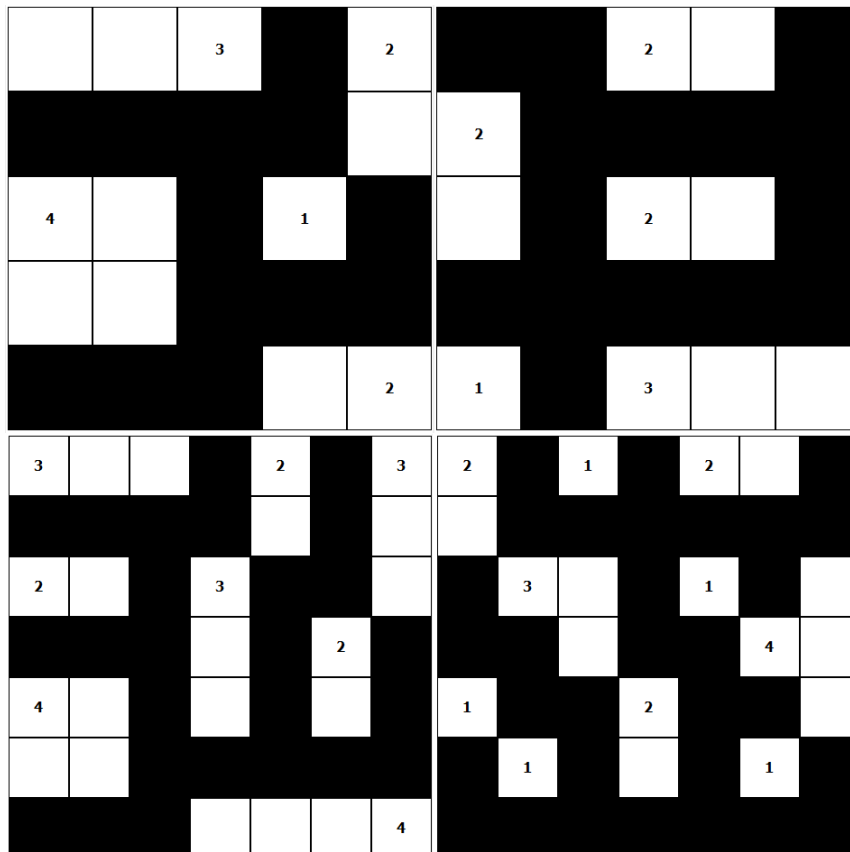


A.I. Solver for Games



Professor Martyn Amos

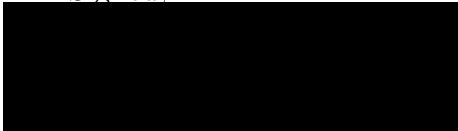
BSc Computer Science Final Report



Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work.

Signed,



Abstract

In biology, the process of natural selection promotes the evolution of the strongest individuals in a generation, and similarly in this project the Japanese puzzle game Nurikabe will be solved with an evolutionary approach. To achieve this an engine is implemented to harness the puzzle, and a genetic algorithm with multiple objectives will be executed on the puzzle boards. Considerable research into the area has revealed appropriate methodologies for completing this task, and a lot of testing on relevant parameter values has been done to achieve a genetic algorithm which can consistently solve small Nurikabe puzzles, and provide strong solutions for more complex puzzles.

Acknowledgements

I extend a huge thank you to my personal tutor Professor Martyn Amos for all of the help, kind words and guidance throughout my time working on this project. I would also like to thank Dr Nicholas Costen for facilitating the final year projects.

List of Figures

1.1	Example of a Nurikabe puzzle and its solution	2
2.1	NOT-gate with information flow from left to right	7
2.2	Example of a simple one-point chromosome crossover	8
2.3	Example of a mutation on the 5th bit	8
2.4	Crossover on stronger Lemmings script chromosomes	12
2.5	Mutation on Lemmings script chromosome offspring	12
2.6	An example of the structure of an artificial neural network, composed of many neurons that process information and return useful outputs .	14
2.7	The two types of structural mutation in NEAT. The top number in each genome is the <i>innovation number</i> of the gene, assigned to new genes with increasing value, which acts as a historical marker to identify the original ancestor of the gene	15
2.8	The progression of the artificial neural network MarI/O (Bling, 2015)	16
2.9	Features of Jenetics as specified on their website (Wilhelmstotter, 2007)	18
2.10	The component structure and life cycle of EO (Merele et al., 1998) .	19
3.1	Examples of random Nurikabe paths encoded as the ocean	22
3.2	Examples of random Nurikabe paths encoded as islands	23

3.3	Genotype structure & gene sum calculation (Wilhelmstotter, 2007)	25
3.4	Class hierarchy of genetic algorithm concept classes	26
3.5	Empty Nurikabe Board	27
3.6	The array representation of the empty Nurikabe board in Figure 3.5	27
3.7	The result of encoding chromosomes C1-C4 in the board from Figure 3.5	29
4.1	Puzzles of varying size occupying the same area	35
4.2	Generating a solution with knowledge of the board	36
4.3	5x5 and 7x7 puzzle solutions found by brute forcing	37
4.4	Finding the number of expected islands in a Nurikabe solution	39
4.5	Finding the number of black squares in a Nurikabe solution	39
4.6	Finding the number of black 2x2s in a Nurikabe solution	40
4.7	Finding the number of disconnected oceans in a Nurikabe solution	41
4.8	Finding the number of connected islands in a Nurikabe solution	42
4.9	Finding the number of satisfied islands in a Nurikabe solution	43
4.10	Generating one thousand solutions for a 7x7 Nurikabe puzzle	46
4.11	The main function for executing a single genetic algorithm on a Nurikabe puzzle	47
4.12	The function to configure the engine for the genetic algorithm	47
4.13	The function to evaluate the fitness of a solution in the genetic algorithm engine	48
4.14	The codec used to inform the engine of how to interpret Nurikabe solutions	48
4.15	The engine function to generate a new Nurikabe solution	49
4.16	The engine function to generate a new Nurikabe move	49
4.17	The results of an execution of the genetic algorithm with default parameters	50
4.18	The statistics of an execution of the genetic algorithm with default parameters	51
4.19	Testing results displaying best chromosomes using boltzmann selection	55

4.20	The best solutions for 12x12 and 9x9 Nurikabe puzzle boards found whilst testing rates of alteration	57
4.21	Ten solutions for a 15x15 puzzle using a very high generation limit and population size	59
4.22	400 solutions for a 9x9 puzzle using a low population size	60
4.23	400 solutions for a 9x9 puzzle using a higher population size	60
5.1	The consistent solution of 5x5 Nurikabe puzzles	62
5.2	A correct solution for a 7x7 Nurikabe puzzle	63
6.1	Solutions for a 9x9 Nurikabe puzzle which could be completed by a person with a working knowledge of Nurikabe	67

List of Tables

3.1	Pool of available genes with bit representation	24
3.2	Example of the structure of a Nurikabe chromosome	24
3.3	Example set of Nurikabe chromosomes	27
3.4	Chromosome C1 split into its individual moves	28
4.1	The final pool of moves available for solution generation	36
4.2	5x5 brute force	37
4.3	7x7 brute force	37
4.4	Results of genetic algorithm execution with various selectors on 7x7 puzzle	52
4.5	Results of genetic algorithm execution with various selectors on 9x9 puzzle	53
4.6	Results of genetic algorithm execution combining boltzmann with other selectors, to re-enforce the appropriate choice of selector	54
4.7	Searching rates of alteration on multiple runs of the genetic algorithm on a 9x9 Nurikabe puzzle	56
4.8	Searching rates of alteration on multiple runs of the genetic algorithm on a 12x12 Nurikabe puzzle	57

4.9	Ten runs on 15x15 Nurikabe puzzle with a population size of 1000 and generation limit of 400	59
4.10	Ten runs on 15x15 Nurikabe puzzle with a population size of 5000 and generation limit of 2000	59
5.1	A demonstration of the strength of the genetic algorithm on Nurikabe puzzles with varying dimensions	63

Contents

Abstract

Declaration

Acknowledgements

List of Figures **i**

List of Tables **iv**

1 Introduction **1**

1.1 Problem Statement and Proposed Solution 1

1.2 Aim and Objectives 2

1.2.1 Aim 2

1.2.2 Objectives 2

1.3 Background 3

1.3.1 Artificial Intelligence 3

1.3.2 Nurikabe 4

2 Literature Review **6**

2.1 Nurikabe NP-Complete 6

2.2	Genetic Algorithms	7
2.2.1	Multi-Objective Optimization	9
2.2.2	Genetic Algorithms in Automated Game Solving	10
2.3	Rule-Based Systems	12
2.3.1	Rule-Based Systems in Automated Game Solving	13
2.4	Neural Networks	13
2.4.1	Artificial Neural Networks in Automated Game Solving	15
2.5	Existing Work	17
2.5.1	Zen Puzzle Garden	17
2.5.2	Nurikabe	17
2.6	Implementation Libraries	18
2.6.1	Jenetics	18
2.6.2	ECJ	19
2.6.3	Evolving Objects (EO)	19
3	Product Design	20
3.1	Nurikabe	20
3.2	Implementation	21
3.3	Population Generation	21
3.3.1	Gene	23
3.3.2	Chromosome	24
3.3.3	Genotype	24
3.3.4	Phenotype	25
3.3.5	Population	25
3.3.6	Class Hierarchy Diagram	26
3.3.7	Encoding the Chromosomes	26
3.4	Fitness Function	29
3.5	Individual Selection	31
3.6	Alteration	32
3.6.1	Crossover	32

3.6.2	Mutation	32
3.7	Verification	33
4	Implementation	34
4.1	Nurikabe Engine	34
4.1.1	Generating Nurikabe Solutions	35
4.1.2	Brute Forcing Nurikabe Puzzles	36
4.1.3	Encoding Input	37
4.2	Fitness Function	38
4.2.1	The Number of Expected Islands	38
4.2.2	The Number of Black Squares	39
4.2.3	The Number of Black 2x2 Areas	40
4.2.4	The Number of Disconnected Oceans	40
4.2.5	The Number of Connected Islands	41
4.2.6	The Number of Complete, Satisfied Islands	42
4.2.7	Calculating the Fitness Value	43
4.3	Genetic Algorithm	46
4.3.1	Jenetics Library Implementation	46
4.3.2	Survivor and Offspring Selection Testing	51
4.3.3	Alteration Testing	55
4.3.4	Further Testing	58
5	Evaluation	62
5.1	Nurikabe Complexity	62
5.2	Testing Evaluation	64
5.2.1	Survivor and Offspring Selection Testing	64
5.2.2	Alteration Testing	64
5.3	Objective Achievement	65
6	Conclusions	66
6.1	Completed Work	66

6.2	Limitations	67
6.2.1	Tweaking Final Solutions	67
6.2.2	Execution Time	67
6.3	Final Conclusion	68
6.4	Future Work	68
6.5	Summary	68
7	Bibliography	69

Appendix A: Terms of Reference

Appendix B: Ethics Form

Appendix C: Nurikabe Engine User Guide

CHAPTER 1

Introduction

1.1 Problem Statement and Proposed Solution

Puzzle games have been played globally for hundreds of years, predating all modern technology, and they all share something in common. At their core every puzzle requires a player to find a certain configuration of pieces, be that numbers, shapes, colours or sounds to achieve a final, correct solution.

Whilst puzzle games are generally easily explained, their solutions can be quite complex and require a certain set of strategies to complete them. One example of a puzzle with a complicated set of constraints is Nurikabe, a binary determination puzzle which requires the user to place black squares on non-numbered squares to achieve a set of numbered 'islands' surrounded by a black 'ocean'.

Nurikabe is a difficult puzzle with a vast scope and many rules to follow and Holzer et al. (2004) has proven it to be NP-complete. The combination of these factors provides a strong platform for research and in my project I will use an aspect of A.I. (Artificial Intelligence) called genetic algorithms to implement an automated solver for Nurikabe puzzles.

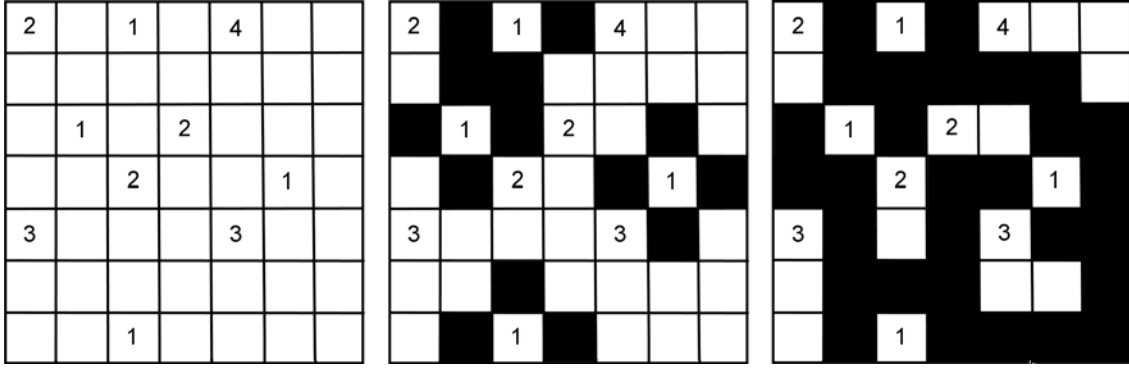


Figure 1.1: Example of a Nurikabe puzzle and its solution

1.2 Aim and Objectives

1.2.1 Aim

The aim of my project is to research and create an automated solver for Nurikabe, using multi-objective genetic algorithms which will be able to effectively adhere to the many constraints involved in solving the puzzle. I will examine existing solvers for Nurikabe as well as other similar puzzles, comparing the results of my solution with other automated solving techniques to draw a conclusion about the effectiveness of using multi-objective genetic algorithms for this purpose.

1.2.2 Objectives

To be successful in the project I will need to complete the following steps:

- Perform a literature review, reviewing other people's work in this area, particularly pre-existing solvers for Nurikabe and research regarding multi-objective genetic algorithms.
- Define a project plan, deciding which software and development languages to use to run the genetic algorithms. Design a Nurikabe engine, providing a platform to test the solver and analyze the results.
- Develop the Nurikabe engine and run extensive unit tests to ensure that it adheres to the strict constraints of the puzzle, this is important to avoid any

inconsistencies in the data that come from testing the solver as it could invalidate the tests.

- Implement the automated solver, recording the results of test runs on puzzles of varying complexity. Adjust key factors such as the difficulty of the puzzles and the size of the grids to acquire an expansive data-set that has multiple results for each puzzle variation, to eliminate any data anomalies.
- Evaluate the data gathered from testing, comparing how the different solvers perform at different levels of complexity. If the data is reliable and decisive draw a conclusion about the efficiency and viability of using genetic algorithms to solve NP-complete puzzle games such as Nurikabe, and in other applications as well.
- Reflect on the project and consider how successfully the aims and objectives were met.

1.3 Background

1.3.1 Artificial Intelligence

Humans have the tendency to use computers for tasks that we consider to be too difficult to complete in a reasonable amount of time, such as complex mathematics or payroll processing, because a computer can do these very easily. Previously we have designated more abstract tasks to humans and not machines, for example recognizing if a person is lying using cues such as their voice and facial movements, because these are considerably more difficult for a computer to automate.

Cawsey (1997) states that artificial intelligence is involved in the resolution of these tasks, because they require an advanced knowledge and refined reasoning processes. There are many reasons for attempting to automate human intelligence, for example, simulating aspects of our own behavior allows us to better understand the

implications of our own actions, and by studying human reasoning we are able to develop and improve techniques to enhance our ability to solve difficult problems.

AI is a broad field which overlaps with many different subject areas, ranging from mathematics to philosophy, and the applications of AI can be theorized by simply considering problematic areas that humans have to deal with on a day-to-day basis. A lot of these problems are areas where people excel, things that humans do without having to summon much if any thought, for example navigating around other people in a crowded or busy environment, or interpreting a sentence from somebody else and instantly deriving meaning from it. These problems are extremely difficult for a computer to effectively simulate because there are a seemingly infinite number of variables involved.

Other problem areas are more complicated, some tasks demand a specialized skill set, knowledge and training, and unlike the above these are usually referred to as 'expert' tasks. Whilst expert tasks are harder for an average human to understand, they can be easier for a computer to simulate given enough information from experts than instinctive actions such as accurately recognizing a person's face. The most appropriate AI technique to be applied to a given problem depends on the type of problem, the information available and the complexity of the solution.

1.3.2 Nurikabe

Nurikabe is a complex, NP-complete puzzle game and its constraints are going to be a main factor in the implementation of an automated A.I. solver. It is important for the solver to adhere to these constraints, otherwise the results of the solver will be invalid:

- Numbered squares can not be filled in.
- A number tells the number of continuous white cells. Each area of white cells contains only one number in it and they are separated by black cells, as determined by Nikoli (2006).

- The black cells must form a continuous path.
- There can not be a 2x2 area of black cells.

From these constraints we can draw conclusions about the nature of a completed puzzle:

- The number of expected white squares (including numbered squares) is equal to the sum of the values of the numbers on the board.
- The number of expected black squares (size of the path) is equal to the total area of the board minus the total number of expected white squares (For example, given a 5x5 Nurikabe puzzle with a numbered square sum of 9 we can assume that there will be 16 black squares in a given correct solution.)

NP-Completeness

An NP-Complete problem is one that is both NP and NP-Hard. The term NP comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines, as explained by Sipser (2006). Nondeterminism refers to a set of rules that present more than one action for a given problem, for example Nurikabe is nondeterministic, at a given point you may choose to continue the path in any available direction, resulting in different correct solutions.

CHAPTER 2

Literature Review

2.1 Nurikabe's NP-Completeness

Holzer et al. (2004) explain that a number of pencil puzzles (puzzles that are solved by drawing on some figure) are NP-complete, including Slitherlink (Takayuki, 2000), Number Place (aka. Sudoku) (Takayuki and Takahiro, 2003) and Pearl (Friedman, 2002). It can be seen that Nurikabe is contained within NP, it is obvious that a Turing machine has the ability to guess a black and white pattern and the validity of a solution can be verified within polynomial time.

To conclude that Nurikabe is NP-complete, all that remains is to prove hardness. Holzer et al. (2004) do this by reducing the Planar 3SAT problem to Nurikabe, as Planar 3SAT is already proved to be NP-complete by Lichtenstein (1982). By transforming the associated graph of a formula into a Boolean circuit using logic gates (see Figure 2.1), mapping the Boolean circuit to a Nurikabe board, it can be seen that there is a one-to-one correspondence between the Nurikabe puzzle and satisfying assignments of the Boolean circuit, proving it to be NP-complete.

It has been proved that all Nurikabe variants are also NP-complete. There are

4 variants of the puzzle, found by solving the puzzle with variations of two of the rules; black cells must be connected and black cells can not form 2 by 2 squares. By considering these variants and applying the same reasoning, Holzer et al. (2004) were able to prove this theorem.

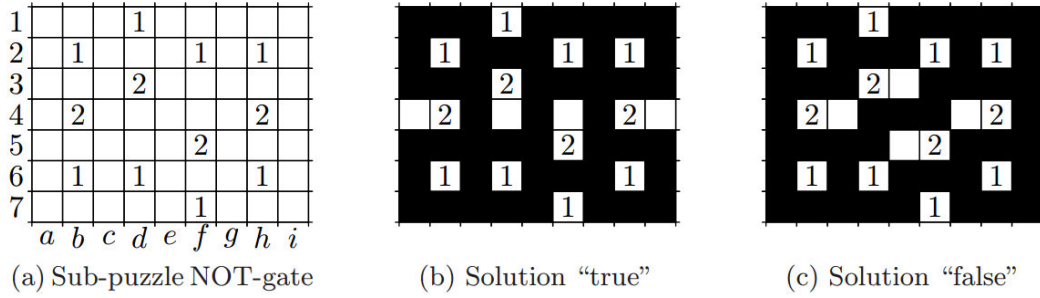


Figure 2.1: NOT-gate with information flow from left to right

2.2 Genetic Algorithms

Genetic algorithms (GA) apply principles of evolutionary biology, namely natural selection to problem solving, a biological process in which the strongest individuals in a given group are more likely to succeed, as stated by Man et al. (1996).

GA presumes that a potential solution for a problem is independent and is represented by a set of parameters. The parameters are ideally structured as a string of values in binary form, each depicting genes within a chromosome (also known as genome). A chromosome is given a "fitness" value, representing how close to a desired solution it is, determined using a "fitness" or "strength" heuristic.

The use of simulated evolution allows GA to search the solution space of a problem, applying the survival of the fittest strategy with the goal of improving the population of each new generation. Houck et al. (1995) discusses how it is possible for inferior individuals to survive a generation and reproduce by chance, reducing the elitism of the solutions. Promising chromosomes can be exploited using crossovers with other chromosomes, gene mutation and specific selection operations to solve both linear and non-linear problems.

Generally the execution of GA begins by generating a completely random pool

of chromosomes, evaluating each using a fitness heuristic. Strong members of the population will be crossbred at random (see Figure 2.2) until a set of new genomes is acquired, then a gene mutation is introduced (see Figure 2.3). This process is repeated until the fitness heuristic returns a value that satisfies the desired threshold.

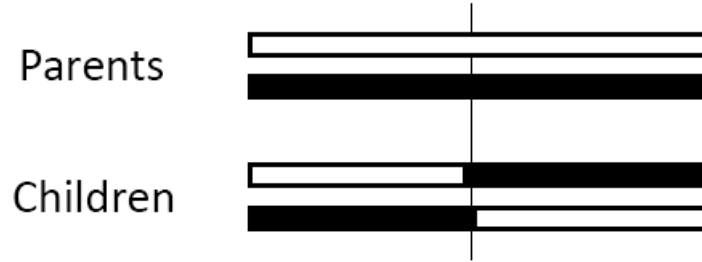


Figure 2.2: Example of a simple one-point chromosome crossover

Crossover simulates genetic recombination in human reproduction, and the ratio of the next generation's population born by a crossover operation is defined by the crossover rate. It is also possible to perform a multi-point crossover operation, splitting single chromosomes in multiple places.

0	1	0	1	1	0	1	1
↓							
0	1	0	1	0	0	1	1

Figure 2.3: Example of a mutation on the 5th bit

Similarly, mutation simulates gene mutation in human reproduction, creating genes that are untraceable to a parent to avoid chromosome convergence and elitism. Each gene from every chromosome is checked for gene mutation, using random number generation and a defined mutation rate e.g. 0.01 to determine if the value is going to be changed. These variables are altered depending on the requirements of the problem space, and the benefit of crossover can be lost if mutation is not enough to find a solution and the population begins to converge on similar chromosomes.

When selecting the strongest chromosomes for crossover there are a few options to consider regarding the expected fitness heuristic value. Some problems may be

solvable with a high level of elitism, which means selecting a small group of the strongest chromosomes for the next iteration, although this will lead to convergence in a lot of cases. Another technique is steady state selection, in which the weakest members of the population are cut out at each selection phase, which takes longer than employing elitism but could more reliably generate a solution. Generally, genetic algorithms use fitness proportion selection which is a roulette wheel type technique, encouraging a Darwinian survival of the fittest process.

There are multiple types of mutation, some are more effective than others however they may all be more successful in different problem spaces;

- Displacement Mutation

Take a random section of genes from a chromosome and insert into a random location in another.

- Insertion Mutation

Take a single gene from a chromosome and insert into a random location in another.

- Inversion Mutation

Take a random section of genes from a chromosome and reverse it.

- Inversion Displacement Mutation

Take a random section of genes from a chromosome, reverse it and insert into a random location in another.

2.2.1 Multi-Objective Optimization

Some problem areas (including Nurikabe) have multiple objectives to consider in their solution, Deb et al. (2002) explains that these give rise to a set of optimal solutions, widely known as Pareto-optimal solutions. None of these solutions can be assumed to be better than the others, so the user must find as many solutions as possible. This sort of genetic algorithm attempts to satisfy multiple, potentially

conflicting objectives simultaneously, as determined by Alfaris (2010), trading off optimization in one objective to ensure that the algorithm performs well with each of the other objectives.

2.2.2 Genetic Algorithms in Automated Game Solving

The 1991 puzzle-platformer game 'Lemmings' has been referred to as a 'Drosophila' of A.I. by McCarthy (1998), meaning that similarly to the common fruit fly it serves as an accessible, experimental platform that can be used to gain knowledge about other, potentially more complex problems. In each level of the game the user is presented with a map full of obstacles, a lemming spawn point and a door, and the goal is to help the lemmings navigate the map and reach the door without dying. A game of Lemmings ends when the map is clear of all lemmings, so it may be necessary to 'blow up' any remaining, stuck lemmings. If the user saves a certain percentage of the lemmings, they have completed the level.

To achieve this, the user has the ability to assign one of 8 special types to the lemmings, depending on the requirements of the map (basher, blocker, bomber etc.,) allowing the lemmings to perform actions such as building sets of stairs, or falling from a height using an umbrella as a parachute. Kendall and Spoerer (2004) implemented a genetic algorithm solution for Lemmings, and it can be seen that the complexity of the game demands various design considerations when finalizing the format of the chromosomes and fitness heuristics.

One limitation to consider is that the game's maps are typically larger than the screen, so knowledge of the game state at any given time is partial. Also, the move-speed of lemmings is only an approximation, and the time available for each map is strictly limited so the automated solver has to work quickly. McCarthy (1998) observes that human players develop strategies before the game is played, suggesting that a planned approach is more appropriate than a reactive approach, so an evolutionary path to a solution is fit for this problem.

The chromosomes in this case are scripts, which define the actions of the lem-

lings through a controller. The lemmings are unaware of the environment around them as well as other lemmings, they only listen to the controller. If the controller receives a move from a chromosome which can not be completed, or targets a lemming that does not exist or is already dead, it will simply be ignored. The chromosomes are rated using a complicated fitness function, after being tested on the same test map, and this takes the form $(\alpha + \beta + \delta + \gamma)/10$. Please see equations (2.1, 2.2, 2.3, 2.4) for the definitions of these variables.

$$\alpha = (numStartingLemmings * T) * numLemmingsSaved * 10 \quad (2.1)$$

$$\beta = (numStartingLemmings * timeRemaining) * 10 \quad (2.2)$$

α rewards each lemming that reaches the exit door. If the map is unsolved, β is set to 0, otherwise it provides a bonus if the map is solved faster. $\alpha + \beta$ rewards success although it doesn't reward any lemmings that don't reach the exit door.

$$\delta = \sum_i (lemming[i].life) \quad (2.3)$$

δ rewards lemmings that are active in the map for longer amounts of time, summing the total life of all of the lemmings. The index i represents each lemming that entered the map, and a lemming's life is incremented for each time step that it remains active. If a single lemming reaches the exit door α will always evaluate to more than δ when all lemmings stay on the map for the length of the game, without any making it to the exit.

$$\gamma = \sum_{x,y} (explored[x][y]) * 10 \quad (2.4)$$

γ is set to reward greater map exploration, especially horizontal platform exploration as that is typically an indicator of a 'good' chromosome. See 2.5 for the definition of $explored[x][y]$.

$$explored[x][y] = \begin{cases} 0 & \text{if no lemmings entered } cell[x][y] \text{ during the game} \\ 1 & \text{if any lemming entered } cell[x][y] \text{ during the game} \\ 2 & \text{if any lemming entered } cell[x][y] \text{ during the game} \\ & \text{and } cell[x][y] \text{ is also a horizontal platform} \end{cases} \quad (2.5)$$

Once the chromosomes have been evaluated they are selected for crossover and mutation. McCarthy (1998) uses one point crossover on the chromosome scripts (see Figure 2.4), so two stronger chromosomes are subject to crossover to create two scripts of offspring. The offspring are mutated using flip mutation (see Figure 2.5), and this is repeated until a chromosome is evolved to the point of completing the Lemmings map successfully.

$$\begin{aligned} ScriptP_1 &= (1, 2) (4, 1) \parallel (2, 5) (2, 1) (3, 2) \\ ScriptP_2 &= (7, 1) (5, 2) \parallel (8, 0) (1, 2) (4, 1) \\ ScriptC_1 &= (1, 2) (4, 1) (8, 0) (1, 2) (4, 1) \\ ScriptC_2 &= (7, 1) (5, 2) (2, 5) (2, 1) (3, 2) \end{aligned}$$

Figure 2.4: Crossover on stronger Lemmings script chromosomes

$$\begin{aligned} ScriptC_1(\text{pre-mut}) &= (1, 2) (4, 1) \mathbf{(8,0)} (1, 2) (4, 1) \\ ScriptC_1(\text{post-mut}) &= (1, 2) (4, 1) \mathbf{(6,1)} (1, 2) (4, 1) \end{aligned}$$

Figure 2.5: Mutation on Lemmings script chromosome offspring

2.3 Rule-Based Systems

Also known as knowledge-based systems, rule-based systems are software or hardware that store knowledge and use it to make inferences. The knowledge is provided by human experts, books, data mining etc., and whilst a general program runs algorithms against data to provide a result, rule-based systems run an inference engine and the knowledge base against data instead. There may also be domain knowledge

available, containing specific knowledge about the domain, and temporary knowledge from the current operation. An inference engine is a program that activates knowledge in a knowledge base, providing answers, suggestions and predictions, much like a human expert.

There are two types of inference, forward and backward chaining. Forward chaining begins at the start of the rule set and draws new conclusions from each rule, reverting back to the first available satisfied condition in each cycle, repeating until the goal has been achieved. Conversely, backward chaining begins with the goal and chains backwards through rules, moving through a set of hypotheses until one has been proved.

2.3.1 Rule-Based Systems in Automated Game Solving

Rule-based systems have been applied to the automated solution of puzzle games, for example, Merrit (2016) implemented a rule-based system using Prolog for the Rubik's Cube. The rules for the system were derived from an expert source (Black and Taylor, 1980), and it can be seen that a rule-based system is an appropriate solution for this puzzle because of the monumental number of possible cube manipulations, only a few of which leading to a solved puzzle. It is necessary to use sets of heuristic (intelligent) rules for the solution of a Rubik's Cube, as a blind search strategy which selects rules at random would not be successful, because the knowledge base drawn from the expert source is too vast.

2.4 Neural Networks

An artificial neural network (ANN) is a set of interconnected nodes, which is similar to the huge network of the approximate 10^{11} neurons in a human brain, designed to process information. ANNs are used in many fields outside of computer science, such as mathematics, engineering, physics and even psychology, and like people, ANNs learn by example (Stergiou and Siganos, 1996). Animals use their nervous

systems to react adaptively to external and internal environmental changes, and ANNs aspire to mimic this behavior using models and simulations of the nervous system to find solutions to complex problems.

Neurons work by processing information, receiving and providing this information via spikes. ANNs link many of these neurons together according to a specified network architecture, and the objective is to turn sets of inputs into informative, usable outputs, learning by adaptation (see Figure 2.6.)

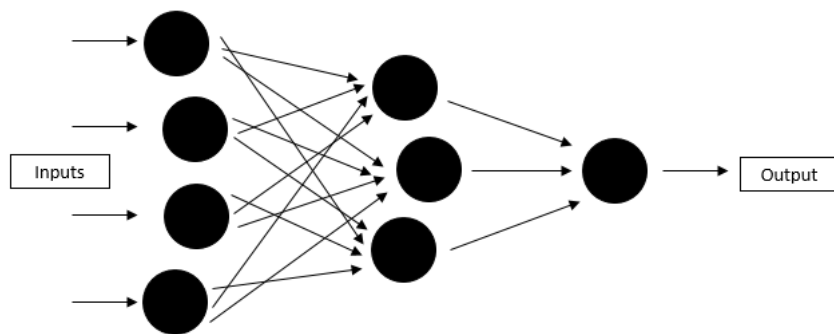


Figure 2.6: An example of the structure of an artificial neural network, composed of many neurons that process information and return useful outputs

As a basic example, an animal may learn that a certain type of tree isn't fit for climbing, as its branches are dangerously thin. This is information that will be processed by the animal's nervous system, and it could influence changes in the animal's choice of route for finding food, which could in turn improve its chances of survival. Without processing the information and learning by adapting to it, the animal would not learn that the trees are not fit for purpose, and that could be detrimental.

The same logic can be applied to problems targeted by ANNs, for example, teaching a robot to navigate an obstacle course. The robot will inevitably find problems, such as falling over a rock, and the result of the problems can be considered information to be processed by the ANN, hopefully improving the level of success of the robot's next attempts.

The adaptive learning of ANNs can be perceived as optimization, as new information improves the success of every aspect of a solution, even if it wasn't necessarily

'bad' in the first place. ANNs can be considered as linear combinations of nonlinear basis functions, in which the parameters are found by applying the optimization methods, with respect to an approximation error measure. The error measure takes into account the areas in the solution that are not desired, much like a fitness heuristic in genetic algorithms.

2.4.1 Artificial Neural Networks in Automated Game Solving

The classic game 'Super Mario' has been used as a platform for developing artificial neural networks, specifically using a method known as NEAT (NeuroEvolution of Augmenting Technologies), created by Stanley and Miikkulainen (2002). Neuroevolution is the artificial evolution of neural networks, using genetic algorithms. The chromosomes in the genetic algorithm in this case are randomly generated neural networks, each allowed a certain amount of time to attempt the Super Mario level, and are rated by a fitness heuristic as usual. The genes represent neural network nodes and there are two types of structural mutation in NEAT, the first is adding nodes, and the second is adding connections between nodes (see Figure 2.7.)

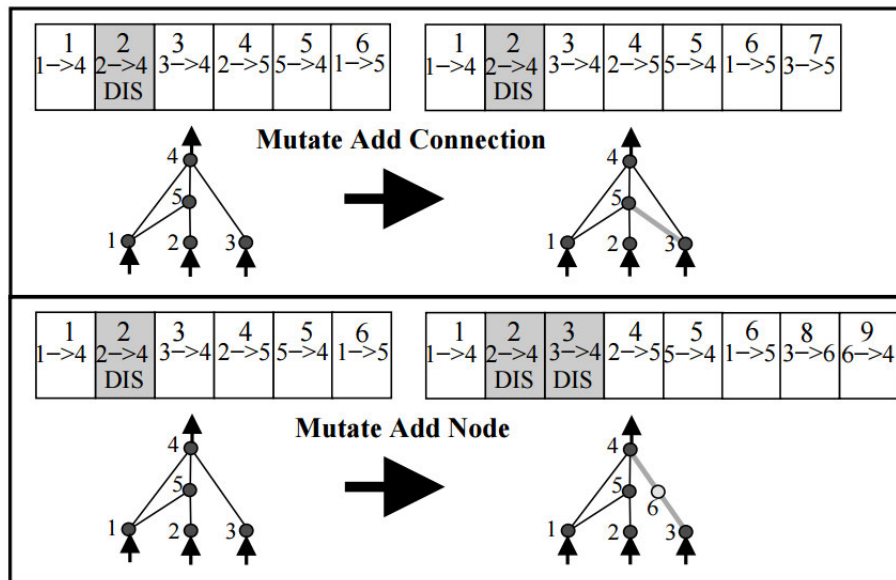


Figure 2.7: The two types of structural mutation in NEAT. The top number in each genome is the *innovation number* of the gene, assigned to new genes with increasing value, which acts as a historical marker to identify the original ancestor of the gene

The project, 'MarI/O: Machine Learning for Video Games' was created by Bling (2015): YouTube introduction at <https://www.youtube.com/watch?v=qv6UVOQ0F44>, with source code available at <http://pastebin.com/ZZmSNaHX>.

The ANN was allowed a 24 hour 'learning' period, in which it began attempting the level with no knowledge at all about what to expect, or how to do anything. The only available information at the start is that it is only successful if it reaches the end of the level, but it isn't aware of what the different buttons will do, or of any of the failure conditions within the game.

It can be seen from a demonstration of this learning period that at the start, the ANN simply presses the available buttons at random, using the results of the button presses as information to adapt during the next run (see Figure 2.8.) If the ANN is idle making no progress for too long, it will simply terminate and the next chromosome will be used. Over time the genetic algorithm selects stronger ANN chromosomes and performs crossover and mutation to create new populations, repeating these steps until an ANN chromosome reaches the end of the level, completing the intentions of the project.

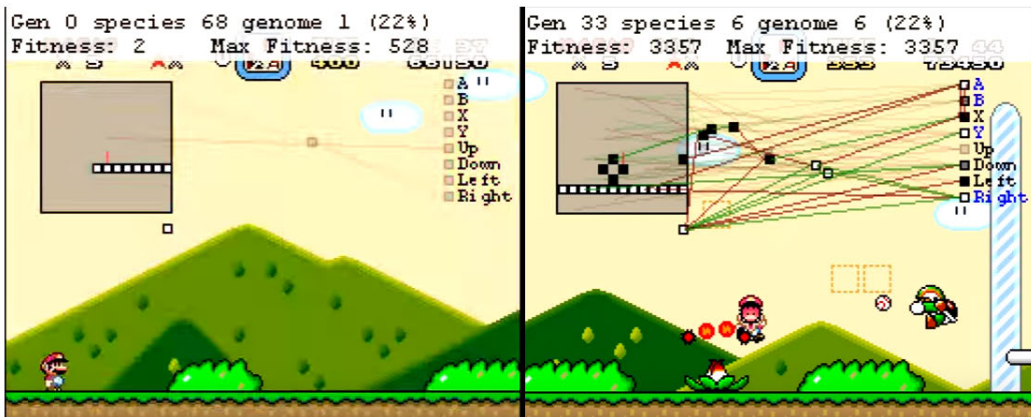


Figure 2.8: The progression of the artificial neural network MarI/O (Bling, 2015)

2.5 Existing Automated Puzzle Game Solvers

2.5.1 Zen Puzzle Garden

A genetic algorithm was used in the solution of the puzzle game Zen Puzzle Garden (ZPG). The starting board in a ZPG puzzle is a grid representing a garden, and the objective is to move a monk around the garden, raking each grid on the square, navigating obstructions on the way. The genetic algorithm was implemented in Java, using a specialized evolution library.

It can be seen that Amos and Coldridge (2012) have represented a move on the board as a gene for the genetic algorithm, and a chromosome is comprised of a number of genes, as well as a starting location. In order to evaluate the success of an individual solution, the path script is encoded into a ZPG board, and the results are analyzed to provide a resulting fitness value based on the success of the board characteristics.

The genetic algorithm used in this solution is successful in solving ZPG puzzles, and it outperforms other instances of automated ZPG solvers. This solution could also have implications in mobile robotics, where a self-avoiding path must be chosen, weighing path lengths against hazards, similar to the monk in a ZPG puzzle.

2.5.2 Nurikabe

Automated solvers for Nurikabe have been implemented in the past, for example, it can be seen that Huttar and Kathy (2005) use a rule-based deterministic approach, attempting to 'solve' each square of a Nurikabe board to determine if it should be black or white. A list of the constraints of a Nurikabe puzzle is defined and a large number of rules are applied to the squares, checking for any contradictions. A contradiction is a violation of any of the constraints, the most common contradiction being an attempt to mark a square black once it had already been marked white.

To progress a solution it can be seen that a hypothesis-testing mechanism is used, in the case that the solver has no definite idea of what cell to color black or

white, a duplicate of the board is taken and used as a 'trial' board. The solver can continue on the trial board and if a contradiction is hit, then the solver knows that this particular hypothesis is false. It should be noted that Huttar and Kathy (2005) discuss how it is difficult to know how far to take the new hypothesis, as it can only be proven true if taken to a solution to the puzzle, however between two hypotheses, if one is false then the other is assumed to be true.

The simpler rules to verify are the first to be applied in order to approve the efficiency of the solver, and this process is repeated until each square on the board has been validated. At this point the entire board is checked again and if no rules are fired, either a solution has been found or the solver has failed to solve the puzzle, although this doesn't necessarily mean that the puzzle is unsolvable. One of the boards tested is of size 24x19, and it is yet to be solved by this solver, as Nurikabe solutions become extremely complex on larger sized boards.

2.6 Genetic Algorithm Implementation Libraries

2.6.1 Jenetics

Jenetics is an advanced Genetic Algorithm, respectively an Evolutionary Algorithm, library written in modern day Java (Wilhelmstotter, 2007). It is designed with clear separation between the core concepts of GA, namely Gene, Chromosome, Genotype, Phenotype, Population and Fitness Function. Jenetics stands out as it uses the concept of an evolution stream for executing the steps, using the Java stream interface within the Java stream API. See Figure 2.9 for a list of Jenetics' features.



Figure 2.9: Features of Jenetics as specified on their website (Wilhelmstotter, 2007)

2.6.2 ECJ

ECJ is an evolutionary computation research system, also written in Java. Designed to be very flexible, with all classes and settings dynamically determined at run-time by the user's defined parameter file, all structures in the system are arranged to be easily modifiable.

2.6.3 Evolving Objects (EO)

EO is a template-based, evolutionary computation library written in ANSI-C++ (Merelo et al., 1998). Designing GA in EO involves selecting specific components that are required for the problem space and building the structure around them. If there is existing code available for a problem, the components can be taken to form an algorithm and then connected to the fitness function. Even for more difficult problems, most of the required operators are readily available in EO.

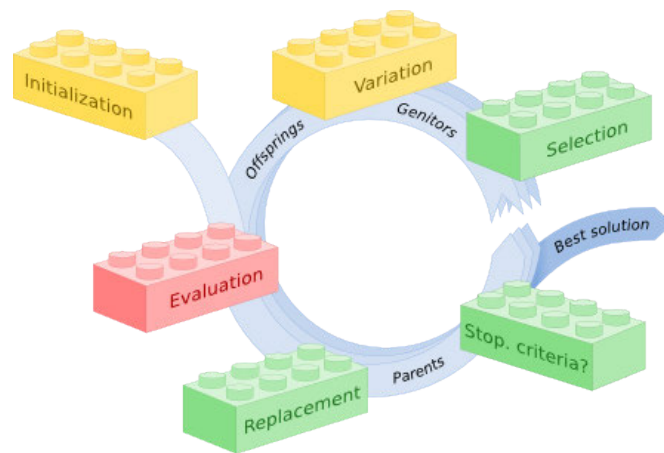


Figure 2.10: The component structure and life cycle of EO (Merelo et al., 1998)

CHAPTER 3

Product Design

3.1 Nurikabe

In order to design a genetic algorithm for solving Nurikabe it will be important to highlight the rules of the game, the expected state of a correct solution, and the aspects of a solution that make it invalid.

Nurikabe can be played on boards of varying size, and the size of the board does not matter for the genetic algorithm, however it will be important to make sure enough moves are generated to create a path long enough to form a potential solution.

A complete, correct solution to Nurikabe will conform to these laws, and the execution of the genetic algorithm will terminate when all laws are satisfied:

- The black (a.k.a "ocean") areas should form one completely connected "wall".
- Black cells should never form a 2x2 area.
- Every area of white cells (a.k.a "island") must contain only one number, therefore multiple numbers should never be connected by white squares.

- The number of white cells in a single island must be equal to the value of the number.

Using these laws it is possible to come to conclusions regarding the nature of a completed solution;

1. The total number of white squares must equal the sum of all of the numbers on the board.
2. Therefore, the number of black squares in a complete solution should equal the area of the board, minus the total number of white squares.

3.2 Implementation

The genetic algorithm implementation library *Jenetics* will be used for this project, there is a lot of clear, concise documentation available for the development of a genetic algorithm using *Jenetics* and it is an appropriate library for a Nurikabe solver because it includes a clear separation of the required concepts, specifically Gene, Chromosome, Genotype, Phenotype, Population and Fitness Function.

The code for the Nurikabe solver will be written in Java using the Eclipse IDE, this is a language that I am familiar with and it is also suitable for the library as *Jenetics* is written in modern day Java.

3.3 Population Generation

It is necessary to define the evolutionary aspects of Nurikabe as distinct definitions, to be compatible with the library's methods and to highlight how I can represent a Nurikabe solution as a binary string.

To begin the implementation of a genetic algorithm for solving Nurikabe, a population of chromosomes must be initially generated. These chromosomes will define a set of 'moves' to be made on a Nurikabe board relative to a given starting square, to create a potential solution.

I decided to implement a prototype Nurikabe solution generator in Java, to be used as a testing ground for generating chromosomes which can be interpreted as attempted solutions and mapped to an empty Nurikabe puzzle. I added a GUI to the generator to display the generated chromosomes on various Nurikabe boards, although the attributes of a board do not change any aspect of the chromosomes as the evolution is to be done by the genetic algorithm.

Initially the program was coded to create a string of moves at random (from a move pool of up, down, left and right), then starting from a random empty square on a Nurikabe board the referenced square would move, changing each square hit to black. The only constraints were that it could not change the color of numbered squares, or go through the boundaries of the board, so moves were ignored accordingly.

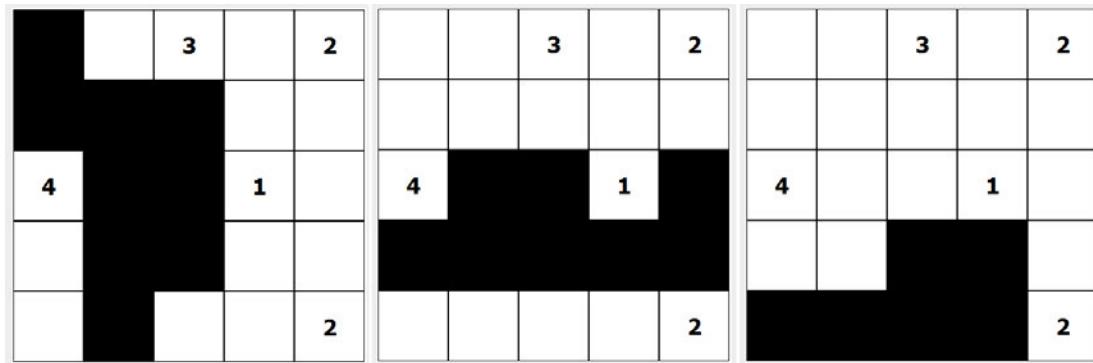


Figure 3.1: Examples of random Nurikabe paths encoded as the ocean

It can be seen from Figure 3.1 that when the paths are mapped to create oceans (adding black squares), they look nothing like realistic attempts at a Nurikabe solution. There is a considerable difference in each generated path, and they all break the rules of the game, so this was not the correct way to encode the paths to the puzzle.

After further experimentation it became clear that a more appropriate way to encode the paths would be required. Using numbers as reference squares, starting with all empty squares changed to black, the generated moves are interpreted as instructions to change a square from black to white, until each individual number on the board has been satisfied. To satisfy a number, its island must contain a

number of white squares equal to the number's value. The paths encoded in this way are considerably more realistic, and the variance between each path isn't as great, displayed in Figure 3.2.

It is also worth noting that the generated paths used in Figures 3.1 and 3.2 are of equal lengths. This shows how encoding the paths as islands is also more computationally efficient than encoding the paths as oceans.

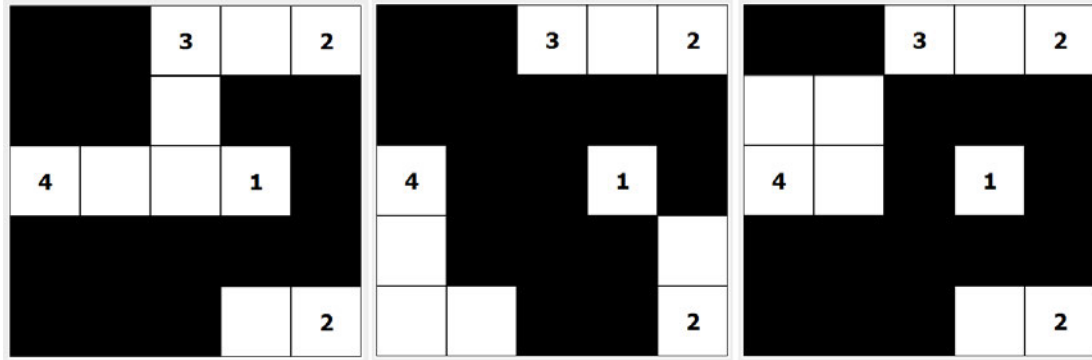


Figure 3.2: Examples of random Nurikabe paths encoded as islands

3.3.1 Gene

A single gene will be used to represent a 'move' on a Nurikabe board, relative to a starting position. The first moves to consider are the directional keys, however it could be seen from early experiments in mapping generated solutions to a board that this is not sufficient. Using only up, down, left and right, the generated paths tended to be grouped up around the starting point, as the constraints in hitting walls and numbers restricted moves, and the path rarely moved far from the starting square.

To improve the variance of the paths, a new type of move was added, which acts as a marker to tell the algorithm that the next move will not result in changing a square's state in any way. This allowed the paths to go backward on themselves, and this change allowed the program to generate paths that look more like trees, with smaller branches of squares drawn without creating a loop. This move is known as an **N** move.

Finally, after seeing how the **N** move affected the newly generated paths, a few

extra instructions called repeats were added to the generator and tested. These repeat instructions prepared the algorithm to repeat the last move that it made, and the idea behind this was to encourage the paths to be even more diverse, and travel further on the board from the given starting square.

These moves need to be represented in bit form to be suitable for the genetic algorithm, and at the end of testing there is a set of eight moves to consider.

Table 3.1: Pool of available genes with bit representation

Up	Down	Left	Right	N	Repeat (R1)	Repeat (R2)	Repeat (R3)
000	001	010	011	100	101	110	111

3.3.2 Chromosome

In biology a chromosome is an organized structure containing the DNA of a living organism, and similarly, in the implementation of an automatic solver for Nurikabe, a chromosome will contain a string of genes (the DNA of this problem space), representing moves.

The genes will be executed sequentially, and in the Jenetics library the interface also contains a variable to track the number of genes in a given chromosome (length). There is also no restriction on the length of a chromosome in Jenetics, which is a desirable feature for solving Nurikabe as it is impossible to predict or calculate the number of moves (genes) required in a correct solution. Please see this example of the structure of a chromosome, displaying how its genes translate to Nurikabe moves:

Table 3.2: Example of the structure of a Nurikabe chromosome

Genes	001	011	111	100	111	101	011	101	001	110	010
Moves	Down	Right	R3	N	R3	R1	Right	R1	Down	R2	Left

3.3.3 Genotype

The genotype is the structural, immutable representative of an individual, consisting of one to n chromosomes. The chromosomes of a genotype do not necessarily have

to be of the same length, although all genes must be the same type and have the same constraints; e.g. the same minimum and maximum values for numerical genes. The genotype also contains a method to calculate the total number of genes in all chromosomes within the genotype. See Figure 3.3 for an example of a genotype's structure, containing chromosomes of varying length.

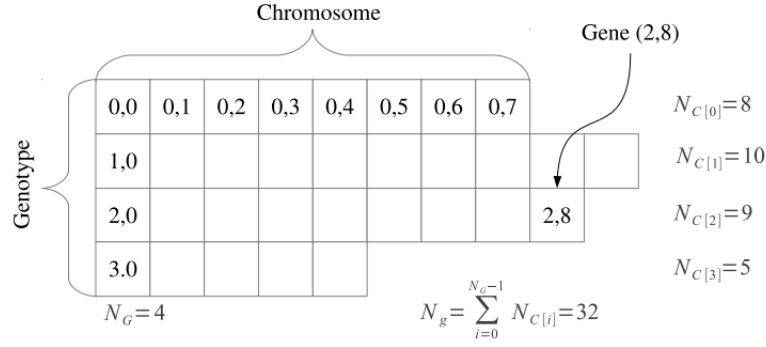


Figure 3.3: Genotype structure & gene sum calculation (Wilhelmstotter, 2007)

In solving Nurikabe, however, only one chromosome is required for each genotype. A single string of genes will be sufficient in creating a solution to be reviewed by a fitness function. The genotype is the central class of the library, and it is mainly this class that the evolution engine works with.

3.3.4 Phenotype

In the Jenetics library, a phenotype is an immutable type consisting of a genotype paired with a fitness function, where the fitness function represents the environment in which the genotype lives. The phenotypes maintain a natural order, defined by their fitness values (given by the fitness function), using the Java Comparable interface.

3.3.5 Population

At the end of the class hierarchy of the Jenetics library is the population, a collection of phenotypes, acting as the start and end point of an evolution step. It is from the population that individuals are selected for crossover and mutation, and it includes

methods to sort the population by fitness value, to highlight the phenotype with the highest fitness at the end of each step.

3.3.6 Class Hierarchy Diagram

Please see Figure 3.4 for a diagram of the class hierarchy of the genetic algorithm concept classes as defined in Jenetics.

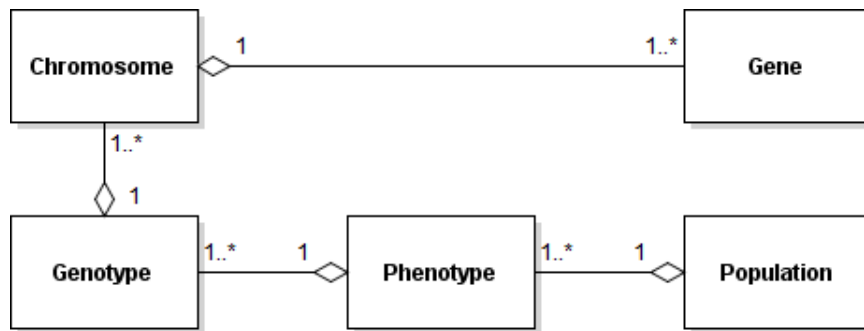


Figure 3.4: Class hierarchy of genetic algorithm concept classes

3.3.7 Encoding the Chromosomes

Once a population of individuals has been generated, the fitness function will evaluate each chromosome, based on how 'good' of a solution it is to a given Nurikabe puzzle. In order to perform this evaluation, the chromosomes must first be encoded into the puzzle, just as a person would fill their answers in with a pen.

A simple way to computationally represent a Nurikabe puzzle is to use a 2-dimensional array. White squares will be represented as zeros, numbered squares as their value and black squares as the number -1. See Figure 3.6 for an example of the array representation of an un-solved Nurikabe puzzle (Figure 3.5).

		3		2
4			1	
				2

Figure 3.5: Empty Nurikabe Board

{ 0 0 3 0 2 }

{ 0 0 0 0 0 }

{ 4 0 0 1 0 }

{ 0 0 0 0 0 }

{ 0 0 0 0 2 }

Figure 3.6: The array representation of the empty Nurikabe board in Figure 3.5

To encode each chromosome, a start position must be identified and the chromosome separated into individual genes. In this case, the genes represent moves, or rather the direction to attempt to mark the next square as white. The start position will always be the first number on the board, starting from the top left square. During the design of the generator, the start position was also generated at random and represented in binary form as a part of the chromosome, however this was removed after the decision to encode the path relative to the islands was made. As previously decided, the first step in encoding a path will be to mark each non-numbered square black, because the paths will be mapped relative to the islands, not the ocean.

First, I will generate a small population of chromosomes to be used as examples for the design of the genetic algorithm. These chromosomes only contain between 15 and 20 steps, which will not be sufficient for forming solutions to large puzzles, but because they are generated independently of any information, they are usable on a puzzle of any size.

Table 3.3: Example set of Nurikabe chromosomes

C1 (12)	000010011100001010100011001011100000
C2 (20)	100010001001101011001000011100011010100100100010101101001011
C3 (15)	010011000010010101010100100000001000000101011
C4 (17)	001010101100001000011011000001010011011100011000000

A chromosome will be split into a set of genes, and the genes will be sequentially iterated over to encode each move into a new Nurikabe solution. Take chromosome **C1** as an example:

Table 3.4: Chromosome C1 split into its individual moves

010	010	010	001	100	011	010	010	011	011	101	010
Left	Left	Left	Down	R3	R1	Right	R1	Down	R2	Left	Left

It is important that every chromosome is encoded in the same way every time a population is generated, because if they aren't, the fitness function will be 'rating' inconsistent solutions, and it is unlikely that the genetic algorithm will function properly. Following a set of rules, the path will be mapped to the board:

1. If the move is an N move, set a flag preparing the next move to result in no change to square state.
2. If the move is one of the Repeat moves, set the move to equal the previous move.
3. Check the next square in the direction of the move defined by the gene, if it is a wall or a number, ignore the move and restart with the next gene. If it is already a white square, move **only** if the N move flag is set and mark it as the new reference square.
4. If the next square is black, change it to white and mark it as the new reference square.
5. If the current island has been satisfied, move to the next number on the board and mark it as the new reference square.
6. Repeat until either all genes have been dealt with, or all numbers have been satisfied.

Using these rules, encoding chromosomes **C1-C4** into the 5x5 Nurikabe puzzle from Figure 3.5 results in the solutions displayed in Figure 3.7. These solutions are all incorrect, however they are appropriate individuals to be used for selection in the genetic algorithm.

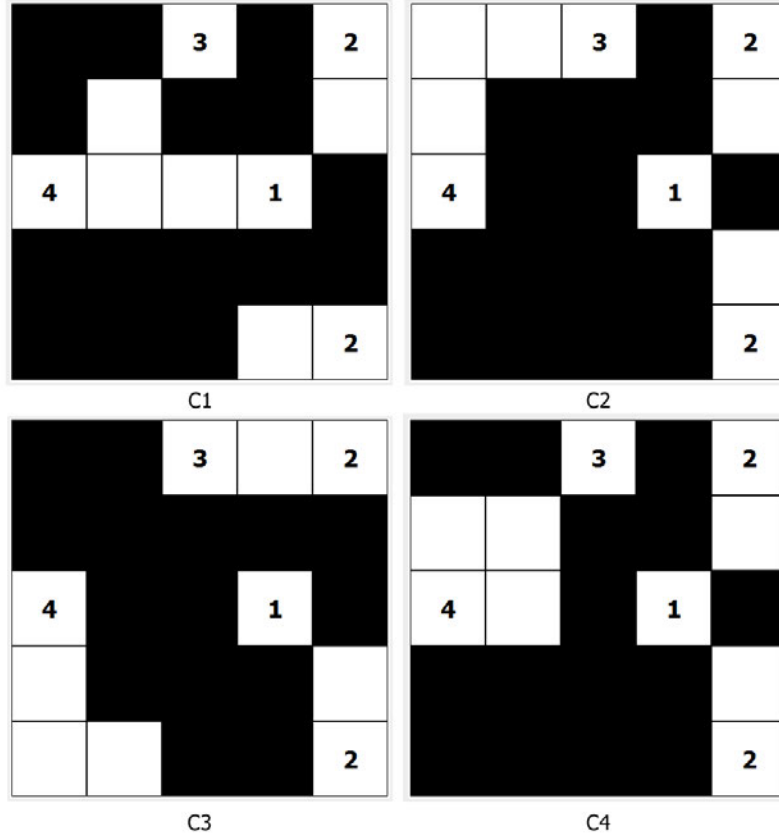


Figure 3.7: The result of encoding chromosomes C1-C4 in the board from Figure 3.5

3.4 Fitness Function

After a population has been generated, each individual is assigned a fitness value given by a fitness function. This fitness function must take into account the attributes of a Nurikabe solution which make it incorrect, and create a function to turn the evaluation of the attributes into a representative value. The value can be used to tell the genetic algorithm how 'good' a solution is, and that will be important when selecting individuals for alteration at the end of an evolutionary stage.

There are multiple objectives to consider in rating a Nurikabe solution:

\mathbf{v} = The total number of 2x2 areas of black squares.

\mathbf{w} = The total number of disconnected ocean areas.

\mathbf{x} = The total number of connected islands.

y = The total number of satisfied numbers.

z = The total number of satisfied islands (disconnected islands with satisfied numbers.)

The fitness function must be sufficiently fast to compute, as it will be called many times by the genetic algorithm on large populations of chromosomes. This means that it must be simple whilst maintaining a reliable resulting number to represent how fit each individual is, relative to the other individuals in the population.

As Nurikabe is a complex problem space with multiple objectives, the fitness function must weigh the values against each other, because an improvement in one area may diminish the value in another. For example, if a chromosome that produces a solution with 9 satisfied islands is altered to produce a chromosome with 10 satisfied islands, it may result in an increase of the number of connected islands. It is not necessarily a more fit solution, and that must be reflected in the fitness value.

Using the above parameters, taking into account the conflicting areas of Nurikabe, a fitness function for Nurikabe chromosomes can be defined as follows:

$$\alpha = 1 - \frac{x}{y} \quad (3.1)$$

Alpha is used to track the number of satisfied numbers against connected islands, and it will converge to 1 when the number of connected islands reaches 0, which is the ideal scenario for a complete solution.

$$\beta = \frac{1}{w} \times v \quad (3.2)$$

Beta is concerned with the solution's fitness in regard to the ocean, rewarding solutions that contain less 2x2 areas of black squares, and have fewer disconnected sections of ocean squares. A correct Nurikabe solution will have only one connected ocean, and a lower value in beta indicates a more desirable solution.

$$\sigma = \alpha \times \beta \quad (3.3)$$

Sigma rewards solutions that have higher values in both the number of satisfied islands, and the fitness of the ocean. Sigma promotes the tendency to reward solutions that have a decent level of fitness in each of the multiple-objectives.

$$\Gamma = 1 - (z \times \sigma) \quad (3.4)$$

Gamma will give the final fitness value for an individual, taking into account all of the objectives that define the fitness of a single Nurikabe solution. In the case that sigma is equal to zero, the value of \mathbf{z} (number of disconnected, satisfied islands) is irrelevant, as it can be assumed to be a correct solution.

The fitness function as designed may be required to be adjusted during the implementation, as it is difficult at this stage to judge how successful it will be. As Nurikabe is a problem space with multiple objectives, the balance of each individual part of the fitness function is very important, and it could heavily impact the assigned fitness value given to each solution.

3.5 Individual Selection

After the entire population has been rated using the fitness function, selectors are responsible for selecting a certain number of individuals, dividing the population into *survivors* and *offspring*. In the Jenetics library, the selection process acts on phenotypes, and indirectly acts on genotypes via the fitness function.

There are a number of available selector implementations, such as the Tournament selector, Boltzmann selector, Probability selector and Roulette Wheel selector, and these are all supported by the Jenetics library. The appropriate selector to be used in this problem space will become evident during the implementation of the genetic algorithm.

Different selectors can be tried, even in combination with each other, in reaction to any problems that are discovered when the algorithm is selecting individuals. It will also be necessary to test different selectors in the case that one is more effi-

cient than another, especially when running the genetic algorithm on large Nurikabe puzzles.

3.6 Alteration

Alterers determine how the search space can be traversed, encouraging genetic diversity among selected individuals.

3.6.1 Crossover

Crossover is used to combine parts of two desirable parent chromosomes, to create an offspring chromosome to be part of the next population. It will be necessary to experiment with the recombination probability, $P(r)$, which determines the probability that a given individual will be selected for crossover.

The appropriate type of crossover to be performed on the chromosomes in the genetic algorithm will be decided during the implementation, as testing will be required to see if single-point crossover is more appropriate for Nurikabe chromosomes than multi-point crossover.

Other types of crossover include Uniform crossover, in which bits are copied at random from each parent, and Arithmetic crossover, using arithmetic operators such as AND, OR or XOR to create the offspring. These types of crossover will also be tested when attempting to improve the efficiency of the genetic algorithm.

3.6.2 Mutation

The second alteration type used in Genetics is mutation, which alters one or more of a chromosome's genes to encourage genetic diversity. Mutation helps to prevent elitism, which is the act of selecting chromosomes that are all strong in the same areas, and this can cause the genetic algorithm to slow down or even cease to execute before reaching a correct solution.

The mutation probability, $P(m)$, is a parameter to be optimized which defines

the probability that a gene within a given population is mutated. Finding an optimal value for $P(m)$ will be a crucial step in the implementation, to preserve the genetic diversity required without altering the chromosomes too much. This can cause too many desirable genes to be lost and solutions could too often become worse after the mutation, negating the original intention.

3.7 Verification

This process of population generation, fitness evaluation, selection and alteration is repeated, until an individual is found with a fitness value of 1, indicating that the evolutionary stage is complete and a correct solution has been found. Using the results from each execution of the algorithm, alterations can be made to variables such as the crossover and mutation probabilities, as well as the selection methods to increase the success of the genetic algorithm.

CHAPTER 4

Implementation

4.1 Nurikabe Engine

To facilitate the execution of a genetic algorithm in Nurikabe, I implemented an engine using Java, developed in the Eclipse IDE. A Nurikabe board is stored as an array, with the number 0 representing white squares, and numbers in place of islands. The boards are displayed in a GUI, allowing the user to visualize the results of attempts at a solution.

Using iteration, each element of the array is mapped to a board with the same number of squares as dimensions in the array, allowing a flexible puzzle size without having to alter the size of the display. This means that any Nurikabe puzzle can be quickly imported into the engine to be used for testing, without code refactoring. Please see Figure 4.1 for an example of how boards of different size are encoded to occupy the same space.

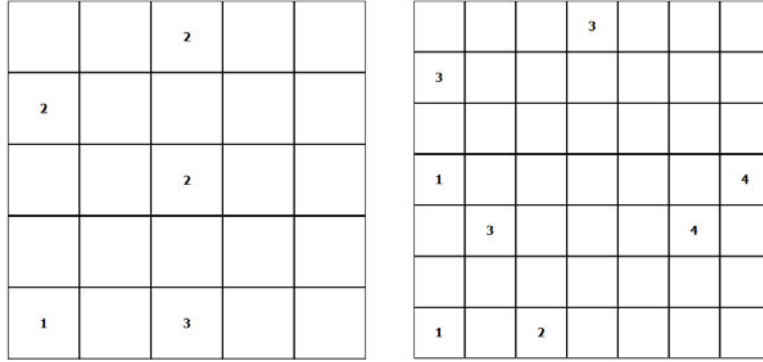


Figure 4.1: Puzzles of varying size occupying the same area

4.1.1 Generating Nurikabe Solutions

The original idea for generating Nurikabe solutions was to simply specify a desired path length (representing the number of moves), generate that number of random moves, and encode the path as the ocean of the puzzle, until the expected number of black squares had been mapped. Any invalid moves were simply ignored, and as discussed in the product design, encoding the path as the puzzle's ocean was ineffective, so the decision to encode the path relative to the islands was made.

Whilst refactoring the code in this area, I realized that the solution generation could be streamlined to vastly improve the performance of the engine. Instead of generating a fixed set of random moves, I re-wrote the code to generate the solution with knowledge of the Nurikabe board, and moves are randomly selected from a pool of the available moves at any given square. This preserved the randomness of the solutions, whilst ensuring that each solution is exactly as long as it needs to be, and every move is now used in the encoding.

Figure 4.2 displays an example of an unfinished Nurikabe puzzle solution, to highlight how each move is generated. Whilst the square to the right of the number 4 is under consideration, the available moves are evaluated, and in this case the available moves are up, down, or left. One move from the pool is selected at random, encoded to the board, and this process is repeated until an entire solution has been generated and each island is satisfied.

After generating a number of puzzles with the updated code, it became clear

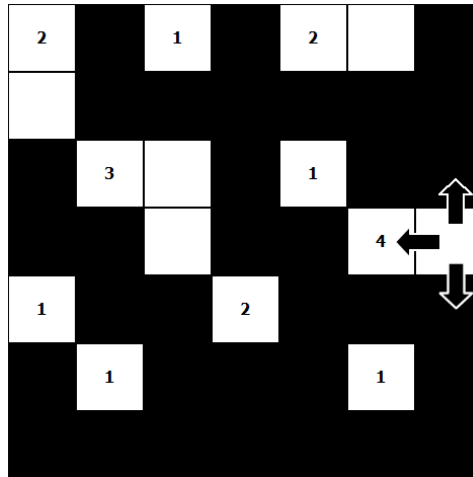


Figure 4.2: Generating a solution with knowledge of the board

that the repeat bits were no longer necessary, as the importance of branching the path out is considerably lower for the islands than it was for the oceans. A lot of code refactoring made it possible for the N move to be removed as well, and the encoding of the solutions was updated to reflect this. If a path attempts to move into a number, or back on to itself, it is allowed but makes no change to the state of the square. The final move pool, including each move's binary representation can be seen in Table 4.1.

Table 4.1: The final pool of moves available for solution generation

Up	Down	Left	Right
000	001	010	011

4.1.2 Brute Forcing Nurikabe Puzzles

To test the representation of solutions, I wrote an algorithm to attempt to brute force a Nurikabe puzzle, by generating random solutions and checking their validity, until a correct solution is found. This also served as a demonstration of how an increase in puzzle complexity affects the time of execution, which can be seen in Figure 4.3. When a correct solution is found, it is encoded into the puzzle, and displayed to the user.

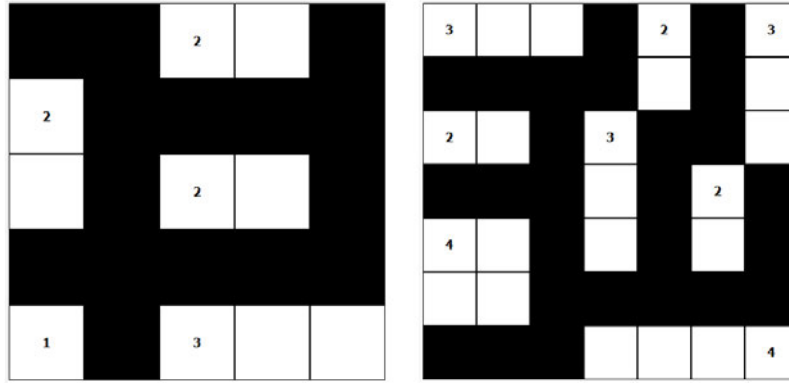


Figure 4.3: 5x5 and 7x7 puzzle solutions found by brute forcing

Table 4.2: 5x5 brute force

Execution	Solution	Time (ms)
1	49	38
2	390	48
3	380	40
4	91	26
5	103	30

Table 4.3: 7x7 brute force

Execution	Solution	Time (ms)
1	3888	200
2	959	61
3	702	53
4	8155	280
5	11498	387

The results in Tables 4.2 and 4.3 highlight the variance in execution time on boards with only slightly differing complexity. In fact, when executing the brute forcing algorithm on another 7x7 board with a more complicated configuration of islands, it tried billions of solutions without ever finding the correct one.

Whilst it is difficult to write the code to verify a solution, it is straightforward to confirm if a solution is correct using the human eye, which reinforces the accuracy of the validation done by the engine. Writing the code to validate a correct solution provided a basis for the fitness function, which allocates a value to each solution, representing how close to a correct solution it is.

4.1.3 Encoding Input

It is possible within the engine to input a user-defined solution, in order to make alterations to a path, or to view the results of encoding a chromosome from testing.

This functionality is intended to make testing the genetic algorithm more straightforward and user friendly.

4.2 Fitness Function

Nurikabe is a complicated problem space with multiple objectives, which can each impact the overall success of a solution. The fitness function is fundamentally important to the execution of the genetic algorithm, therefore it is important to thoroughly plan, code and test each section.

The representative value of each of the multiple objectives of Nurikabe is defined by an integer, and the value is found by exploring the Nurikabe board after a solution has been encoded into it, searching for different characteristics depending on the nature of the objective.

The code for the following functions can be found in the Java file 'NurikabeSolution', within the source files for the program, and the execution of each function will be visualized in this report. A 7x7 Nurikabe puzzle will be used to provide a working example of the calculation of the fitness value.

4.2.1 The Number of Expected Islands

This value corresponds to the number of individual numbers on a given Nurikabe board, which can be found by iterating over each square, and checking if it contains a number or not. This can be seen in Figure 4.4, each square is evaluated from left to right, and if it is found to be a number the counter is incremented.

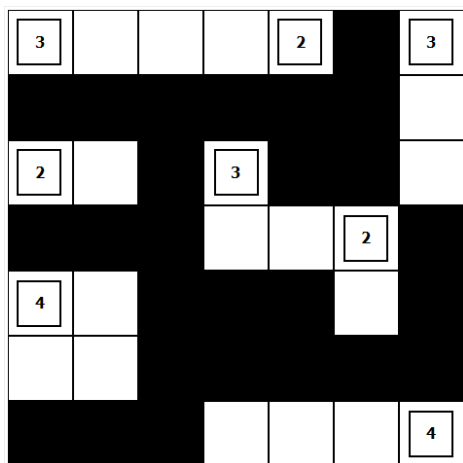


Figure 4.4: Finding the number of expected islands in a Nurikabe solution

As there are 8 numbers on the Nurikabe board, we can expect 8 islands in a correct solution.

This value allows a human eye to confirm that this solution is incorrect, as there are only 6 islands on the board, however a computer is unable to draw the same conclusion in a simple manner.

4.2.2 The Number of Black Squares

This value represents the number of black squares found within a given Nurikabe solution. It is calculated simply by iterating through each square on the board, incrementing a counter if it is found to be a black square, which can be seen in Figure 4.5.

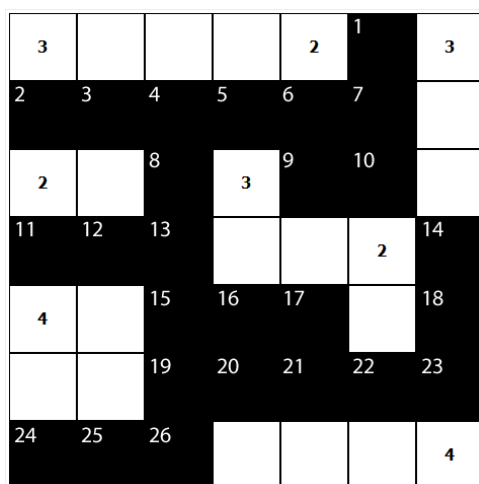


Figure 4.5: Finding the number of black squares in a Nurikabe solution

This solution contains 26 black squares, which doesn't directly tell us much about the strength of the solution, however it is required to inform other methods in the fitness function.

Preceding some code refactoring in solution generation, this value may have been useful, however all complete solutions that are generated are guaranteed to have the same (correct) number of black 2x2s on the board.

4.2.3 The Number of Black 2x2 Areas

As the black ocean path must be continuous, span the entire puzzle board without disconnecting, and maintain a width of 1, a correct Nurikabe solution will have a value of 0 for this objective.

This value is found by exploring the Nurikabe solution, iterating through the squares from left to right, treating each square as a reference at the top left of a 2x2 area. If the entire area is black, including the reference square, then a counter is incremented as displayed in Figure 4.6, and the next square is selected. The iteration considers the first ($n - 1$) squares of each row and column, with n representing the width or length of the puzzle, to avoid null pointer exceptions.

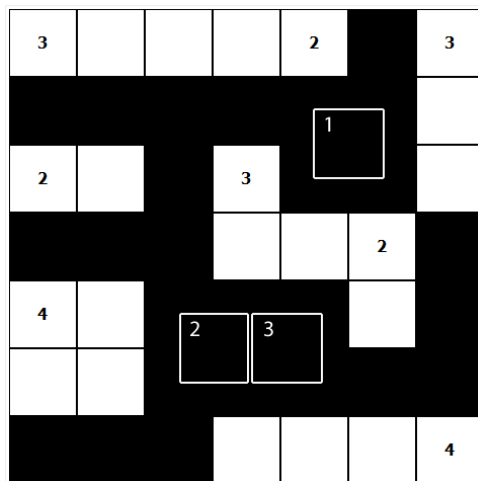


Figure 4.6: Finding the number of black 2x2s in a Nurikabe solution

There are 3 black 2x2 areas in this solution, another value which indicates that this solution is incorrect, as there can be no black 2x2 areas in a correct solution.

It is important to calculate this value, as it does highlight that this solution is likely stronger than one that is found to have a higher value in the same objective.

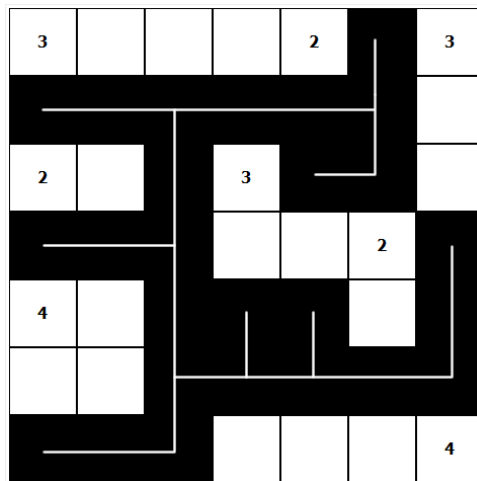
4.2.4 The Number of Disconnected Oceans

As previously discussed, the black ocean path must be continuous, without disconnecting. In a complete Nurikabe solution, this objective will have a value of 1, which is the single black path which spans the puzzle board.

The calculation of this value is achieved by evaluating the position of each of the black squares on the board. A function has been written to iterate over the black squares on the board, and explore the area of black squares on which it falls, which

can be seen in Figure 4.7. After each black square of the island has been identified, it is marked as having been 'dealt with' in a boolean array with the same dimensions as the puzzle board.

After an area of black squares has been fully explored, and each square identified, the counter for the number of disconnected oceans is incremented, and the iterator continues to search for black squares that haven't been marked as 'dealt with' in the boolean array. This process is repeated until the iterator explores the entire solution without finding a black square that hasn't already marked, it then terminates and the value of the objective is set to equal the counter.



In this particular solution there is only one disconnected ocean area, which is a desired characteristic of a correct solution, although this does highlight that a resulting value of 1 does not necessarily mean that the solution is correct.

Figure 4.7: Finding the number of disconnected oceans in a Nurikabe solution

4.2.5 The Number of Connected Islands

In a correct Nurikabe solution, each island must be disconnected, therefore each island may only house one number. This means that the desired value for this objective is 0.

Finding this value is done in a similar way to finding the number of disconnected oceans, with some differences. In this case, the function iterates over each white square on the board, including squares containing numbers. The island on which the white square falls is explored, and if it contains only more than one number, the counter tracking the number of connected islands is incremented by the total

number of numbers on the island. A visualization of this is shown in Figure 4.8. As with the previous function, the state of each square is maintained in a boolean marker array, so each square is only checked once.

After each white or numbered square in the board has been dealt with, the function terminates and the value of the connected island objective is set to equal the counter.

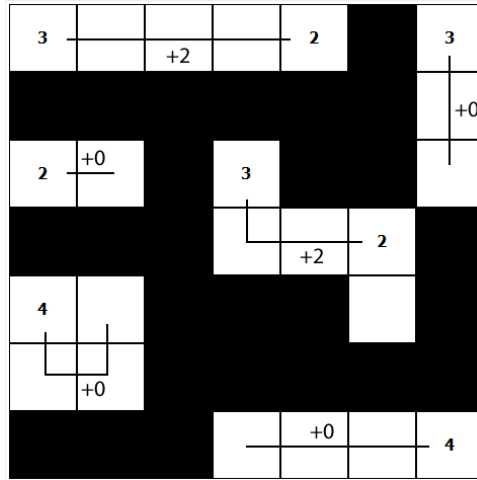


Figure 4.8: Finding the number of connected islands in a Nurikabe solution

The execution of this method results in a value of 4 for this solution, as there are two islands both containing two numbers.

This is another value that confirms the solution is incorrect, as the desired outcome is a value of 0, however it is useful in weighing the strength of this solution against others with different values in this objective.

4.2.6 The Number of Complete, Satisfied Islands

The number of satisfied islands represents the number of islands on the board that have both the correct number of white squares, and are disconnected from all other islands (containing only one number.) The expected value for this objective is equal to the number of expected islands, which is a value found previously to executing this function.

In order to find the value of this objective, a similar process is followed to the previous two functions. An iterator evaluates each white square on the board, and evaluates the island that it falls on. If the island contains only one number, and the number of white squares within the island is equal to that number's value, the counter for the number of satisfied islands is incremented by one. This means that

any island containing more than one number or any island with an incorrect sizing is not counted as satisfied, as seen in Figure 4.9.

After each square has been evaluated once, it is marked as true at a corresponding location in a boolean marker array. If the iterator finds a white square, it will first check to see if it has already been dealt with before attempting to explore its island. When all squares on the board have been evaluated, the function terminates and the value of the objective is set to equal the counter.

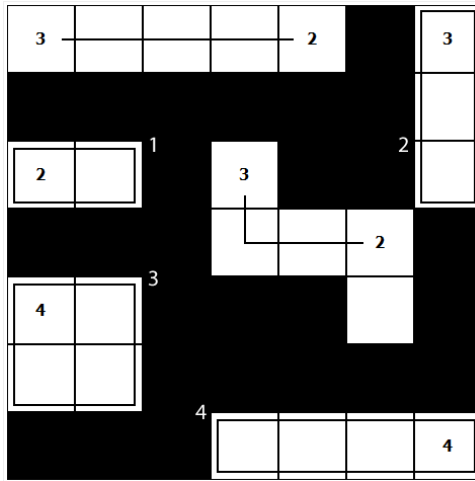


Figure 4.9: Finding the number of satisfied islands in a Nurikabe solution

After exploring the solution it can be seen that there are only four islands which are both satisfied and disconnected. Four of the numbers fall on islands that are invalid, so they are considered non-satisfied.

Again, this value confirms that the solution is incorrect, however it is one of the more important indicators of a solution's strength, and this will be a useful value in calculating its overall fitness.

4.2.7 Calculating the Fitness Value

After finding the value of each of the desired objectives, these values can be used to calculate a double value which represents the overall strength of a given Nurikabe solution.

To reach this final version of the fitness function involved a lot of trial and error. Millions of solutions were generated, and the resulting values were observed so that the function could be refined to provide a more representative value to each solution.

The required variables are set to equal the value of the different objectives:

$\text{numBlack2x2s} = 3$ (The total number of 2x2 areas of black squares.)

$\text{numNumbers} = 8$ (The total number of numbers on the board.)

$\text{numDisconnectedOceans} = 1$ (The number of disconnected ocean areas.)

$\text{numSatisfiedIslands} = 4$ (The total number of satisfied islands.)

$$\alpha = \frac{\text{numSatisfiedIslands}}{\text{numNumbers} * 100} \quad (4.1)$$

$$\beta = 1 - \frac{\text{numBlack2x2s}}{\text{numBlack2x2s} + 50} \quad (4.2)$$

$$\sigma = \frac{1}{\text{numDisconnectedOceans}} \quad (4.3)$$

$$\gamma = (\alpha * \beta) * \sigma \quad (4.4)$$

The fitness function can be applied to the working example, using the resulting values found from each method, representing the multiple objectives of Nurikabe. The values are substituted into the relevant equations as follows:

$$\alpha = \frac{4}{8} * 100 = 50 \quad (4.5)$$

The value of alpha represents a weighting between the number of satisfied islands, and the number of expected satisfied islands in a correct solution, which is equal to the number of numbers on the board. The ideal value for alpha is 100, as this means that the number of satisfied islands equals the expected number.

$$\beta = 1 - \frac{2}{2 + 50} = \frac{25}{26} \quad (4.6)$$

Beta is used to allow the presence of black 2x2 areas to negatively impact the overall fitness score. Ideally beta will have a value of 1, which would confirm that

there are no black 2x2 areas in the solution.

$$\sigma = \frac{1}{1} = 1 \quad (4.7)$$

Sigma simply provides a value representing the strength of the solution in regard to the number of disconnected oceans on the board. As a correct solution may only have one disconnected ocean, a value higher than that will drastically affect the final fitness value.

$$\gamma = (\alpha * \beta) * \sigma = (0.005 * \frac{25}{26}) * 1 = 48.07692307... = 48.077 \quad (4.8)$$

Gamma is the final value which represents the overall fitness of a solution, by weighing each of the multiple objectives against each other.

It can be seen that the final fitness value for the worked solution is 48.077%. This result seems appropriate given the strength of the solution, as whilst there is only one disconnected ocean area which is a desired characteristic of a correct solution, there are only half as many satisfied islands as expected islands, and two black 2x2 areas. The contrast between the values in different objectives places this solution at a middling strength, and it is obvious that there is a lot of room for improvement, which is reflected in the fitness value.

With the operation of the fitness function implemented into the engine, it is possible to generate as many solutions as desired for a specific Nurikabe board, and each solution's fitness will be calculated with the objective values displayed in a table. This can be seen in Figure 4.10, in which 1000 solutions are generated, the table is populated and sorted, and the board from each solution can be displayed in the GUI by selecting any row from the table. In this case, the best solution generated has a higher fitness than the worked example in the same puzzle board.

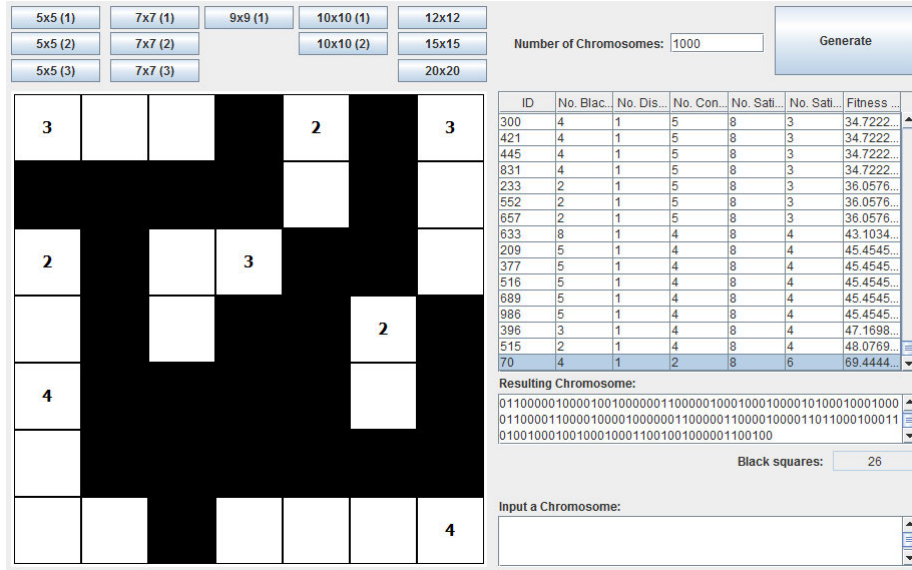


Figure 4.10: Generating one thousand solutions for a 7x7 Nurikabe puzzle

4.3 Genetic Algorithm

The fitness function contains a majority of the execution of the genetic algorithm, which has been implemented using the Jenetics library. A single run of the genetic algorithm requires a few different parameters, the output of the function is the best chromosome from the final population, and the Jenetics library provides a statistics container so various attributes of the run can be evaluated.

4.3.1 Jenetics Library Implementation

The code for the execution of a single run of the genetic algorithm can be seen in Figure 4.11. The required parameters are the population size, generation limit, the crossover rate and the mutation rate. The default values for these parameters are 1000, 400, 0.15 and 0.35 respectively, and the values can be changed easily within the GUI.

```

//Run the GA given a Nurikabe board, population size, generation limit, crossover rate and mutation rate
public void executeGeneticAlgorithm(int[][] nurikabeBoard, int popSize, int genLimit, double crossoverRate, double mutationRate) {
    //Create evolution statistics container
    final Consumer<? super EvolutionResult<AnyGene<String>, Double>> statistics = EvolutionStatistics.ofNumber();

    //Configure the engine
    final Engine<AnyGene<String>, Double> nurikabeEngine = generateEngine(popSize, crossoverRate, mutationRate);

    //Execute the GA engine
    final Phenotype<AnyGene<String>, Double> nurikabeResult = nurikabeEngine.stream()
        //Stop the evolution stream after N "steady" (similar rated fitness) generations
        .limit(limit.bySteadyFitness(genLimit))
        .peek(statistics)
        //Reduce the evolution stream to its best phenotype
        .collect(EvolutionResult.toBestPhenotype());

    //Display the result
    System.out.println(statistics);
    //These strings are formatted: [[0101010101010]]
    String winningChromosome = nurikabeResult.getGenotype().toString();
    //Remove the surrounding brackets
    String choppedChromosome = winningChromosome.substring(2, winningChromosome.length() - 2);
    //Encode the winning chromosome into the board for evaluation
    encodeInputChromosome(choppedChromosome);
}

```

Figure 4.11: The main function for executing a single genetic algorithm on a Nurikabe puzzle

To record the results of the execution of the genetic algorithm a statistics container is declared. The genetic algorithm engine is then configured, which I separated into its own function as seen in Figure 4.12.

```

//Build GA Engine given population size, crossover rate and mutation rate
public Engine<AnyGene<String>, Double> generateEngine(int popSize, double crossoverRate, double mutationRate) {
    //Changing the selectors and alterers here will change them for all GA runs
    Engine<AnyGene<String>, Double> nurikabeEngine = Engine
        .builder(
            NurikabeGeneticAlgorithm::eval, nurikabeCODEC(nextNurikabeMove()))
        .populationSize(popSize)
        .optimize(Optimize.MAXIMUM)
        .survivorsSelector(new TournamentSelector<>())
        .offspringSelector(new RouletteWheelSelector<>())
        .alterers(
            new SinglePointCrossover<>(crossoverRate),
            new Mutator<AnyGene<String>, Double>(mutationRate)
        )
        .build();

    return nurikabeEngine;
}

```

Figure 4.12: The function to configure the engine for the genetic algorithm

A custom class has been created named *nurikabeSolution*, to store all of the attributes of a single solution, including all of the values for each of the multiple objectives required by the fitness function, and all of the parameters which define the board itself. This class also contains all of the functions required to find the values of the objectives, so that the fitness function used by the genetic algorithm engine can be considerably more usable and readable, as seen in Figure 4.13.

```

//Calculate the fitness of the solution
private static Double eval(final String nurikabeSolution) {
    Double fitness = 0.0;

    //Prepare a solution to evaluate its success
    NurikabeSolution solutionContainer = new NurikabeSolution();

    solutionContainer.chromosomeString = nurikabeSolution;
    solutionContainer.finalSolution = composeSolution(currentNurikabe, solutionContainer.chromosomeString);
    //Set the number of black 2x2s
    solutionContainer.getNumBlack2x2s();
    //Set the number of black squares
    solutionContainer.getNumBlackSquares();
    //Set the number of disconnected ocean areas
    solutionContainer.getDisconnectedOceans();
    //Set the number of connected islands
    solutionContainer.getConnectedIslands();
    //Set the number of satisfied numbers
    solutionContainer.getSatisfiedNumbers();
    //Set the number of satisfied islands
    solutionContainer.getSatisfiedIslands();
    //Set the fitness value
    solutionContainer.getFitnessValue();

    fitness = solutionContainer.fitnessValue;

    return fitness;
}

```

Figure 4.13: The function to evaluate the fitness of a solution in the genetic algorithm engine

Within the engine builder, the fitness function has been referenced (*eval*), and a codec has been included (*nurikabeCODEC*) as seen in Figure 4.14, to inform the engine about what a single Nurikabe gene is, and how to generate a complete Nurikabe chromosome. This is important, as the engine must be able to create new instances of a gene in order to properly alter the chromosomes in a population.

```

//Define the codec for an abstract chromosome for Nurikabe
static Codec<String, AnyGene<String>> nurikabeCODEC(String nextNurikabeMove) {
    return Codec.of(
        Genotype.of(AnyChromosome.of(NurikabeGeneticAlgorithm::nextSolution)),
        gt -> gt.getChromosome().getGene().getAllele()
    );
}

```

Figure 4.14: The codec used to inform the engine of how to interpret Nurikabe solutions

The codec defines a genotype, using the container named *AnyChromosome* provided by *Jenetics*, which calls the function *nextSolution* to generate a new chromosome. *AnyChromosome* has been selected for use as the format of a Nurikabe chromosome is unsuitable for the more basic chromosome container types available in *Jenetics*, such as the *BitChromosome* or *NumericChromosome* classes. As *AnyChromosome* is a more abstract chromosome class, it is necessary to provide

the engine the functions to generate new genes and chromosomes in a way that is specific to Nurikabe, using the function *nextSolution* as seen in Figure 4.15.

```
//Get the next random path
private static String nextSolution() {
    //Generate a new path
    final AnyGene<String> nurikabeGene = AnyGene.of(NurikabeGeneticAlgorithm::nextNurikabeMove);
    String newSolution = generateChromosome(currentNurikabe, nurikabeGene);

    return newSolution;
}
```

Figure 4.15: The engine function to generate a new Nurikabe solution

The function *nextSolution* declares an *AnyGene* variable, used to represent a single Nurikabe move, and because the *AnyGene* class is abstract as well it is necessary to provide a function to generate individual genes, named *nextNurikabeMove*. This function simply returns a random Nurikabe move from the full move pool, which is the necessary way to inform the engine on what a Nurikabe move is, as seen in Figure 4.16.

```
//Get the next random move
private static String nextNurikabeMove() {
    final Random random = RandomRegistry.getRandom();
    int rnd = random.nextInt(5);
    String nextMove = "";

    switch (rnd) {
        case 0: nextMove = bitUp;
                break;
        case 1: nextMove = bitDown;
                break;
        case 2: nextMove = bitLeft;
                break;
        case 3: nextMove = bitRight;
                break;
        case 4: nextMove = bitN;
                break;
    }

    return nextMove;
}
```

Figure 4.16: The engine function to generate a new Nurikabe move

Once the engine has been configured, the genetic algorithm is ready to be executed. In this implementation using the Jenetics library, the phenotype is the container for the best individual from the entire run of the algorithm. In biology, the phenotype is the set of observable characteristics of an individual after its interaction with the environment, and in this context it represents the characteristics of the strongest Nurikabe solution after many generations of evolution in populations of chromosomes with a gradual improvement in various objectives.

The engine is streamed, and there are many available options to configure the attributes of the stream. It is possible to limit the execution by a specific number of generations, however it seems more appropriate to limit the execution when the gradient of success reaches a plateau. This is achieved by limiting the stream by a specified generation limit, which will terminate after the fitness of solutions plateaus for that number of generations.

The strongest solution that is found by the end of the execution is encoded into the puzzle board on the GUI, and the table is populated with its attributes, so it is easy to evaluate the results. The statistics of the run are also printed to the console, providing information on how the fitness of the chromosomes improved over time. An example of the GUI output at the end of the run can be seen in Figure 4.17.

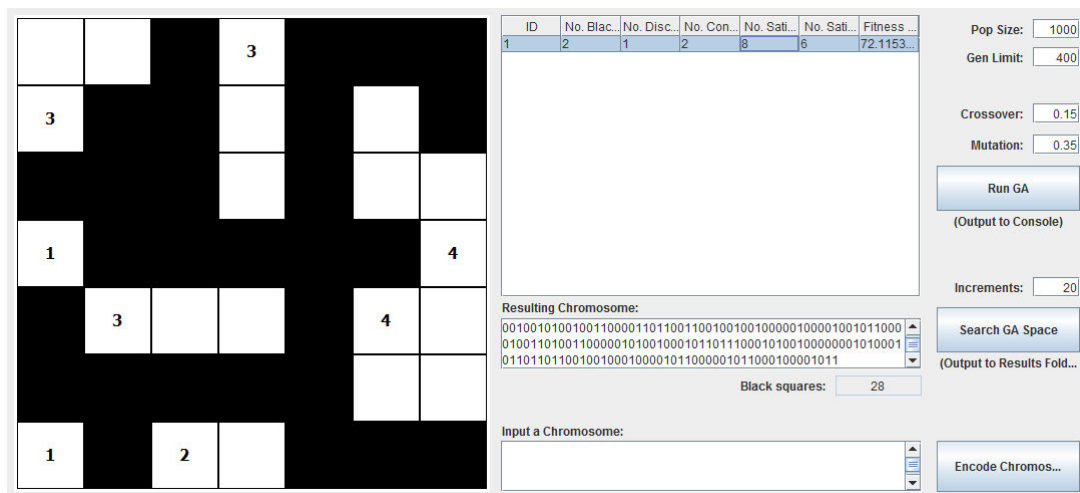


Figure 4.17: The results of an execution of the genetic algorithm with default parameters

This run used the default parameter values, and the best solution found had a fitness of 72.01%. The statistics that are output to the console can also be seen in Figure 4.18.

Time statistics
Selection: sum=0.171150153000 s; mean=0.000319906828 s
Altering: sum=2.824845497000 s; mean=0.005280085041 s
Fitness calculation: sum=1.591442093000 s; mean=0.002974658118 s
Overall execution: sum=4.547287901000 s; mean=0.008499603553 s
Evolution statistics
Generations: 535
Altered: sum=208,890; mean=390.448598131
Killed: sum=0; mean=0.000000000
Invalids: sum=0; mean=0.000000000
Population statistics
Age: max=38; mean=1.536426; var=5.228997
Fitness:
min = 0.000000000000
max = 72.115384615385
mean = 55.508003694608
var = 760.962768379091
std = 27.585553617412

Figure 4.18: The statistics of an execution of the genetic algorithm with default parameters

These statistics provide an advanced breakdown of the evolution stream, including the time taken to execute individual parts of the engine, and details on the number of generations, as well as the age of the winning chromosome. It can be seen that this solution was 38 generations old and the algorithm terminated on generation 535, where a plateau was reached and the stream terminated.

4.3.2 Survivor and Offspring Selection Testing

There are a number of ways to customize the evolution engine in attempt to improve the results, including different selection techniques for the survivors of a generation, and the individuals selected from the pool of offspring for alteration. As it was necessary to use the AnyChromosome class to implement the codec for Nurikabe, the number of selectors available in Jenetics was slightly limited.

To decide on the appropriate selection techniques for both the survivors and offspring, a number of tests were done using multiple selectors, with the same default values for all parameters to maintain consistent results, which can be seen in Table 4.4. All tests were executed on both the 7x7 Nurikabe puzzle which was used as the working example, and another 9x9 puzzle, to see if the complexity of the puzzles has an impact on how effective the different selection techniques are.

Table 4.4: Results of genetic algorithm execution with various selectors on 7x7 puzzle

Survivor Selector	Offspring Selector	Average Best Fitness (%)
Tournament	Roulette Wheel	84.32
Tournament	Linear Rank	88.20
Tournament	Monte Carlo	93.69
Tournament	Tournament	79.23
Roulette Wheel	Tournament	83.93
Roulette Wheel	Linear Rank	91.19
Roulette Wheel	Monte Carlo	91.44
Roulette Wheel	Roulette Wheel	80.65
Linear Rank	Roulette Wheel	81.97
Linear Rank	Monte Carlo	87.01
Linear Rank	Tournament	88.93
Linear Rank	Linear Rank	91.39
Monte Carlo	Roulette Wheel	81.86
Monte Carlo	Linear Rank	81.82
Monte Carlo	Tournament	84.12
Monte Carlo	Monte Carlo	86.29

It can be seen from the results of executing the genetic algorithm with different selection combinations that there are a number of combinations that don't work. There is an issue with these tests, as the genetic algorithm commonly solved the puzzle, achieving a fitness of 100%, so the averages were sometimes weighted inappropriately in cases that multiple solutions were found to be correct. For this reason, it is necessary to repeat the tests on a more complex puzzle, so the stronger combinations are to be used in genetic algorithm runs on a 9x9 puzzle board, as seen in Table 4.5.

The Monte Carlo selector will not be considered in these more accurate tests, as

it is to be used as a baseline selector, that all other selectors should out perform. Another selector will be tested, to see if it has any impact on the result as well, providing a more comprehensive review of the available selectors.

Table 4.5: Results of genetic algorithm execution with various selectors on 9x9 puzzle

Survivor Selector	Offspring Selector	Average Best Fitness (%)
Tournament	Roulette Wheel	54.71
Tournament	Linear Rank	57.57
Tournament	Boltzmann	55.79
Roulette Wheel	Tournament	59.65
Roulette Wheel	Linear Rank	55.45
Roulette Wheel	Boltzmann	56.05
Linear Rank	Tournament	56.58
Linear Rank	Linear Rank	56.66
Linear Rank	Boltzmann	58.51
Boltzmann	Tournament	60.81
Boltzmann	Roulette Wheel	55.33
Boltzmann	Linear Rank	59.78

These results reveal more than the previous set, although it can be seen that the choice of selector doesn't seem to have much impact on the overall success of the genetic algorithm. This may not turn out to be the case after testing the other parameters and functionality of the engine, but as the *boltzmann* selector seems to provide the highest fitness values overall, it will be used as the selection method for further testing.

To re-enforce this choice and to highlight a suitable combination for the Boltzmann selector, a set of refined tests have been completed calculating the average from more runs of the genetic algorithm, as seen in Table 4.6. In these tests the best fitness found from any of the genetic algorithm executions is noted as well,

because it may provide useful information in judging the strongest combination.

Table 4.6: Results of genetic algorithm execution combining boltzmann with other selectors, to re-enforce the appropriate choice of selector

Survivor Selector	Offspring Selector	Best Fitness	Average Best Fitness (%)
Boltzmann	Roulette Wheel	77.16	55.61
Boltzmann	Tournament	66.96	57.73
Boltzmann	Linear Rank	69.44	57.18
Roulette Wheel	Boltzmann	66.96	57.77
Tournament	Boltzmann	66.96	58.24
Linear Rank	Boltzmann	68.18	57.92

It can be seen from these results that the best performing combination is to use tournament selection for the survivors, and boltzmann selection for the offspring. It is also worth noting that a combination of boltzmann and roulette wheel for the survivors and offspring respectively resulted in the single highest fitness value for a single solution, which could mean that this combination is more likely to provide spiking results, which may mean a correct solution in some situations. It is also possible that this value is purely an anomaly in the data, as a large number of puzzles are generated in each run.

To facilitate these tests, a new button was added to the GUI to run multiple executions of the genetic algorithm on the same board, outputting the results and average fitness value to the console. The table is also populated with the best chromosome from each run, an example during the testing for the roulette wheel selection can be seen in Figure 4.19.

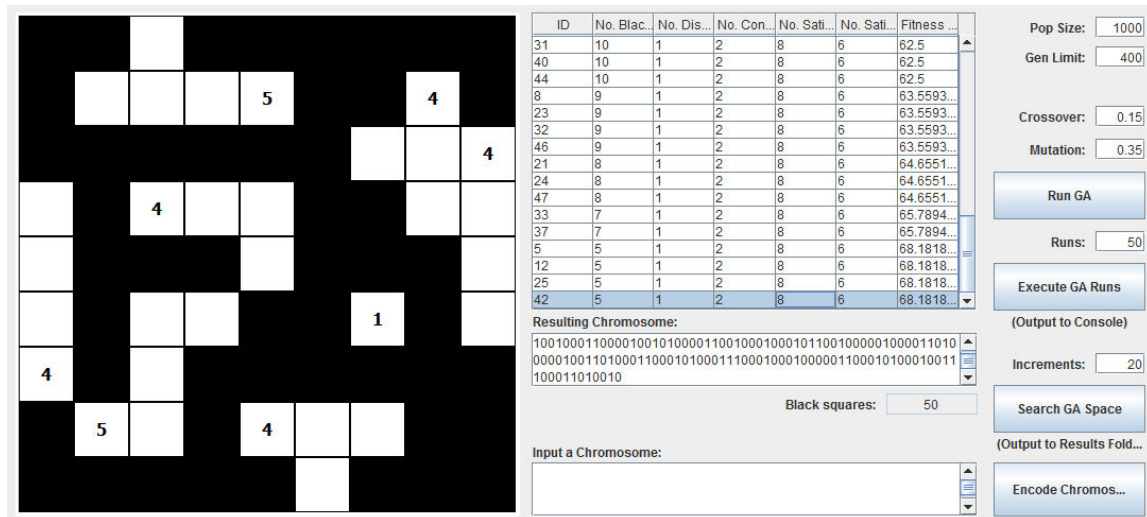


Figure 4.19: Testing results displaying best chromosomes using boltzmann selection

4.3.3 Alteration Testing

Perhaps the most important values to refine in order to improve the success of the genetic algorithm are the rates of alteration. Now that a consistently successful selection method has been chosen, it is possible to run another set of tests to determine the optimal values for the crossover and mutation rates. All of the tests in this section will be performed using tournament selection for the survivors, and boltzmann selection for the offspring.

In order to harness the execution of these tests, functionality has been included to execute multiple instances of the genetic algorithm at incremental rates of mutation and crossover. The data about the most successful chromosome of each run is printed to a file, and the overall strongest chromosome is updated at the top of the file at the end of each run, so the progression of the fitness values over the increments can be seen.

These tests should serve to pinpoint the most appropriate values for these parameters, and a number of different boards will be used for these tests to provide a reliable range of results. It will be interesting to see if the complexity of a puzzle impacts the required crossover and mutation rates, as that will confirm that Nurikabe is a very complex problem space which could need a scaling factor for the values.

The first puzzle to be tested will be a 9x9 puzzle, and the genetic algorithm will run with 10 increments for the mutation and crossover rates between 0 and 1, as seen in Table 4.7. The number of results in the table has been limited to the best of each increment of the crossover rate, to avoid the result sets becoming too long.

Table 4.7: Searching rates of alteration on multiple runs of the genetic algorithm on a 9x9 Nurikabe puzzle

Crossover Rate	Mutation Rate	Best Fitness Value (%)
0.00	0.60	66.96
0.10	0.80	66.96
0.20	0.20	64.66
0.30	0.20	70.75
0.40	0.80	68.19
0.50	0.70	69.44
0.60	0.60	68.19
0.70	0.90	72.12
0.80	0.50	66.96
0.90	0.50	65.79

These results indicate that a high level of crossover and mutation are the most successful, as the best fitness found from all runs was 72.12%, with a crossover of 0.70 and mutation of 0.90. It can also be seen that a solution was found with a percentage of 70.75% at considerably lower rates of alteration, indicating that lower rates could also work well, although even such a slight difference in success could mean substantial improvement in the solution. The same process was repeated on a 12x12 Nurikabe puzzle, displayed in Table 4.8.

Table 4.8: Searching rates of alteration on multiple runs of the genetic algorithm on a 12x12 Nurikabe puzzle

Crossover Rate	Mutation Rate	Best Fitness Value (%)
0.00	0.40	59.52
0.10	0.30	56.50
0.20	0.90	58.33
0.30	0.70	57.38
0.40	0.80	63.64
0.50	0.50	56.55
0.60	0.90	53.76
0.70	0.80	60.34
0.80	0.90	56.50
0.90	0.50	57.47

The test data from the runs on a 12x12 puzzle tend to confirm that a higher mutation rate is more appropriate, as the only two rates to achieve a fitness above 60 had mutation rates of 0.80. It seems that a middling crossover with a high mutation rate is the best combination for the genetic algorithm, so for future tests the crossover rates will be between 0.30 and 0.60, and the mutation rates between 0.80 and 1. The best puzzle solutions from each set of tests were encoded back into the boards as seen in Figure 4.20.

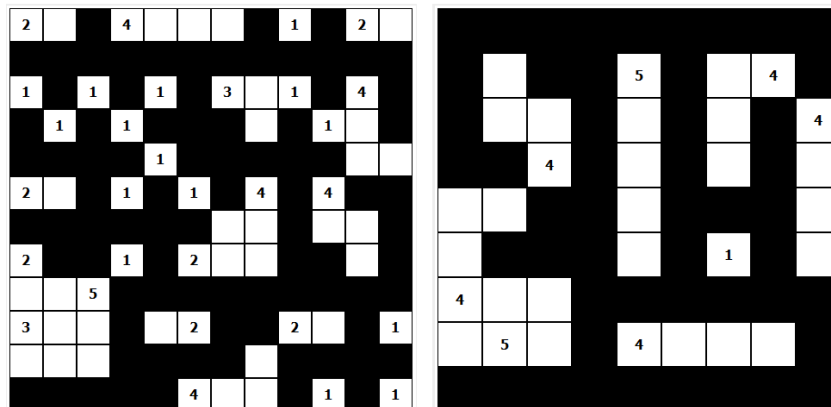


Figure 4.20: The best solutions for 12x12 and 9x9 Nurikabe puzzle boards found whilst testing rates of alteration

It can be seen from these solutions that they are both fairly strong, with small groups of problem areas which impact the results of the objectives, bringing their fitness value down. In the case of the 9x9 solution, all but two of the numbers have had their islands satisfied, and the board is only a few changes away from becoming a correct solution.

4.3.4 Further Testing

To allow the genetic algorithm to search more solutions during a single execution, it is possible to increase the size of each population, and to increase the generation limit, allowing more generations of chromosomes to be evaluated before the run terminates at a plateau. Previous tests used a population size of 1000 and generation limit of 400, which was sufficient for finding appropriate parameter values.

To see if the number of individuals per population and the number of generations impacts the results, these values have been increased to 5000 and 2000 respectively, and a number of executions were performed on a complex, 15x15 puzzle board. To provide contrasting results, the same number of tests were run with the previous parameter values, which can be seen in Tables 4.9 and 4.10. These tests are using the most appropriate engine parameters as found in the previous tests, in order to promote a result with the strongest results.

Table 4.9: Ten runs on 15x15 Nurikabe puzzle with a population size of 1000 and generation limit of 400

Run No.	Final Chromosome Fitness (%)	Run No.	Final Chromosome Fitness (%)
1	28.94	1	37.17
2	27.06	2	32.34
3	30.03	3	31.91
4	26.28	4	33.78
5	27.06	5	34.72
6	24.31	6	35.71
7	35.24	7	31.75
8	26.71	8	32.34
9	29.76	9	33.26
10	25.93	10	37.54

Table 4.10: Ten runs on 15x15 Nurikabe puzzle with a population size of 5000 and generation limit of 2000

These results provide evidence to confirm that a higher generation limit results in better solutions, although the downside is that the generation limit massively impacts the genetic algorithm's execution time. The set of chromosomes from the run with the higher generation limit can be seen in Figure 4.21.

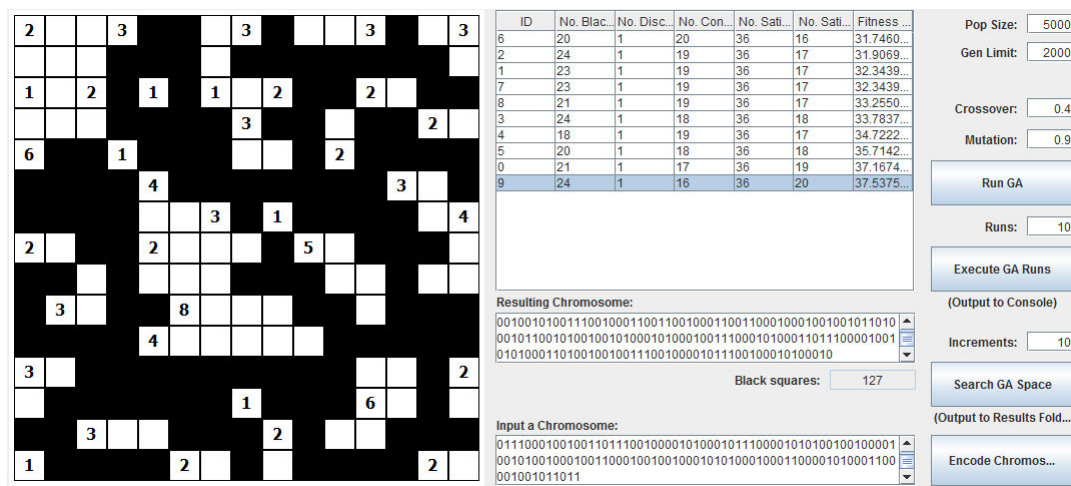


Figure 4.21: Ten solutions for a 15x15 puzzle using a very high generation limit and population size

It is clear that a higher generation limit results in stronger solutions, although

it is not clear if the same can be said for population size. To test this, 400 solutions have been generated for a 9x9 puzzle with a population size of 200 and default generation limit, with results displayed in Figure 4.22.

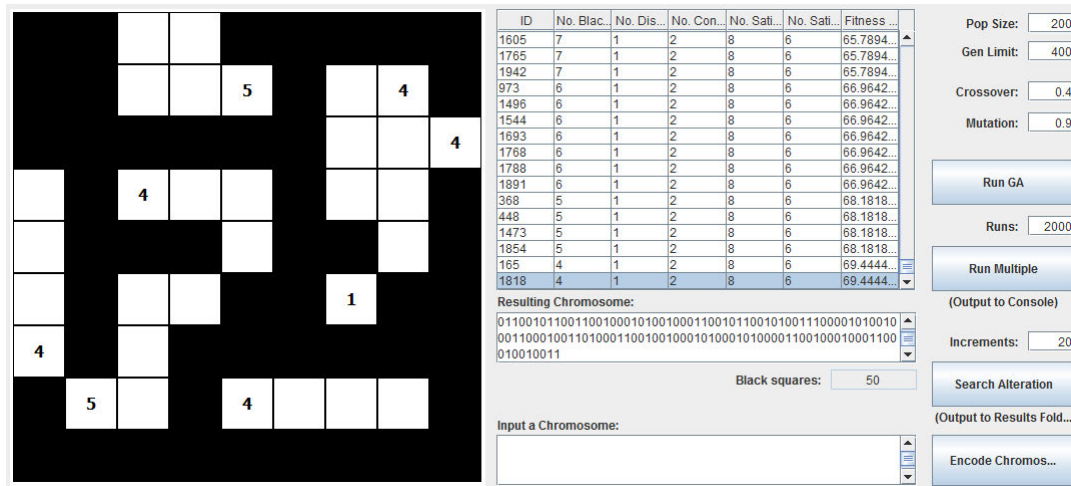


Figure 4.22: 400 solutions for a 9x9 puzzle using a low population size

The best of the 400 solutions has a fitness of 69.44%. It is a strong chromosome with a good fitness, and it can be seen that the solution is quite close to being correct, and all but two of the islands are satisfied. It should be noted that whilst it would be easy for a human to fix the puzzle by eye, it's considerably harder for a computer. In order to provide contrasting results, another set of runs were completed using a population size of 1000, as seen in Figure 4.23.

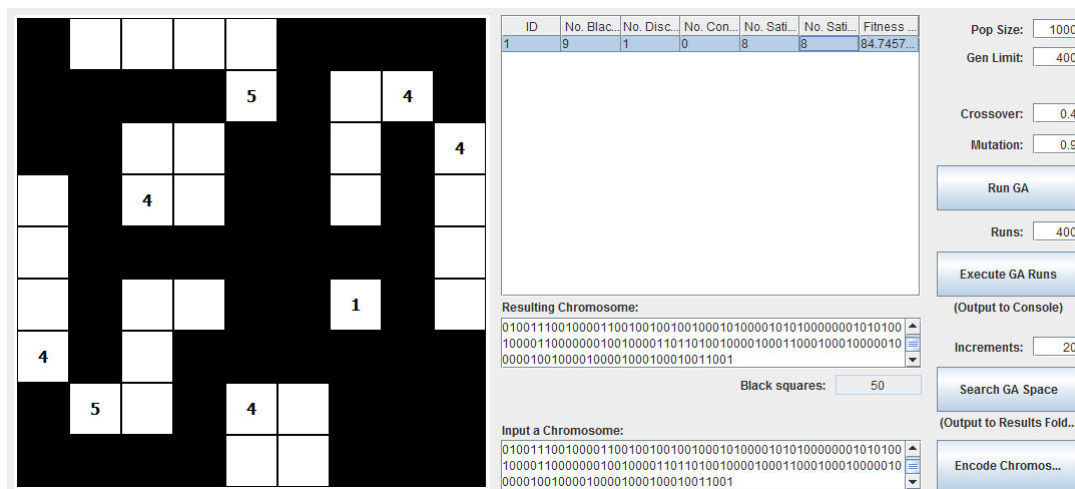


Figure 4.23: 400 solutions for a 9x9 puzzle using a higher population size

It is evident by looking at the best chromosome of the runs with a higher popula-

tion size that a higher population size tends to benefit the strength of the algorithm. This final solution is very strong, each island is complete and satisfied, however the solution is still incorrect as there are a number of black 2x2 areas on the board. This concludes the implementation chapter, the Nurikabe genetic algorithm engine has been configured, and the results of testing can be evaluated.

5.1 Nurikabe Complexity

The single biggest roadblock in improving the genetic algorithm is that Nurikabe puzzles become considerably harder to solve with even slight differences in board size and island complexity. It can be seen from Figure 5.1 that the algorithm can consistently solve 5x5 puzzles.

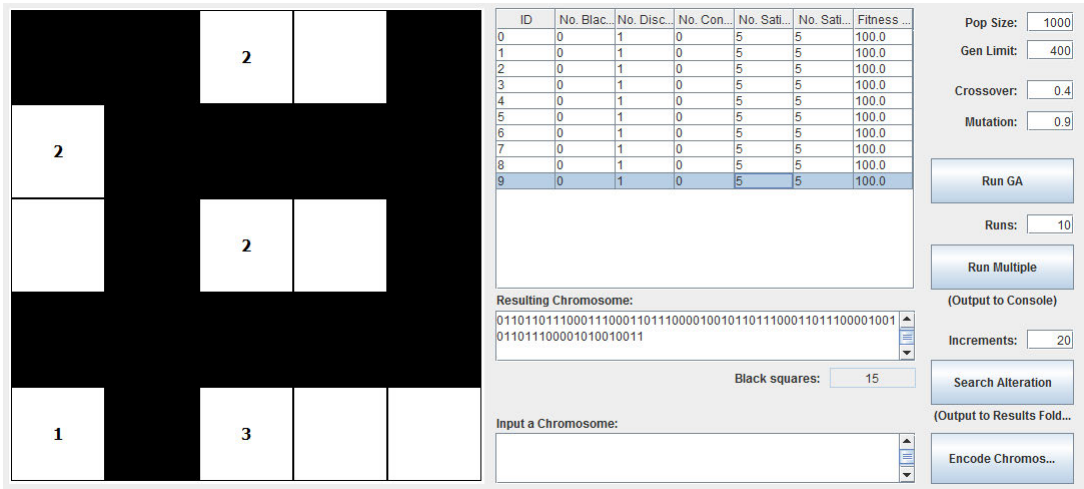


Figure 5.1: The consistent solution of 5x5 Nurikabe puzzles

By increasing the size of the puzzle to a 7x7, a dip in performance is already evident, as seen in Figure 5.2. The genetic algorithm is able to solve this puzzle, although not as consistently as it can solve the 5x5 puzzle.

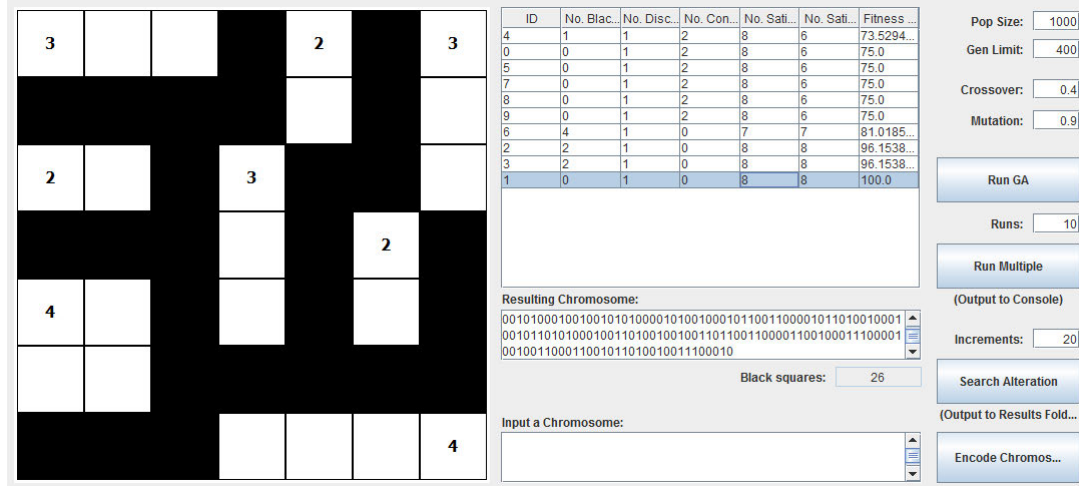


Figure 5.2: A correct solution for a 7x7 Nurikabe puzzle

This trend continues as the complexity of the puzzle to be solved is increased. This can be seen in Table 5.1, where a number of tests were run on puzzles of varying size with the same parameters, to highlight the performance curve.

Table 5.1: A demonstration of the strength of the genetic algorithm on Nurikabe puzzles with varying dimensions

Puzzle Dimensions	Average Fitness Value (%)	Best Fitness Value (%)
5x5	100.00	100.00
7x7	78.74	100.00
9x9	61.58	66.96
10x10	56.23	61.21
12x12	53.24	55.56
15x15	28.42	31.30
20x20	18.82	20.81

These results clearly display that the complexity of a Nurikabe puzzle drastically impacts the effectiveness of the genetic algorithm. Smaller puzzle boards can be

solved in short periods of time, whereas executions on larger, more complex puzzles take considerably more time, and do not achieve a correct solution. The reason for this is that larger puzzles become extremely complex very quickly, and there can be multitudes of possible combinations of paths, only one of which providing a correct solution.

As there is generally only one solution to a Nurikabe puzzle, and that solution is extremely specific, the genetic algorithm tends to reach a plateau in execution, achieving similar fitness values on each individual board. Each island in a solution tends to be complete in the best solution from each run, but the black 2x2 areas are rarely handled, and a lot of islands are left connected and unsatisfied.

5.2 Testing Evaluation

5.2.1 Survivor and Offspring Selection Testing

It was beneficial to run tests to select an appropriate combination of selectors, as it provided a stronger basis of execution for all tests that came after it. In solving Nurikabe, the most successful selector for the survivors is the tournament selector, and it is the boltzmann selector for the offspring. The choice of selection doesn't seem to impact the success of the execution by that large a margin, although any improvement is important in this type of project.

5.2.2 Alteration Testing

Testing the genetic algorithm with incremental levels of alteration revealed a suitable rate for both the crossover and mutation of the evolution engine. This improved the performance of the algorithm, resulting in stronger solutions.

5.3 Objective Achievement

At the outset of this research project, a number of objectives were established, and the success of these objectives can be retrospectively discussed.

The first objective was to perform a literature review, to review other work in the same area. This was successful, and provided an in-depth understanding of the features of a genetic algorithm, as well as required artificial intelligence techniques. Acquiring this knowledge was fundamental to implementing the genetic algorithm for Nurikabe, as it is a difficult goal to achieve, and a number of choices that were made throughout the progress were influenced by methodologies highlighted in the review.

Next, a project plan was to be defined, in order to provide the designs for the genetic algorithm and the Nurikabe testing harness. Then the engine was to be implemented, and a number of tests done to confirm that the puzzle and solution encoding worked as expected. Developing the engine was a testing process, but it is successful in allowing the generation, verification and alteration of Nurikabe puzzles.

The next objective was to implement the automated solver to the engine, recording test results whilst altering parameters, to judge the feasibility of using a genetic algorithm to solve Nurikabe puzzles. This was mostly successful - there were a number of road blocks during the implementation, because the nature of the code is fairly complicated, and it was a big learning curve, which paid off by the end.

The final objective to consider is to evaluate the data found from testing, and discuss the viability of using genetic algorithms to solve Nurikabe. This objective has been met, and the test data gathered during the implementation has been evaluated to reveal that whilst a genetic algorithm may not be the most ideal technique to solve Nurikabe, it is definitely suitable, and in combination with other techniques it could provide a very accurate basis for quickly solving Nurikabe puzzles.

CHAPTER 6

Conclusions

6.1 Completed Work

The original intention for this project was to research and create an automated solver for the Japanese puzzle game, Nurikabe, using a multi-objective genetic algorithm. Extensive research has been completed, resulting in a comprehensive literature review which revealed that there are a number of uses for genetic algorithms in automated game solving.

A Nurikabe engine has been implemented in Java, which facilitates the execution of a genetic algorithm on any given Nurikabe puzzle. The genetic algorithm is successful in solving small Nurikabe puzzles, and it shows promise in solving larger puzzles.

6.2 Limitations

6.2.1 Tweaking Final Solutions

When executing runs on puzzle boards that the genetic algorithm is unable to solve, the final solutions are very interesting, and suggest that whilst a genetic algorithm alone isn't very successful, a combination of techniques may be more fruitful. As seen in Figure 6.1, and as mentioned throughout the implementation, the best solutions on larger boards could be fixed by a person with a working knowledge of Nurikabe in a short time. These solutions are only several changes away from being correct, although it isn't straightforward for a computer to make those changes.

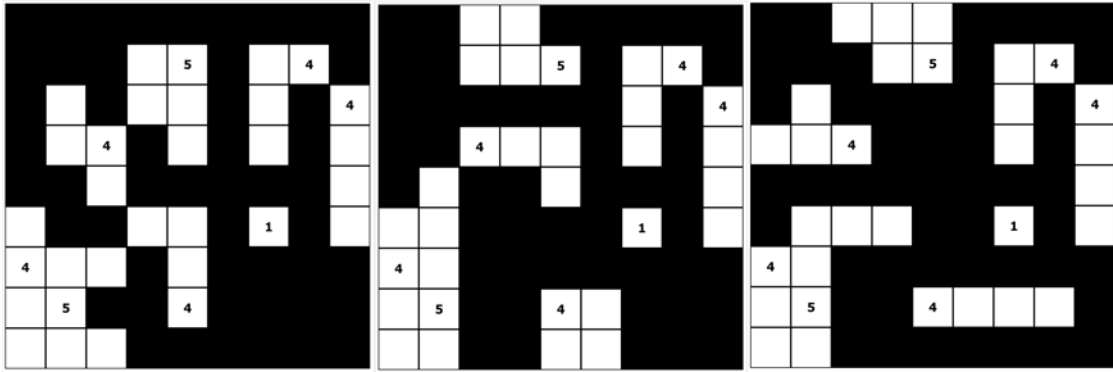


Figure 6.1: Solutions for a 9x9 Nurikabe puzzle which could be completed by a person with a working knowledge of Nurikabe

6.2.2 Execution Time

Whilst it hasn't become an issue during the course of this project, the time taken to run the genetic algorithm could be an issue. When using high parameter values for the population size and generation limit, or when running the genetic algorithm on a large puzzle board, it can take several minutes, hours or even days for some of the testing to be completed. A single run is relatively fast, but the code could be re-factored to improve its performance further.

6.3 Final Conclusion

Nurikabe is a suitable puzzle game to be used to test genetic algorithms, and whilst the genetic algorithm implemented in this project is successful in solving small puzzles, it consistently reaches a plateau in fitness when attempting to solve more complex puzzles. This inspires the conclusion that whilst genetic algorithms can be used to solve Nurikabe, they may not be the most successful or worthwhile approach.

This suggests that a Nurikabe solver could be comprised of a genetic algorithm to achieve a strong solution very quickly, in combination with a rule based system to prune the attributes of the solution, providing a reliable platform for solving Nurikabe, as well as other similar number-based puzzle games.

6.4 Future Work

In the future development of the Nurikabe solver, a number of changes and improvements can be made in attempt to achieve a more consistently successful result. It will be worthwhile to investigate the use of other techniques to work with the genetic algorithm, in order to achieve more correct puzzle solutions. The testing functionality could also be improved, allowing results to be output in various different formats, making the test results more readable, and potentially more usable for visualizing the data. The Nurikabe engine could also be updated to allow the user to alter a path in an easier way, for example by clicking on elements in a grid, which would be difficult to achieve.

6.5 Summary

In summary, the aim to research and implement an automated solver for Nurikabe has been completed, with interesting results. The ability to consistently solve small puzzles is a major accomplishment, and there are a number of doors open for future iterations of the product to solve larger puzzles, and different games as well.

CHAPTER 7

Bibliography

- A. Alfari. Multidisciplinary system design optimization. page 6, 2010.
- M. Amos and J. Coldridge. A genetic algorithm for the zen puzzle garden game. *Natural Computing*, 11(3):353–359, 2012.
- M. R. Black and H. Taylor. *Unscrambling the Cube*. Zephyr Engineering Design, 1980.
- S. Bling. Mario - machine learning for video games, 2015. URL <https://www.youtube.com/watch?v=qv6UV0Q0F44>.
- A. Cawsey. *The Essence of Artificial Intelligence*. Prentice Hall PTR, 1997.
- K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2): 182–197, 2002.
- E. Friedman. Pearl puzzles are np-complete. *Unpublished manuscript, August*, 2002.
- M. Holzer, A. Klein, and M. Kutrib. On the np-completeness of the nurikabe pencil

- puzzle and variants thereof. In *Proceedings of the 3rd International Conference on FUN with Algorithms*, pages 77–89, 2004.
- C. R. Houck, J. Joines, and M. G. Kay. A genetic algorithm for function optimization: a matlab implementation. *NCSU-IE TR*, 95(09), 1995.
- L. Huttar and L. Kathy. Nurikabe: Generating and auto-solving, 2005. URL <http://www.huttar.net/lars-kathy/puzzles/nurikabe.html>.
- G. Kendall and K. Spoerer. Scripting the game of lemmings with a genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 117–124, 2004.
- D. Lichtenstein. Planar formulae and their uses. *SIAM journal on computing*, 11(2):329–343, 1982.
- K.-F. Man, K.-S. Tang, and S. Kwong. Genetic algorithms: concepts and applications. *IEEE transactions on Industrial Electronics*, 43(5):519–534, 1996.
- J. McCarthy. Partial formalizations and the lemmings game. Technical report, Citeseer, 1998.
- J. J. Merelo, M. Keijzer, and M. Schoenauer. Evolving objects, 1998. URL <http://eodev.sourceforge.net/>.
- D. Merrit. Prolog in action, 2016. URL <http://www.amzi.com/articles/rubik.htm>.
- Nikoli. Rules of nurikabe puzzle, 2006. URL <http://www.nikoli.com/en/puzzles/nurikabe/rule.html>.
- M. Sipser. Introduction to the theory of computation, second edition. page 266, 2006.
- K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.
- C. Stergiou and D. Siganos. Neural networks, 1996.

- Y. Takayuki. On the np-completeness of the slither link puzzle. *IPSJ SIGNotes ALgorithms*, 74:25–32, 2000.
- Y. Takayuki and S. Takahiro. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.
- F. Wilhelmstotter. Jenetics, 2007. URL <http://jenetics.io/>.

Appendices

Appendix A: Terms of Reference



List of Figures

1	Example of a Nurikabe puzzle and its solution.	1
---	--	---

Contents

1	Project Background	1
2	Learning Outcomes	2
3	Aim	2
4	Objectives	2
5	Problems	3
6	Timetable and Deliverables	4
7	Required Resources	4

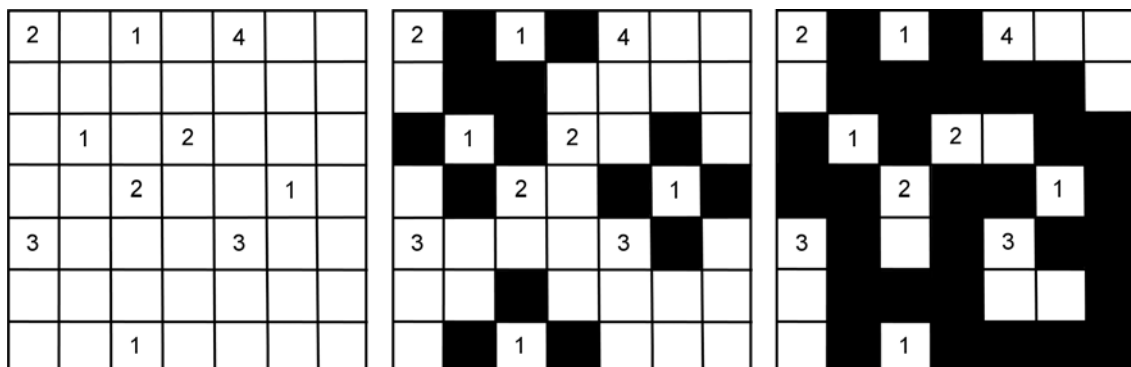
1 Project Background

Puzzle games have been played globally for hundreds of years, predating all modern technology, and they all share something in common. At their core every puzzle requires a player to find a certain configuration of pieces, be that numbers, shapes, colours or sounds to achieve a final, correct solution. The difficulty, length and scope of every puzzle is different and whilst they are usually simple to explain, some puzzles can be extremely challenging to complete.

The variance and complexity of the strategies involved in solving a puzzle provides a great opportunity to investigate automated solving techniques, an example being the popular numeric puzzle game 'Sudoku', in which the player is given a partially complete grid of numbers which must be completed, with some constraints. Another example of a challenging puzzle game is 'Nurikabe', originating from Japanese folklore, in which the player is presented with a grid that is also partially populated by numbers. To complete the puzzle the grid must be filled with black squares (usually referred to as the sea), which must all be connected, allowing space for a number of white squares to be connected to the numbers dictated by their value (forming islands), and this must be done following a complicated set of constraints.

Nurikabe is a difficult puzzle with a vast scope and many rules to follow, research has proven it to be NP-complete [1] and the grid can be expanded to increase the difficulty further. I think that the combination of these factors provides a strong platform for research and development and in my project I will use an aspect of A.I. (artificial intelligence) called genetic algorithms to implement an automated solver for Nurikabe puzzles.

Figure 1: Example of a Nurikabe puzzle and its solution.



2 Course-Specific Learning Outcomes

On completion of my project I will be able to do the following:

1. Demonstrate extensive knowledge of how A.I. can be used to solve complicated problems which can't be solved efficiently using normal computing methods.
2. Develop knowledge of computer hardware and an understanding of how the selection of hardware will affect the performance of an application.
3. Display an ability to work and develop new skills independently of teaching.
4. Investigate the interaction between hardware and software and the influence of this interaction on the design of computer systems.
5. Assess previous work to highlight weaknesses and areas that could be improved, to gauge progress and to advance the ability to complete large projects in the future.
6. Develop an understanding of the nature of databases and be able to develop, maintain and interrogate databases.

3 Aim

The aim of my project is to research and create an automated solver for Nurikabe, using multi-objective genetic algorithms which will be able to effectively adhere to the many constraints involved in solving the puzzle. I will examine existing solvers for Nurikabe as well as other similar puzzles, comparing the results of my solution with other automated solving techniques to draw a conclusion about the effectiveness of using multi-objective genetic algorithms for this purpose.

4 Objectives

To be successful in the project I will need to complete the following steps:

- Perform a literature review, reviewing other people's work in this area, particularly pre-existing solvers for Nurikabe and research regarding multi-objective genetic algorithms.

- Define a project plan, deciding which software and development languages to use to run the genetic algorithms. Design a Nurikabe engine, providing a platform to test the solver and analyze the results.
- Develop the Nurikabe engine and run extensive unit tests to ensure that it adheres to the strict constraints of the puzzle, this is important to avoid any inconsistencies in the data that come from testing the solver as it could invalidate the tests.
- Implement the automated solver, recording the results of test runs on puzzles of varying complexity. Adjust key factors such as the difficulty of the puzzles and the size of the grids to acquire an expansive data-set that has multiple results for each puzzle variation, to eliminate any data anomalies.
- Perform similar test runs using pre-existing automated solvers that don't necessarily use any aspects of A.I. in their execution, recording the results from these as well.
- Evaluate the data gathered from testing, comparing how the different solvers perform at different levels of complexity. If the data is reliable and decisive draw a conclusion about the efficiency and viability of using genetic algorithms to solve NP-complete puzzle games like Nurikabe, and in other applications as well.
- Reflect on the project and consider how successfully the aims and objectives were met.

5 Problems

There are a number of problems that could arise whilst working on my project:

- This is a research based project and the viability of a solution may only become clear fairly late in to its progression. To circumvent any potential failures I will need to be confident throughout my research that I will be able to complete my aim even if it requires me to alter my approach.
- I have decided to use multi-objective genetic algorithms to develop a solution, if this method turns out to be unsuitable for solving Nurikabe then I will have to look for viable substitutes. This can be avoided by extensively analyzing other areas of A.I. during my literature review to highlight effective alternatives.

6 Timetable and Deliverables

Finish	Requirement	Description
17/10/2016	Terms of Reference	The initial plan for the project
31/10/2016	Literature Review	Research into existing solvers and A.I. techniques
31/11/2016	Product Design	The design for my automated Nurikabe solver
19/12/2016	Engine Implementation	Implement the Nurikabe engine
30/01/2016	Solver Implementation	Implement the automated solver
14/02/2016	System Testing	Test the engine and solver, record all results for validation
25/03/2016	Experimental Comparisons	Evaluate and compare results
24/04/2017	Project Submission	The final submission of the project
24/04/2017	Project Presentation	The presentation of the completed project

The deliverables of my project are as follows:

- Comprehensive research into using A.I. to solve puzzle games such as Nurikabe.
- An automated solver for Nurikabe using multi-objective genetic algorithms.
- An evaluation of the final results and comparison between my automated solver and other pre-existing solvers.

7 Required Resources

- A standard computer to run the genetic algorithms.
- Implementation platforms and software to be decided in the product design.

References

- [1] Markus Holzer, Andreas Klein, and Martin Kutrib. “On the NP-completeness of the nurikabe pencil puzzle and variants thereof”. In: *Proceedings of the 3rd International Conference on FUN with Algorithms*. 2004, pp. 77–89.

Appendix B: Ethics Form



Manchester
Metropolitan
University

ETHICS CHECKLIST

This checklist must be completed **before** commencement of **any** research project. This includes projects undertaken by **staff and by students as part of a UG, PGT or PGR programme**. Please attach a Risk Assessment.

Please also refer to the [University's Academic Ethics Procedures](#); [Standard Operating Procedures](#) and the [University's Guidelines on Good Research Practice](#)

Full name and title of applicant:	<div style="background-color: black; width: 100%; height: 100%;"></div>	
University Telephone Number:		
University Email address:		
Status: <small>All staff and students involved in research are strongly encouraged to complete the Research Integrity Training which is available via the Staff and Research Student Moodle areas</small>	Undergraduate Student <input checked="" type="checkbox"/> Postgraduate Student: Taught <input type="checkbox"/> Postgraduate Student: Research <input type="checkbox"/> Staff <input type="checkbox"/>	
Department/School/Other Unit:	Computing and Digital Technology Network	
Programme of study (if applicable):	Computer Science	
Name of DoS/Supervisor/Line manager:	Professor Martyn Amos	
Project Title:	A.I. Solver for Games	
Start & End date (cannot be retrospective):	01/10/2016 - 28/04/2017	
Number of participants (if applicable):		
Funding Source:		
Brief description of research project activities (300 words max): Perform a literature review, researching genetic algorithms as well as reviewing pre-existing automated solvers for Nurikabe and other similar puzzle games. Define a project plan, deciding which software and development languages to use. Develop the Nurikabe engine and run extensive unit tests. Implement an automated solver, recording the results of test runs on puzzles of varying complexity. Analyze all of the test results to reach a final conclusion.		
	YES	NO
Does the project involve NHS patients or resources? If 'yes' please note that your project may need NHS National Research Ethics Service (NRES) approval. Be aware that research carried out in a NHS trust also requires governance approval. Click here to find out if your research requires NRES approval Click here to visit the National Research Ethics Service website To find out more about Governance Approval in the NHS click here	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Does the project require NRES approval? If yes, has approval been granted by NRES? Attach copy of letter of approval. Approval cannot be granted without a copy of the letter.	<input type="checkbox"/>	<input checked="" type="checkbox"/>


NB Question 2 should only be answered if you have answered YES to Question 1. All other questions are mandatory.		YES	NO
1.	Are you are gathering data from people?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
For information on why you need informed consent from your participants please click here			
2.	If you are gathering data from people, have you:	<input type="checkbox"/>	<input type="checkbox"/>
	a. attached a participant information sheet explaining your approach to their involvement in your research and maintaining confidentiality of their data?	<input type="checkbox"/>	<input type="checkbox"/>
	b. attached a consent form? (not required for questionnaires)	<input type="checkbox"/>	<input type="checkbox"/>
Click here to see an example of a participant information sheet and consent form			
3.	Are you gathering data from secondary sources such as websites, archive material, and research datasets?	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Click here to find out what ethical issues may exist with secondary data			
4.	Have you read the guidance on data protection issues?	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	a. Have you considered and addressed data protection issues – relating to storing and disposing of data?	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	b. Is this in an auditable form? (can you trace use of the data from collection to disposal)	<input checked="" type="checkbox"/>	<input type="checkbox"/>
5.	Have you read the guidance on appropriate research and consent procedures for participants who may be perceived to be vulnerable?	<input checked="" type="checkbox"/>	<input type="checkbox"/>
	a. Does your study involve participants who are particularly vulnerable or unable to give informed consent (e.g. children, people with learning disabilities, your own students)?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
6.	Will the study require the co-operation of a gatekeeper for initial access to the groups or individuals to be recruited (e.g. students at school, members of self-help group, nursing home residents)?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click for an example of a PIS and information about gatekeepers			
7.	Will the study involve the use of participants' images or sensitive data (e.g. participants personal details stored electronically, image capture techniques)?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here for guidance on images and sensitive data			
8.	Will the study involve discussion of sensitive topics (e.g. sexual activity, drug use)?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here for an advisory distress protocol			
9.	Could the study induce psychological stress or anxiety in participants or those associated with the research, however unlikely you think that risk is?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read about how to deal with stress and anxiety caused by research procedures			
10.	Will blood or tissue samples be obtained from participants?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read how the Human Tissue Act might affect your work			
11.	Is your research governed by the Ionising Radiation (Medical Exposure) Regulations (IRMER) 2000?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to learn more about IRMER			
12.	Are drugs, placebos or other substances (e.g. food substances, vitamins) to be administered to the study participants or will the study involve invasive, intrusive or potentially harmful procedures of any kind?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read about how participants need to be warned of potential risks in this kind of research			
13.	Is pain or more than mild discomfort likely to result from the study? Please attach the pain assessment tool you will be using.	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Click here to read how participants need to be warned of pain or mild discomfort resulting from the study and what do about it.		
14. Will the study involve prolonged or repetitive testing or does it include a physical intervention?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to discover what constitutes a physical intervention and here to read how any prolonged or repetitive testing needs to managed for participant wellbeing and safety		
15. Will participants to take part in the study without their knowledge and informed consent? If yes, please include a justification.	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read about situations where research may be carried out without informed consent		
16. Will financial inducements (other than reasonable expenses and compensation for time) be offered to participants?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read guidance on payment for participants		
17. Is there an existing relationship between the researcher(s) and the participant(s) that needs to be considered? For instance, a lecturer researching his/her students, or a manager interviewing her/his staff?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Click here to read guidance on how existing power relationships need to be dealt with in research procedures		
18. Have you undertaken Risk Assessments for each of the procedures that you are undertaking?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
19. Is any of the research activity taking place outside of the UK?	<input type="checkbox"/>	<input checked="" type="checkbox"/>
20. Does your research fit into any of the following security sensitive categories: <ul style="list-style-type: none"> • commissioned by the military • commissioned under an EU security call • involve the acquisition of security clearances • concerns terrorist or extreme groups If Yes, please complete a Security Sensitive Information Form	<input type="checkbox"/>	<input checked="" type="checkbox"/>

I understand that if granted, this approval will apply to the current project protocol and timeframe stated. If there are any changes I will be required to review the ethical consideration(s) and this will include completion of a 'Request for Amendment' form.

- ☒ have attached a Risk Assessment
☒ have attached an Insurance Checklist

If the applicant has answered YES to ANY of the questions 14 – 17 then they must complete the [MMU Application for Ethical Approval](#)

Signature of Applicant:  Date: 31/10/2016 (DD/MM/YY)

Independent Approval for the above project is (please check the appropriate box):

Granted

☒ I confirm that there are no ethical issues requiring further consideration and the project can commence.

Not Granted

☐ I confirm that there are ethical issues requiring further consideration and will refer the project protocol to the Faculty Research Group Officer.

Signature:  Date: 21/04/17 (DD/MM/YY)
Digitally signed by Nicholas Costen
 DN: cn=Nicholas Costen, o=MMU, ou=SCMDT,
 email=n.costen@mmu.ac.uk, c=GB
 Date: 2017.04.21 08:51:03 +01'00'

Print Name: Nicholas Costen Position: Unit coordinator

Approver: Independent Scrutiniser for UG and PG Taught/ PGRs RD1 Scrutiniser/
 Faculty Head of Ethics for staff.

Appendix C: Nurikabe Engine User Guide



Contents

1 Building the Engine	i
2 Using the Engine	i

1 Building the Engine

- Open the Eclipse IDE.
- Select 'File' - 'Import'.
- Select 'Existing Projects into Workspace' under the 'General' folder.
- Select 'Next'.
- Toggle 'Select archive file' and navigate to the project ZIP folder (NurikabeGA.zip).
- Select 'Finish'.

2 Using the Engine

- Specify GA parameters, or leave default (Pop Size, Gen Limit, Crossover, Mutation).
- Specify desired number of runs, or alteration rate increments, if required.
- Execute the GA by selecting 'Run GA' for a single execution, 'Run Multiple' for multiple executions, or 'Search Alteration' to search the alteration rates incrementally.
- Multiple GA runs result in the data printed to text file, found in the 'Results' folder of the project.