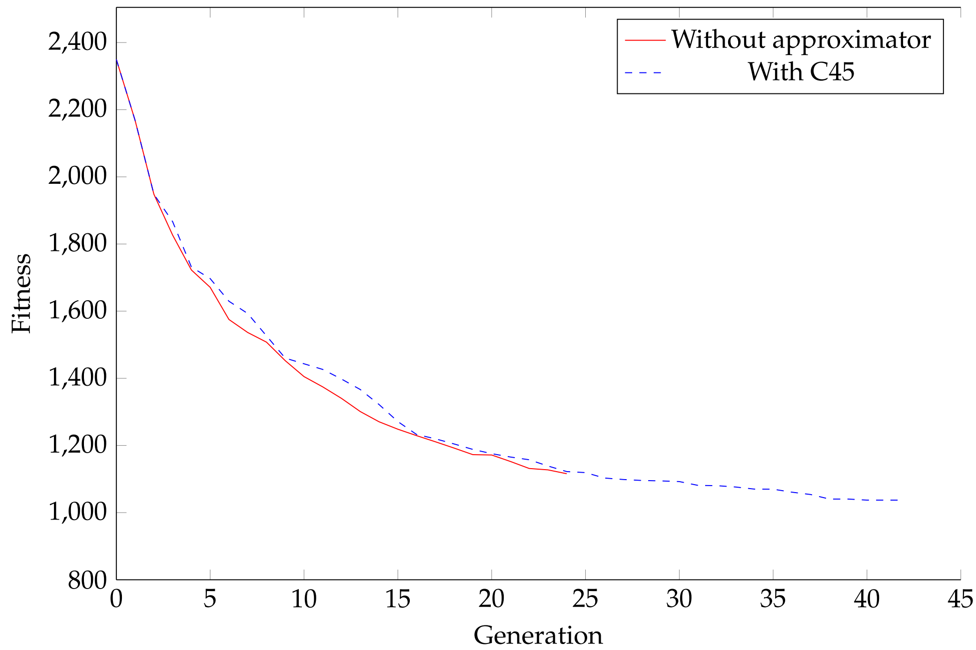

FIGURE 6.6: Average best fitness values achieved after a given number of generations by the GAs without and with the C4.5 decision trees approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment

both the relationship between best fitness and total evaluations used can be seen, as well as the aforementioned statistical significance at every time point. This data is comparing the unoptimised *Ms. Pac-Man* GA results without an approximator to its counterpart where a neural network approximator, with an accuracy threshold $Acc_{Min} = 0.75$ and a prediction threshold $Pred_{Min}$ of median, was used.

We tracked the percentage of time that the best fitness with the approximator is statistically different ($p <= 0.05$ with the Wilcoxon Signed Rank Test) from the best fitness without it for different segments of time. The 2000 evaluation budget was split in 4 segments: the early quarter (0 to 500 evaluations), the middle-early quarter (501 to 1000 evaluations), the middle-late quarter (1001 to 1500 evaluations) and the late quarter (1501-2000 evaluations).

Another important element is the number of predictions made by each approximator, as well as how often they were wrong (false positives). These results can be seen in Table 6.8.

These results allow us to gain more insight into the changes the GA behaviour experiences when using the various approximators.

TABLE 6.8: Average number of predictions made by each approximator configuration tested in the *Ms. Pac-Man* experiments, as well as the average number of false negatives generated as a result.

Approx	Acc_{Min}	$Pred_{Min}$	Predictions	False negatives
NN	60	First	2568.9	1.45%
NN	60	Median	1586	4.97%
NN	60	Third	106.45	8.45%
NN	75	First	2035.3	1.78%
NN	75	Median	1333.15	5.00%
NN	75	Third	93.95	6.60%
NN	85	First	390.15	2.92%
NN	85	Median	103.95	6.16%
NN	85	Third	10.4	3.37%
C45	60	First	5395.75	6.06%
C45	60	Median	1851.65	13.46%
C45	60	Third	722.9	0.00%
C45	75	First	4241.85	5.44%
C45	75	Median	1691.2	13.19%
C45	75	Third	631.05	0.00%
C45	85	First	700.9	4.02%
C45	85	Median	540.15	13.97%
C45	85	Third	181.6	0.00%
KNN	60	First	4292.1	4.29%
KNN	60	Median	1383.65	7.34%
KNN	60	Third	441.7	0.00%
KNN	75	First	1758.15	3.36%
KNN	75	Median	866.75	7.00%
KNN	75	Third	232.4	0.00%
KNN	85	First	44.3	2.48%
KNN	85	Median	41.85	9.20%
KNN	85	Third	8.75	0.00%

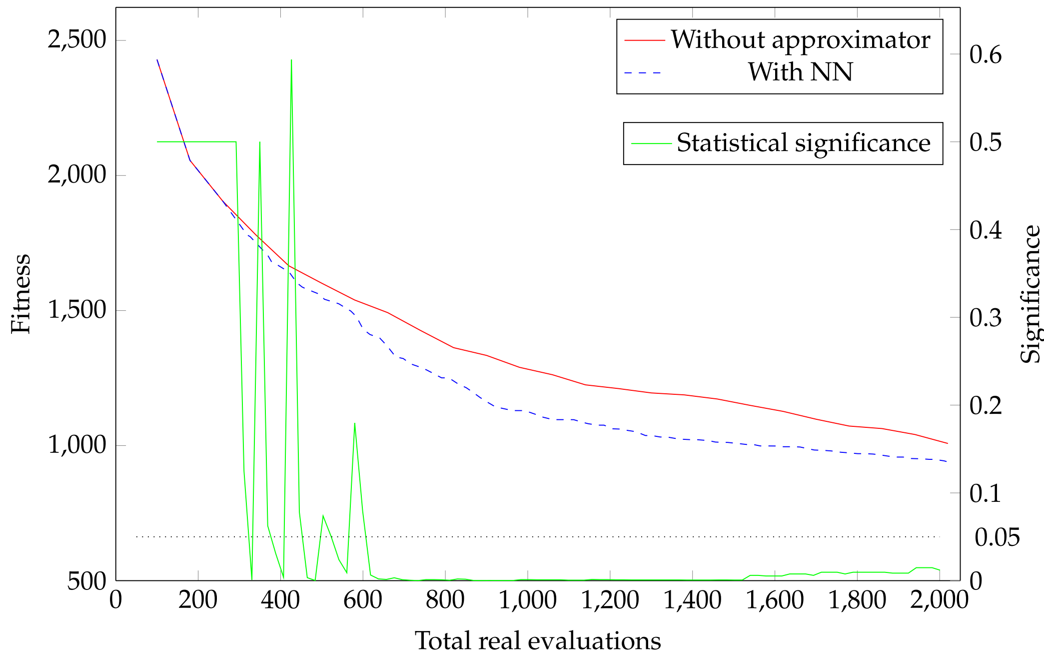

FIGURE 6.7: Average best fitness values achieved after a given number of evaluations by the GAs without and with the neural network approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment. Significance of difference between the paired results is also plotted. Lower values are better

Performance of Various Approximators

Neural Networks The neural network approximator behaved admirably, managing significant quality increases for the same computational cost under most configurations. This can be seen from the aggregate results in Table 6.9. Particularly, when the prediction threshold was set to median and the accuracy threshold was not too high, the results were significantly better at all points fairly early in each run.

At no point were runs using the neural network approximator worse than runs without it, as made clear by the results. The worst scenario was with the prediction threshold set to third quartile and the highest accuracy threshold (0.85), where the use of the approximator brought no advantage or disadvantage compared to not using it.

Further optimising the neural network structure or architecture could very likely further improve results.

TABLE 6.9: Percentage of time that the neural network approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)	Best segment(s)
0.60	FirstQuart	47%	0%	Middle-early
0.60	Median	82%	0%	Middle-late and late
0.60	ThirdQuart	15%	0%	Middle-late
0.75	FirstQuart	42%	0%	Middle-early
0.75	Median	81%	0%	Middle-late and late
0.75	ThirdQuart	14%	0%	Middle-late
0.85	FirstQuart	21%	0%	Middle-early
0.85	Median	17%	0%	Middle-early
0.85	ThirdQuart	0%	0%	N/A

TABLE 6.10: Percentage of time that the C4.5 decision tree approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)	Best segment(s)
0.60	FirstQuart	53%	0%	Middle-early
0.60	Median	58%	0%	Middle-early
0.60	ThirdQuart	50%	0%	Middle-late
0.75	FirstQuart	69%	0%	Middle-early, middle-late
0.75	Median	75%	0%	Middle-late
0.75	ThirdQuart	40%	0%	Middle-late
0.85	FirstQuart	6%	0%	Middle-early, middle-late
0.85	Median	22%	0%	Middle-late
0.85	ThirdQuart	8%	2%	Middle-early

TABLE 6.11: Percentage of time that the k-nearest neighbour approximator proved to be significantly better, or worse, than using no approximator in the *Ms. Pac-Man* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)	Best segment(s)
0.60	FirstQuart	50%	0%	Middle-early
0.60	Median	85%	0%	Middle-early to late
0.60	ThirdQuart	57%	1%	Middle-late
0.75	FirstQuart	34%	0%	Middle-late
0.75	Median	44%	0%	Middle-late and late
0.75	ThirdQuart	46%	0%	Late
0.85	FirstQuart	0%	0%	N/A
0.85	Median	0%	0%	N/A
0.85	ThirdQuart	0%	0%	N/A

C4.5 Decision Trees Looking at the results in Table 6.10, the C4.5 decision trees behaved quite differently. Significantly better results appeared earlier, in the middle-early quarter of evaluations (more data is available in Appendix C.1). However, results eventually converged with the ones that did not use an approximator in the late quarter (1501 to 2000 evaluations).

On average, runs using the decision trees approximators were also significantly better more often compared to those using the neural network approximators (35.5% average for neural networks compared to 42.3% average for decision trees). The best performing configuration, however, was slightly worse (82% best for neural networks compared to 75% for decision trees).

The results in the early quarter were also much better with decision trees compared to neural networks. All this marks decision trees as great for achieving good results fast, at least for this task.

The configuration where $Acc_{Min} = 0.85$ and $Pred_{Min}$ set as the third quartile resulted in a couple of snapshots where the runs with the decision tree approximator were significantly worse than without. This entire configuration proved to be weak overall at improving performance.

k-Nearest Neighbours Using the *k*-nearest neighbour algorithm as the approximator had results more similar to the neural network than the decision trees. These can

TABLE 6.12: Average proportion of time in which each combination of approximator and accuracy threshold was statistically significant, regardless of prediction threshold, in the *Ms. Pac-Man* experiments.

Approximator	Acc_{Min}	Average proportion of significance (%)
NN	0.60	48.0%
NN	0.75	45.6%
NN	0.85	12.6%
C45	0.60	53.6%
C45	0.75	61.3%
C45	0.85	12.0%
KNN	0.60	64.0%
KNN	0.75	41.3%
KNN	0.85	0.0%

be seen in Table 6.11. However, it had the best late quarter (1501 to 2000 evaluations) results of the 3 machine learning algorithms.

The combination of KNN, $Acc_{Min} = 0.60$ and $Pred_{Min}$ set as the third quartile resulted in the second configuration in which, at one evaluation snapshot, the fitnesses with the approximator were significantly worse than without. However, the following quarters were entirely dominated by the approximator runs.

It also had the configuration with the highest period of time significant (at 85% of the time) when $Acc_{Min} = 0.60$ and $Pred_{Min}$ set to median. Except for the early segment, all time snapshots showed significantly better results with the approximator than without.

Given these results, this approximator seems best suited for runs with a higher budget, as the best results appear after more generations compared to the other 2.

Performance of Using Various Accuracy Thresholds

Looking at an average of the results observed in the previous sections, and available in Table 6.12, it is immediately obvious that an accuracy threshold of 0.85 is too high. This is the result of the approximators being unable to reach that high of an accuracy often enough to get used at all.

The other two values of the threshold were much better, with $Acc_{Min} = 0.60$ being only slightly better than 0.75. There was no obvious relation between this

TABLE 6.13: Average proportion of time in which each combination of approximator and prediction threshold was statistically significant, regardless of prediction threshold, in the *Ms. Pac-Man* experiments, as well as average predictions computed.

Approximator	$Pred_{Min}$	Average proportion of significance (%)	Predictions
NN	First	36.6%	1664.78
NN	Median	60.0%	1007.70
NN	Third	9.6%	210.80
C45	First	42.6%	3446.17
C45	Median	51.6%	1361.00
C45	Third	32.6%	511.85
KNN	First	28.0%	2031.52
KNN	Median	43.0%	764.08
KNN	Third	34.3%	227.62

threshold and which segment the approximator yielded best results in (i.e. using one value over another affecting the quality of results in middle-late stage of the GA).

Given these results, there might be other values that would be valuable to explore, some potentially lower than 0.5 accuracy. These could be worth exploring in future work.

Performance of Using Various Prediction Thresholds

Looking at the results in Table 6.13 shows that the number of predictions went down as the prediction threshold went up. This was unsurprising, as a majority of new individuals in a population inhabit the area between the first quartile and median of fitnesses.

Having the prediction threshold as median or first quartile proved to be much better than having it set to third quartile, with both more predictions and better fitnesses over time. Overall, having this threshold set to median had the best results when comparing GA performance. Despite producing more predictions, having the prediction threshold set to the first quartile did not result in the top proportion of significance.

However it is interesting to note that having the threshold set as third quartile resulted in almost no false negatives, exception being a small number of them when

TABLE 6.14: In the *Ms. Pac-Man* approximator experiments using the optimised GA parameters: percentage of time that using the approximator resulted in significantly better results; the average number of predictions generated by the approximator each run and the average percentage of which were false negatives.

Approximator	% Significant	Predictions	False Negatives
Neural Network	0%	9.6	28.6%
C45	2%	610.6	10.1%
KNN	0%	206.1	8.7%

the neural network was used as approximator. This can be seen in Table 6.8.

Overview of Runs on the Optimised GA Configuration

The configuration chosen for the approximators was the one with $Acc_{Min} = 0.75$ and $Pred_{Min}$ set to median. This one proved to be one of the better ones and would be a good benchmark for testing our approach on the GA with optimised parameters.

A brief look at the results in Table 6.14 show that the approximators did not impact the performance of the GA by much, if at all, in either direction. The results were virtually never significantly better or worse with any of the 3 configurations. This means that having optimal GA parameters results in not needing the added complexity of the approximators. Further results can be seen in Figure 6.8.

It is also interesting to note that the number of predictions plummeted drastically compared to when approximators were applied to the unoptimised GA. We believe this is due to a couple of factors.

Firstly, due to the optimisation of the GA, it converges faster to better solutions. This gives the approximator less time to bring its own improvements, as the GA hones onto better areas of the search space sooner, despite being sampled at the same rate.

Secondly, exploring the search space faster results in a harder environment for the approximators to map. New information about the relationships between parameters is being discovered and, as a result, approximator accuracy suffers.

This is a reassuring result, as one of the initial plans for this methodology was not only to improve performance, but, more importantly, not to hamper performance.

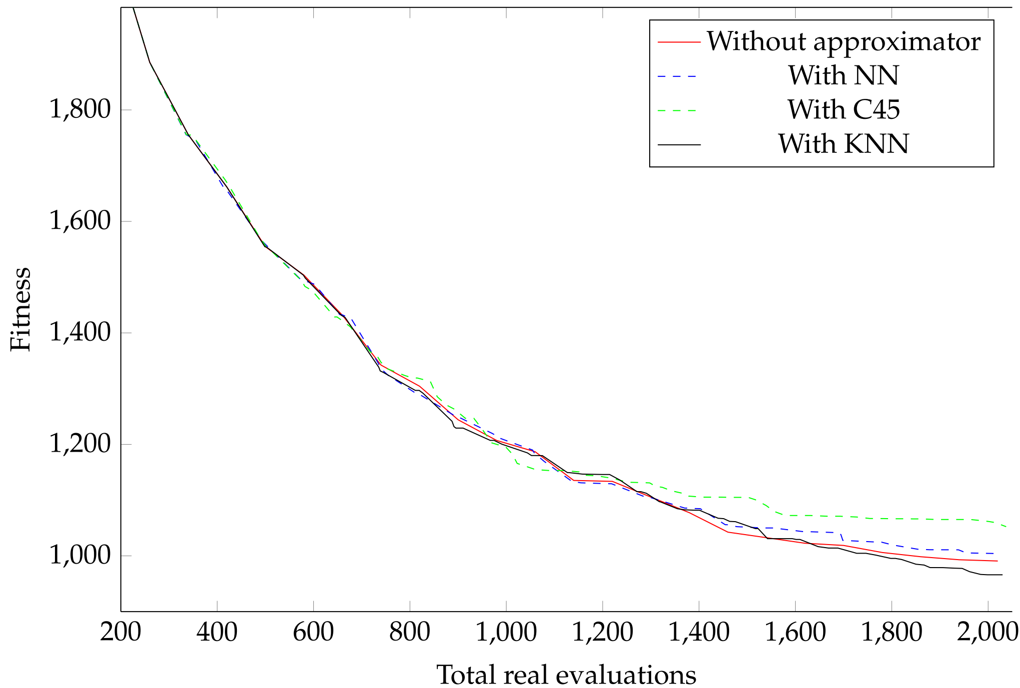

FIGURE 6.8: Average best fitness values achieved after a given number of evaluations by the GAs without and with each of the tested approximators, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *Ms. Pac-Man* experiment with the optimised GA parameter set. Lower values are better.

The approximator did not negatively impact the GA run, even when the GA's parameters were optimised, something extremely valuable.

Comparing Approximators on Unoptimised GA Parameter Set to Optimised GA Parameter Set

It was interesting to compare the runs with approximators on the unoptimised GA parameter set to the set of runs without an approximator, but with the optimised GA parameter set. First of all, comparing the unoptimised runs, without any approximators, directly to the optimised one showed clear dominance by the optimised ones 70% of the time.

Most approximator runs, with the exception of a select few for each approximator type, proved to be worse than simply running the GA with an optimised parameter set.

There were, however, 5 exceptions (out of 27): 3 of them using various configurations of C4.5 decision trees, and one for each of the other 2 approximators. While the improvements were minimal, they were present. This is likely caused by the

approximators being used in avoiding weak individuals that would have been generated by the unoptimised GA, individuals that the optimised one would not have created in the first place.

This is reassuring, as this shows that in a time-critical scenario, optimised parameters could be ignored in favour of an approximator.

6.5 *TORCS* Experiments

6.5.1 Environment

The environment is virtually identical to the one described in Section 5.4.1. The only addition to this is the presence of approximators in some configurations.

Similarly, the GA employed is the same one used in the *Ms. Pac-Man* experiments. Its structure and parameters are described in more detail in Section 6.4.1. No changes were required due to the flexibility of the balance specification language used.

6.5.2 Experiments

An identical setup to the one in the previous section, with *Ms. Pac-Man* experiments, was used. We ran 20 GA runs without an approximator, and then the same number of runs with each combination of approximator, accuracy threshold and prediction threshold. All runs were paired, such that the ith run of each experiment had the same random seed and starting populations. The same metrics were collected.

Afterwards we took one of the better performing approximator configurations and ran 20 runs for each of the 3 approximators on the optimal GA parameter set to see how performance changed in this scenario.

Finally, we compared results from approximator runs on the unoptimised GA to the vanilla optimised GA ones. This was to see if using the approximator could possibly counteract the effects of inefficient GA parameters.

6.5.3 Results

Data

Data on which the results described are based on is available in Appendix C.

Overview of Runs on Unoptimised GA Configuration

Just as with the *Ms. Pac-Man* experiments, run times were not heavily affected by the presence of an approximator. The average times for each configuration can be seen in Table 6.15.

TABLE 6.15: Average run times when using each of the 3 machine learning algorithms as approximators, compared to using no approximator, in the *TORCS* experiments.

Experiment	Average run time
TORCS Base	5m40s / run
TORCS C45	5m42s / run
TORCS Neural Network	5m46s / run
TORCS KNN	5m42s / run

The data collection was done in a very similar manner as done with the *Ms. Pac-Man* experiments. Results, however, painted a very different picture.

The results for all approximators can be seen in Table 6.16. The immediately obvious observation is the fact that neural networks barely, if at all, managed to be employed. In most runs they did not achieve the required accuracy even once.

Performance of Various Approximators

Neural Networks As mentioned previously, the neural networks did not make a single prediction in any run of any of the configurations where it was the approximator of choice. This resulted in GA runs that, with or without the approximator, behaved identically. These results can be seen in Table 6.17.

Further analysis of the results highlighted the reason behind this behaviour. Individuals with extremely bad fitnesses, while correctly identified by trained neural networks as undesirable, were assigned fitness values fairly far from the real values. This resulted in failing the validation tests, even when the accuracy threshold was at its lowest ($Acc_{Min} = 0.60$), due to the accuracy being calculated based on the difference between true answer and predicted answer, rather than an individual's fitness class.

While the neural network was accurately mapping the relationships between parameters and fitness for individuals below the third quartile of fitnesses, the wide spread of bad individuals caused it to never be used.

C4.5 Decision Trees The C4.5 decision tree approximator was, however, successful in providing improved results over the vanilla GA.

TABLE 6.16: Average number of predictions made by each approximator configuration tested in the *TORCS* experiments, as well as the average number of false negatives generated as a result.

Approx	Acc_{Min}	$Pred_{Min}$	Predictions	False negatives
C45	60	First	5214.65	192.20
C45	60	Median	1950.70	144.70
C45	60	Third	728.85	0.00
C45	75	First	4557.00	174.00
C45	75	Median	1907.20	139.25
C45	75	Third	715.95	0.00
C45	85	First	2421.60	87.50
C45	85	Median	1211.00	119.40
C45	85	Third	455.00	0.00
KNN	60	First	4692.50	353.95
KNN	60	Median	1733.80	405.20
KNN	60	Third	659.65	0.00
KNN	75	First	2635.85	135.65
KNN	75	Median	1265.80	281.40
KNN	75	Third	492.30	0.00
KNN	85	First	46.60	1.05
KNN	85	Median	31.30	7.80
KNN	85	Third	15.65	0.00
NN	60	First	0.00	0.00
NN	60	Median	0.00	0.00
NN	60	Third	0.00	0.00
NN	75	First	0.00	0.00
NN	75	Median	0.00	0.00
NN	75	Third	0.00	0.00
NN	85	First	0.00	0.00
NN	85	Median	0.00	0.00
NN	85	Third	0.00	0.00

TABLE 6.17: Percentage of time that the neural network approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)
0.60	FirstQuart	0%	0%
0.60	Median	0%	0%
0.60	ThirdQuart	0%	0%
0.75	FirstQuart	0%	0%
0.75	Median	0%	0%
0.75	ThirdQuart	0%	0%
0.85	FirstQuart	0%	0%
0.85	Median	0%	0%
0.85	ThirdQuart	0%	0%

TABLE 6.18: Percentage of time that the C45 decision tree approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)	Best segment(s)
0.60	FirstQuart	31%	0%	Middle-early
0.60	Median	32%	0%	Middle-early
0.60	ThirdQuart	34%	0%	Middle-early
0.75	FirstQuart	26%	0%	Early
0.75	Median	32%	0%	Middle-early
0.75	ThirdQuart	29%	0%	Middle-early
0.85	FirstQuart	22%	0%	Early
0.85	Median	28%	0%	Middle-early
0.85	ThirdQuart	43%	2%	Middle-early

Table 6.18 presents the performance results from each configuration. It showed the best results exclusively in the first two segments, early and middle-early, with a good proportion of the time showing significantly better results.

Just as the runs on *Ms. Pac-Man*, this highlights the value this approximator can bring when better suggestions are needed faster.

While in *Ms. Pac-Man* experiments a high accuracy threshold resulted in fewer improvements, this time there did not seem to be any dominant configuration, with all of them presenting good results overall.

One configuration ($Acc_{Min} = 0.85$ and $Pred_{Min}$ set to Third Quartile), while presenting the best results overall out of all configurations, also had 2 observations where the results were significantly worse than without an approximator. This is not problematic, but it is worth mentioning.

k-Nearest Neighbours Table 6.19 presents the performance results from each configuration in detail.

Results were fairly mixed when using the k-nearest neighbours algorithm. Most configurations brought more improvements than damage, but there were many points at which the use of this approximator was less desirable than leaving the GA as default.

TABLE 6.19: Percentage of time that the k-nearest neighbour approximator proved to be significantly better, or worse, than using no approximator in the *TORCS* experiments, based on approximator configuration, followed by the segment in which performance proved best.

Acc_{Min}	$Pred_{Min}$	Significantly better (%)	Significantly worse (%)	Best segment(s)
0.60	FirstQuart	17%	28%	Early
0.60	Median	25%	6%	Middle-early
0.60	ThirdQuart	9%	2%	Early
0.75	FirstQuart	16%	0%	Middle-early
0.75	Median	43%	0%	Middle-late
0.75	ThirdQuart	8%	14%	Middle-early
0.85	FirstQuart	0%	0%	N/A
0.85	Median	0%	0%	N/A
0.85	ThirdQuart	0%	0%	N/A

One entire subset of configurations, where the accuracy threshold was set to 85%, proved to not get used at all. This is due to the approximator never being able to achieve that prediction accuracy.

Several other configurations, however, had ups and downs. Most of them behaved admirably in the early and middle-early segments, however dipped slightly in the middle-late and late segments.

Performance of Using Various Accuracy Thresholds

Aggregate results can be seen in Table 6.20.

Overall, there was a fairly clear sign that an accuracy threshold set too high could cause issues with the algorithm, however too low and it might also lose effectiveness.

While C4.5 decision trees managed to present good results across the board, k-nearest neighbours preferred lower values, with the sweet spot most likely around 70-75%. Neural networks, of course, did not manage any measure of success with any of the configurations tested.

TABLE 6.20: Average proportion of time in which each combination of approximator and accuracy threshold was statistically significant, regardless of prediction threshold, in the *TORCS* experiments.

Approximator	Acc_{Min}	Average proportion of significance (%)
NN	0.60	0.0%
NN	0.75	0.0%
NN	0.85	0.0%
C45	0.60	32.3%
C45	0.75	29.0%
C45	0.85	31.0%
KNN	0.60	17.0%
KNN	0.75	22.3%
KNN	0.85	0.0%

TABLE 6.21: Average proportion of time in which each combination of approximator and prediction threshold was statistically significant, regardless of prediction threshold, in the *TORCS* experiments, as well as average predictions computed.

Approximator	$Pred_{Min}$	Average proportion of significance (%)	Predictions
NN	First	0.0%	0.0
NN	Median	0.0%	0.0
NN	Third	0.0%	0.0
C45	First	26.3%	4064.08
C45	Median	30.7%	1689.64
C45	Third	35.3%	633.27
KNN	First	11.0%	2458.32
KNN	Median	22.7%	1010.30
KNN	Third	5.7%	389.20

Performance of Using Various Prediction Thresholds

Similarly to the *Ms. Pac-Man* results, the higher the prediction threshold, the lower the number of predictions made, for identical reasons.

This time, however, there were no false negatives when using the third quartile as the threshold at all in any of the configurations. This result requires potential further investigation, as that could prove very desirable, as mentioned in the last section.

While all 3 thresholds had similar results overall, using median proved to be much better with k-nearest neighbours, while being almost as good as using third

TABLE 6.22: In the *TORCS* approximator experiments using the optimised GA parameters: percentage of time that using the approximator resulted in significantly better results; the average number of predictions generated by the approximator each run and the average percentage of which were false negatives

Approximator	% Significant	Predictions	False Negatives
Neural Network	0%	13.5	5.6%
C45	27%	995.2	8.1%
KNN	2%	854.2	7.3%

quartile in C4.5 decision trees.

Overview of Runs on the Optimised GA Configuration

Compared to the *Ms. Pac-Man* experiments, there did not seem to be any one or two dominant configurations for the approximators. As a result, the configuration chosen for the approximators was the one with $Acc_{Min} = 0.75$ and $Pred_{Min}$ set to median, identical to the previous section, as it had the most consistently good results across both games tested.

Despite the seemingly harder task, the approximators managed to bring significant improvements even on the optimised GA parameter set. Particularly, the C4.5 decision trees proved effective at further improving the quality of the GA. This can be seen in Table 6.22 and Figure 6.9.

The other two approximators brought minimal, if any, improvements to the GA. They did not, however, negatively impact performance.

Comparing Approximators on Unoptimised GA Parameter Set to Optimised GA Parameter Set

Similar to the previous section, comparing results between runs with approximators on an unoptimised GA and no approximator on an optimised GA proved very interesting.

This time there was no one approximator configuration that was entirely better than the optimised GA. Most configurations behaved worse than the optimised GA, with one exception: the C45 decision trees with $Acc_{Min} = 0.60$ and $Pred_{Min}$ set to first quartile. It managed to be better than the optimised GA 23% of the time, but in

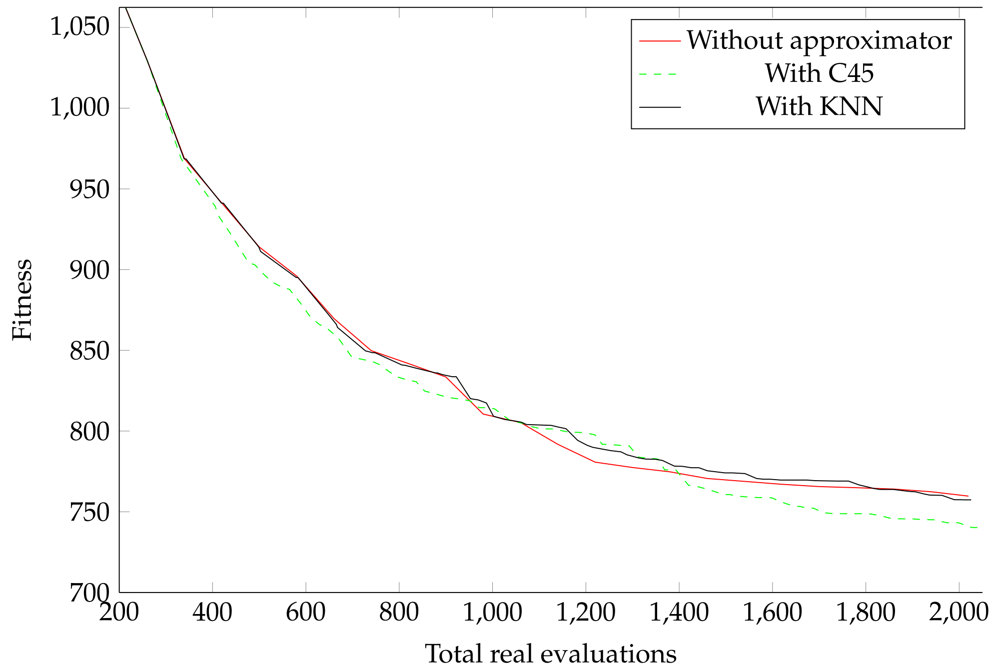

FIGURE 6.9: Average best fitness values achieved after a given number of evaluations by the GAs without and with the neural network approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *TORCS* experiment with the optimised GA parameter set. Lower values are better.

the early segments it was worse 5% and in the middle-late segments it was worse 3% of the time.

Most decision tree configurations were better than the optimised GA in the middle-early segments, but ended up being always worse towards the late segments. This is in line with the newly formed expectation that the decision trees are very good at hastening the early generations of a GA run. K-nearest neighbour and neural network configurations were almost never better.

6.6 *StarCraft* Experiment

6.6.1 Introduction

To further assess the findings of the research done on *Ms. Pac-Man* and *TORCS*, an additional experiment was run, on *StarCraft*. This was done to compare the results from Section 5.3.3 in Chapter 5 with a new set of runs with an identical GA configuration, but also using the best approximator configuration overall.

The approximator configuration chosen was the one with a C4.5 decision tree, $Acc_{Min} = 0.75$ and $Pred_{Min} = Median$.

Given the extremely expensive computational cost of running *StarCraft* experiments, only 10 runs were done, to mimic the limitations of the previous experiments with the game.

6.6.2 Results

Results proved interesting and can be seen in Figure 6.10. The approximator runs were significantly better only 1% of the time, but never significantly worse. Average fitness values were about 10% lower in middle-early and middle-late segments, but never enough to result in significance. This is likely due to the small number of data points (10 runs).

It is worth mentioning again that run times were virtually identical between the two configurations, especially now that the approximator's percentage of computational time used was infinitesimally small compared to the percentage of time used by the game.

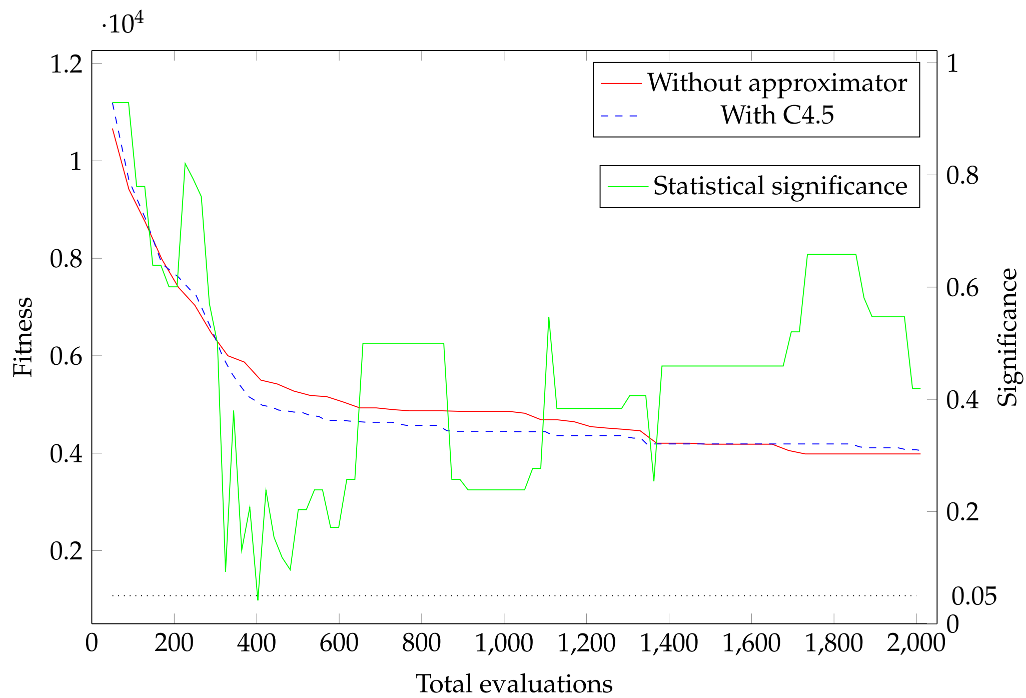

FIGURE 6.10: Average best fitness values achieved after a given number of evaluations by the GAs without and with the C4.5 approximator, with $Acc_{Min} = 0.75$ and $Pred_{Min} = Median(Fitnesses_{Gen-1})$, in the *StarCraft* experiment. Significance of difference between the paired results is also plotted. Lower values are better

6.7 Discussion and Conclusion

Our goal is to create tools and techniques that are usable by industry. For this we need good accuracy, but also much greater speed and scalability. This method of combining GAs and other machine learning algorithms brings significant benefits in these respects. While the graphs might not look impressive, the statistical analysis proves the ability of this approach to further optimise on already efficient approaches.

By using an approximator to predict the fitness of individuals in a GA population, then evaluating only the predicted best, we have managed to allow the GA to explore more individuals in the same amount of time. This has proven highly effective, regardless of algorithm, for *Ms. Pac-Man*, with only one approximator having difficulties in *TORCS*. Results are promising and optimisations to our methodology should bring further speed improvements.

It is valuable to note how there is no single best approximator. While the neural network was very effective in the *Ms. Pac-Man* experiment, it failed to bring improvements in the *TORCS* experiment. However, both *k*-nearest neighbours and C4.5 decision trees were able to be very valuable for both tasks.

Scalability represents the ability of an algorithm to work well on progressively larger scale problems, delivering more or less the same advantages on them. One could argue that the standard OneMax and Trap problems are at the bottom of the scale, *Ms. Pac-Man* a level above, *TORCS* yet another level above, and *StarCraft* another. To some degree, both C4.5 decision trees and k-nearest neighbours managed to deliver good results at most scales.

Given the very small impact on overall performance, with high potential upside and minimal downside, it is very fair to say that our approximators are valuable in speeding up game balance suggestions, as well as other similar areas.

The speed increase could be higher if the approximators were reused between runs. It is something worth looking into in future research, as those outside academia that would benefit from this would likely want to run multiple runs as quickly as possible and reusing data can be extremely valuable.

The quality of approximations could be further improved by potentially using a

committee approach, where multiple models are created in parallel and the majority decision is taken. This is an avenue of research that could be very interesting to test.

It would have been possible to only have 2 classes for approximators where this is relevant: individuals worse than the median and individuals better than the median. However, we decided to have 4 classes to offer slightly more granularity to the algorithm. Exploring whether changing this is valuable could be something done in future work.

One area that we are interested in further assessing is how the training data the predictor uses is managed. While it is purely rolling at this point, it can result in approximators that overfit areas of the landscape and fail to correctly predict individuals of interest. An idea worth following is curated lists, where the data is split in categories, each category accepting and removing entries based on given requirements. These requirements could have one category keep the best fitnesses ever generated by the GA, with another storing the worst fitnesses.

Should the approximator represent an accurate simulation of the fitness landscape, the designer would be left with a model capable of reasonably predicting the fitness of other, custom parameter sets. This approximator would be another tool available to the designer to assess any changes they would want to apply to their games before they go through the much more expensive task of doing AI-driven plays or, even more costly, human player driven plays. Assessing the possibility of this becoming a reality is work worth attempting in the future.

On the topic of fitness classification, we made use of 4 classes. It would be interesting to explore how more (or fewer) classes would influence the quality of the algorithms it was used in.

The fact that the unoptimised runs with an approximator managed to, at times, achieve or surpass the performance of an optimised GA run is also worth mentioning again. Being able to present good suggestion fast, despite suboptimal GA parameters, can be extremely valuable when the tools are used by people unfamiliar with such machine learning algorithms.

Overall, this is an optimisation that can benefit the creation of tools for use in games design, while also having potential for use outside the digital entertainment sector.

6.8 Summary

By combining GAs with other machine learning algorithms, we were able to significantly improve GA performance. While this has been shown on game balance tasks in 2 different games, *Ms. Pac-Man* and *TORCS*, with a short experiment on *StarCraft* as a sanity check, instinct says this is not an optimisation exclusive to games.

Mapping parameter vectors to metric vectors can be applied to a variety of different problems. Potentially saving precious computational time can lead to problems getting valuable good results faster. Unless the problem itself is very cheap computationally, attaching a decision tree algorithm or a simple neural network will rarely be damaging.

Chapter 7

Other Applications of Automated Balancing

7.1 Introduction

Chapter 5 focused on the exploration of automated game design balance using genetic algorithms. This chapter expands on that work and presents several other applications of the algorithms described in there.

The first set of experiments (Section 7.2) continues the work done in Chapter 5 by taking all the resulting knowledge and understanding of the methodology and applying it to a commercial strategy video game, *ComPet*.

The second scenario (Section 7.3) proposed is a departure from game design and a short exploration of game agent development. That section begs the question whether the methodology and specification languages developed can aid in automated agent design, altering the behaviour of an artificial player towards desired goals.

7.2 Commercial Application of Automated Game Balance with *ComPet*

7.2.1 Introduction

Work done on *Ms. Pac-Man*, *StarCraft* and *TORCS* proved that automated game design balance is not limited to just one genre or small games. MindArk, developers of several highly successful commercial games, after initial discussions, proved interested in the research and its potential impact. Their question was straightforward: can these algorithms aid them in better balancing the pacing of their newest game, *ComPet*?

The game itself was presented in detail in Section 3.4. As described there, developers wanted players to spend more time enjoying the player versus player (PvP) element of the game and to use the campaign mode as a motivation to improve the power of their pets. This meant that balance is closely tied to the pacing of the game and allowed us to come up with some interesting metrics, described later in this section.

7.2.2 Environment

While the gauntlet described in Section 3.4 and presented in Appendix A is a condensation of the game's campaign mode, the combat between pets and beasts is based on the game's actual code and is an accurate representation of combat in the game.

We designed a gauntlet that would have the player go through 9 beasts, each with varying skills and strengths (see Table 7.1). The beasts chosen for this experiment were the same as the first 9 beasts created for the actual game and would represent a player's first few hours playing the game.

The main designer expectation is that after 3 of the gauntlet's beasts, the player would be unable to progress for a while, prompting them to gain any missing experience in PvP mode. There is another similar challenge with the 8th beast in the series. This experience could also be gained by fighting previously defeated beasts

repeatedly. As a result, to simulate the PvP repetitiveness, our algorithm used fighting previous beasts for experience gathering.

The pet the player was given at the start of this gauntlet was level 1, with the default abilities a pet of that level would have in the actual game. The rate at which it gained attribute points was also predicated by the game's pace, so as to be realistic.

To get an idea of the current state of the campaign, we simulated an unchanged version of this game 100 times. The results are presented in Table 7.2. The main metrics collected were the number of attempts required to defeat the beast in question and how many times, on average, did the pet manage to eventually beat the beast during a gauntlet run before the maximum number of battles (150 for these experiments) was exceeded. The values collected highlight the relative difficulty of the 9 beasts, with the 4th, 8th and 9th proving particularly difficult, requiring more than 20 attempts to beat on average. The last were also not always successfully defeated before the budget of 150 battles was exhausted.

The designers looking at the results decided the difficulty levels of Ricktick and Thomas Short-Tail were too high and would need to be reduced, while keeping the rest of the beasts at current levels. The number of attempts it takes to defeat them and how often they are beaten every run would represent the metrics by which success is measured. All the metrics considered, alongside the desired values and their importance (Weight), are listed in Table 7.3. The weights were chosen by the designers according to what they considered more important.

Elements of the game that were proposed for change were the health, endurance

TABLE 7.1: *ComPet* beasts in the experiment gauntlet

Name	Level	Health	Ferocity	Endurance
Angel	1	21	0	15
Starbright	1	24	15	25
Jethro	2	29	25	15
Ricktick	3	41	15	20
Forage	3	46	30	6
Harold	4	48	32	30
Fierce Frank	5	45	70	12
Thomas Short-Tail	5	58	15	20
Brutal Bill	6	80	15	50

TABLE 7.2: *ComPet* metrics collected on the gauntlet playing the unchanged version of the game

Name	Defeated After # Attempts	Wins Per Run
Angel	1	1
Starbright	1.5	1
Jethro	10	1
Ricktick	20	1
Forage	1.5	1
Harold	5	1
Fierce Frank	7	1
Thomas Short-Tail	30	0.2
Brutal Bill	20	0.1

TABLE 7.3: *ComPet* metrics to be used in evaluation alongside their desired values and weights

Metric	Name	Desired Value	Weight
M_1	Defeated Ricktick after # attempts	10	5
M_2	Defeated Forage after # attempts	1.5	1
M_3	Defeated Harold after # attempts	5	1
M_4	Defeated Fierce Frank after # attempts	7	1
M_5	Defeated Thomas Short-Tail after # attempts	15	5
M_6	Defeated Brutal bill after # attempts	20	1
M_7	# Wins per Run against Thomas Short-Tail	1	10

and ferocity of two beasts in the gauntlet (Ricktick and Thomas Short Tail's), as well as, for analysis purposes, the cooldown of one of Thomas Short-Tail's abilities (Vampiric Bite). This meant the evolution of 7 parameters (see Table 7.4): Ricktick's Health (RH), his Ferocity (RF), his Endurance (RE), Thomas Short Tail's Health (TH), his Ferocity (TF), his Endurance (RE), as well as the cooldown for Vampiric Bite (VC).

Fitness Evaluation

The same methodology as the one presented in Chapter 5 was used. This meant using the same GAs as described there, as well as the specification language from Chapter 4.

As opposed to the *Ms. Pac-Man* and *StarCraft* experiments, favouring small changes to existing parameters was not required in the *ComPet* experiment. The most important aspect was having the final metrics achieve the designer-set values.

TABLE 7.4: *ComPet* parameters to be changed, their displacement ranges, their decimal accuracy and their weight in the fitness evaluation

Parameter	Variable	Min	Max	Accuracy	Weight
Vampiric Bite Cooldown	VC	+0	+2	10^0	0
Ricktick's Health	RH	-20	+20	10^0	0
Ricktick's Ferocity	RF	-10	+10	10^0	0
Ricktick's Endurance	RE	-10	+10	10^0	0
Thomas Short Tail's Health	TH	-30	+30	10^0	0
Thomas Short Tail's Ferocity	TF	-10	+10	10^0	0
Thomas Short Tail's Endurance	TE	-10	+10	10^0	0

As a result, for each of the metrics, the closer they are to the desired values, the better the fitness should be.

Formally, the fitness function for *ComPet* can be written as:

$$Fitness_{ComPet} = \sum_{i=1}^{m} |M_i - DM_i| \times C_i \tag{7.1}$$

where m = number of metrics considered for comparison (7 for this experiment), M_i = value of metric i from the games played with the evolved version of the game, DM_i = desired value for the metric i, and C_i = weight or importance given to that metric.

As before, the GA considers smaller fitness values to be better, with 0 representing a perfect solution.

Genetic Algorithm

The best GA configuration from the *Ms. Pac-Man* experiments in Section 5.2 were used to generate suggestions for balancing *ComPet*. This was the one where elitism was applied to 20% of the population, mutation was applied to 40% of the population, at a rate of 20%, and crossover was applied to 40% of the population.

Population size was 100, with a tournament size of 6. Experiments ran until 2000 evaluations of the fitness function were used.

7.2.3 Experiment

The main experiment involved balancing a section of the *ComPet* campaign to fit the designer's requirements. The goal, following the requirements and discussion presented in Section 7.2.2, was to find good changes to the proposed parameters, in respect to the metrics presented in Table 7.3.

7.2.4 Results

In this experiment 10 runs were done, each with a different seed. This took a lot of computational effort, as the simulations took a significant time to complete each game. Each generation took approximately 11 minutes to complete.

The results of the experiment are split into two table: the parameter changes recommended by the best individual in each run shown in Table 7.5, and the individual components of the multi-objective fitness function being presented in Table 7.6.

Looking at the results, particularly parameters RH, RF and RE in Table 7.5 and the impact on objective fitness M_1 in Table 7.6, the immediate observation is that changing the beast Ricktick can be done in a variety of ways. Most runs did not, on average, require big changes to Ricktick's statistics, suggesting that that particular beast is not too far from a balanced state and requires only small adjustments. Indeed, the few runs where Ricktick received very big changes resulted in worse fitnesses for that particular objective (M_1).

The interesting results come when analysing the suggestions for beast Thomas Short-Tail. The runs were evenly split between two major strategies: to weaken the beast by making its main attack usable less often, but increase its other statistics (mostly health), or the exact opposite, where no change is made to the cooldown of that ability, but penalties are given to health, ferocity and/or endurance.

These two strategies are also obvious when visualising the 10 runs as a graph. By using Euclidean distance to find the similarity between the best individuals evolved in each run and feeding the resulting values into an algorithm such as GMap [163] to display the run results as a graph, then a similarity map can be generated, such as the one in Figure 7.1. Such a visual tool can further allow a designer to analyse multiple run results and observe patterns in the suggestions.

TABLE 7.5: Main *ComPet* experiment results, highlighting the changes recommended by each run. Green highlighting represents big changes by adding to the original value, red highlighting represents big changes by subtracting from the original value, while colours in-between represent smaller intensity changes. Each column (except Run) represents the displacement to one of the evolved parameters

Run	VC	RH	RF	RE	TH	TF	TE
1	+1	−2	+3	+3	+16	−9	−5
2	+1	+5	+3	−7	+16	+6	−5
3	+1	+5	+10	+2	+10	+10	+10
4	+1	−7	+10	+0	+30	+10	−6
5	+0	+2	+6	−1	−11	+5	−6
6	**+0**	**+2**	**+5**	**−1**	**−14**	**−2**	**+8**
7	+0	+2	−4	+7	−15	+5	−5
8	+0	−11	+10	+7	−11	−1	−5
9	+0	+3	−10	+0	−14	−6	−4
10	+1	−7	+10	−10	+7	+10	−4

TABLE 7.6: Main *ComPet* experiment results, highlighting the individual fitness objectives and scores achieved by each run. Columns, except Run, represent the fitness objectives described in Table 7.3. The bolded row represents the run with the best fitness achieved out of all runs

Run	M_1	M_2	M_3	M_4	M_5	M_6	M_7	Fitness
1	2	0.5	1	3.6	1.25	3.33	6	17.68
2	1.25	0	0.25	0.5	1.25	12.66	6	21.91
3	6.25	0.5	0.5	2	0	7.25	6	22.5
4	4	0.25	0.5	2.5	5	3.5	6	21.75
5	3	0.3	0.6	3	3	5.75	5	20.65
6	**2**	**0.75**	**0.5**	**1**	**3.75**	**1.33**	**6**	**15.33**
7	5	0.3	3.4	2.4	1	2	5	19.1
8	5	5.3	0.6	2	0	1	5	18.9
9	2	0.7	1	1.2	3	5.75	5	18.65
10	0	0.1	0.8	2.4	1	12.6	5	21.9

For this set of runs, one of the immediate conclusions was that increasing the cooldown of Vampiric Bite (parameter VC) would not result in desirable results. Decreasing it is not an option, as a value less than 1 does not make sense within the game. This parameter could now be omitted in future runs.

It is also worth noting how some fitness objectives, particularly M_1 (number of attempts required to defeat Ricktick) and M_6 (number of attempts required to defeat Brutal Bill), were much harder to optimise compared to the rest (see Table 7.6). The second one is most likely due to the fact that by making the previous fight simpler, it allowed a weaker pet to arrive at the Brutal Bill fight, increasing the number of

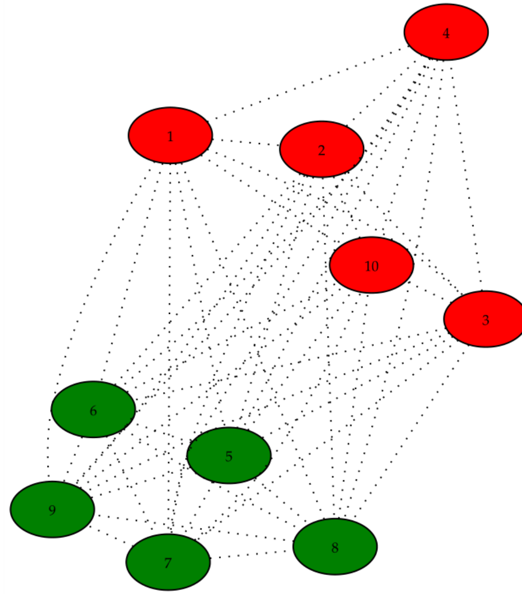

FIGURE 7.1: Similarity map representing the Euclidean distance between the 10 *ComPet* run parameter change suggestions, with colour-coded clusters highlighting the 2 main balancing strategies proposed

attempts required before success. These would require additional analysis from the designer, as different experiments or parameter sets might be necessary to achieve the goals.

Metric M_7 (number of wins on average against Thomas Short Tail) had the most consistent results, however nowhere near optimal. This points towards more changes being required before the goal of getting a win every run is achievable with the current parameter constraints.

Every other metric was much easier to maintain at desired values, with minor exceptions, such as metric M_5 in the best run (run 6) being quite far from 0.

These are all results that the designer will find useful, as they offer valuable insight into the relationships between the various parameters. The designers behind *ComPet* decided to apply the results from run 6 with slight tweaks of their own, as they were closest to what they were hoping to achieve. They also planned on designing more experiments in the future, while also providing valuable feedback on how these algorithms could be even better for commercial use. This included a clear call for simpler tools and better explained pipelines for receiving balance suggestions, mentioning that the specification language was a step in the right direction and that going further would be a win for both academia and industry.

The underlying algorithms (be they GAs, particle swarm optimisers, hill climbers or some other optimisation technique), while important, were not their highest priority. Most valuable were the suggestions offered to them and how well explained they were. The presence of quantifiable metrics and easy visualisation was beneficial to understanding the algorithm's suggestions.

7.3 Evolving Game Agents with Diverse Behaviours

7.3.1 Introduction

Motivation

Balancing games towards designer requirements has been described in detail in previous chapters. However, other elements beyond game mechanics can be tweaked and optimised to offer a more rewarding gaming experience to players. This section looks at using the techniques and tools previously presented to change not the parameters of a game, but the parameters of an agent playing a game to create a version of that agent that behaves in a designer-specified manner.

Game developers often offer varying levels of difficulty for the computer-controlled opponents present in their games. These are elements that require a lot of trial and error to get right. By using simple variations of the techniques presented in previous chapters to alter parameters or values within an agent and using the metrics generated from playing games with said agents, it is possible to generate new and interesting variations to said agents that fit a game developer's requirements. These new agents can be used to automatically collect game metrics faster than human play is able to, can be added as opponents for multiplayer games, or can aid in the goal of balancing game mechanics by acting as proxy human players of arbitrary skill levels, as required by the developers.

We will demonstrate this using *Ms. Pac-Man*. We have made one small change to the rules of the game: the PacMan has no extra lives. Once the PacMan has touched a ghost that is not scared, the game ends and the score is recorded. The reasoning behind this was to streamline the experiment by focusing on getting as many points as possible without being eliminated.

Existing *Ms. Pac-Man* Agents

There have been many agents developed to play *Ms. Pac-Man*, many described in Section 3.1. All of these agents have attempted to be as good as possible at playing *Ms. Pac-Man*. This work, while it could be applied in the same manner, does not have the same aims. The novelty comes from the flexibility this technique presents

and how it puts the game designers first and works towards their vision, regardless of what it might be, instead of continuing the trend of attempting to "solve" a game by creating a perfect agent. The scenario we chose for this section aims to be realistic and interesting in the context of *Ms. Pac-Man*.

It is also important to note that this is not the same as dynamic agent skill scaling, a technique that allows an AI agent to adapt to the opponent's skill level and offer a challenging, but not overpowering difficulty, as highlighted by several researchers [99] [164]. This is design of an agent's behaviour in the stages before the release of a game, resulting in an agent of predictable, and static, skill.

We decided to use a neural network for playing the game, with its framework based on work by Lucas [131], with changes to it presented in Section 7.3.2. The reason behind using a neural network is the ability it has to discover, or evolve, interesting characteristics and rules that might not have been obvious to a designer beforehand. These same techniques could be used with evolving rules in a rule-based agent or the parameters of MCTS agents, but neural networks were a good choice for balancing simplicity of use to simulation speed.

We did not compare the results achieved by our approach to the results described by Lucas as they were both done on two different implementations of *Ms. Pac-Man*, as well as their goals are completely different. Lucas sought to create agents that played the game as well as possible exclusively, while we aim to define designer constraints wherever possible and generate agents fulfilling them.

While research is being done on controlling the ghosts in the game [165], it is less fleshed out than the alternative of altering the PacMan itself. This allows us to quickly compare our resulting agents to a couple of existing artificial agents, namely the rule-based implementation and an MCTS implementation.

Table 7.7 presents the average scores and standard deviations of said scores achieved by the rule-based agent and the MCTS agent, with two extra rows presenting what will be targeted for evolution. Given the performance of the previously mentioned agents, it would seem there is a large gap between the rule-based agent, which could be considered 'easy' difficulty (comparable to a player of basic skill), and the MCTS one, the 'hard' difficulty (comparable to a player of advanced skill). To better cover the spectrum of challenge, a designer might want to create an agent

TABLE 7.7: Average scores, and their standard deviation, achieved by a rule-based agent and MCTS over 1000 games, as well as the desired values for the evolved agents

Agent	Average Score	Standard Deviation
Rule-based	1067	1376
MCTS	2482	1641
Desired New Agent 1	1750	1000
Desired New Agent 2	1750	100

that is in-between those two, as a 'medium' difficulty offering. Attempting to evolve an agent that achieves 1750 points on average is very close to the mid-point between the two existing agents and this will be the goal of a couple of the experiments.

7.3.2 Methodology

Pipeline

The same pipeline as the one described in Section 5.2.1 was used, with one major change: instead of changing the game parameters, the evolved vector is passed to the constructor of the neural network agent described in the next paragraph. This initialises the network using those values, as each value in the vector represents a different weight in the network. A number of games is then played with that agent to collect the relevant metrics.

Neural Network Agent

The agent we designed, based on work by Lucas [131], makes use of a neural network as an evaluator of nodes (accessible points on the map) the PacMan should go towards. While Lucas' approach only tested for the immediate 2 to 4 neighbour nodes the PacMan could go to at any given point, our approach tests every single node on the map. All the valid nodes are considered and evaluated by the network, then the one with the highest score is marked as the destination. This is another critical difference to Lucas' work: while his approach meant that there was no pathfinding required, as the best node would already be a neighbouring valid destination, our approach requires some way of getting the PacMan towards the desired node.

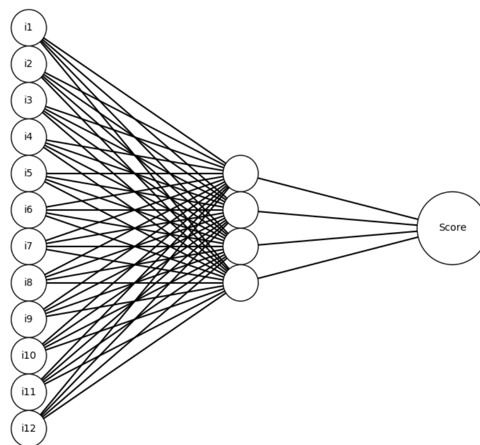

FIGURE 7.2: Structure of the neural network used to score each node

For the previously mentioned pathfinding the agent makes use of the A* algorithm [166] to find the shortest route to the chosen destination. With the shortest route to the chosen node found, the PacMan starts moving on that route until given new instruction. This process is repeated every few frames, each time possibly giving the PacMan a new route and direction to take.

The A* algorithm considers empty spaces as costing 10, spaces with pills or pellets to cost only 5 (thus being more desirable), spaces with chasing ghosts to cost 2000, while spaces with fleeing ghosts to cost only 1.

Choosing the network's topology is one of the main tasks when working with neural networks [161]. The one employed for these experiments is a feed-forward neural network with an input layer of 12 neurons, a single hidden layer with 4 neurons and one output layer with one neuron. The structure can be seen in Figure 7.2. As a result of this structure, there are 57 weights to evolve (this also accommodates for the biases for each hidden and output neuron). The activation function used is the sigmoid activation function.

For each node 12 features are considered and used as inputs to the neural network: the distance from the node to the PacMan; whether the node in question is a junction or not (1 for yes, -1 for no); for each ghost the distance from it to the node, as well as whether that ghost is hunting or hunted; the distance between the nearest power pill and the node; and finally the distance between the nearest small pill and the node. All distances are calculated by taking the shortest path in the maze between the two positions considered. All these features are relevant to the state of

TABLE 7.8: Features chosen for evaluating each node

Feature	Min	Max	Name
Distance Node to Pacman	0	100	$i1$
Is Junction Node	-1 (false)	1 (true)	$i2$
Distance Node to Ghost 1	0	100	$i3$
Is Ghost 1 Hunting	-1 (false)	1 (true)	$i4$
Distance Node to Ghost 2	0	100	$i5$
Is Ghost 2 Hunting	-1 (false)	1 (true)	$i6$
Distance Node to Ghost 3	0	100	$i7$
Is Ghost 3 Hunting	-1 (false)	1 (true)	$i8$
Distance Node to Ghost 4	0	100	$i9$
Is Ghost 4 Hunting	-1 (false)	1 (true)	$i10$
Distance Node to Nearest Power Pill	0	100	$i11$
Distance Node to Nearest Pellet	0	100	$i12$

the game and can greatly impact the desirability of going to a node. They can have values as described in Table 7.8. Should the feature be unavailable for a node (for example: there are no power pills left for there to be a nearest power pill distance), it reverts to being set the maximum value possible.

The output is a single value representing the neural network's evaluation of the node, or simply put: a score. The higher the score, the better it is for the PacMan to go to the evaluated node.

Genetic Algorithm

The representation used was an array of the 57 weights and biases within the neural network. Each element was a real value constrained to be between -5 and 5. These constraints are there to allow for smoother behaviour in the active neurons of the network, to promote generalisation, as the tasks at hand are focused on prediction and scoring, not on classification.

The evolutionary algorithm employed is a variant of a generational GA with two-point crossover [25] (applied with a rate of 35%), a specialised mutation operator (applied with a per-individual rate of 35%) and elitism (applied to the top 15% of the population). The final 15% of the each generation is randomly sampled from the search space through reinitialisation [152]. This is similar to the approach used in Chapter 5.

The mutation operator was applied with a (per allele) mutation rate of 0.5 (meaning that on average 50% of the elements of an individual would be mutated). At each application of the operator, a displacement is randomly generated by a random number generator within the range of acceptable values (between -5 and 5) for that allele and added to the corresponding parameter value.

The experiment used tournament selection, with a tournament size of 6. The population size was 50, with no more than 50 generations for each run. For each experiment, we did 10 runs using the configuration described above, each with a new random seed.

Fitness Evaluation

For each individual the array in an individual was decoded, a neural network was initialised with the given values, then 50 games were played with this new agent, storing all the scores the agent achieved. All the scores would represent the agent's skill level.

Our GA considers smaller fitness values to be better, with 0 representing a perfect solution.

Formally, the fitness function f can be written as:

$$f = f_s + f_d \tag{7.2}$$

$$f_s = |Mean(S) - DS| \tag{7.3}$$

$$f_d = |StdDev(S) - DD| \tag{7.4}$$

Where f_s represents the mean score component and f_d represents the standard deviation component. In the component f_s, S is the array of scores achieved by the individual, $Mean(S)$ is the average of those scores, and DS is the desired average score to be achieved. In the component f_d, $StdDev(S)$ is the standard deviation of the scores, and DD is the desired standard deviation to be achieved.

Experiments

We performed three experiments with *Ms. Pac-Man*.

The first experiment we ran was to generate an agent that is as strong as possible. This meant the aim of the evaluation was to score as many points as possible. The reason behind this was to see what the upper limit of this approach is and to replicate results by other researchers. The value of DS was set to 6000, as that is a high enough value with a very low probability of being achieved. For this experiment, the f_d element of the fitness function is completely ignored, as there is no desired standard deviation to target.

The second experiment involved a designer-led requirement to have an agent that would behave better than the rules-based agent presented in Table 7.7, but not by a massive margin. The desired value of DS was set to 1750, as described earlier in Section 7.3.1. This agent should still have very good games and very bad games, an element highlighted by the relatively high standard deviation of $DD = 1000$.

The third experiment is almost identical to the second one, with the major difference being the targeted standard deviation of $DD = 100$, instead of 1000. This, instinctively, is a much more difficult task given the stochastic nature of the game and the results documented for the other two agents used for comparison (see Table 7.7). Such a low standard deviation would represent a highly consistent agent, something that can be valuable when analysing and balancing games.

7.3.3 Results

Experiment 1: The Strongest Neural Network Evolved

The 10 runs each resulted in agents that would play the game at a much better level than the rules-based agent and its average of 1067 points and even better most times than the MCTS agent, which achieved only 2482 points on average. The best performing agent, from run 2, averaged 2804.2 points. On average, the best agents generated by the 10 runs achieved 2602 points, with a median of 2607.1 and a standard deviation of 90.82. The results are individually documented in Table 7.9.

These results represent the upper limit that can be achieved with this architecture and methodology. Knowing what this upper limit of our approach is, we could then decide on a value for DS for the second and third experiment. Choosing 1750 was based on two factors: it is a value smaller than 2479.6 (the worst of the results for this

TABLE 7.9: Experiment results for the experiment attempting to generate as good a player as possible given the architecture evolved, with the best performing run in bold

Run	$Mean(S)$	$StdDev(S)$	f
1	2,625.0	1,063.0	3,375.0
2	**2,804.2**	**1,182.0**	**3,195.8**
3	2,644.0	1,027.0	3,356.0
4	2,603.2	943.1	3,396.8
5	2,611.0	1,149.0	3,389.0
6	2,479.6	666.4	3,520.4
7	2,575.6	1,100.0	3,424.4
8	2,621.4	1,090.0	3,378.6
9	2,572.8	980.1	3,427.2
10	2,482.4	776.1	3,517.6

experiment), thus theoretically achievable, and it would be a value close to the midpoint between the two agents we compared in Section 7.3.1, representing another tier of skill that we did not have tapped.

Using the Wilcoxon Signed Rank Test with the null hypothesis that the median of the results is lower than 1750 (our desired score for the second experiment), we achieve a p-value of 0.0009765625. Every single run managed to achieve an agent significantly better than the one we are attempting to evolve in the second experiment.

Due to the black box nature of neural networks, it is hard to compare between the various agents generated by each run without simply observing their behaviour. There is a similar behaviour being evolved in all 10 runs. The PacMan attempts to find the best path to each power pill, sometimes taking a short detour to eat a ghost or two on the way. Once it has eaten all 4 power pills, it is simply playing to avoid the ghosts. Without more information, such as previous locations and decisions taken, it is unable to evolve any better strategies, as it is playing frame to frame.

Experiment 2: A Balanced Neural Network with High Variance

The results of the second experiment are individually documented in Table 7.10. This experiment had two objectives to evolve towards, and almost every single run managed to find a solution within 3% of the desired values of $DS = 1750$ and $DD = 1000$, with the notable exception of run 1.

TABLE 7.10: Experiment results for the second experiment, where an agent is evolved to fit given the designer requirements of $DS = 1750$ and $DD = 1000$, with the best performing run in bold

Run	$Mean(S)$	$StdDev(S)$	f_s	f_d	f
1	1,750.4	885.2	0.4	114.8	115.2
2	1,742.0	998.1	8.0	1.9	9.9
3	1,744.0	988.5	6.0	11.5	17.5
4	1,753.4	981.6	3.4	18.4	21.8
5	1,763.8	1,001.4	13.8	1.4	15.2
6	1,751.2	1,009.6	1.2	9.6	10.8
7	1,749.2	1,009.1	0.8	9.1	9.9
8	1,748.6	1,003.5	1.4	3.5	4.9
9	**1,750.2**	**999.0**	**0.2**	**1.0**	**1.2**
10	1,740.4	1,007.0	9.6	7.0	16.6

On average, the best agents generated by the 10 runs achieved a mean fitness of 22.3, with a median of 13 and a standard deviation of 33.21. While the target score seemed easier to achieve, successfully managing the standard deviation requirements as well is moderately surprising and welcome.

After running the best agent from each run, it is fairly clear how they managed to achieve the scores they did. As opposed to the previous experiment, these agents are a lot more aggressive with their positioning, staying close to the ghosts, but not so close to get caught. This allows them to get the required amount of points fairly reliably, but also gets them killed quicker as they get stuck between two chasing ghosts. It is definitely an interesting strategy that results in more exciting agents that also fit the requirements.

Experiment 3: A Balanced Neural Network with Low Variance

On average, the best agents generated by the 10 runs achieved a fitness of 358.12, with a median of 355.2 and a standard deviation of 66.99. The results are individually documented in Table 7.11.

Just as the previous experiment, achieving the target score is extremely easy. However, with such a small target standard deviation, it was much harder to find agents that fit the requirement. This is in no small part due to the heavily stochastic nature of the game. Even the highly skilled MCTS agent had really bad games and really good games. It is still impressive to see our approach manage to get quite close to what some would call a consistent agent.

TABLE 7.11: Experiment results for the third experiment, where an agent is evolved to fit a different set of designer requirements of $DS = 1750$ and $DD = 100$, with the best performing run in bold

Run	$Mean(S)$	$StdDev(S)$	f_s	f_s	f
1	1,739.2	388.3	10.8	288.3	299.1
2	1,739.0	442.6	11.0	342.6	353.6
3	1,727.4	360.6	22.6	260.6	283.2
4	1,610.0	326.1	140.0	226.1	366.1
5	1,772.2	434.6	22.2	334.6	356.8
6	**1,743.6**	**355.4**	**6.4**	**255.4**	**261.8**
7	1,658.4	346.7	91.6	246.7	338.3
8	1,691.0	469.4	59.0	369.4	428.4
9	1,745.0	521.7	5.0	421.7	426.7
10	1,763.8	553.4	13.8	453.4	467.2

Of course, the best run achieved an average score that might be considered lucky, despite the low standard deviation of the scores. To double check the results, we ran that particular agent for another 50 games. The new results did not stray too far from the original results, achieving an average score of 1782.2, with a standard deviation of 366.6. This is close to the original values and within expected ranges.

Looking at some of the behaviours evolved, there seems to be less aggression. The PacMan aims to collect all power pills while ignoring fleeing ghosts. It stays as far as it can from all 4 ghosts, which proves to work in its favour to consistently achieve the points needed.

It is also worth mentioning that we do not supply an agent's current score as an input, so the GA is unable to evolve agents with behaviours that make them end the game when reaching the desired number of points while playing as well as they can.

7.4 Discussion

7.4.1 Industrial Applications

By integrating a specification language and a bridge with an existing genetic algorithm developed in academia, we were able to demonstrate applicability in industry through a cooperation with MindArk and their commercial product, *ComPet*. The successful application of these techniques to a commercial game in development is reassuring, as the goal is for them to be valuable in the real world, helping the development of games. A lot of valuable feedback was received from MindArk during the placement there, and much of it is reflected in this thesis.

Having already used this methodology in industry, the benefits are obvious: less time spent explaining research and more time spent solving immediate problems. The researcher has a clear goal, to develop algorithms that solve balance problems, while the game designer needs only present their problem using a simple language.

While each run, regardless of strategy evolved, presented the best result as a list of numeric changes, there are potentially better ways of presenting this information. Replacing the numbers with slightly broader suggestions (such as replacing a +16 to Thomas Short-Tail's Health with "Significantly increase Thomas Short-Tail's Health") can make results a lot easier to interpret by designers. Presenting the results in a less spartan manner can make it much more likely that the algorithm is usable by people with less technical training. Displaying balance suggestion using fuzzy language can ease designers into the process of implementing automated game balancing into their pipeline.

This successful use of research in an industrial context is a sign that not only is there need for more research on the topic, but that also there can be demand for it in the real world, given proper presentation and tools.

7.4.2 Evolving Game Agents

This work presents a useful tool for game designers to use in their quest to turn their vision of a game into reality as accurately and easily as possible. It does, in no way, replace human designers as they still need to define the requirements and double-check the results of the automated system.

An immediate use for this work is to generate agents of various skills for games, and this is particularly valuable for cases where no pre-existing agent exists. These could then be used to balance games to fit designer requirements, using the methodology described in Chapter 5, which requires the availability of automated agents. By applying the balancing technique first to agents, then using the resulting AI players to play the game, a wider variety of designer tasks can be fulfilled, at a fraction of the effort and cost.

The experiments described in this section focused on evolving behaviours for the PacMan character, the one controlled by the player. However, this methodology could just as well be applied to non-player characters, such as enemies or allies in games. Also, given the positive results we reported on the flexibility of the neural network, it would seem possible to design artificial agents that could control multiple entities in a game at once.

In addition, more objectives, such as time alive, number of ghosts eaten, or number of pellets collected, could be added to the evaluation to further customise the behaviour of the resulting agent. This is extremely important to highlight, as behaviour can be defined by more than just scores and their variance. More metrics can allow for finer tuning of the resulting agents, within the limitations of the game.

An obvious idea for improving the agent we described and developed involves adding extra inputs to represent the state of the game, and decisions taken, at previous ticks in the game. This recurrence could give the neural network memory and could unlock new strategies to be evolved.

It would also be interesting to evolve the weights for the A* algorithm used, the ones defining the cost of going to a node, alongside the neural network itself. This would further unlock strategies for a designer to use.

Of course it would be desirable to alter the architecture of the agent itself. Using emergent tangled graph representation [167], for example, a technique that has proven to have a very high performance ceiling, would allow for even more flexibility from a designer's requirements and expectations.

Less important, but definitely very exciting, was the fact that the evolved best agents were able to achieve better results than MCTS, an algorithm considered extremely good at playing *Ms. Pac-Man*, at a tiny fraction of its computational cost.

Further improving the structure of the neural network would most definitely bring further incremental improvements. Adding memory could potentially greatly improve performance, as the current version lacks both memory and a forward model, something the MCTS agent has, yet still managed to beat it.

This has further shown that GAs are viable tools for use in game design, complementing human intuition and offering valuable techniques for future problem-solving in game design and beyond.

7.5 Summary

The previously presented two scenarios are fairly different from one another. One is an industrial application of game design balance, another attempts to evolve artificial intelligence with various behaviours.

However, both solutions share the same underlying GA and specification language to solve their respective requirements. This is testament to the flexibility of our approach, as well as the versatility of the concept of "balance". By abstracting a balance task as numbers to change and numbers to aim for, one can theoretically solve a number of problems, in many different fields.

Chapter 8

Conclusions and Future Work

The methodology presented in Chapter 5, optimised in Chapter 6, then applied in two further scenarios in Chapter 7, proved successful in dealing with a wide variety of tasks, most of them related to games balance.

The area of automated game balancing is being explored in depth at this point in time by many researchers and this can only benefit the games development world. Manual testing will, most likely, never be obsolete, but designers will be able to focus on much more interesting tasks while letting computational intelligence do the less exciting elements of balance.

This thesis has further shown that GAs are viable tools for use in game design, complementing human intuition and offering a route to future problem-solving of complex scenarios.

While much of this research aimed to produce results worthy of use in industry, and, to some extent, a significant portion of it was exactly that, there is still a lot to improve in future work. The partnership with MindArk on *ComPet* was a particularly valuable opportunity to better understand how to bridge the gulf between industry and academia. The specification language presented in Chapter 4 and used throughout this entire thesis may seem simplistic, but it is critical to remember that simplicity will allow for mainstream adoption of academic research in industry and potential for further growth. Tighter cooperation between industry and academia can only benefit both.

The work done on approximators (Chapter 6) proved to be of much value, with many interesting results highlighting much more potential in this field. Mixing different machine learning algorithms with the goal of emphasising each of their

strengths and mitigate some of the weaknesses proved quite powerful.

While no obvious "winner" emerged among the tested approximators, there is likely much that could be improved in the methodology, as well as other approximator models to test.

This work also proved of interest to other research areas. The task-agnostic approach, while only tested on toy problems and games, could easily be valuable in other fields. As long as one can present the problem as a list of numbers, then map it to either a fitness class (the fuzzy approach) or a fitness value (the exact approach), then GA use can be enhanced by other ML methods.

It was also a pleasant surprise to see the methodology developed being able to tackle agent generation as well. Not only did the evolved agents from Section 7.3 successfully achieve the required balance goals, the ones generated during the search for the best agent possible beat some top-of-the-line agents quite handily. With a better network structure and more features, such as memory, it is very likely that results could be even better.

Overall, this thesis succeeded in its goals of identifying problems within game design, particularly related to game balance, present viable methods of mitigating those problems, then start the conversation and implementation of approaches to bridge the gap between academia and industry. The optimisations to GAs in general were a valuable process that happened to bring more value to the game balance research.

There are still many challenges ahead for those that will continue research in this field. While some industrial cooperation was achieved during the research reported in this thesis, it was minimal. And despite the tests done on several different games, some of which real commercial games, in several different genres, they were not comprehensive. They do not promise that this work will translate at the same level of performance for other games.

Appendix A

ComPet Example Gauntlet

This document is an XML file describing the beasts one would have to fight to complete the campaign designed for the *ComPet* experiment in Section 7.2.

```xml
<ComPetGauntlet>
  <RetryCount>0</RetryCount>
  <GrindAfterFailureCount>2</GrindAfterFailureCount>
  <MaximumBattlesTotal>150</MaximumBattlesTotal>
  <GainXPOverTime>true</GainXPOverTime>
  <NumberOfRuns>10</NumberOfRuns>

  <PlayerPets>
    <ComPetPet>
      <BeastID>DefaultPetBalanced</BeastID>
      <Health>20</Health>
      <Mojo>30</Mojo>
      <Endurance>15</Endurance>
      <Ferocity>15</Ferocity>
      <HealthPerLevel>7.5</HealthPerLevel>
      <MojoPerLevel>2</MojoPerLevel>
      <EndurancePerLevel>0.75</EndurancePerLevel>
      <FerocityPerLevel>2.5</FerocityPerLevel>
      <Experience>0</Experience>
      <Level>1</Level>
      <Abilities>
```

```
        <AbilityPair>
          <AbilityID>1000</AbilityID>
        </AbilityPair>
        <AbilityPair>
          <AbilityID>270</AbilityID>
          <UseFromLevel>1</UseFromLevel>
          <UseUntilLevel>2</UseUntilLevel>
        </AbilityPair>
        <AbilityPair>
          <AbilityID>20</AbilityID>
          <UseFromLevel>3</UseFromLevel>
          <UseUntilLevel>7</UseUntilLevel>
        </AbilityPair>
        <AbilityPair>
          <AbilityID>220</AbilityID>
          <UseFromLevel>3</UseFromLevel>
          <UseUntilLevel>7</UseUntilLevel>
        </AbilityPair>
      </Abilities>
    </ComPetPet>
  </PlayerPets>

  <Beasts>
    <ComPetPet>
      <BeastID>102</BeastID>
    </ComPetPet>

    <ComPetPet>
      <Enabled>true</Enabled>
      <BeastID>Level1BeastWHeal</BeastID>
      <Health>21</Health>
```

```xml
        <Mojo>15</Mojo>
        <Endurance>15</Endurance>
        <Ferocity>10</Ferocity>
        <Level>1</Level>
        <Abilities>
          <AbilityPair>
            <AbilityID>1000</AbilityID>
          </AbilityPair>
          <AbilityPair>
            <AbilityID>2000</AbilityID>
          </AbilityPair>
        </Abilities>
      </ComPetPet>
  </Beasts>
</ComPetGauntlet>
```

Appendix B

Diplomatic Turn-Based Strategy Games

B.1 Introduction

As we mentioned in Section 3.5, there was some work done on exploring the area of believable diplomatic agents. This Appendix presents the results of that work.

B.2 A Description of DTBG

For the purpose of this article, a game that used to run up until 6 years ago, but sadly went down due to the developers disappearing, called *Genesis* [1] , will be described. *Genesis* had each player in control of a single town, capable of expanding it with various military and economic buildings, of recruiting armies and researching improvements to its economy and units. The game world was split in systems, each system with 5 planets, each housing 20 players. The game would support hundreds of players at once. A simple screenshot of how *Genesis* used to look can be seen in Figure B.1.

The game had two major factions: good and evil, each with a number of different races available. Players of opposing factions could not be directly allied with each other, as a way to facilitate conflict. Each planet, as a result, housed 10 players of the

[1] Due to its age and lack of marketing, most references to *Genesis* have disappeared. A few remain, at http://www.gamespot.com/genesis-2006/ and http://www.gamesindustry.biz/articles/genesis-new-mmo-game-launch

good faction, and 10 players of the evil faction, creating the game's original motivation: the desire to be the stronger faction when access to the entire system of planets was available.

Victory would only be achieved once one single player would successfully reach the end of the research tree and constructed a particular building, then successfully defended it against the rest of the world for 3 full days. On their own, a player would never be able to succeed. Successfully arriving to the end-game building would mean the player has no military resources at all. An amazing result of this is the fact that, even though they would not win in the traditional sense of the term, many players would choose to support that single player in their quest to the finish line, defending them and taking some indirect victory from the achievement. Of course, multiple such groups would form, all with the same goal: be the group supporting the eventual winner.

The greatest pull to this sort of game, as well as its differentiating feature, is its meta-game and social aspect. By interacting with a number of people larger than the usual MMO, players must understand and cooperate with their friends or enemies differently, often making compromises for the good of more, or the opposite, gaining personal benefit at the cost of others. In a long-term game, for example, grudges against players could escalate into war between groups, something usually not possible in the more linear MMO environments.

The social aspect also helps create bonds of friendship and camaraderie that could help lead to a group's victory, or, just as important, a more enjoyable game experience during a round. Tapping into what makes humans form these links with others offers great potential for even more innovative video game mechanics, making use of what, for example, attracts people to board games.

B.3 Conflicts as a Gameplay Mechanic

Most games have mechanics that allow players to either move closer to a desired end-state or balance state [12]. However, *Genesis* allowed only one player to officially "win" the game, leaving hundreds of players as nothing but second place. The assumption many would make is that once a player is close to victory, the rest

FIGURE B.1: Genesis planet view. Every little tower is a different player

will turn on them and deny them the chance to win. Given the immense number of players, this creates a loop and victory is never achieved by anyone, as any military resources expended during one denial can be recuperated by the time it is needed again.

As mentioned previously, players would team up for the indirect achievement of supporting another potential victor. Why this develops the way it does is something worth looking into in more depth, as during the many rounds experienced in *Genesis*, rarely did betrayal happen. When it did happen, it was either extremely subtle and unnoticed until it was too late for the receiving party, or harsh treatment was exerted on the acting party. This group interaction is very exciting, as it does mimic society in a manner. Players want to be accepted in their group, even if success might favour someone else.

The separation of players in planets and systems created a fascinating diplomatic dynamic. Conflict would exist for the first stage of the game, the planet stage, between 10 evil players and 10 good players. The 10 players in each group would form a camaraderie in their common desire to prove superior. Most often, one of the 10 would assume leadership. Once the whole system was available, 80 more players entered the fray, 40 evil, 40 good. The logical evolution was, usually, to merge the 5 alliances of each faction. This is where the first major conflict of interest would arise, as players can grow attached to even the name of their small group, most often having to lose it in the merge. Not only that, but for the purpose of administration,

only one player could be official leader, regardless of how decision making was done internally. This caused other debates and, sometimes, conflict.

All of this internal conflict had to be solved quickly, as the opponents were always in a similar situation, hoping to take advantage of weaknesses during this transition period, striving to be more efficient than their enemies. A lot of compromise would happen purely for the sake of hastening the process of union between the 50 players of their faction.

B.4 Discussing Potential Player Types

Each new entry in the video game world brings a different type of players with it. Properly identifying the various groups that might emerge and catering to all their requirements is critical in maintaining a balanced community. This is also important for AI research, as, at the time of writing, no one AI can "pretend" to be any type of human. AIs try to emulate certain personalities and play styles, inspired by human behaviour itself.

Bartle [168] described MMORPG players as falling into 4 categories: explorers, killers, achievers and socialisers. Most of these would readily apply to DTBGs, but their aims and "engines" behave differently.

This chapter aims to provide a very simple and broad assumption of the player types emerging in this genre, as experienced personally during the playing of *Genesis*.

Of course, a player can be part of multiple of these representations at once, in varying percentages, by choice or by group necessity.

The identified player types are leaders, followers, diplomats, aggressors, warriors and strategists.

B.4.1 Leaders (Socializers / Achievers)

The players that want to directly influence the path of their group and the interactions between different groups, that want to be figures of authority among the many players in the game, are called Leaders. They share qualities of both socializers and achievers. They will be happy to interact with members of their group, to know

who they are leading and what each of them enjoys. But their end goal is to take that group of people as far as possible up the achievement ladder, preferably the victory itself.

B.4.2 Followers (Explorers)

Followers play the game to discover what interactions will emerge between themselves and the environment, between their group and the environment, between groups themselves. They will happily be led by others and follow orders given for the experience of the game and the various storylines that emerge. They are most often the players that eventually end up supporting the candidates for winning the game.

This is the class of players that would, at a first glance, be the easiest to emulate with an AI. Their social interaction requirements (such as forums or public chat) are minimal, while their desire to come up with better alternatives to what their leaders are saying is close to none. They are very similar, in behaviour, to individuals in a swarm, following those around them and rarely questioning the decision of the group.

B.4.3 Diplomats (Socializers / Explorers)

Diplomats are very similar to Leaders in many respects, except for their end game desire. Diplomats don't particularly want to win as much as they want to see what they can discover or do while interacting with other players, be they allies or enemies. The questions diplomats ask themselves is how far can they drag negotiations in their favour, or who can they talk to for even greater opportunities to success.

B.4.4 Aggressors (Socializers / Killers)

These players are similar to diplomats, but they will use conversation to create in-game conflict, or war. Their end goal is to cause a war in their favour, preferring trial by combat over peace by words. Sometimes, however, they will attempt to stir controversy in other groups, to benefit their own. They are rare, or alternatively hard to identify, due to others choosing not to trust their kind.

B.4.5 Warriors (Killers)

They will eagerly forgo the chance to be the winners of the game for the opportunity to have a large army and help in the success of their group through military action. This player type is very similar to the Follower type, but differs greatly in what motivates them. Warriors want a big army and a chance to use it profitably.

B.4.6 Strategists (Explorers / Achievers)

By manipulating the resources available to their group, both military and diplomatic, strategists want to discover as many great ways to push the game in their favour and to achieve new strategic heights. Their desire is to outsmart their opponents and find novel ways of maximizing their group's power.

B.5 Game Design Space

By defining and opening a new game genre, both researchers and game designers receive access to a new world of possible research and products. The DTBG design space is, at a glance, quite broad. It's only inherent requirement is that many people have to interact in a meta-game environment to further their gameplay goals. This creates many possibilities for building new experiences, something greatly beneficial to the industry and to the furthering of ludology [169].

Alongside the social requirement, elements that designers could 'play' with during their design of a new game or game variation include:

1. Having an economy that players must maintain, through orders involving buildings, research, trading and/or espionage.

2. Having a military that players must build and use in conquering, depriving of resources or weakening of opposing forces.

3. Having a group structure to push players into social, military or economic factions.

4. The presence of a ranking system to differentiate players in different categories.

5. The existence of a dynamic or static narrative to support the gameplay elements of the game.

6. Having player statuses to define their position in the game's "world order".

7. Whether players can be removed from the game completely through military, economic or diplomatic means.

These are but few of the parameters the DTBG genre would have to creating new and exciting game worlds and all of them mesh really well with the presence of many players. Not only that, but each of these elements allows for even more research. For example, the military aspects of a game could be tweaked in real time by an AI algorithm, to further improve the experience of players.

Each new game in the genre would allow for different player types to emerge, as it is definitely possible that some of the ones defined in the previous section might not have room to exist in the different game environment created.

B.6 Discussion

A lot of the work defining video game genres started just slightly over a decade ago [170] and some would argue it is nowhere near as complete as artistic mediums such as movies or music. The world of video games is ever expanding and becoming part of every day society.

This game genre would create new ways for game design to evolve, for psychology, sociology and politics research to gather data to further human insight in social behaviour, as well as a wonderful test-bed for artificial intelligence research. One can only imagine, at this point, what an AI that leads humans would behave like. Research in this genre would eventually make that a reality.

On a broad scope, creating a complete AI for a game like this would be a merging of many other areas of research, such as natural language processing, data analysis and prediction, emotion detection, and more. This could prove to be a great challenge, but is something that the game genre itself allows leeway. There is no direct feature, beyond the ability for people to chat, that the genre relies on exclusively.

By removing various game elements, such as a requirement for economy, or the presence of military, AI researchers could focus on particular elements of DTBG and, as a result, create new games in the genre, or better understand how some features influence game design and human behaviour.

Appendix C

Aggregate Data for Approximator Experiments

C.1 Introduction

This appendix contains aggregate results from the approximator experiments with *Ms. Pac-Man* and *TORCS* from Chapter 6.

TABLE C.1: Results when comparing between approximator runs with the unoptimal GA parameter set and both optimal and unoptimal GA runs, in the *Ms. Pac-Man* experiments. Better and worse values represent the percentage of time that the second configuration proved significantly better, or worse respectively, than the first configuration

First	Second	Better	Worse	Analysis
Optimal	Unoptimal	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc60-First	11%	31%	Better 23% of the time in quarter 0. Better 19% of the time in quarter 1. Worse 23% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc60-Median	4%	43%	Better 9% of the time in quarter 0. Better 7% of the time in quarter 1. Worse 73% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc60-Third	0%	2%	Equal in quarter 0. Equal in quarter 1. Worse 7% of the time in quarter 2. Equal in quarter 3.
Optimal	C45Acc75-First	10%	0%	Better 9% of the time in quarter 0. Better 30% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	C45Acc75-Median	0%	0%	Equal in quarter 0. Better 7% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	C45Acc75-Third	0%	46%	Equal in quarter 0. Worse 19% of the time in quarter 1. Worse 65% of the time in quarter 2. Worse 100% of the time in quarter 3.

Optimal	C45Acc85-First	0%	43%	Equal in quarter 0. Worse 65% of the time in quarter 1. Worse 84% of the time in quarter 2. Worse 23% of the time in quarter 3.
Optimal	C45Acc85-Median	0%	68%	Equal in quarter 0. Worse 73% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc85-Third	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc60-First	6%	12%	Better 23% of the time in quarter 0. Equal in quarter 1. Worse 3% of the time in quarter 2. Worse 42% of the time in quarter 3.
Optimal	KNNAcc60-Median	8%	0%	Equal in quarter 0. Better 30% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	KNNAcc60-Third	0%	13%	Equal in quarter 0. Worse 50% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	KNNAcc75-First	0%	64%	Equal in quarter 0. Worse 57% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc75-Median	0%	67%	Equal in quarter 0. Equal in quarter 1. Worse 7% of the time in quarter 2. Equal in quarter 3.
Optimal	KNNAcc75-Third	0%	59%	Equal in quarter 0. Worse 73% of the time in quarter 1. Worse 92% of the time in quarter 2. Worse 69% of the time in quarter 3.

Optimal	KNNAcc85-First	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc85-Median	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc85-Third	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc60-First	4%	31%	Better 4% of the time in quarter 0. Better 11% of the time in quarter 1. Worse 23% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc60-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	NNAcc60-Third	0%	69%	Equal in quarter 0. Worse 76% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc75-First	1%	35%	Equal in quarter 0. Better 3% of the time in quarter 1. Worse 38% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc75-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Optimal	NNAcc75-Third	0%	69%	Equal in quarter 0. Worse 76% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc85-First	0%	60%	Equal in quarter 0. Worse 65% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 73% of the time in quarter 3.

Optimal	NNAcc85-Median	0%	67%	Equal in quarter 0. Worse 69% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc85-Third	0%	70%	Equal in quarter 0. Worse 80% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Unoptimal	C45Acc60-First	53%	0%	Better 52% of the time in quarter 0. Better 100% of the time in quarter 1. Better 61% of the time in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc60-Median	58%	0%	Better 38% of the time in quarter 0. Better 100% of the time in quarter 1. Better 84% of the time in quarter 2. Better 7% of the time in quarter 3.
Unoptimal	C45Acc60-Third	50%	0%	Better 23% of the time in quarter 0. Better 80% of the time in quarter 1. Better 88% of the time in quarter 2. Better 7% of the time in quarter 3.
Unoptimal	C45Acc75-First	69%	0%	Better 42% of the time in quarter 0. Better 100% of the time in quarter 1. Better 100% of the time in quarter 2. Better 34% of the time in quarter 3.
Unoptimal	C45Acc75-Median	75%	0%	Equal in quarter 0. Better 96% of the time in quarter 1. Better 100% of the time in quarter 2. Better 53% of the time in quarter 3.
Unoptimal	C45Acc75-Third	40%	0%	Better 9% of the time in quarter 0. Better 73% of the time in quarter 1. Better 76% of the time in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc85-First	6%	0%	Equal in quarter 0. Better 11% of the time in quarter 1. Better 11% of the time in quarter 2. Equal in quarter 3.

Unoptimal	C45Acc85-Median	22%	0%	Equal in quarter 0. Better 26% of the time in quarter 1. Better 38% of the time in quarter 2. Better 23% of the time in quarter 3.
Unoptimal	C45Acc85-Third	8%	2%	Equal in quarter 0. Better 11% of the time in quarter 1. Wild results in quarter 2. Better 11% of the time in quarter 3.
Unoptimal	KNNAcc60-First	50%	0%	Better 52% of the time in quarter 0. Better 100% of the time in quarter 1. Better 46% of the time in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc60-Median	85%	0%	Better 38% of the time in quarter 0. Better 100% of the time in quarter 1. Better 100% of the time in quarter 2. Better 100% of the time in quarter 3.
Unoptimal	KNNAcc60-Third	57%	1%	Better 14% of the time in quarter 0. Wild results in quarter 1. Better 100% of the time in quarter 2. Better 73% of the time in quarter 3.
Unoptimal	KNNAcc75-First	34%	0%	Better 14% of the time in quarter 0. Better 57% of the time in quarter 1. Better 65% of the time in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc75-Median	44%	0%	Equal in quarter 0. Better 80% of the time in quarter 1. Better 100% of the time in quarter 2. Better 100% of the time in quarter 3.
Unoptimal	KNNAcc75-Third	46%	0%	Equal in quarter 0. Better 23% of the time in quarter 1. Better 61% of the time in quarter 2. Better 100% of the time in quarter 3.
Unoptimal	KNNAcc85-First	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc85-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.

Unoptimal	KNNAcc85-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc60-First	47%	0%	Better 42% of the time in quarter 0. Better 100% of the time in quarter 1. Better 46% of the time in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc60-Median	82%	0%	Better 33% of the time in quarter 0. Better 96% of the time in quarter 1. Better 100% of the time in quarter 2. Better 100% of the time in quarter 3.
Unoptimal	NNAcc60-Third	15%	0%	Equal in quarter 0. Better 19% of the time in quarter 1. Better 30% of the time in quarter 2. Better 11% of the time in quarter 3.
Unoptimal	NNAcc75-First	42%	0%	Better 33% of the time in quarter 0. Better 100% of the time in quarter 1. Better 34% of the time in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc75-Median	81%	0%	Better 4% of the time in quarter 0. Better 84% of the time in quarter 1. Better 100% of the time in quarter 2. Better 100% of the time in quarter 3.
Unoptimal	NNAcc75-Third	14%	0%	Equal in quarter 0. Better 19% of the time in quarter 1. Better 26% of the time in quarter 2. Better 11% of the time in quarter 3.
Unoptimal	NNAcc85-First	21%	0%	Better 9% of the time in quarter 0. Better 42% of the time in quarter 1. Better 23% of the time in quarter 2. Better 7% of the time in quarter 3.
Unoptimal	NNAcc85-Median	17%	0%	Better 4% of the time in quarter 0. Better 26% of the time in quarter 1. Better 15% of the time in quarter 2. Better 19% of the time in quarter 3.
Unoptimal	NNAcc85-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.

TABLE C.2: Results when comparing between approximator runs with the unoptimal GA parameter set and both optimal and unoptimal GA runs, in the *TORCS* experiments. Better and worse values represent the percentage of time that the second configuration proved significantly better, or worse respectively, than the first configuration

First	Second	Better	Worse	Analysis
Optimal	Unoptimal	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc60-First	23%	2%	Wild results in quarter 0. Better 50% of the time in quarter 1. Worse 3% of the time in quarter 2. Equal in quarter 3.
Optimal	C45Acc60-Median	20%	37%	Wild results in quarter 0. Better 61% of the time in quarter 1. Worse 38% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc60-Third	5%	14%	Wild results in quarter 0. Better 15% of the time in quarter 1. Worse 19% of the time in quarter 2. Worse 19% of the time in quarter 3.
Optimal	C45Acc75-First	17%	45%	Wild results in quarter 0. Wild results in quarter 1. Worse 100% of the time in quarter 2. Worse 73% of the time in quarter 3.
Optimal	C45Acc75-Median	17%	41%	Wild results in quarter 0. Better 61% of the time in quarter 1. Worse 53% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc75-Third	5%	45%	Wild results in quarter 0. Better 15% of the time in quarter 1. Worse 69% of the time in quarter 2. Worse 92% of the time in quarter 3.

Optimal	C45Acc85-First	4%	45%	Worse 9% of the time in quarter 0. Better 15% of the time in quarter 1. Worse 69% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc85-Median	4%	43%	Worse 19% of the time in quarter 0. Better 15% of the time in quarter 1. Worse 53% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	C45Acc85-Third	4%	43%	Worse 19% of the time in quarter 0. Better 15% of the time in quarter 1. Worse 53% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc60-First	14%	62%	Wild results in quarter 0. Wild results in quarter 1. Worse 100% of the time in quarter 2. Worse 92% of the time in quarter 3.
Optimal	KNNAcc60-Median	4%	55%	Worse 14% of the time in quarter 0. Wild results in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc60-Third	0%	52%	Worse 33% of the time in quarter 0. Worse 3% of the time in quarter 1. Worse 69% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc75-First	0%	34%	Worse 42% of the time in quarter 0. Worse 11% of the time in quarter 1. Worse 69% of the time in quarter 2. Worse 11% of the time in quarter 3.
Optimal	KNNAcc75-Median	0%	50%	Worse 47% of the time in quarter 0. Worse 15% of the time in quarter 1. Worse 38% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc75-Third	0%	65%	Worse 47% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 92% of the time in quarter 2. Worse 100% of the time in quarter 3.

Optimal	KNNAcc85-First	0%	66%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 88% of the time in quarter 3.
Optimal	KNNAcc85-Median	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	KNNAcc85-Third	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc60-First	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc60-Median	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc60-Third	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc75-First	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc75-Median	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc75-Third	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.

Optimal	NNAcc85-First	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc85-Median	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Optimal	NNAcc85-Third	0%	69%	Worse 57% of the time in quarter 0. Worse 19% of the time in quarter 1. Worse 100% of the time in quarter 2. Worse 100% of the time in quarter 3.
Unoptimal	C45Acc60-First	31%	0%	Better 52% of the time in quarter 0. Better 61% of the time in quarter 1. Equal in quarter 2. Better 11% of the time in quarter 3.
Unoptimal	C45Acc60-Median	32%	0%	Better 47% of the time in quarter 0. Better 80% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc60-Third	34%	0%	Better 38% of the time in quarter 0. Better 96% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc75-First	26%	0%	Better 52% of the time in quarter 0. Better 50% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc75-Median	32%	0%	Better 47% of the time in quarter 0. Better 80% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc75-Third	29%	0%	Better 33% of the time in quarter 0. Better 80% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.

Unoptimal	C45Acc85-First	22%	0%	Better 47% of the time in quarter 0. Better 42% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc85-Median	28%	0%	Better 38% of the time in quarter 0. Better 73% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	C45Acc85-Third	43%	0%	Better 38% of the time in quarter 0. Better 73% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc60-First	17%	28%	Better 52% of the time in quarter 0. Wild results in quarter 1. Worse 88% of the time in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc60-Median	25%	6%	Better 38% of the time in quarter 0. Better 61% of the time in quarter 1. Equal in quarter 2. Worse 22% of the time in quarter 3.
Unoptimal	KNNAcc60-Third	9%	2%	Better 23% of the time in quarter 0. Better 11% of the time in quarter 1. Equal in quarter 2. Worse 7% of the time in quarter 3.
Unoptimal	KNNAcc75-First	16%	0%	Equal in quarter 0. Better 65% of the time in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc75-Median	43%	0%	Equal in quarter 0. Better 65% of the time in quarter 1. Better 100% of the time in quarter 2. Better 7% of the time in quarter 3.
Unoptimal	KNNAcc75-Third	8%	14%	Equal in quarter 0. Better 30% of the time in quarter 1. Equal in quarter 2. Worse 57% of the time in quarter 3.

Unoptimal	KNNAcc85-First	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc85-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	KNNAcc85-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc60-First	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc60-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc60-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc75-First	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc75-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc75-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc85-First	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc85-Median	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.
Unoptimal	NNAcc85-Third	0%	0%	Equal in quarter 0. Equal in quarter 1. Equal in quarter 2. Equal in quarter 3.

Bibliography

[1] M. Morosan and R. Poli, "Automated Game Balancing in Ms PacMan and StarCraft using Evolutionary Algorithms", in *Applications of Evolutionary Computation*, G. Squillero and K. Sim, Eds., Springer, Cham, 2017, pp. 377–392, ISBN: 978-3-319-55849-3. DOI: 10.1007/978-3-319-55849-3_25. [Online]. Available: http://link.springer.com/10.1007/978-3-319-55849-3_25.

[2] ——, "Speeding Up Genetic Algorithm-based Game Balancing using Fitness Predictors", in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17, New York, NY, USA: ACM, 2017, pp. 91–92, ISBN: 978-1-4503-4939-0. DOI: 10.1145/3067695.3076011. [Online]. Available: http://doi.acm.org/10.1145/3067695.3076011.

[3] ——, "Evolving a Designer-Balanced Neural Network for Ms PacMan", in *2017 9th Computer Science and Electronic Engineering (CEEC)*, Sep. 2017, pp. 100–105. DOI: 10.1109/CEEC.2017.8101607.

[4] ——, "Online-Trained Fitness Approximators for Real-World Game Balancing", in *Applications of Evolutionary Computation*, K. Sim and P. Kaufmann, Eds., vol. 10784 LNCS, Cham: Springer International Publishing, 2018, pp. 292–307, ISBN: 978-3-319-77538-8. DOI: 10.1007/978-3-319-77538-8_21.

[5] A. Iacob, M. Morosan, F. Sepulveda, and R. Poli, "Genetic Optimisation of BCI Systems for Identifying Games Related Cognitive States", in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ACM, 2018, pp. 237–238.

[6] M. Morosan and R. Poli, "Lessons from Testing an Evolutionary Automated Game Balancer in Industry", in *2018 IEEE Games, Entertainment, Media Conference (GEM) (2018 IEEE GEM)*, Galway, Ireland, Aug. 2018.

[7] G. Mountain, *Tactics in Fable Legends*, 2015. [Online]. Available: http://gwaredd.github.io/nuclai_mcts/root/index.html (visited on 09/02/2017).

[8] C. Kerr, *DeepMind wants to answer the big ethical questions posed by AI*, 2017. [Online]. Available: https://www.gamasutra.com/view/news/307008/DeepMind_wants_to_answer_the_big_ethical_questions_posed_by_AI.php (visited on 11/06/2017).

[9] R. Cohen, "History and genre", *Neohelicon*, vol. 13, no. 2, pp. 87–105, 1986.

[10] D. Clearwater, "What Defines Video Game Genre? Thinking about Genre Study after the Great Divide", *Loading...*, vol. 5, no. 8, pp. 29–49, 2011. [Online]. Available: http://journals.sfu.ca/loading/index.php/loading/article/viewArticle/67.

[11] A. Tychsen and M. Hitchens, "Game Time: Modeling and Analyzing Time in Multiplayer and Massively Multiplayer Games", *Games and Culture*, vol. 4, no. 2, pp. 170–201, 2009, ISSN: 1555-4120. DOI: 10.1177/1555412008325479.

[12] M. Sicart, "Game Studies - Defining Game Mechanics", *Game Stud.*, vol. 8, pp. 1–15, 2008, ISSN: 16047982. [Online]. Available: http://gamestudies.org/0802/articles/sicart.

[13] T. H. Apperley, "Genre and Game Studies: Toward a Critical Approach to Video Game Genres", *Simulation & Gaming*, vol. 37, no. 1, pp. 6–23, 2006, ISSN: 1046-8781, 1552-826X. DOI: 10.1177/1046878105282278. [Online]. Available: http://sag.sagepub.com/content/37/1/6.short.

[14] I. Schreiber, *Game Balance Concepts*, 2010. [Online]. Available: https://gamebalanceconcepts.wordpress.com/2010/07/07/level-1-intro-to-game-balance/.

[15] A. Cincotti, H. Iida, A. Cincotti, and H. Iida, "Outcome Uncertainty and Interestedness in Game-playing: A Case Study Using Synchronized Hex", *New Mathematics and Natural Computation (NMNC)*, vol. 02, no. 02, pp. 173–181, 2006, ISSN: 1793-7027.

[16] T. Mahlmann, J. Togelius, and G. N. Yannakakis, "Evolving Card Sets Towards Balancing Dominion", *2012 IEEE Congress on Evolutionary Computation*, pp. 1–8, 2012.

[17] K. Burgun, *Understanding Balance in Video Games*, 2011. [Online]. Available: http://www.gamasutra.com/view/feature/134768/.

[18] D. Sirlin, *Balancing Multiplayer Games*, 2009. [Online]. Available: http://www.sirlin.net/articles/balancing-multiplayer-games-part-1-definitions.

[19] M. Beyer, A. Agureikin, A. Anokhin, C. Laenger, F. Nolte, J. Winterberg, M. Renka, M. Rieger, N. Pflanzl, M. Preuss, and V. Volz, "An Integrated Process for Game Balancing", *IEEE Conference on Computational Intelligence and Games*, 2016.

[20] A. M. Turing, "I.—Computing Machinery and Intelligence", *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. [Online]. Available: https://academic.oup.com/mind/article-lookup/doi/10.1093/mind/LIX.236.433.

[21] N. A. Barricelli, "Symbiogenetic Evolution Processes Realized by Artificial Methods", *Methodos*, vol. 9, no. 35-36, pp. 143–182, 1957.

[22] ——, "Numerical Testing of Evolution Theories, Part II Preliminary Tests of Performance", *Symbiogenesis and Terrestrial Life, Acta Biotheoretica*, vol. 16, pp. 99–126, 1962.

[23] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[24] D. Whitley, "A Genetic Algorithm Tutorial", *Statistics and Computing*, vol. 4, no. 2, pp. 65–85, Jun. 1994, ISSN: 0960-3174. DOI: 10.1007/BF00175354. [Online]. Available: http://link.springer.com/10.1007/BF00175354.

[25] D. E. Goldberg *et al.*, *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-wesley Reading Menlo Park, 1989, vol. 412.

[26] R. L. Haupt, S. E. Haupt, and S. E. Haupt, *Practical Genetic Algorithms*. Wiley New York, 1998, vol. 2.

[27] C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcazar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preuß, J. Togelius, and G. N. Yannakakis, Eds., *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6024, ISBN: 978-3-642-12238-5. DOI: 10.1007/978-3-642-12239-2. [Online]. Available: http://link.springer.com/10.1007/978-3-642-12239-2.

[28] C. Di Chio, A. Agapitos, S. Cagnoni, C. Cotta, F. F. de Vega, G. A. Di Caro, R. Drechsler, A. Ekárt, A. I. Esparcia-Alcázar, M. Farooq, W. B. Langdon, J. J. Merelo-Guervós, M. Preuss, H. Richter, S. Silva, A. Simões, G. Squillero, E. Tarantino, A. G. B. Tettamanzi, J. Togelius, N. Urquhart, A. Ş. Uyar, and G. N. Yannakakis, Eds., *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7248, ISBN: 978-3-642-29177-7. DOI: 10.1007/978-3-642-29178-4. [Online]. Available: http://link.springer.com/10.1007/978-3-642-29178-4.

[29] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A field guide to genetic programming*. Lulu.com, 2008.

[30] J. R. Koza, "Introduction to Genetic Programming", in *Advances in Genetic Programming*, 1994, pp. 21–42, ISBN: 0-262-58133-7. DOI: doi:10.1145/1388969.1389057. [Online]. Available: http://www.genetic-programming.com/jkpdf/aigp1994intro.pdf.

[31] M. Sipper, "Evolving Game-Playing Strategies with Genetic Programming", *ERCIM News*, vol. 64, pp. 28–29, 2008. [Online]. Available: http://www.ercim.eu/publication/Ercim_News/enw64/EN64.pdf.

[32] F. Fernandez-de-Vega, G. G. Gil, J. A. G. Pulido, and J. L. Guisado, "Control of Bloat in Genetic Programming by Means of the Island Model", in *Parallel Problem Solving from Nature - PPSN VIII*, vol. 3242, 2004, pp. 263–271, ISBN: 3-540-23092-0. DOI: doi:10.1007/b100601.

[33] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill, "Grammar-based Genetic Programming: A Survey", *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010, ISSN: 13892576. DOI: 10.1007/s10710-010-9109-y.

[34] L. Brandy, *Using Genetic Algorithms to Find Starcraft 2 Build Orders*, 2010. [Online]. Available: http://lbrandy.com/blog/2010/11/using-genetic-algorithms-to-find-starcraft-2-build-orders/.

[35] K. O. Stanley, B. Bryant, and R. Miikkulainen, *Evolving Neural Network Agents in the NERO Video Game*, 2005. [Online]. Available: ftp://www.cs.utexas.edu/pub/neural-nets/papers/stanley.cig05.pdf (visited on 09/25/2015).

[36] D. B. Fogel, "Blondie24: Playing at the Edge of AI", in *Blondie24*, ser. The Morgan Kaufmann Series in Artificial Intelligence, Morgan Kaufmann, 2002, pp. 273–298, ISBN: 9781558607835. DOI: 10.1016/B978-155860783-5/50016-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781558607835500167.

[37] J. Žegklitz and P. Pošík, "Model Selection and Overfitting in Genetic Programming: Empirical Study", p. 8, Apr. 2015. arXiv: 1504.08168. [Online]. Available: http://arxiv.org/abs/1504.08168.

[38] C. U. Lim, R. Baumgarten, and S. Colton, "Evolving Behaviour Trees for the Commercial Game DEFCON", in *European Conference on the Applications of Evolutionary Computation*, Springer, vol. 6024 LNCS, 2010, pp. 100–110.

[39] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Evolving Behaviour Trees for the Mario AI Competition using Grammatical Evolution", in *Applications of Evolutionary Computation*, vol. 6624 LNCS, 2011, pp. 123–132, ISBN: 978-3-642-20525-5. DOI: 10.1007/978-3-642-20525-5_13.

[40] D. A. Savic and G. A. Walters, "Genetic Algorithms for Least-Cost Design of Water Distribution Networks", en, *Journal of Water Resources Planning and Management*, vol. 123, no. 2, pp. 67–77, Mar. 1997, ISSN: 0733-9496. DOI: 10.1061/(ASCE)0733-9496(1997)123:2(67). [Online]. Available: http://ascelibrary.org/doi/abs/10.1061/(ASCE)0733-9496(1997)123:2(67).

[41] B. L. Miller and D. E. Goldberg, "Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise", *Evolutionary Computation*, vol. 4, no. 2, pp. 113–131, Jun. 1996. DOI: 10.1162/evco.1996.4.2.113. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/evco.1996.4.2.113.

[42] A. Wright, J. Rowe, and J. Neil, "Analysis of the Simple Genetic Algorithm on the Single-peak and Double-peak Landscapes", in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02*, vol. 1, IEEE, 2002, pp. 214–219, ISBN: 0-7803-7282-4. DOI: 10.1109/CEC.2002.1006236. [Online]. Available: http://ieeexplore.ieee.org/document/1006236/.

[43] K. Deb and D. E. Goldberg, "Analyzing Deception in Trap Functions", in, vol. 2, 1993, pp. 93–108. DOI: 10.1016/B978-0-08-094832-4.50012-X. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/B978008094832450012X.

[44] N. Nedjah and L. d. M. Mourelle, "Evolutionary Multi-objective Optimisation: A Survey", *International Journal of Bio-Inspired Computation*, vol. 7, no. 1, p. 1, 2015, ISSN: 1758-0366. DOI: 10.1504/IJBIC.2015.067991. [Online]. Available: http://www.inderscience.com/link.php?id=67991.

[45] X.-S. Yang, "Bat Algorithm for Multi-objective Optimisation", *International Journal of Bio-Inspired Computation*, vol. 3, no. 5, pp. 267–274, 2011.

[46] J. E. Fieldsend, R. M. Everson, and S. Singh, "Using Unconstrained Elite Archives for Multi-objective Optimisation", 2003.

[47] A. K. Dubey and V. Yadava, "Multi-objective Optimisation of Laser Beam Cutting Process", *Optics & Laser Technology*, vol. 40, no. 3, pp. 562–570, 2008.

[48] R. P. C. Moreira, E. F. Wanner, F. V. C. Martins, and J. F. M. Sarubbi, "The Menu Planning Problem: A Multiobjective Approach for Brazilian Schools Context", in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ACM, 2017, pp. 113–114.

[49] S. Giarola, A. Zamboni, and F. Bezzo, "Spatially Explicit Multi-objective Optimisation for Design and Planning of Hybrid First and Second Generation Biorefineries", *Computers & Chemical Engineering*, vol. 35, no. 9, pp. 1782–1797, 2011.

[50] R. Marler and J. Arora, "Survey of Multi-objective Optimization Methods for Engineering", *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, Apr. 2004, ISSN: 1615-147X. DOI: 10.1007/s00158-003-0368-6. [Online]. Available: http://link.springer.com/10.1007/s00158-003-0368-6.

[51] N. B. Urquhart, "Evaluating the Performance of an Evolutionary Tool for Exploring Solution Fronts", in *International Conference on the Applications of Evolutionary Computation*, Springer, 2018, pp. 523–537.

[52] Y. Jin, "A Comprehensive Survey of Fitness Approximation in Evolutionary Computation", *Soft Computing*, vol. 9, no. 1, pp. 3–12, 2005, ISSN: 1432-7643. DOI: 10.1007/s00500-003-0328-5.

[53] M. Bhattacharya, "Evolutionary Approaches to Expensive Optimisation", *Arxiv - Computers & Society*, vol. 2, no. 3, pp. 53–59, 2013, ISSN: 21654069. DOI: 10.14569/IJARAI.2013.020308.

[54] B. Johanson and R. Poli, "GP-Music: An Interactive Genetic Programming System for Music Generation with Automated Fitness Raters", Tech. Rep., 1998.

[55] B. Jin, Yaochu and Sendhoff, "Reducing Fitness Evaluations using Clustering Techniques and Neural Network Ensembles", *Genetic and Evolutionary Computation–GECCO 2004*, pp. 688–699, 2004, ISSN: 03029743. DOI: 10.1007/978-3-540-24854-5_71.

[56] D. R. Carvalho and A. A. Freitas, "A Hybrid Decision Tree/Genetic Algorithm Method for Data Mining", *Information Sciences*, vol. 163, no. 1, pp. 13–35, 2004, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2003.03.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025503004146.

[57] K. Deb, A. Sinha, P. J. Korhonen, and J. Wallenius, "An Interactive Evolutionary Multiobjective Optimization Method Based on Progressively Approximated Value Functions", *IEEE Transactions on Evolutionary Computation*, vol. 14, no. 5, pp. 723–739, Oct. 2010, ISSN: 1089-778X. DOI: 10.1109/TEVC.2010.2064323. [Online]. Available: http://ieeexplore.ieee.org/document/5585740/.

[58] P. K. S. Nain and K. Deb, "A Multi-Objective Optimization Procedure with Successive Approximate Models", [Online]. Available: http://www.iitk.ac.in/kangal.

[59] J. Dias, H. Rocha, B. Ferreira, and M. d. C. Lopes, "A Genetic Algorithm with Neural Network Fitness Function Evaluation for IMRT Beam Angle Optimization", *Central European Journal of Operations Research*, vol. 22, no. 3, pp. 431–455, Sep. 2014. DOI: 10.1007/s10100-013-0289-4. [Online]. Available: http://link.springer.com/10.1007/s10100-013-0289-4.

[60] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.", *Psychological review*, vol. 65, no. 6, pp. 386–408, Nov. 1958, ISSN: 0033-295X. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/13602029.

[61] J. J. Weng, N. Ahuja, and T. S. Huang, "Learning Recognition and Segmentation of 3-D Objects from 2-D Images", in *Computer Vision, 1993. Proceedings., Fourth International Conference on*, IEEE, 1993, pp. 121–128.

[62] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[63] S. Behnke, *Hierarchical Neural Networks for Image Interpretation*. Springer, 2003, vol. 2766.

[64] Y. Liu and X. Yao, "Evolutionary Design of Artificial Neural Networks with Different Nodes", in *Proceedings of IEEE International Conference on Evolutionary Computation*, IEEE, 1996, pp. 670–675, ISBN: 0-7803-2902-3. DOI: 10.1109/ICEC.1996.542681. [Online]. Available: http://ieeexplore.ieee.org/document/542681/.

[65] J. Schaffer, D. Whitley, and L. Eshelman, "Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art", in *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, IEEE Comput. Soc. Press, 1992, pp. 1–37, ISBN: 0-8186-2787-5. DOI: 10.1109/COGANN.1992.273950. [Online]. Available: http://ieeexplore.ieee.org/document/273950/.

[66] D. Floreano and F. Mondada, "Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural Network Driven Robot", in *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, The MIT Press, 1994, pp. 421–430.

[67] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks Through Augmenting Topologies", *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[68] J. R. Quinlan, "C4.5: Programming for Machine Learning", *Morgan Kauffmann*, vol. 38, 1993.

[69] N. S. Altman, "An Introduction to Kernel and Nearest-neighbor Nonparametric Regression", *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[70] G. Yannakakis and J. Togelius, "A Panorama of Artificial and Computational Intelligence in Games", *IEEE Transactions on Computational Intelligence and AI in Games*, no. c, pp. 1–1, 2014, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2014.2339221. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6855367.

[71] G. N. Yannakakis, "Game AI Revisited", in *Proceedings of the 9th Conference on Computing Frontiers*, 2012, pp. 285–292, ISBN: 978-1-4503-1215-8. DOI: 10.1145/2212908.2212954. [Online]. Available: http://doi.acm.org/10.1145/2212908.2212954.

[72] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A Mixed-initiative Level Design Tool", in *FDG '10 - Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 2010, pp. 209–216, ISBN: 9781605589374. DOI: 10.1145/1822348.1822376. [Online]. Available: http://dl.acm.org/citation.cfm?id=1822376.

[73] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient Sketchbook: Computer-aided Game Level Authoring", in *Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG 2013)*, 2013, pp. 213–220. [Online]. Available: http://www.fdg2013.org/program/papers/paper28_liapis_etal.pdf.

[74] Q. X. Q. Xiong and X.-y. H. X.-y. Huang, "Speed Tree-Based Forest Simulation System", *Electrical and Control Engineering (ICECE), 2010 International Conference on*, 2010. DOI: 10.1109/iCECE.2010.738.

[75] J. Togelius, "A Procedural Critique of Deontological Reasoning", in *Proceedings of DiGRA 2011 Conference: Think Design Play*, 2011.

[76] C. Browne and F. Maire, "Evolutionary Game Design", English, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1–16, Mar. 2010, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2010.2041928. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5404867.

[77] B. Bouzy and B. Helmstetter, "Monte-Carlo Go Developments", in *Advances in Computer Games*, H. J. Herik, H. Iida, and E. A. Heinz, Eds., Boston, MA: Springer US, 2004, pp. 159–174, ISBN: 978-1-4757-4424-8. DOI: 10.1007/978-0-387-35706-5. [Online]. Available: http://link.springer.com/10.1007/978-0-387-35706-5.

[78] G. Chaslot, J. T. Saito, J. W. H. M. Uiterwijk, B. Bouzy, and H. J. van den Herik, "Monte-Carlo Strategies for Computer Go", in *Belgian/Netherlands Artificial Intelligence Conference*, 2006.

[79] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI.", *AIIDE*, pp. 216–217, 2008. [Online]. Available: http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf.

[80] S. Branavan, D. Silver, and R. Barzilay, *Non-Linear Monte-Carlo Search in Civilization II*. [Online]. Available: http://people.csail.mit.edu/regina/my_papers/civ_ijcai2011.pdf (visited on 03/31/2015).

[81] N. Sephton, P. I. Cowling, E. Powley, and N. H. Slaven, "Heuristic Move Pruning in Monte Carlo Tree Search for the Strategic Card Game Lords of War", 2014.

[82] B. Cowley, "Player Profiling and Modelling in Computer and Video Games", PhD thesis, 2009, p. 299. [Online]. Available: http://www.researchgate.net/publication/253240709_Player_Profiling_and_Modelling_in_Computer_and_Video_Games.

[83] K. P. Sycara, "Negotiation Planning: An AI Approach", *European Journal of Operational Research*, vol. 46, no. 2, pp. 216–234, May 1990, ISSN: 03772217. DOI: 10.1016/0377-2217(90)90133-V. [Online]. Available: http://www.sciencedirect.com/science/article/pii/037722179090133V.

[84] C. M. Jonker, K. V. Hindriks, P. Wiggers, and J. Broekens, *Negotiating Agents*, en, Sep. 2012. DOI: 10.1609/aimag.v33i3.2421. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2421.

[85] J. Long, N. R. Sturtevant, M. Buro, and T. Furtak, *Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search*, 2010. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/viewFile/1876/1942 (visited on 09/25/2015).

[86] F. Kolbe, "Goal Oriented Task Planning for Autonomous Service Robots", PhD thesis, 1990, p. 100. DOI: 10.1007/BF03192151. [Online]. Available: http://users.informatik.haw-hamburg.de/~kolbe_j/thesis/RGOAP-Felix_Kolbe-Masterthesis-2013.pdf.

[87] J. Orkin, "Three States and a Plan: The AI of FEAR", *Game Developers Conference*, vol. 2006, no. 1, pp. 1–18, 2006, ISSN: 00099104. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.8551&rep=rep1&type=pdf.

[88] C. García-García, L. Torres-López, V. Larios-Rosillo, and H. Luga, "A GOAP Architecture for Emergency Evacuations in Serious Games", in *11th International Conference on Intelligent Games and Simulation, GAME-ON 2010*, 2010, pp. 10–12, ISBN: 9789077381588. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84877933845&partnerID=40&md5=e0b403a616252005ac6406c35eacff98.

[89] P. Bjarnolf, "Threat Analysis Using Goal-Oriented Action Planning : Planning in the Light of Information Fusion", *Information FUsion*, p. 68, 2008. [Online]. Available: http://his.diva-portal.org/smash/record.jsf?pid=diva2:2228&dswid=7514.

[90] G. J. Olsder and G. P. Papavassilopoulos, "A Markov Chain Game with Dynamic Information", *Journal of Optimization Theory and Applications*, vol. 59, no. 3, pp. 467–486, Dec. 1988, ISSN: 0022-3239. DOI: 10.1007/BF00940310. [Online]. Available: http://link.springer.com/10.1007/BF00940310.

[91] M. J. Nelson and M. Mateas, "Recombinable Game Mechanics for Automated Design Support", *Proceedings of the 4th Artificial Intelligence and Interactive Digital Entertainment Conference*, vol. 4, pp. 84–89, 2008. [Online]. Available: http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-014.pdf.

[92] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, *Towards a Video Game Description Language*, Nov. 2013. [Online]. Available: http://strathprints.strath.ac.uk/45278/1/dagstuhl_vgdl.pdf.

[93] T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning", English, in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, IEEE, Aug. 2013, pp. 1–8, ISBN: 978-1-4673-5311-3. DOI: 10.1109/CIG.2013.6633610. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6633610.

[94] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI Championship: Level Generation Track", English, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 332–347, Dec. 2011, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2011.2166267. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6003769.

[95] M. M. Kate Compton, "Procedural Level Design for Platform Games", [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.501.9465.

[96] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Automatic Content Generation in the Galactic Arms Race Video Game", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 245–263, 2009, ISSN: 1943068X. DOI: 10.1109/TCIAIG.2009.2038365.

[97] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of Go with Deep Neural Networks and Tree Search", *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016, ISSN: 0028-0836. DOI: 10.1038/nature16961. [Online]. Available: http://dx.doi.org/10.1038/nature16961.

[98] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 General Video Game Playing Competition", *IEEE Transactions on Computational Intelligence and AI in Games*, no. c, pp. 1–1, 2015, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2015.2402393. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7038214.

[99] J. K. Olesen, G. N. Yannakakis, and J. Hallam, "Real-time Challenge Balance in an RTS Game Using rtNEAT", English, in *2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008*, IEEE, Dec. 2008, pp. 87–94, ISBN: 9781424429745. DOI: 10.1109/CIG.2008.5035625. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5035625.

[100] M. Sipper, *Evolved to Win*. Lulu, 2011. [Online]. Available: http://www.lulu.com/.

[101] S. Polberg, M. Paprzycki, and M. Ganzha, "Developing Intelligent Bots for the Diplomacy Game", *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 589–596, 2011.

[102] S. Fisher, *Working With Conflict: Skills and Strategies for Action*. Zed Books, 2000, vol. 4, p. 185, ISBN: 1856498379. [Online]. Available: https://books.google.com/books?id=YCPEoKBlS54C&pgis=1.

[103] P. Hingston, "A Turing Test for Computer Game Bots", English, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 3, pp. 169–186, Sep. 2009, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2009.2032534. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5247069.

[104] E. Haller and T. Rebedea, "Designing a Chat-bot that Simulates an Historical Figure", in *Proceedings - 19th International Conference on Control Systems and Computer Science, CSCS 2013*, 2013, pp. 582–589, ISBN: 978-1-4673-6140-8. DOI: 10.1109/CSCS.2013.85.

[105] S. Cagnoni, A. B. Dobrzeniecki, R. Poli, and J. C. Yanch, "Genetic Algorithm-based Interactive Segmentation of 3D Medical Images", *Image and Vision Computing*, vol. 17, no. 12, pp. 881–895, 1999.

[106] D. S. Linden and E. E. Altshuler, "Automating Wire Antenna Design using Genetic Algorithms", *Microwave Journal*, vol. 39, no. 3, pp. 74–81, 1996.

[107] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based Procedural Content Generation: A Taxonomy and Survey", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011, ISSN: 1943068X. DOI: 10.1109/TCIAIG.2011.2148116.

[108] M. Cook, "A Vision For Continuous Automated Game Design", Jul. 2017. arXiv: 1707.09661. [Online]. Available: http://arxiv.org/abs/1707.09661.

[109] C. Bauckhage, K. Kersting, R. Sifa, C. Thurau, A. Drachen, and A. Canossa, "How Players Lose Interest in Playing a Game: An Empirical Study Based on Distributions of Total Playing Times", English, in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Sep. 2012, pp. 139–146, ISBN: 978-1-4673-1194-6. DOI: 10.1109/CIG.2012.6374148. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6374148.

[110] B. Shen, *StarCraft - Beyond WarCraft in Space*, 2003. [Online]. Available: http://web.stanford.edu/group/htgg/sts145papers/bshen_2003_1.pdf (visited on 09/25/2015).

[111] J. Juul, "Zero-Player Games", in *The Philosophy of Computer Games Conference 2012*, 2012, pp. 1–11. [Online]. Available: https://blip.tv/arsgames/8439_staffan-bjork-5948058.

[112] V. Volz, G. Rudolph, and B. Naujoks, "Demonstrating the Feasibility of Automatic Game Balancing", Mar. 2016. arXiv: 1603.03795. [Online]. Available: http://arxiv.org/abs/1603.03795.

[113] H. Chen, Y. Mori, and I. Matsuba, "Solving the Balance Problem of On-Line Role-Playing Games Using Evolutionary Algorithms", *Journal of Software Engineering and Applications*, vol. 05, no. 08, pp. 574–582, 2012, ISSN: 1945-3116. DOI: 10.4236/jsea.2012.58066. [Online]. Available: http://www.scirp.org/journal/PaperDownload.aspx?DOI=10.4236/jsea.2012.58066.

[114] J. Liu, J. Togelius, D. Pérez-Liébana, and S. M. Lucas, "Evolving game skill-depth using general video game ai agents", in *2017 IEEE Congress on Evolutionary Computation (CEC)*, Jun. 2017, pp. 2299–2307. DOI: 10.1109/CEC.2017.7969583.

[115] J. I. E. Ramos, R. A. Vázquez, and D. F. México, "Locating Seismic-sense Stations through Genetic Algorithms", in *Proceedings of the GECCO*, vol. 11, 2011, pp. 941–948.

[116] S. Preble, M. Lipson, and H. Lipson, "Two-dimensional Photonic Crystals Designed by Evolutionary Algorithms", *Applied Physics Letters*, vol. 86, no. 6, p. 61 111, 2005.

[117] O. E. David, H. J. van den Herik, M. Koppel, and N. S. Netanyahu, "Genetic Algorithms for Evolving Computer Chess Programs", *Evolutionary Computation, IEEE Transactions on*, vol. 18, no. 5, pp. 779–789, 2014.

[118] *OpenAI Five Benchmark: Results*. [Online]. Available: https://blog.openai.com/openai-five-benchmark-results/ (visited on 08/22/2018).

[119] T. L. Taylor, *Raising the Stakes: E-Sports and the Professionalization of Computer Gaming*. MIT Press, 2012, p. 336, ISBN: 0262300478. [Online]. Available: https://books.google.com/books?id=CiL8aPrSeKcC&pgis=1.

[120] D. Lee and L. J. Schoenstedt, "Comparison of eSports and Traditional Sports Consumption Motives.", en, *ICHPER-SD Journal of Research*, vol. 6, no. 2, pp. 39–44, Nov. 2010, ISSN: ISSN-1930-4595. [Online]. Available: http://eric.ed.gov/?id=EJ954495.

[121] G. Cheung and J. Huang, "Starcraft from the Stands: Understanding the Game Spectator", in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, 2011, pp. 763–772.

[122] J. Rambusch, P. Jakobsson, and D. Pargman, *Exploring E-sports : A Case Study of Game Play in Counter-Strike*, eng, 2007. [Online]. Available: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A25495&dswid=-1723.

[123] T. Weiss and S. Schiele, "Virtual Worlds in Competitive Contexts: Analyzing eSports Consumer Needs", in *Electronic Markets*, vol. 23, 2013, pp. 307–316. DOI: 10.1007/s12525-013-0127-5.

[124] C. Souza, A. Kirillov, M. D. Catalano, and A. Contributors, *The Accord.NET Framework*, 2014. DOI: 10.5281/zenodo.1029480. [Online]. Available: http://accord-framework.net.

[125] L. Shelton, *PacmanAI-MCTS*. [Online]. Available: https://github.com/LoveDuckie/PacmanAI-MCTS.

[126] M. Morosan, *PacMan-CSharp*. [Online]. Available: https://github.com/mihail-morosan/PacMan-CSharp.

[127] H. V. Seijen, J. Romoff, M. Fatemi, R. Laroche, T. Barnes, J. Tsang, and M. Maluuba, "Hybrid Reward Architecture for Reinforcement Learning", in *Advances in Neural Information Processing Systems*, 2017, pp. 5392–5402.

[128] L. L. DeLooze and W. R. Viner, "Fuzzy Q-learning in a Nondeterministic Environment: Developing an Intelligent Ms. Pac-Man Agent", in *2009 IEEE Symposium on Computational Intelligence and Games*, IEEE, Sep. 2009, pp. 162–169, ISBN: 978-1-4244-4814-2. DOI: 10.1109/CIG.2009.5286478. [Online]. Available: http://ieeexplore.ieee.org/document/5286478/.

[129] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning", in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html.

[130] T. Pepels, M. H. M. Winands, and M. Lanctot, "Real-Time Monte Carlo Tree Search in Ms Pac-Man", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 3, pp. 245–257, Sep. 2014, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2013.2291577. [Online]. Available: http://ieeexplore.ieee.org/document/6731713/.

[131] S. Lucas, "Evolving a Neural Network Location Evaluator to Play Ms. Pac-Man", *IEEE Symposium on Computational Intelligence and Games*, pp. 203–210, 2005.

[132] J. Schrum and R. Miikkulainen, "Evolving Multimodal Behavior with Modular Neural Networks in Ms. Pac-Man", in *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*, New York, New York, USA: ACM Press, 2014, pp. 325–332, ISBN: 9781450326629. DOI: 10.1145/2576768.2598234. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2576768.2598234.

[133] M. Wittkamp, L. Barone, and P. Hingston, "Using NEAT for Continuous Adaptation and Teamwork Formation in Pacman", in *2008 IEEE Symposium On Computational Intelligence and Games*, IEEE, Dec. 2008, pp. 234–242, ISBN: 978-1-4244-2973-8. DOI: 10.1109/CIG.2008.5035645. [Online]. Available: http://ieeexplore.ieee.org/document/5035645/.

[134] F. Mourato, M. P. dos Santos, and F. Birra, "Automatic Level Generation for Platform Videogames using Genetic Algorithms", in *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology - ACE '11*, New York, New York, USA: ACM Press, 2011, p. 1, ISBN: 9781450308274. DOI: 10.1145/2071423.2071433. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2071423.2071433.

[135] *StarCraft AI, the StarCraft BroodWar Resource for custom AIs*. [Online]. Available: http://www.starcraftai.com/.

[136] BWAPI, "BWAPI", [Online]. Available: https://bwapi.github.io/.

[137] MasterOfChaos, "Chaoslauncher", 2011. [Online]. Available: https://github.com/mihail-morosan/Chaoslauncher.

[138] L. Zezula, *StormLib*. [Online]. Available: https://github.com/ladislav-zezula/StormLib.

[139] B. G. Weber, M. Mateas, and A. Jhala, "Applying Goal-Driven Autonomy to StarCraft.", in *AIIDE*, 2010.

[140] N. Justesen and S. Risi, "Learning Macromanagement in Starcraft from Replays using Deep Learning", *ArXiv preprint arXiv:1707.03743*, 2017.

[141] D. Churchill and M. Buro, "Incorporating Search Algorithms into RTS Game Agents", in *AI and Interactive Digital Entertainment Conference, AIIDE (AAAI)*, 2012.

[142] N. Justesen and S. Risi, "Continual Online Evolutionary Planning for In-game Build Order Adaptation in StarCraft", in *Proceedings of the Genetic and Evolutionary Computation Conference*, ACM, 2017, pp. 187–194.

[143] P. Garcia-Sanchez, A. Tonda, A. M. Mora, G. Squillero, and J. Merelo, "Towards Automatic StarCraft Strategy Generation Using Genetic Programming", in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, Aug. 2015, pp. 284–291, ISBN: 978-1-4799-8622-4. DOI: 10.1109/CIG.2015.7317940. [Online]. Available: http://ieeexplore.ieee.org/document/7317940/.

[144] B. Wymann, *TORCS*. [Online]. Available: http://torcs.sourceforge.net/.

[145] MindArk, *Compet Game*, 2018. [Online]. Available: https://competgame.com/.

[146] *Metaheuristics in the Large*, 2018. [Online]. Available: http://www.mitlware.org/.

[147] J. Swan, S. Adriaensen, M. Bishr, E. K. Burke, J. A. Clark, P. De Causmaecker, J. Durillo, K. Hammond, E. Hart, C. G. Johnson, Z. A. Kocsis, B. Kovitz, K. Krawiec, S. Martin, J. J. Merelo, L. L. Minku, E. Ozcan, G. L. Pappa, E. Pesch, P. García-Sánchez, A. Schaerf, K. Sim, J. E. Smith, T. Stützle, S. Voß, S. Wagner, and X. Yao, "A Research Agenda for Metaheuristic Standardization", in *MIC 2015: The XI Metaheuristics International Conference*, 2015. [Online]. Available: http://www.cs.nott.ac.uk/~pszeo/docs/publications/research-agenda-metaheuristic.pdf.

[148] D. Gravina and D. Loiacono, "Procedural Weapons Generation for Unreal Tournament III", in *2015 IEEE Games Entertainment Media Conference (GEM)*, IEEE, Oct. 2015, pp. 1–8, ISBN: 978-1-4673-7452-1. DOI: 10.1109/GEM.2015.7377225. [Online]. Available: http://ieeexplore.ieee.org/document/7377225/.

[149] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt (visited on 07/02/2018).

[150] T. Thompson, L. McMillan, J. Levine, and A. Andrew, "An Evaluation of the Benefits of Look-ahead in Pac-Man", in *IEEE Symposium Computational Intelligence and Games, 2008*, IEEE, 2008, pp. 310–315, ISBN: 9781424429738. DOI: 10.1109/CIG.2008.5035655.

[151] L. Shelton, "Implementation of High-level Strategy Formulating AI in Ms Pac-Man", Tech. Rep., 2013. [Online]. Available: http://lucshelton.com/assets/Uploads/Dissertation-Main-Copy.pdf.

[152] H. G. Cobb and J. J. Grefenstette, "Genetic Algorithms for Tracking Changing Environments", in *Proceedings of the 5th International Conference on Genetic Algorithms*, 1993, pp. 523–530, ISBN: 1-55860-299-2. DOI: 10.1.1.48.6501. [Online]. Available: http://dl.acm.org/citation.cfm?id=657576.

[153] C. Coxe, *ZZZKBot*, 2015. [Online]. Available: https://github.com/chriscoxe/ZZZKBot.

[154] AIIDE, *2015 AIIDE StarCraft AI Competition Report*, 2015. [Online]. Available: http://www.cs.mun.ca/~dchurchill/starcraftaicomp/report2015.shtml.

[155] L. Davis, *Handbook of genetic algorithms*. CUMINCAD, 1991.

[156] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*. IOP Publishing Ltd., 1997.

[157] Hecht-Nielsen, "Theory of the Backpropagation Neural Network", in *International Joint Conference on Neural Networks*, IEEE, 1989, 593–605 vol.1. DOI: 10.1109/IJCNN.1989.118638. [Online]. Available: http://ieeexplore.ieee.org/document/118638/.

[158] K. Hornik, M. Stinchcombe, and H. White, "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989. DOI: 10.1016/0893-6080(89)90020-8.

[159] H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, *Neural Network Design*, 2nd. USA: Martin Hagan, 2014.

[160] K. Vora, S. Yagnik, and M. Scholar, *A Survey on Backpropagation Algorithms for Feedforward Neural Networks*, 2015.

[161] S. Lawrence, C. L. Giles, and A. C. Tsoi, "What Size Neural Network Gives Optimal Generalization ? Convergence Properties of Backpropagation", *Networks*, no. UMIACS-TR-96-22 and CS-TR-3617, pp. 1–37, 1996, ISSN: 0885-6125. DOI: 10.1023/A:1010884214864.

[162] A. Bogomolny, *Number of Trials to First Success*, 2014. [Online]. Available: http://www.cut-the-knot.org/Probability/LengthToFirstSuccess.shtml (visited on 02/05/2017).

[163] E. R. Gansner, Y. Hu, and S. Kobourov, "GMap: Visualizing Graphs and Clusters as Maps", in *2010 IEEE Pacific Visualization Symposium (PacificVis)*, IEEE, Mar. 2010, pp. 201–208, ISBN: 978-1-4244-6685-6. DOI: 10.1109/PACIFICVIS.2010.5429590. [Online]. Available: http://ieeexplore.ieee.org/document/5429590/.

[164] C. H. Tan, K. C. Tan, and A. Tay, "Dynamic Game Difficulty Scaling Using Adaptive Behavior-Based AI", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 4, pp. 289–301, Dec. 2011, ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2011.2158434. [Online]. Available: http://ieeexplore.ieee.org/document/5783334/.

[165] P. R. Williams, D. Perez-Liebana, and S. M. Lucas, "Ms. Pac-Man Versus Ghost Team CIG 2016 Competition", in *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, 2016, pp. 420–427.

[166] S. J. Russell, P. Norvig, and E. Davis, *Artificial Intelligence : A Modern Approach*, 3rd ed. Upper Saddle River NJ: Prentice Hall, 2010, p. 1132, ISBN: 9780136042594. [Online]. Available: https://www.worldcat.org/title/artificial-intelligence-a-modern-approach/oclc/359890490.

[167] S. Kelly and M. I. Heywood, "Emergent Tangled Graph Representations for Atari Game Playing Agents", in, Springer, Cham, Apr. 2017, pp. 64–79. DOI: 10.1007/978-3-319-55696-3_5. [Online]. Available: http://link.springer.com/10.1007/978-3-319-55696-3_5.

[168] R. Bartle, "Hearts, Clubs, Diamonds, Spades: Players who Suit MUDs", *Journal of MUD Research*, vol. 1, no. 1, p. 19, 1996. DOI: 10.1007/s00256-004-0875-6. [Online]. Available: http://mud.co.uk/richard/hcds.htm.

[169] M. Mateas, *Build It to Understand It: Ludology Meets Narratology in Game Design Space*, May 2005. [Online]. Available: http://summit.sfu.ca/item/254.

[170] M. J. P. Wolf, "Genre and the Video Game", in *The Medium of the Video Game*, 2002, ch. 6, ISBN: 978-0292791503. [Online]. Available: http://www.robinlionheart.com/gamedev/genres.xhtml.