University of Derby

Department of Electronics, Computing, & Mathematics

A project completed as part of the requirements for BSc (Hons) Computer Games Programming

entitled

Genetic Decks: Can a Genetic Algorithm Create Viable Decks for the Game Hearthstone?

By

Elliott James Andrew Bowman

April 2018

Abstract

The deckbuilding aspect of Collectible Card Games (CCG), is a staple feature within this particular genre of games. This concept brings about a unique optimisation problem inside an increasingly large and varied search space. Throughout this study, we cover the implementation of a genetic algorithm, that is tasked with the design and production of high quality decks for the game Hearthstone: Heroes of Warcraft. Our implementation is inspired by issues discovered from past implementations of genetic algorithms within this research area, and manages to successfully generate decks that can obtain win rates over human designed counterparts, although we identify issues and potential future work that would improve the solution.

Acknowledgements

After completing the mountain of work in this absolutely monumental project, I would like to thank a few notable people who helped me throughout this entire journey.

Firstly, I would like to thank my supervisor, Chris Windmill, who apart from providing me with the topic of this study itself, also provided me with geniunely helpful advice that helped me focus and refine my project, and helped me find ways of testing that I would never even had thought of.

Secondly, I would like to thank Github user Demilich1 for creating and developing Metastone, as well as Pablo García-Sánchez et al for helping me find inspiration with my own implementation.

Thirdly, I would like to thank my Mum and Nick, who have been supportive of me all throughout my time at university, and who have somehow managed to tolerate me this whole time. Thank you for taking the time out of your busy lives to provide me with valuable help.

Finally, I would like to thank my best friends. Although we may not see each other a lot, you have always been there for me, even when I thought I was having a breakdown from stress. Thank you for helping me when the times got tough, and when I needed the extra support. I don’t think I could have finished university without you guys.

Table of Contents

1. Introduction

1.1. Project Rationale

Procedural Content Generation (PCG), defined as the 'algorithmic creation of game content with limited or indirect user input' (Togelius, Shaker and Nelson, 2016, p1), has been a popular technique used within video games and a well-known area of study for many years. The game Hearthstone: Heroes of Warcraft has been a popular platform in the context of artificial intelligence research, especially in fields relating to search algorithms and machine learning. One concept that has not been explored as thoroughly is the concept of *deckbuilding.* The work of Pablo García-Sánchez et al (García-Sánchez et al, 2016) proposed the usage of a genetic algorithm to create decks that could attain 'good' win rates when compared to human designed decks. Their project was able to generate successful decks, however they concluded that there were some problems that arose with their implementation, relating to further analysis of their solutions.

This study aims to take inspiration from their methodology, with the intent of improving upon the issues that they came across during their time on the project, and to see if the eventual results become more refined.

1.2. Hearthstone: Heroes of Warcraft

Hearthstone: Heroes of Warcraft is a Digital Collectible Card Game (DCCG) developed and published by Blizzard Entertainment (Blizzard Entertainment, 2014). The game is a turn based card game played between two opponents, with the goal of reducing the opponent's health to zero, by making use of various minion and spell cards. The player can choose to play as nine distinct 'Hero' classes that have their own specific cards and abilities.

One of the main mechanics of the game is the ability for users to create their own custom decks, by selecting cards from a list of cards that they own. This usage of deckbuilding allows players to create a deck that suits their specific playstyles.
The main gameplay is split into two distinct modes, 'Ranked' and 'Wild'; Ranked is the 'main' game mode of the game, and restricts the amount of cards a player can use to the most recently released card sets; Wild allows a player to use any card from any set, but they are unable to use these decks within the Ranked mode.

Throughout this study we will be referring to both the original Hearthstone: Heroes of Warcraft (Hearthstone) as well as the open-source clone of the game, Metastone.

1.3. Metastone

Metastone is an open-source re-implementation of Hearthstone created by Github user Demilich1 (2015). Demilich1 describes Metastone as 'a useful tool for card analyzation, deck building and performance evaluation'. Although it is not the only open source reimplementation of Hearthstone, it is the one with the most up to date card set, whilst also being the simulator that was used to generate decks by Pablo García-Sánchez et al.

1.4. Genetic Algorithms

Genetic algorithms are a specialised form of optimisation algorithms that can also be used to generate content over time. They will be referred to as *Genetic Algorithms* throughout this study.

Genetic algorithms generally work by randomly generating an initial population, where each item in the population is evaluated to see how well they meet some optimum criteria. They are then edited via a set of information exchanges to output a new population. This process has an end goal of producing a candidate that meets the optimum criteria set by the user.

1.5. Aims and Objectives

Pablo García-Sánchez et al presented a strong framework for the usage of evolutionary algorithms with Hearthstone.
In order to successfully manage the development phase and evaluation and analysis of our techniques, we have split the project into an aim and various objectives.

The aim of our project is:

To design and develop a genetic algorithm that has the capability to generate decks of varying archetypes, with the hope that they can gain effective win rates over human designed decks.

To achieve this aim, the following objectives have been identified:

- Conduct an extensive literature review of genetic algorithms and their uses, within the context of video games.
- Develop a program which will handle various functions, including constructing decks into their required formats, transfer of pre-game values and retrieval of post-game statistics.
- Develop a genetic algorithm with the ability to generate and evaluate decks over a series of generations.
- Conduct extensive testing of the decks by using AI-controlled players within an open source clone of the game Hearthstone.
- Evaluate the outcome of testing, to determine if card choices within the decks are sound, along with an analysis of different types of level of control on the algorithm, to see whether results change.
- Perform a play-by-play analysis of solution decks to determine whether the AI within Metastone has an effect on the outcome of fitness calculations.

1.6. Hypothesis

From the objectives specified in section 1.5, we can propose the following hypothesis:

Can a genetic algorithm generate decks for the game Hearthstone; that can effectively beat human created decks?

2. Literature Review

2.1. Aim of Literature Review

The aim of this literature review is to analyse genetic algorithms and their potential implementations. In order to do this, we will provide an introduction to the history of genetic algorithms, along with a general overview of how they function. This will allow us to evaluate how genetic algorithms have been implemented in previous years within game related applications.

2.2. Genetic Algorithms: An Introduction

Genetic algorithms fall under the umbrella of 'Evolutionary Computation', a grouping of optimisation algorithms that are inspired by biological evolution. They also come under the subset of algorithms known as 'Evolutionary Algorithms', which make use of more typical biological processes, such as mutation and reproduction.

In the context of optimisation algorithms, linear programming techniques are very good at solving problems where the constraints are clearly defined. For example, a simple problem such as finding a way to maximise profits in a company product line, given available resources. However, when the problem space becomes too complex, such as finding the best and most efficient shape of spacecraft antennae, computers are unable to find an accurate solution, due to the increased amount of potential constraints and combinations.

Genetic algorithms take inspiration from biological processes. This is visible when looking at the various terminology used in the field. For example, a string structure used to define the actual data (and eventual solution) is labelled as a 'chromosome', a vector of data which is edited over time is labelled as 'allele', and an individual vector entry within an allele is labelled as a 'gene'. (Goldberg, 1989) However, alleles can also be labelled as 'Individuals'.

The idea of mimicking how species change within nature through natural selection was first hypothesised by Alan Turing in his seminal paper 'Computing Machinery and Intelligence' (Turing, 1950). Turing proposed the idea of a program that would simulate the 'mind of a child', in that it would be educated over time, noting that its initial 'child' would not be a good solution, and that successor children would be checked to see if they were better or worse. He linked this idea to natural evolution with survival of the fittest.

It was not until the work of Professor John Henry Holland however that genetic algorithms gained popularity. His work, titled 'Adaptation in Natural and Artificial Systems' (Holland, 1992), introduced the theoretical foundations of evolutionary algorithms, which eventually became genetic algorithms as we know them today.

Over the years, numerous publications have been released, acting as guides and introductory texts to the field of evolutionary algorithms. Goldberg defined genetic algorithms as 'search algorithms based on the mechanics of natural selection and natural genetics'. He follows this by noting that genetic algorithms function by combining survival of the fittest along with a 'structured yet randomized information exchange', most notably being Reproduction, Crossover and Mutation (Goldberg 1989, p. 10).

In order for a new population to gain any kind of increase in the quality of results, a value needs to be defined for each individual. This value, also known as *fitness*, acts as a measure of some sort of profit that we want to maximise (Goldberg 1989, p. 10). Individuals with a higher fitness value mean that they meet the criteria of the solution better than others within the population, so they have a higher chance of being chosen for the reproduction stage.

2.2.1. Selection, Crossover and Mutation

In order for a genetic algorithm to provide good results, some core operators are required. The selection function is the first part of the reproduction cycle of a generation. During this stage, a number of individuals are selected from the current population for later breeding during the crossover stage. There are various different selection methods, with the most used methods being Roulette Wheel selection and Tournament Selection.

Roulette Wheel selection is a method known as *fitness proportionate selection*. This is where an individual's fitness value is used to assign a probability value of potential selection against other individuals in the population, as shown in Figure 1. This is usually completed by dividing a particular individual's fitness by the total fitness of the population, normalizing them to 1. As the name implies, it can be visualised similar to a casino roulette wheel, as seen in Figure 2.

TABLE 1.1 Sample Problem Strings and Fitness Values

No.	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

Figure 1 – Table of sample problem chromosomes and their assigned fitness values. The '% of total' column is the percentage of the total fitness value for each individual (Goldberg 1989, p. 11).

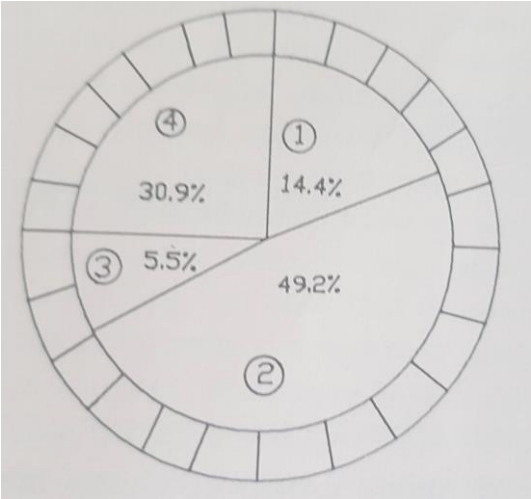

Figure 2 – A visualisation of the data from Figure 1. Each slice of the wheel represents the percentage for each individual, with bigger slices meaning a higher chance of being selected (Goldberg 1989, p. 11).

This 'wheel' is spun various times until a required amount of parents have been selected. Roulette Wheel selection is expanded upon by the selection methods of Stochastic Universal Sampling (SUS). This is a similar method to Roulette Wheel, however instead of one parent being selected, multiple fixed points are used which increases the amount of selected parents in one 'spin of the wheel'.

Methods that fall within the *fitness proportionate selection* umbrella do not work with fitness values that can become negative values, as you cannot assign a slot on the 'wheel' if its value is not above zero.

Tournament Selection is another highly used method. Here, a group of individuals are randomly chosen from the current population, where the selected individuals can also be duplicates of each other. The selected potential parents take part in a 'tournament', where the 'winner' is the individual with the highest fitness out of all the other participants. This process is repeated until the new population is full.

In the case where only two individuals take part, it is known as a *binary tournament.* This selection method can be implemented very efficiently, as no sorting of the population is required (Xie and Zhang, 2013). However, Binary Tournament Selection has the potential downside of not being guaranteed to produce the best solution when compared to other methods (Alabsi and Naoum, 2012, p. 5).

Tournament selection also has the choice of selecting the size of the tournament pool. A smaller pool will increase the chances of individuals with lower fitness being selected, which will increase population diversity, but can lead to slower convergence on the solution. Whereas, a larger tournament pool will lead to a lower chance of weak individuals being chosen, meaning a lower overall population diversity, and also an increased chance on converging on local maxima (Xie and Zhang, 2013).

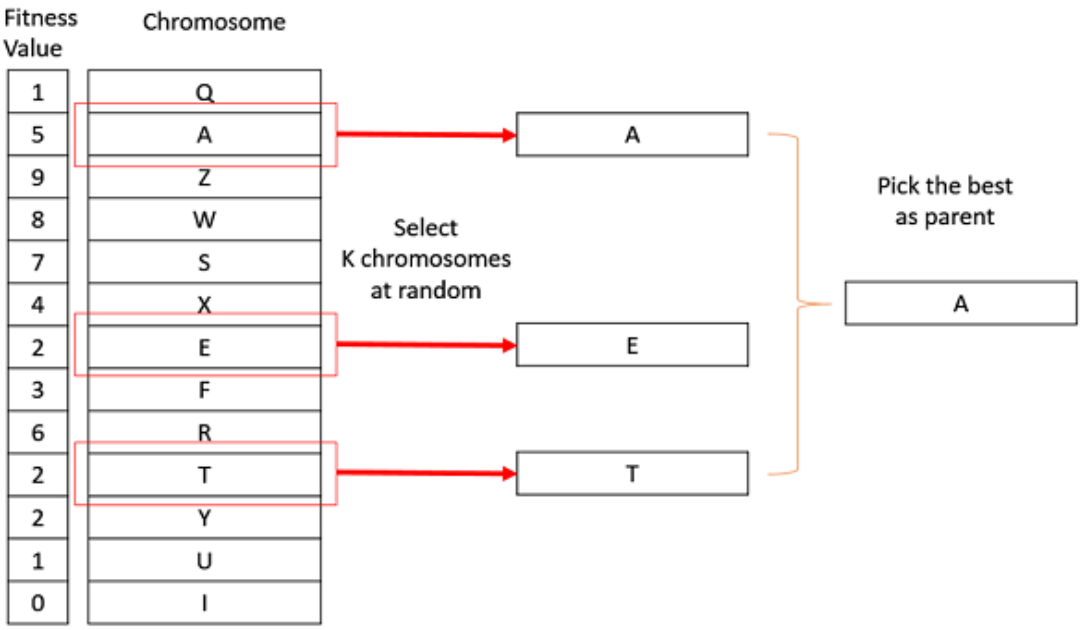

Figure 3 – Showcases the example steps for selecting a parent using tournament selection. K potential parents are chosen at random, and their fitness values are compared with each other (www.tutorialspoint.com, 2018).

The next required function for a genetic algorithm is Crossover. During this stage, two parents have their genes split at a randomly chosen point, and then recombined to produce a number of offspring individuals. Crossover is carried out in according to a user defined probability value, which is usually tested against a randomly generated number.

Single point Crossover is where a crossover point is randomly chosen for both parents. Every gene from the start of the chromosome to the crossover point is then copied from one parent, and every gene afterwards is copied from the other parent.

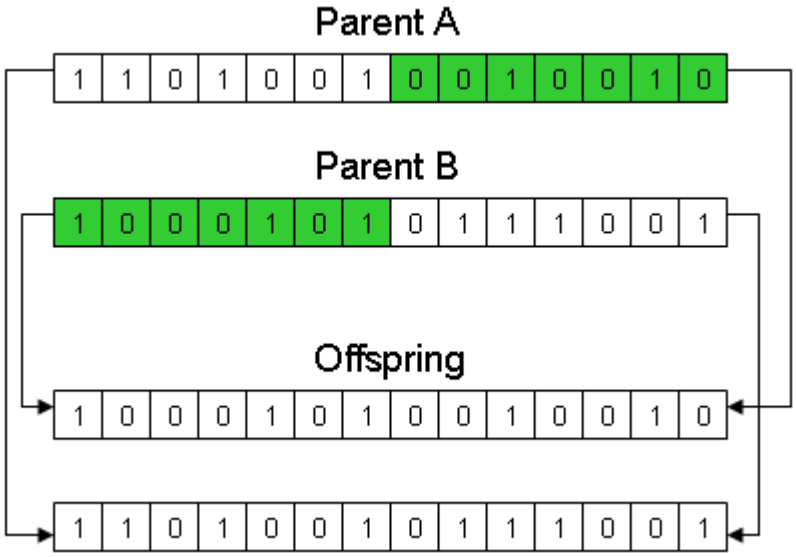

Figure 4 – Showcases the steps to perform the single point crossover operation on two selected parents. Parent A and Parent B have been split at identical points, which was chosen randomly. The Offspring strings have received the respective strings from both parents (McCullock, 2012).

Figure 4 shows a visualised example of a single point crossover operation. Here both parents are spilt at an identical randomly chosen point, and their genes are copied into each child, with each child being a reversed copy of the other.

An evolution of this type of crossover is *K-point Crossover*. This is similar to single-point crossover; however, *K* amount of identical points are randomly chosen within both parents. When more than one crossover point is chosen, genes are copied over in alternating turns, starting with the first parent, and then continuing with the second parent.

The final function required is Mutation. This operator, like crossover, makes changes to a chromosome's genes by randomly selecting one or more values, and then changing them in some way. Mutating a bit within a chromosome's gene is also selected based on a user defined probability value, which is checked against a randomly generated number. Goldberg (Goldberg 1989, p. 14) noted that the mutation probability should be around 1 in 1000 per bit, linking it to real life mutations and their secondary effect on evolution. He also noted that mutation is required as performing crossover and selection could potentially lead to loss of useful bits.

Crossover and Mutations are crucial in the evolution process, as they help prevent the potential problems of hitting local maxima and local minima in the problem space. These are known as the highest point and the smallest point within a range of values, respectively. As Figure 5 shows, without crossover and mutation, the genetic algorithms will begin to converge on a perceived global optimum solution where chromosomes will start becoming too similar to each other, and will lead to slowing of the evolution process.

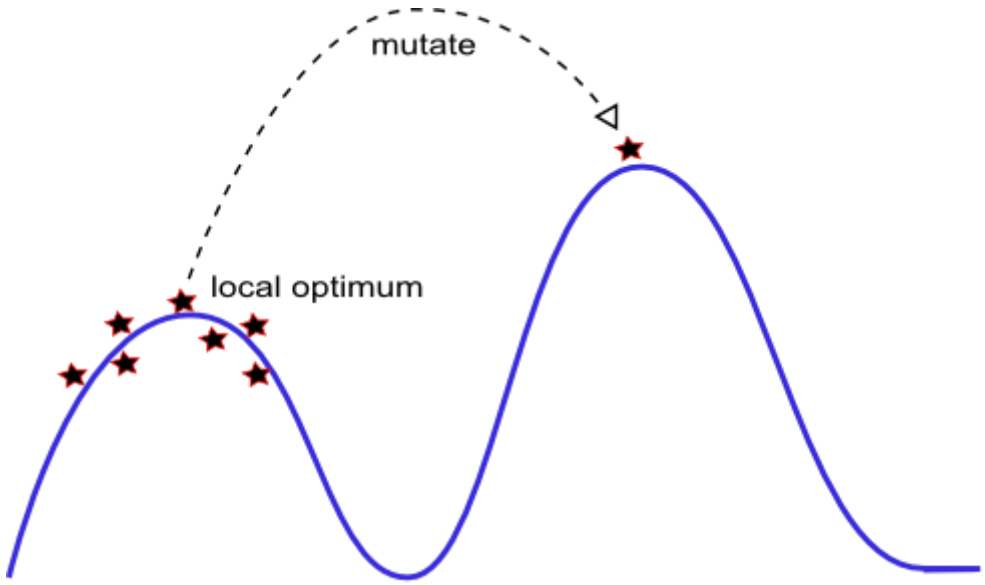

Figure 5 – A visualisation of an example search space for an optimisation problem. The example generation has reached local maxima, however a mutation has caused a new point that is closer to the best solution to be discovered (Li, 2016).

2.3. Application of Computer Generated Content in Video Games

The usage of procedurally generated content in video games has become an increasingly popular trend in recent years. The ability to generate almost every component of a game, from terrain, to items and characters with little to no human input, makes generated content a huge advantage to take. In this section, we will explore the main concepts of PCG content, and how they could affect our implementation.

Togelius et al (2016) proposed a taxonomy of PCG, where the type of generation is defined by a number of different properties. These properties will be shown below.

The first property is whether the content is *offline* or *online*. These terms are not to be confused with the case of playing with an internet connection, and instead refer to content generation before a game session and during a game session, respectively. In the case of offline generated content, complex game components such as maps and characters may be generated during development or before the game starts. In contrast, online generated content refers to content that is generated whilst the player is playing the game. This can refer to altering of enemy behaviour to reflect player progress, or procedural map generation over time.

The second property is two distinct differences for when generating content in games. These two differences are whether the content is *necessary* or *optional*. Necessary content is defined as the key content of a game that allows the player to continue progress in the game such as collecting specific items and the layout of levels. Optional content is self-explanatory, in that the content does not directly affect whether the player can progress through the rest of the

game. Optional content can come in various forms, such as the coins and mushrooms in the game Super Mario Bros.

Togelius et al (2016) continues on to note that Necessary content must always be correctly generated. If it were to be generated incorrectly, it could harm or outright stop the player from progressing in a game. Whereas optional content can be generated in a way where it provides no benefit to the player.

The third property is the level of control given to the PCG algorithm in question. This property carries two extremes. One, is that the algorithm uses a randomly generated seed number as the input, whilst the other is the use of a set of parameters. An algorithm with a completely random input will simply generate a unique seed for each piece of content it requires, whereas an algorithm with a set amount of parameters can be generated according to how the user sets the parameters. As an example, a dungeon generator may have options for number of rooms, enemies and weapons which the user can assign themselves, whilst a random seed will place a completely random amount.

The fourth property they defined is whether the generation algorithm is *deterministic* or *stochastic*. These terms essentially refer to the 'randomness' of the output of the algorithm, with more stochastic algorithms outputting content that would be hard to replicate, and deterministic being easier to replicate with the same input parameters.

The fifth property is whether the algorithm is described as *constructive* or *generate-and-test*. A constructive algorithm would generate content once, with testing done before it is deployed to make sure that it works properly. A generate-and-test algorithm will alternate between generating content and then testing that content to check whether it meets some sort of completion criteria. Here, the algorithm will loop, discarding and modifying candidates that fail to meet the success criteria, until it is complete.

The final property relates to how much input the algorithm receives from the user. Here, they have been split into *Automatic Generation* and *Mixed Authorship*. Automatic generation refers to more typical algorithms that take limited input from the user, with that input usually affecting the parameters of the algorithm itself. Mixed authorship, however is where the user has a direct influence on the generation process. The map creation framework Sketchaworld is an example of a mixed authorship algorithm, where the user first creates their own landscape by hand, then the rest of the map is generated whilst ensuring that it stays 'realistic', e.g. a bridge would automatically be generated over the top of a river where the two parts cross over each other (TNO, n.d.).

In addition to proposing an in depth taxonomy for PCG algorithms, Togelius et al (2011) also analysed 'search-based procedural content generation' methods, giving a detailed overview on evolutionary algorithms. Togelius et al (2011) continues on to outline three main methods of evaluating an individuals' fitness within a population:

- *Direct Evaluation*: This is where features of the individuals are mapped directly to the fitness function. For example, a maze generator may have a fitness criteria of a number of viable exits, which it tests the individuals against.

- *Simulation-Based*: This is where each individual is tested in a simulated environment of the problem space, and its performance is then used to calculate its fitness value.

- *Interactive*: This is where each individual is tested for fitness based on interaction with the player during actual gameplay.

2.4. Implementations of Genetic Algorithms within Video Games Environments

As previously mentioned in section 2.3, games are a very suitable environment to create procedural content generation algorithms. In this section, we will be analysing various examples that have been implemented with a genetic algorithm in mind, before focusing on examples that relate more closely to our problem space.

The game *invAIders* (Martian Face Games, 2011) released onto the Xbox Live Indie Game marketplace, is an action shooter where the goal of the game is to defeat all enemies to advance to the next level. The creator of the game, Michael Martin, proposed the usage of a genetic algorithm to provide an adaptive AI that alters its behaviour based on the playstyle of the player.

The potential search space is separated into two main components: the spawning parameters, and the behaviour of the enemies. The spawning parameters refer to the initial starting location of enemies, as well as the timings between spawns, which are both randomly generated and parameterised. The enemy behaviour is to determine what each enemy does after they have been spawned, which is visualised as a list of two behaviours: moving and shooting. These can range from simply moving and shooting in random directions, or a direction more relative to the player's location. This component has an input of a combination of random generation and parameters.

Martin goes on to define the fitness function for the algorithm, which makes use of the player itself to calculate fitness. The values themselves are calculated by measuring how many times a particular AI defeats the player, along with the average time spent alive, before eventually being destroyed. This fitness is then added cumulatively with the previous fitness values that were generated (Martin, 2011).

The reproduction process makes usage of the three main operators: selection, mutation and crossover. However, the process is extremely specific and defined for this problem space. Martin proposed the new population to be generated by taking children from one of each of the three operators, instead of using all of them to generate children. Here, selection generates three children, crossover generates one child, and mutation generates three children, before generating three new individuals at random to create the new population (Martin, 2011).

The selection method that was used is very rudimentary, as it simply selects the top three individuals based on their fitness values. The crossover operator works in an extremely specific way; as a child is generated by taking the list of behaviours from the individual with the highest fitness, and then combining that with the spawn parameters of a randomly chosen individual. The selection and crossover implementation here could be seen as a poor approach, as it can lead to the convergence to local maxima, as discussed in Section 2.2. The

mutation operator is also specifically defined for this problem space. Here, a random probability value is used to determine the number of potential changes to both the behaviour list and to the spawn parameters. The input individuals used are not ones generated from selection and crossover, but are instead direct copies of the best and second best individuals from the previous population (Martin, 2011).

Martin justified this unique take on the algorithm by noting that he did not want the player to become too adjusted to the difficulty of the AI, meaning a good solution would be unable to be found. He continued on to note that adding three randomly generated individuals per generation would enable the algorithm to escape this potential problem (Martin, 2011).

Nicholas Cole, Sushil J. Louis and Chris Miles (2004) proposed the use of a genetic algorithm that would be able alter the parameters of the state machines for the computer controlled bots in the first-person shooter game *Counter-Strike* (Valve Corporation, 2000).

The problem space that was identified was comprised of two separate components: the weapon selection parameters and the aggressiveness parameters. These parameters refer to the probability of selecting a particular type of weapon, and the overall playstyle of the AI, respectively. Each parameter is then encoded into a string of 178 binary bits, with the first 160 bits being each weapon selection probability, the next 15 bits being bot aggressiveness, and the final 3 bits being the usage of grenades (Cole, Louis and Miles, 2004).

The fitness of each individual is evaluated by checking the performance of each bot against a table of rewards and punishments. These can range from killing enemies to defusing the bomb, all the way to killing a friendly teammate and losing a round. These properties allowed fitness to be easily calculated, as individuals that would have been tuned towards poor gameplay standards, such as high aggressiveness, would receive a low fitness value.

Cole made use of unique variation of a genetic algorithm, known as CHC (Louis and Li, 1997). This differs from a genetic algorithm in a number of ways:

- The individuals with the highest fitness from both the parent and child population are selected and copied over, instead of using just the child population.
- If two selected parents are too similar, they are unable to reproduce.
- Mutation is only performed if the algorithm has converged on local minima or maxima.

The outcome of the generation process determined that bots tuned by a genetic algorithm performed very similarly to bots that are tuned by human players, noting that increasing the number of parameters available to the fitness function could produce wildly different results (Cole, Louis and Miles, 2004).

Although we are looking at using Hearthstone for our problem space, other card games have also been used in previous research, along with genetic algorithms. *Magic: The Gathering* (Wizards of the Coast, 1993) is a Trading Card Game that has a similar gameplay loop to Hearthstone, in that the main goal of the game is to reduce the opponent’s health to zero by making use of various minion and spell cards. Sverre Johann Bjørke and Knut Aron Fludal (2017) proposed the usage of a genetic algorithm to optimise the deckbuilding process, in the

hopes of obtaining decks that could be considered comparable to decks constructed by experienced players.

In *Magic: The Gathering*, there is a total of over 15,000 cards which are separated into five separate colours, white, black, red, green and blue. Cards are played by playing 'land' cards, which increase the number of particular mana the player has for a particular colour. When attempting to define their search space, Bjørke proposed the usage of one card set with a total card number of 194, whilst also opting to use the 'sealed' format, which is essentially where players construct their decks at random before the game starts. An individual's chromosome is defined as an array of integers, with each integer corresponding to a key in a hash map of all cards in their problem space (Bjørke and Fludal, 2017).

To perform their fitness calculations, Bjørke decided to make use an open-source AI engine of *Magic: The Gathering*, which would calculate an individual's fitness as the total win percentage of every match it played against an opponent.

Bjørke noted the use of tournament selection to handle the parent selection process, with the tournament pool being of size 3. When deciding on what type of crossover to use, Bjørke noted the potential downsides of using more traditional implementations, such as single and two-point crossover, justifying the decision by noting that crossover could potentially invalidate candidate decks. Instead, they opted to use their own crossover function, which first sorts the two parent arrays into one array, and would then generate child decks by copying all odd numbered card indexes into one child, and all even numbered indexes into the other. Mutation was defined as swapping cards at random points into any other card, until a swap was performed that would invalidate the deck, which would stop the mutation process. (Bjørke and Fludal, 2017).

When discussing their implementation, Bjørke highlighted that they did not see any of the generated decks reach the optimum win rate criteria, even when running for hundreds of generations. They attributed this to the low population size of 10, and the chosen selection method, which would lead to lack of exploration of the search space during the low end of the generation process. They also noted that the playstyle of the AI influenced the outcome of the decks, highlighting that the AI does not play the game like a human player would, such as remembering card synergies. They recommended a number of potential changes for future work, such as using adaptive operator parameters, and usage of a number of different types of AI to determine if decks are created based on the AI's preference (Bjørke and Fludal, 2017).

2.5. Conclusion

The aim of this literature review was to analyse and evaluate the overall implementation of genetic algorithms, along with various real life example applications. With this aim in mind, we have given a general overview of the functionality of the operators applied to genetic algorithms, alongside an exploration into various examples of both traditional and problem specific genetic algorithms, both which are completely separate from our own problem space, and some that correspond more closely.

3. Methodology

3.1. Problem Space Identification

Our potential problem space is very expansive. As of its 1.3.0 release, Metastone has over 700 cards that can be used. This combined with the 30 card deck limit means that a potential solution could take a while to generate. Each deck can have a maximum of two copies of a particular card, and any card with a rarity value of 'legendary' can only have one of each card. To simplify our problem space, we are going to be focusing on the 'Ranked' version of Hearthstone, which will limit our amount of cards to around 690.

In Hearthstone, decks are built up of 30 cards of the player's choice, usually containing a varied amount of minion and spell cards. Many decks in Hearthstone fall under 5 different deck archetypes, which defines the overall playstyle of the deck, and what cards are included inside the deck itself. Along with these three archetypes, there are various sub-archetypes that have more defined strategies. Table 1 below shows some of the various playstyles that have been realised, either by the developers of the game or players themselves.

Deck Archetype	**Overview**
Aggro/Aggressive	High focus on playing aggressively by dealing damage to the opponent as quickly as possible, through a large number of low cost cards.
Control	A main focus on keeping control of the game board, with the main goal of winning towards the end of the game with higher costing cards.
Midrange	A combination between aggro and control, focusing on controlling the game board early, and then becoming aggressive during the midgame.
Combo	Revolves around playing a specific combination of cards, to produce an overwhelming advantage over the opponent.
Zoo	Focuses on a large amount of low cost minions to gain board control early.
…	…

Table 1 – List of potential deck archetypes. Full table in Appendix 7.1.

In Hearthstone, cards are separated into two distinct types: *neutral* and *class*. Neutral cards make up the bulk of the cards within the game, and consist of a variety of minions of varying mana costs. Some cards within this type have special properties that provide the card with an extra ability during gameplay. For example, some cards may have the 'Taunt' ability, which causes all minion and hero attacks to attack that particular minion first.

Class cards consist of a mixture of minion and spell cards that provide special effects for that particular hero, or cards that exclusively interact with one another. For example, the Shaman hero has 'totem' minion cards that have special interactions with other cards, whilst also having cards with the 'Overload' special effect, which causes the player's mana to temporarily be reduced next turn.

Both neutral and class cards receive expanded features with the release of various downloadable content for the game. For example, the expansion 'League of Explorers' released various neutral and class cards with the 'Discover' ability, which presents the player with a choice of one of three randomly generated cards. Table 2 below gives an overview into the unique mechanics available to each hero, whilst Table 3 gives a general overview of the various abilities available to both neutral and class cards.

Class	**Unique Mechanics**
Druid	Has spell cards with the 'Choose One' ability, giving a choice of various effects on a card.
Hunter	Revolves around 'beast' minions that cause interactions to occur from other cards.
Mage	Contains the largest amount of spell cards, along with minions with the 'Spell Power' ability, which increases spell damage by a certain number.
Paladin	Focuses on cards with 'Divine Shield' abilities, causing them to take at least 2 instances of damage to be destroyed.
Priest	Has a strong focus on healing and damage dealing spell cards, but also has access to cards with 'Mind Control' abilities.
Rogue	Has cards with the 'Combo' ability, meaning that playing subsequent cards beforehand triggers an effect.
Shaman	Focuses on cards with the 'Overload' ability, in that you make the choice of playing a stronger card whilst weakening yourself for the next turn.
Warrior	Has a strong focus on 'weapon' cards that allow the hero to attack targets directly. Also has a focus on gaining the temporary health stat 'Armour'.
Warlock	Revolves around 'Demon' cards that deal damage to the player's hero to gain low cost advantages over the opponent.

Table 2 – Short overview of each of the selectable hero classes in Hearthstone.

Card Ability	**Description**	**Neutral**	**Class**
Battlecry	Triggers an effect when the card is played.	Yes	Yes
Charge	Allows the minion to attack as soon as it is played.	Yes	Yes
Choose One	Let's the player choose from 2 unique selections that trigger an event.	No	Yes
Combo	Triggers an event based on whether a card was played beforehand.	No	Yes
Deathrattle	Triggers an event when the minion is destroyed.	Yes	Yes
Discover	Allows the player to choose one of three randomly generated cards, and draw it.	Yes	Yes
Divine Shield	Causes the first instance of damage to be nullified, which also removes the shield.	Yes	Yes
Freeze	Freezes the target for one turn, making them unable to attack.	Yes	Yes
…	…	…	…

Table 3 – Card abilities in Hearthstone. Full table in appendix 7.2.

With the knowledge of these two previous tables, we can take educated guesses on what type of decks suit each class, and what type of cards would usually be included in them.

To further simplify our problem space, if we limit generation of decks to one class type, we can further limit the card space to just the class specific cards and 'neutral' cards, i.e. cards that can be used by any hero.

3.2. Technical Overview

Our program can be split into three separate components: our Windows Form program, the genetic algorithm, and Metastone itself. The Windows Form program will act as the Graphical User Interface for easy usage of the genetic algorithm. It will also handle all functions relating to cards and decks, as well as exchanging data with Metastone, and assembling and generating results, both for individuals and per generation.
The genetic algorithm will be a part of the form application, and will only be used during the generation process.

Metastone will need to be edited slightly for a few reasons:

- We need to retrieve post game statistics which will be used to calculate an individual's fitness.
- We also need to exchange the corresponding current opponent index and player index.
- Finally, we need to edit Metastone to automatically run the simulation process, and then exchange data we need without user input.

3.3. Cards and Decks File Structure

Before we can begin with our entire process, we need to first populate our complete card list into a usable format for our program. In Metastone, cards are stored as JavaScript Object Notation (JSON) files. Each card has a collection of name/value pairs that act as basic properties that are required in all others.
JSON files are constructed/deconstructed by two separate functions: *serialisation* and *deserialisation*; Serialisation is 'the process of translating data structures into a format that can be easily stored and transmitted to another location', with deserialisation being the opposite process (Isocpp.org, n.d.).

In Metastone, each card shares a number of basic properties which all cards require to be valid, as shown in Table 4.

Property	**Usage**
Id	A unique identifier for each card.
Name	The card's actual in-game name.
baseManaCost	The total mana cost of the card.
Type	A card can either be a minion or spell card.
baseAttack	The attack value of the card.
baseHp	The health value of the card.

heroClass	Which hero class this card belongs to. Can either be a specific class, or 'neutral', where it can be used with any class.
Set	The card set that this card belongs to. This denotes whether this card is accessible from within the Ranked or Wild game modes.

Table 4 – Required JSON properties for each card.

The other properties are unique to this particular card, and not useful for our program. For our program, we only require the id, baseManaCost, heroClass and rarity properties. The reasons for this will be discussed further on.

Figure 6 below shows how a card is implemented for use in Metastone. In this example, it shows the 'Fire Elemental' minion card for the Shaman hero class. This card shares the properties discussed Table 4, but also has extra properties which are unique to this particular card.


```
{
	"id": "minion_fire_elemental",
	"name": "Fire Elemental",
	"baseManaCost": 6,
	"type": "MINION",
	"baseAttack": 6,
	"baseHp": 5,
	"heroClass": "SHAMAN",
	"rarity": "FREE",
	"description": "Battlecry: Deal 3 damage.",
	"battlecry": {
		"targetSelection": "ANY",
		"spell": {
			"class": "DamageSpell",
			"value": 3
		},
		"resolvedLate": false
	},
	"attributes": {
		"BATTLECRY": true
	},
	"collectible": true,
	"set": "CLASSIC",
	"fileFormatVersion": 1
}
```


Figure 6 – A JSON implementation of the 'Fire Elemental' card for the Shaman hero (Demilich1, 2015).

Like cards, decks are also stored as JSON files. Decks are stored as an array of card IDs within a 'cards' object, followed by three name/value pairs: the name of the deck, the particular class type, and an arbitrary value. This can be seen in Figure 7 below.

```
{
  "cards": [
    "minion_tunnel_trogg",
    "minion_tunnel_trogg",
    "spell_finders_keepers",
    "spell_finders_keepers",
    "spell_lightning_bolt",
    "spell_lightning_bolt",
    "minion_southsea_deckhand",
    "minion_southsea_deckhand",
    "minion_patches_the_pirate",
    "spell_ancestral_knowledge",
    "spell_ancestral_knowledge",
    "minion_totem_golem",
    "minion_totem_golem",
    "spell_lava_shock",
    "spell_lava_shock",
    "weapon_jade_claws",
    "spell_rockbiter_weapon",
    "spell_rockbiter_weapon",
    "minion_bloodmage_thalnos",
    "spell_lava_burst",
    "spell_lava_burst",
    "spell_feral_spirit",
    "spell_feral_spirit",
    "spell_lightning_storm",
    "spell_lightning_storm",
    "minion_argent_horserider",
    "minion_argent_horserider",
    "weapon_doomhammer",
    "weapon_doomhammer",
    "minion_leeroy_jenkins"
  ],
  "name": "Aggro Shaman",
  "heroClass": "SHAMAN",
  "arbitrary": false
}
```


Figure 7 – An example Shaman deck. The card list is stored as an array of card IDs, with the name and class stored as name/value pairs below (Demilich1, 2015).

3.4. Card and Deck Construction

To handle all JSON functions, deck construction and card storage, we will use a Manager class which will handle all required functionality. This class will follow the Singleton pattern (Msdn.microsoft.com, n.d.), as we only need to store the entire card list once. We will also need to handle deserialization of each JSON file, to be able to retrieve the required data. For this, the NuGet package 'Json.NET' (Newtonsoft, 2018) will be used to enable access to JSON deserialisation/serialisation functions.

3.4.1. Card Construction

As we already have a suitable data structure for our cards, we only need to deserialise the cards required for our algorithm. However, due to the various hero classes and their exclusive cards, we need to have multiple data structures to store each respective class' cards. To save

development time, Json.NET has a 'JObject' class which can parse a string that has been read from a JSON file. It also allows us to retrieve data at each name/value pair.

A pseudocode implementation of our actual implementation is visible in Figure 8 below.

```
for all card directories
    get all card JSON files in current directory
    for all current card JSON files
        read all text in file
        Parse the text into JObject
        retrieve all card properties
        if card ID is null
            card ID equals card file name
        assemble retrieved properties into Card class object
        check current card class type
            add card to respective card list
```

Figure 8 – Pseudocode implementation of the GetAllCards function.

Due to the way Metastone has implemented its newer cards, some cards have no ID name/value pair which we can parse correctly. Because of this, we set the ID to the file name, which coincidentally follows the same format as the ID would.

3.4.2. Deck Construction

Many decks that a player of Hearthstone may create usually contain a varied mixture of both neutral and class cards, due to the fact that some cards that may synergise well with more specific cards. It also reduces the chance of initially generating invalid decks due to the small class card count per class, relative to the neutral card count. Because of this, when generating the initial population, we will randomly choose between whether to retrieve a class card or neutral card.

Figure 9 below shows a pseudocode implementation of the GenerateSpecificDeck function, which generates a single deck of specified class.

```
while index is not at max card count
    if index equals max card count
        if random number is greater than card type probability
            grab neutral card
        else
            grab class card
        add final card to deck list
        validate deck
        for all cards in deck list
            add each card ID to deck string
        assign deck name
        assign deck class type
        assemble final deck with previous variables
        generate deck into JSON format

    if index is not equal to max card count
        if random number is greater than card type probability
            grab neutral card
        else
            grab class card
        add card to deck list
        increment index
    return generated deck
```

Figure 9 – Pseudocode Implementation of GenerateSpecificDeck function.

Our GenerateSpecificDeck function is of type Tuple<string, List<Card>>. We will use this type as it allows us to return both the deck list which will be used by the genetic algorithm, and the string version which will be used by Metastone.

3.4.3. JSON Serialisation of Decks

The final function we need is one that serialises the deck string into a JSON file which we can pass over to Metastone. JSON.Net has a 'JsonTextWriter' class which handles creating JSON files from various inputs. Using this, we can easily create a JSON file at a specified target path, which will be Metastone's deck directory.

3.5. Automation of the Generation Process

Although not necessary, the obvious advantages of automating the entire generation process outweighs the extra time spent developing the ability to do so. In order to automate our process, we need to make changes to both Metastone and our own program.

3.5.1. Information Exchanges with Metastone

To be able to begin with the automation process, we have to perform some data exchanges with Metastone and our program. We need to tell Metastone what hero class our genetic algorithm deck is, and what the current opponent hero class is. We also need to retrieve the post-game statistics after every 16 games per opponent. These statistics will then be added to total values for the actual values we need, which will then be used to calculate an individual's fitness.

To be able to transfer data between our program and Metastone, we first have to look at how Metastone communicates with other components of the program. Metastone uses its own

event system to let each window know what wants to be run next. These events will cause constructors of certain windows to be called, which then calls all other functions that are needed for that particular window. This means that we cannot create a new function that calls one that is already there, unless we call it from a function that is handled by an event.

Due to this, we need to look at how the event system works as a whole, and then make changes to it for our needs.

Metastone's custom event system is constructed as follows:

- An enum of event identifiers.
- A proxy class that handles sending and receiving of events.

The proxy class is then initialised in each respective window that Metastone has, with their own respective event messages. Figure 11 below shows an example usage of Metastone's event system.

```
case SIMULATION_RESULT:
        Platform.runLater(new Runnable() {
                @Override
                public void run() {
                        waitView.getScene().getWindow().hide();
                        SimulationResult result = (SimulationResult) notification.getBody();
                        resultView.showSimulationResult(result);
                        getFacade().sendNotification(GameNotification.SHOW_VIEW, resultView);
                }
        });
        break;
```

Figure 11 – An example usage of Metastone's event system. Here, the 'SIMULATION_RESULT' event is being called, meaning it will get and hide the currently running window, show the simulation result window, and then send the event 'SHOW_VIEW' to actually update the view.

To retrieve the post-game statistics, we need to output the data into a text file that our program will be able to parse. The 'SIMULATION_RESULT' event is used within the 'SimulateGamesCommand.java' file, where luckily Metastone calculates the statistics for us, meaning we only need to write the output. The statistics are also stored inside one string variable, further decreasing the difficulty of the task. Figure 12 below shows how Metastone was edited to handle our requirements.

```
private String filePath = "";
//
// Output game stats to a text file
// luckily game stats are created as a long string
public void WriteStatsToFile(String textLine)
{
        filePath = System.getProperty("user.dir");
        filePath = filePath.concat("/test.txt");

        try
        {
                File path = new File(filePath);
                FileOutputStream fs = new FileOutputStream(path);
                OutputStreamWriter osw = new OutputStreamWriter(fs);
                Writer w = new BufferedWriter(osw);
                w.write(textLine);
                w.close();

        }
        catch(IOException e)
        {
                //error
        }
}
```

Figure 12 – 'WriteStatsToFile' function within 'SimulateGamesCommand.java'. This takes the post-game statistics string and outputs it to a text file, which will be read by our program.

Metastone's player configuration windows make use of various ComboBox components, meaning we only need to edit the starting index for each ComboBox to get our desired outcome.

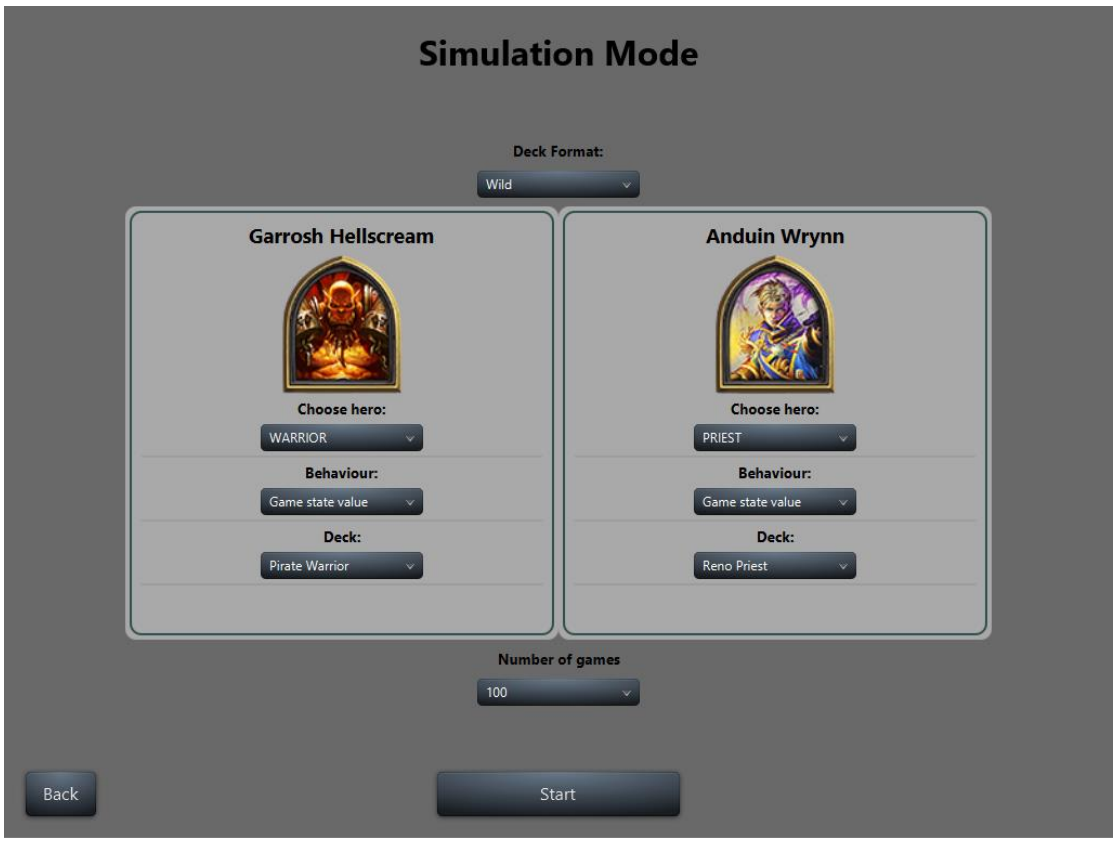

Figure 13 – Example view of the Simulation Mode Selection.

As Figure 13 shows, there are four separate ComboBox components that concern us:

- The hero class, which we need to select our player and opponent.
- The AI of each player, which will determine how deck will be played.
- The deck of each player, which we need to edit to use the correct decks.
- The number of games to simulate.

We can send over the index of the player and the index of the opponent via a text file, with each value corresponding to the index of that particular hero class they are playing. For example, the Warrior hero class has an index of 0, whilst the Warlock class has an index of 8. This means that we can construct a simple for loop in our program that will increment to the next opponent for each game.

When choosing the player AI, there are multiple options to choose from.

- Play Random: Every action per turn is selected randomly.
- Greedy Best Move: Every action is scored and the highest scored action is played.
- No Aggression: The player will draw cards and essentially do nothing every turn.
- Game State Value: Adds scores to moves based on various properties, such as the amount, health and modifiers on minions on the board. These scores are then weighted with potential actions to take.

When discussing their implementation, García-Sánchez et al discovered that, after simulating 11520 games, that the 'Greedy Optimize Turn' AI outperformed the other options, with a winning percentage of 37.5%. However, as they were using a previous version, most of their choices have been omitted from the version we are using. Metastone creator Demilich1 describes the Game State Value AI as the most sophisticated option (Github, 2015). Because of this, and comparing it to how the other options function, we will be choosing this option.

As we are going to have the player simulate 16 games per opponent, we only need to add an extra entry to the total list of selectable games to play.

3.5.2. Automating Command Line Commands

The main component of our automation process is being able to run commands from the command prompt programmatically. To do this we can use a C# Process class object. This class lets you run local system processes, including executable files.

In our current implementation, we use a Process object to run an instance of the Command Prompt. We then input two commands: one to access the directory of Metastone, and one to run the command 'gradlew run'. This will build the Metastone project, run one instance of it, and then close it. Calling the 'WaitforExit' function on the Process object will stop our program from opening another instance of Metastone, and disrupting the flow of the program. This functionality is all stored within a single function, which is called for every individual per 16 games.

3.6. Required Helper Functions

During the development of our program, there were various functions that we developed to help the generation of the solutions. Here we will discuss some of the most prominent functions.

3.6.1. Deck Validity Checks

One highly important feature is that our decks are valid after being generated. As a results, we require a function that checks the validity of the decks, both during the generation of the initial population, and when testing each individual. The following conditions that mark a deck as invalid are as follows:

- If the deck has more or less than 30 cards in total.
- If the deck has more than 2 of the same card.
- If the card has more than 1 of the same card of legendary rarity.
- If the deck contains any cards that are not the same hero class as the chosen hero class.

It should be noted that the conditions above only apply to how decks are constructed prior to commencing a game session. During normal gameplay, Hearthstone allows players to obtain cards that would be considered illegal during the deck construction process.

These conditions should be self-explanatory, as this is also how deck validity is checked with Hearthstone itself. A deck within Hearthstone has to have exactly 30 cards, otherwise the player would be at a disadvantage if they had less, and at an advantage if they had more. This is a simple check of each individual's card count.

As explained previously, letting a player have more than two of the same card, or more than one of a legendary card would also place the player at an advantage over the other. To calculate these particular conditions, we iterate over every card in the list, incrementing a counter for every extra duplicate card. We then gather every 'invalid' card into a list and remove the unnecessary duplicates from their respective card sets. We do this so during the mutation process, we do not mutate into an unnecessary duplicate card and invalidate the deck.

The final condition is not handled within our validity checking function. This is because we store each class' respective cards within their own data structures, meaning we cannot accidentally choose an invalid card.

These validity checks are performed during both the generation of the initial population, and the mutation process, to reduce the chances of generating invalid decks, both at the start and during the overall running of the program.

3.6.2. Resuming the Generation Process after Fatal Errors

One vitally important function we needed was one that provided us the ability to continue the generation process if the program itself was to stop working. Without this function, if our program was to fail towards the end of the generation process, we would lose out on the data generated up to the failure point, along with having to start over from the beginning.

During the main running of the program, the current population that would have been tested in Metastone is not stored in an external file. Because of this, we generate a new population set from the previous completed population. While this means we can generate a new population and continue generating as normal, it also means we discard the previous in progress population.

3.7. Genetic Algorithm Implementation

Our Genetic Algorithm is based upon the methodology of Pablo García-Sánchez et al (García-Sánchez et al, 2016). In this section, I will outline each section of our genetic algorithm, along with how they compare against García-Sánchez's implementation.

A general overview of the structure of the algorithm can be seen below in Figure 14.

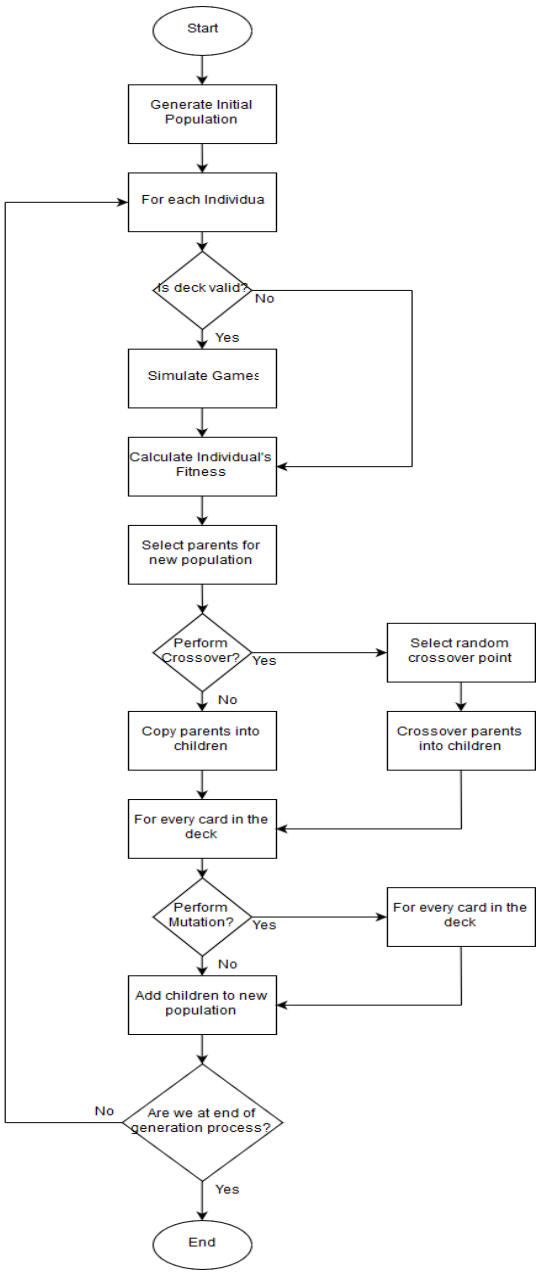

Figure 14 – Flow chart illustrating the structure of the genetic algorithm.

3.7.1. Categorisation of our Algorithm

As seen in section 2.3, Togelius et al (2016) outlined various properties that help categorise procedural content generation algorithms. Taking these properties into consideration, we can begin to analyse our own implementation, and begin to see which properties line up with our algorithm. Our potential problem space is the full amount of neutral cards and a particular hero class' class cards, for the Ranked game mode in Hearthstone. The optimum solution we are looking for is the individual with the highest total wins, lowest standard deviation, and that it is not illegal. These three values are calculated after simulating 144 games per individual.

As decks in Hearthstone are created before you play a game against an opponent, our decks can be seen as *offline* content, as you cannot start playing without creating a deck beforehand.

Following on from this, the decks can be identified as *necessary* content, as the main purpose of the game is to use decks to defeat the other opponent's deck.

In terms of the level of control of our algorithm, we appear to be closer to the extreme of using a set of parameters for the input. In our algorithm, we are able to select the hero class to use as a base for the deck, and we are also able to edit the probabilities for mutation and card type selection.

Although our level of control is quite constrained, our algorithm sits more in the *stochastic* end of the spectrum. This is because the decks are generated randomly initially, and then further mutated and crossed over randomly.

As the general idea of genetic algorithms is to improve on candidate solutions over time, we can easily add our algorithm under the *generate-and-test* property as our decks are not going to meet our solution criteria until later on in the generation process.

Finally, our algorithm fits under the *automatic generation* category as our algorithm only takes limited input from the user, with that input affecting the algorithms main parameters.

3.7.2. Initial Population Generation

Pablo García-Sánchez's implementation of their algorithm is similar to our own in the context of generating the initial decks. Both of our implementations generate initial decks at random. As they did not explicitly say whether they check for deck validity both before and during the deck creation process, we can only assume their implementation is more rudimentary as they only noted that they check for validity during the crossover and mutation process. This is different to our implementation where we check for validity when the deck is first generated, and then fix the errors in the deck before testing it in Metastone.

3.7.3. Fitness Calculation

As with the previous section, our fitness evaluation is similar to Pablo García-Sánchez et al (García-Sánchez et al, 2016) methodology. Pablo García-Sánchez calculates three separate fitness values lexicographically; the validity of the deck, the total wins after playing against all opponents, and the standard deviation of its number of wins against each opponent.

These three fitness values are a good fit for our algorithm, as we only want to find decks that obtain a high number of wins, along with a consistent amount of wins.

The fitness function in our algorithm works by first checking whether the deck legality is equal to 1. If it is, its fitness is no longer calculated and its other values are set to 0. If it is legal, we then calculate its total amount of wins, along with the standard deviation of the number of wins between opponents.

As previously discussed in Section 2.3, Togelius et al (Togelius et al, 2011) defined three separate categories for how candidate fitness is evaluated. Our fitness evaluation firmly sits

within the *Simulation-Based* category, as we make usage of Metastone to evaluate the performance of the decks.

3.7.4. Parent Selection

Our selection method is also fairly similar to Pablo García-Sánchez's function. Here, Pablo García-Sánchez proposed the use of the tournament selection method, with the tournament size between 2 and 4. Although they did not explicitly say which tournament size they decided to use, we can assume that they decided to use a size with higher selection pressure, as their overall results showed a rapid increase in fitness in a relatively short amount of generations.

The same implementation is proposed for our own selection method, however in order to see whether the quality of results change significantly based on selection pressure, we will have the tournament size be a selectable value by the user.

3.7.5. Crossover

Pablo García-Sánchez's implementation is similar as they also use single-point and two-point crossover, however they did not define which one they use.
In our implementation, we use both single-point crossover and two-point crossover. Crossover is performed according to a probability value, which if met, swaps parts from two parents into children. If crossover points are invalid, i.e. they are at the beginning or end of the deck, we reselect another point.

As our decks are sorted alphabetically, we should be mitigating the potential probability to create invalid decks through crossover as duplicate cards should be grouped next to each other.

3.7.6. Mutation

Pablo García-Sánchez proposed the use of a mutation algorithm where a single random card is chosen and then mutated into another random card. This process is then repeated depending on a defined probability value. They also proposed that their mutation probability decrease over time, beginning with a higher probability to increase exploration of the search space, and a lower value towards the end to increase exploitation of individuals with high fitness (Sourceforge.net, 2016).

Our proposed implementation is similar to Pablo García-Sánchez's version; however, we also make use of the potential changes that they noted after analysing their methodology. They proposed that making the mutation function make 'smart' changes, based on particular card values such as card cost, relating the idea similar to how human players may choose alternate cards that are not too dissimilar to the card they were planning to include (García-Sánchez et al, 2016).

Because of this potential change, our implementation mutates random cards a number of times, with the chance of either selecting a random card or a card of similar cost dependant on a probability value. We select either a card of higher cost or lower cost of the current card. If this is not possible, we select a card of random cost.

A pseudocode implementation of this process is shown in Figure 15.

```
get the current deck's duplicate cards
while we can still mutate
    if random number is less than mutation probability
        select a random card to mutate
        if random number is equal to select card of similar cost probability
            get current card's mana cost
            set card lower cost
            set card upper cost
            if current card's cost is too low
                set lower cost to 1
            if current card's cost is too high
                set higher cost to 10
            if random number is less than 0.5
                if random number is less than type selection probability
                    get random class card of lower cost
                        if no available card
                            choose random card
                    else
                        get random neutral card of lower cost
                            if no available card
                                choose random card
            else
                if random number is less than type selection probability
                    get random class card of higher cost
                        if no available card
                            choose random card
                    else
                        get random neutral card of higher cost
                            if no available card
                                choose random card
            replace current card with new card
        else
            replace card with random card
```

Figure 15 – Pseudocode implementation of our mutation function.

3.8. Content Evaluation Methods

In order to evaluate our final generated decks, we need to collect data from each generation. To be able to see how our generations changed over time, we will collect the average total wins and standard deviation across all 10 individuals, along with the individuals with the highest and lowest fitness. For these two individuals, we will also be collecting their decks, to be able to analyse the cards that were chosen.

In order to judge the functionality of our genetic operators, we will be performing four separate tests:

- The first test will generate a Shaman deck, with a mutation rate of 0.5, crossover rate of 0.7, a tournament size of 4, single-point crossover, population size of 10, and a generation count of 50. The mutation rate is decreased by 0.01 per generation.

- The second test will generate a Warlock deck, with a mutation rate of 0.5, crossover rate of 0.7, a tournament size of 2, single-point crossover, population size of 10, and a generation count of 50. The mutation rate is decreased over time by 0.01 per generation.

- The third test will generate a Mage deck, with a mutation rate of 0.02, crossover rate of 0.7, tournament size of 4, two-point crossover, population size of 10, and a generation count of 10. The mutation rate is not decreased over time.

- The fourth test will generate a Paladin deck, with a mutation rate of 0.02, crossover rate of 0.7, tournament size of 6, two-point crossover, population size of 10, and a generation count of 10. The mutation rate is not decreased over time.

To be able to evaluate our decks, we need to select a set of opponent decks to pit each individual against. For our testing, we will be making use of cards from the default cards available to all players, along with the following expansions: *Whispers of the Old Gods, One Night in Karazhan,* and *Mean Streets of Gadgetzan.* With these cards in mind, we will create nine different decks, one for each hero class, with each one representing a deck archetype, as discussed in Section 3.1. Table 4 below outlines each opponent deck, giving an overview of the overall expected strategy for each.

Hero Class	**Deck Type**	**Overview**
Druid	Minion Based	Focus on summoning 'Jade Golems', which gain increased stats for each newly summoned golem.
Hunter	Midrange	Has a main goal of using minions that boost the stats of held cards to summon extremely dangerous minions.
Mage	Tempo	Should focus on keeping board control with 'Mana Wyrm' minions, along with high spell usage.
Paladin	Minion Based	Has a main focus on 'Murloc' cards, which would help provide cheap minions early, with them eventually becoming boosted.
Priest	Combo	Focuses on playing minions with the 'Can't Attack' ability, and then using cards that perform the 'Silence' Ability to turn them into cheap and dangerous minions.
Rogue	Combo + Control	Revolves around the legendary card 'C'thun', which receives boosts from other minions in the deck, with the main goal of summoning C'thun in the late game for a quick win.
Shaman	Zoo + Combo	Mainly functions by using 'evolve' cards, which transform minions into ones of higher cost. The idea is to gain early board control, then evolve to gain a full board of high damage minions.
Warlock	Zoo	A main focus on flooding the board with cheap cards, and keeping the board under control with various spell cards.
Warrior	Aggro + Minion Based	Has the main goal of dealing as much damage as possible in the lowest number of turns, but also revolves around 'Pirate' cards, which provide synergies with each other.

Table 4 – List of opponent decks with their corresponding deck types and preferred strategies.

The goal of the first two tests is to see the difference of results when keeping the process as random as possible, to see if quality candidates are generated without user interference. We will also be testing whether increasing selection pressure, and including a mutation rate that changes over time helps increase the quality of results.

The third test will be to see whether including a varying card type probability will affect the overall deck structures. This probability will be calculated after counting the number of class and neutral cards in each deck. If there is an abundance of a particular type of card, we heavily weight the probability to choose the other type of card. With these tests, the hope is that the final candidates will have more decks with an even spread of the various type of cards that are available.

The final test will be to analyse whether including a varying probability for when to select between minions or spells will also affect overall deck structures, along with the varying card type probability mentioned previously. This probability will be edited depending on the amount of minion/spell cards in the deck.

We will also be analysing each deck from each test which has the highest fitness. We will be evaluating card choices in the deck, and will be seeing whether cards chosen have any synergies, or are considered good cards to choose.

In addition to collecting data from testing, we will also be evaluating whether the AI we chose within Metastone could have an effect on the outcome of testing. During the discussion of their implementation, Pablo García-Sánchez et al proposed that doing play-by-play analysis of selected decks, with the hypothesis that the decks that were being created were influenced by the AI that was chosen, and how the AI itself performed (Pablo García-Sánchez et al, 2016). This was also noted by Bjørke et al (2017) as mentioned previously in Section 2.4, discussing that a potential influence on the final decks is the playstyle of a particular AI, and that it requires further testing.

Due to this, we will be analysing a series of games per deck for the four tests we set up earlier. During this analysis, we will be evaluating how the AI itself plays the decks that were generated, with the hope that the AI makes accurate choices based on the cards within the deck. Although we are not expecting the AI to play as well as a human player, if the AI were to make poor choices on the cards it has at a particular time, we can start to identify whether the AI has a potential influence on the fitness evaluation stage.

4. Results and Analysis

4.1. Results

Throughout this section, we will discuss and give an overview of the four different tests we performed, which we took over the course of several days. For each deck, we will also be analysing the cards chosen in the deck, the mana curve of the deck, i.e. the total number of each card cost in the deck, and an analysis of how the Metastone AI played the deck against an AI opponent. We will also be analysing which archetype each deck could fit into, as discussed in Section 3.1.

4.1.1. Deck 1 – Shaman

The first deck was generated with a mutation probability value that decreases over time, with a maximum generation count of 50. At the 50th generation, the final deck that was generated included a large amount of neutral minion cards. Table 6 below shows the deck list for the Shaman deck with a fitness value of 26.

Card Name	**Rarity**	**Mana Cost**	**Card Ability**
Arcanosmith (N)	Common	4	Battlecry + Summon
Auctionmaster Beardo (N)	Legendary	3	Reset Hero Power
Ancient of Blossoms (N)	Common	6	Taunt
Big Game Hunter (N)	Epic	5	Battlecry + Destroy
Big-time Racketeer (N)	Common	6	Battlecry + Summon
Bomb Squad (N)	Rare	5	Battlecry + Deal Damage + Deathrattle
Cairne Bloodhoof (N)	Legendary	6	Deathrattle + Summon
Core Hound (N)	Common	7	N/A
Emperor Cobra (N)	Rare	3	Poisonous
Frost Elemental (N)	Common	6	Battlecry + Freeze
Finja the Flying Star (N)	Legendary	5	Stealth + Summon
Bloodmage Thalnos (N)	Legendary	2	Spell Damage + Deathrattle + Card Draw
Hired Gun (N)	Common	3	Taunt
Leper Gnome (N)	Common	1	Deathrattle + Deal Damage
Millhouse Manastorm (N)	Legendary	4	Set Attribute
Mogu’shan Warden (N)	Common	4	Taunt
Naga Corsair (N)	Common	4	Battlecry
Novice Engineer (N)	Common	2	Battlecry + Card Draw
Nozdormu (N)	Legendary	9	Players turns last 15 seconds
Red Mana Wyrm (N)	Common	5	Increment Attribute
Reckless Rocketeer (N)	Common	6	Charge
Small-time Buccaneer	Rare	1	Set Attribute

(N)			
Stormpike Commando (N)	Common	5	Battlecry + Deal Damage
Stormwind Knight (N)	Common	4	Charge
Sergeant Sally (N)	Legendary	3	Deathrattle + Deal Damage
Street Trickster (N)	Common	3	Spell Damage
Tinkmaster Overspark (N)	Legendary	3	Battlecry + Transform
Ysera (N)	Legendary	9	Generate
Tauren Warrior (N)	Common	3	Taunt + Enrage
Wisp (N)	Common	0	N/A

Table 6 – Shaman deck final card list. Neutral cards are denoted by (N), and class cards are denoted by (C).

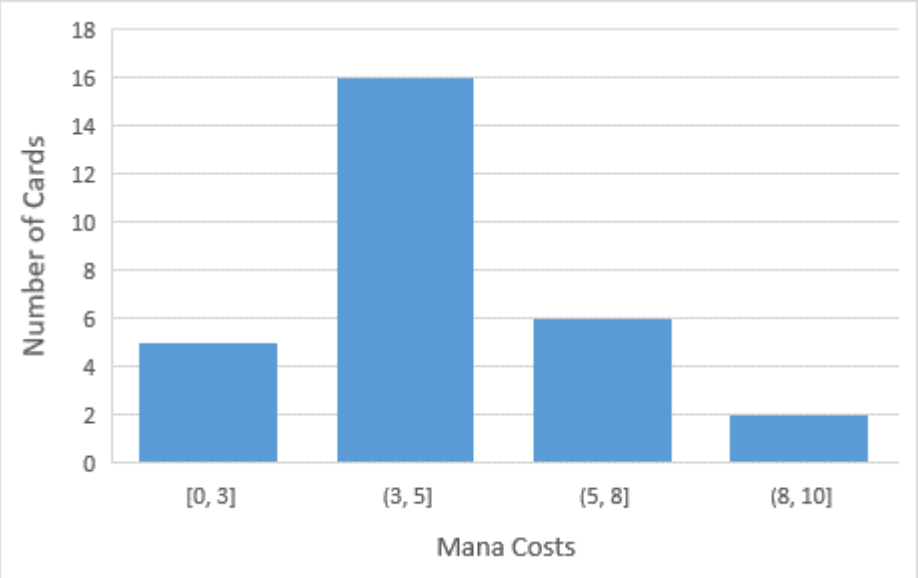

Figure 16 – Mana curve of the evolved Shaman deck.

The mana curve of the generated deck (as seen in Figure 16) shows a high amount of cards that cost from 3 to 5. This begins to weigh our deck in the direction of *aggro* and *midrange* deck types, as both of those decks focus on using low cost minions to gain early game board control. Cards such as Bloodmage Thalnos, Leper Gnome, Small-Time Buccaneer and Wisp provide cheap early game minions, and cards such as Hired Gun and Tauren Warrior provide early game Taunt minions to help protect the other weaker minions. This continues into the higher cost cards, such as Frost Elemental and Cairne Bloodhoof, and finally into the highest costing cards; Ysera and Nozdormu, both considered extremely effective cards to include.

Although this deck contains a number of low costing cards, there are some cards that are considered poor or are outclassed by objectively better cards. Wisp, for example, costs 0 mana but is considered a poor card to play due to cards with better health and attack values being available for a similar cost.

The complete exclusion of class cards also hampers the quality of this deck. Although not necessary, the lack of cards that have synergies with the Shaman mechanics means there is a lack of strategy in the deck beyond playing minions at random.

The extreme lack of spell or weapon cards places this deck at a large disadvantage. The Shaman hero has access to spells that can damage all minions on the board, meaning losing board control early on will almost always lead to the player losing.

Also, the inclusion of very high costing cards, such as Ysera and Nozdormu, goes against the *aggro* and *midrange* archetypes, as those decks have a main focus of winning the match early, or be in a position to win the match very quickly by that point in the game.

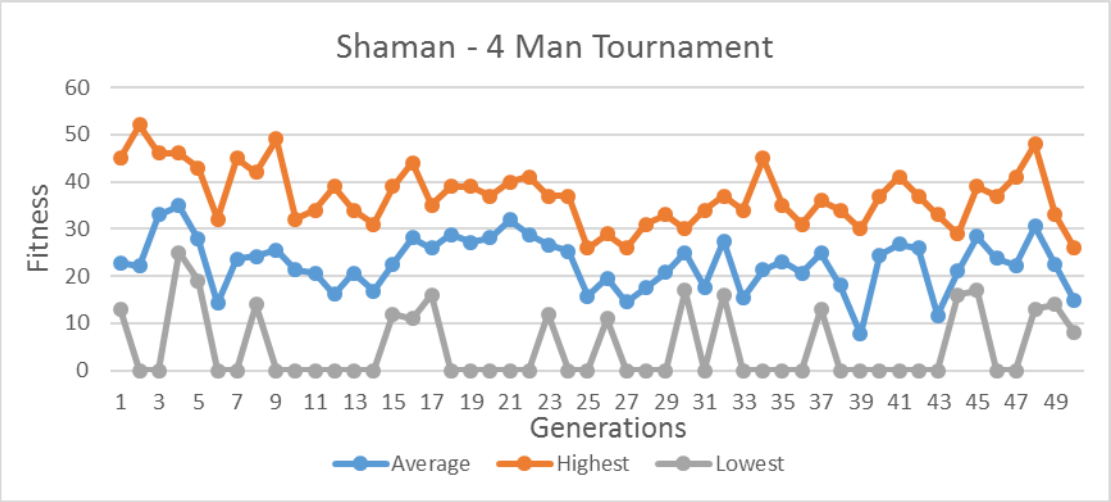

Figure 17 – Evolution of the Shaman deck across fifty generations.

In terms of the overall progress of the generation process, the pattern appears to show no convergence towards the optimum solution. The individual with the highest fitness achieved overall was with a fitness of 52. The average fitness over time appears to be becoming lower over time, with the highest average of 35, and the lowest of 18. The large amount of points with a fitness of 0 can be a potential reason for the lowering average over time, and it could also be due to a potential flaw in the reproduction operators of the algorithm.

The overall pattern appears to become quite erratic after generation 25, with steep increases and decreases with no discernible pattern. However, it appears to stabilise after generation 43. This could be due to mutations into sub-optimal cards and poor crossover points being chosen.

Although the average fitness is decreasing over time, it does not appear to be converging on either a local maxima or minima. This could be due to the maximum generation count of 50, which if increased may have ended up as described.

Overall, the most probable conclusion as to the lack of notable increase of fitness is the complete lack of class and spell cards, making it very unlikely for the deck to consistently win matches, which would lead to low fitness values being calculated.

4.1.2. Deck 1 – Gameplay Analysis

The Shaman deck had a poor performance overall, losing every match it played against its opponents. In terms of the performance of the AI, the AI did not seem to make too many poor decisions, with only minor errors such as playing sub-optimal or useless cards instead of others. Table 7 below outlines the turns taken during the first match, against an aggro Warrior deck.

Turn Number	Card(s) Played	Notes
1	'The Coin' + Millhouse Manastorm	This is a good first turn play, as millhouse manastorm is a strong early game card, and you would not expect many spells in an aggro deck.
2	Hero Power	A typical action to take when nothing else is available.
3	Street Trickster	At this point, board control was lost, due to the opponent using a weapon and a previously summoned minion to remove Millhouse Manastorm.
4	Naga Corsair	Not a very good play, although the deck composition does not really allow for any good actions to be taken.
5	Hero Power	By this point, the player lost all minions, and was facing a nearly full board of enemy minions.
6	Auctionmaster Beardo + Hero Power	An average action to take, although Auctionmaster Beardo's ability is completely useless as it is based around spell card usage.
7	N/A	Lost the Game.

Table 7 – Turns taken by the AI player using the evolved Shaman Deck.

As we can see from Table 7, the AI did not perform too many poor actions. The action on turn 3 is a particularly bad example, as Street Trickster has an attack value of 0, meaning it cannot do anything except take up space on the board. Overall, the AI past turn 3 had no chance of winning, due to having no spell cards to deal with the enemy minions.

The second game has a similar outcome to the first, in that the Shaman does not make too many terrible plays, but still loses nonetheless. Table 8 below outlines the turns taken during the second match, which was played against a midrange Hunter deck.

Turn Number	Card(s) Played	Notes
1	'The Coin' + Millhouse Manastorm	This is also a good play to take, although it may also be a poor one due to midrange decks having an abundance of spell cards.
2	Hero Power	A typical action to take when nothing else is available. By this point, the opponent played 'Scavenging Hyena' which gains stats for every beast minion that dies.
3	Street Trickster	Again, a very poor play due to the uselessness of this card. By this point, the Scavenging Hyena was boosted to a damage value of 12.
4	Hero Power	Player could not play any cards due to having no cheaper cards. A particularly poor choice was made here, as the player could have used Millhouse Manastorm to remove

		the Hyena, but attacked the hero instead.
5	Emperor Cobra	A decent card to use if the opponent does not have a damage spell or if the player has a taunt minion.
6	N/A	By this point, the player lost due to not removing ther Hyena, which had been boosted to a very high damage level of 15.
7	N/A	Lost the Game.

Table 8 – Turns taken by the AI player using the evolved Shaman Deck.

As we see again in Table 8, the Shaman did not make too many bad decisions, barring the Street Trickster being played on turn 3. The player not removing the extremely high threat minion on the board is an issue, as the player was fully capable of doing so, but opted not to. This raises questions about the implemented scoring method.

4.1.3. Deck 2 – Warlock

The second deck was also generated with a mutation probability value that decreases over time, with a maximum generation count of 50. At the 50th generation, the final deck that was generated also included a large amount of neutral minion cards. Table 9 below shows the deck list for the Warlock deck with a fitness value of 24.

Card Name	**Rarity**	**Mana Cost**	**Card Ability**
Arcane Anomaly (N)	Common	1	Increment Attribute
Ancient of Blossoms (N)	Common	6	Taunt
Bloodsail Corsair (N)	Rare	1	Battlecry
Baron Geddon (N)	Legendary	7	Deal Damage
Blood Knight (N)	Epic	3	Battlecry + Increment Attribute
Backstreet Leper (N)	Common	3	Deathrattle
Backstreet Leper (N)	Common	3	Deathrattle
Blowgill Sniper (N)	Common	2	Battlecry + Deal Damage
Captain Greenskin (N)	Legendary	5	Battlecry
Dark Iron Dwarf (N)	Common	4	Battlecry
Grimscale Oracle (N)	Common	1	Set Attribute
Grimscale Oracle (N)	Common	1	Set Attribute
Loot Hoarder (N)	Common	2	Deathrattle + Card Draw
Mad Bomber (N)	Common	2	Battlecry + Deal Damage
Mad Bomber (N)	Common	2	Battlecry + Deal Damage
Mind Control Tech (N)	Rare	3	Battlecry + Mind Control
Mayor Noggenfogger (N)	Legendary	9	All Minions Attack Random Targets
Nozdormu (N)	Legendary	9	Players turns last 15 seconds.
Onyxia (N)	Legendary	9	Battlecry + Summon
Prince Malchezaar (N)	Legendary	5	Summon
Faceless Manipulator	Epic	5	Battlecry + Transform

(N)			
River Crocolisk (N)	Common	2	N/A
Southsea Captain (N)	Epic	3	Set Attribute
Silver Hand Knight (N)	Common	5	Battlecry + Summon
Sunfury Protector (N)	Rare	2	Battlecry + Taunt
Twilight Drake (N)	Rare	4	Battlecry + Set Attribute
Thrallmar Farseer (N)	Common	3	Windfury
Wild Pyromancer (N)	Rare	2	Deal Damage
Wild Pyromancer (N)	Rare	2	Deal Damage
Ysera (N)	Legendary	9	Generate

Table 9 – Warlock deck final card list. Neutral cards are denoted by (N), and class cards are denoted by (C).

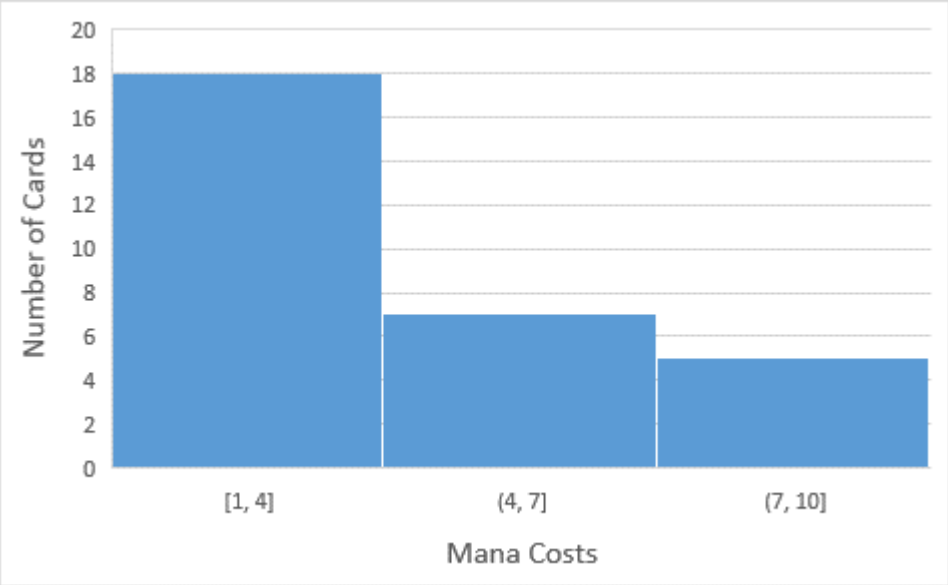

Figure 18 – Mana curve of the evolved Warlock deck.

Unlike the generated Shaman deck, the Warlock decks mana curve (As seen in Figure 18) contains a large amount of low cost minion cards, and a small amount of cards that cost 5 and above. This firmly helps place it in the *zoo* archetype, as that archetype focuses on using a large amount of cheap minions to overwhelm the opponent.

This deck suffers the same problem as the Shaman deck, in that having a number of high costing cards and a lack of spell cards puts the deck at a disadvantage. The lack of damage dealing spells means that the player is unable to destroy other minions without using their own, meaning they can easily lose board control. In addition, the inclusion of high costing cards means they take up space for lower costing cards, which help fit better with deck archetype.

This deck also has completely useless cards, in terms of their abilities. Cards such as Grimscale Oracle are useless as they depend on the deck having a large amount of ‘Murloc’ cards, the Captain Greenskin card is also unecessary as Warlocks do not have access to any

weapon cards. There is also the inclusion of cards that would be deemed novelty, such as Mayor Noggenfogger, which its ability serves no purpose except to cause chaos on the board.

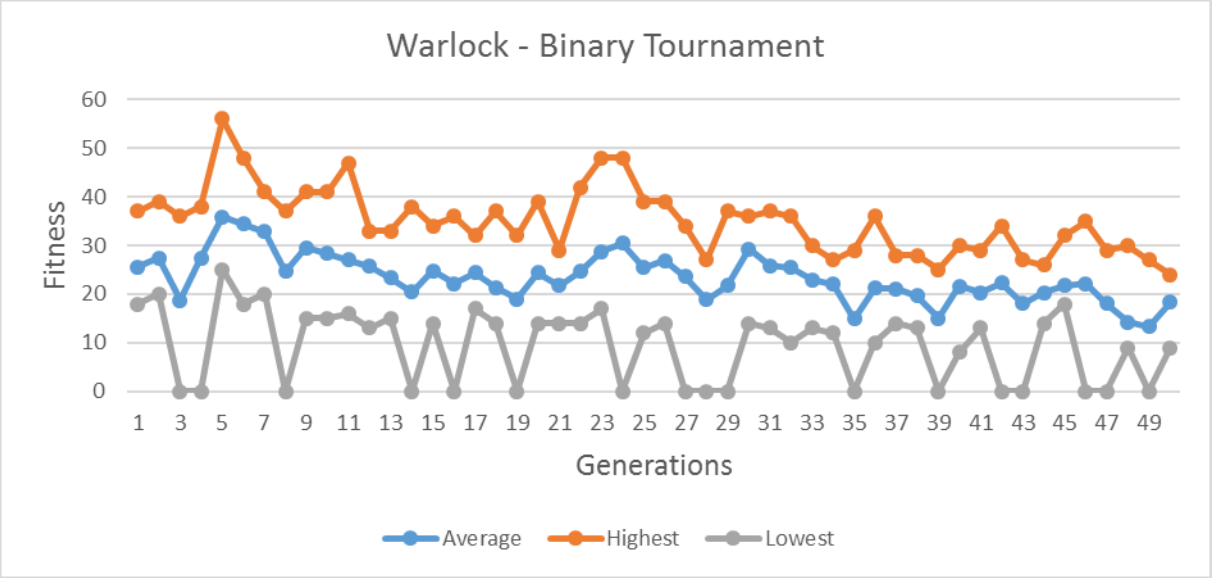

Figure 19 – Evolution of the Warlock deck across fifty generations.

As viewed previously in section 4.1.1, Figure 19 shows the overall generation process of the Warlock deck, which shares a similar pattern to the Shaman deck. The overall average fitness again appears to be declining over time, with a few minor increases where the highest fitness individuals are placed. The highest fitness value gained was 56, which is an extreme difference compared to the other individuals with the highest fitness values.

One difference compared to the Shaman deck is the steady pattern of the overall process. We can correlate this to the lower amount of individuals with a fitness of 0, which lessens the skewing of the calculated average values.

4.1.4. Deck 2 – Gameplay Analysis

The Warlock deck had a marginally better outcome over the previous Shaman deck. Again, the AI did not necessarily make poor choices every time. Table 10 outlines the turns taken by the AI against an aggro Warrior deck.

Turn Number	Card(s) Played	Notes
1	'The Coin' + River Crocolisk	This is not really a good play, as River Crocolisk has no particular abilities. One thing to note here is that the player had Blowgill Sniper, which would have been a much better play as it could have removed an enemy minion.
2	Mad Bomber	An average action to take, Mad Bomber relies on luck to hope it hits the right targets.
3	Blowgill Sniper	Although it removed one minion from the board, it was played much too late, and lost most of its value.
4	Blood Knight	Completely useless against a Warrior deck which has no minions with Divine Shield.

5	Captain Greenskin	Another useless action as Warlock has no weapon cards available to use.
5	N/A	Lost the Game.

Table 7 – Turns taken by the AI player using the evolved Warlock deck.

As we have viewed previously with the Shaman analysis, the AI made a few poor choices, which could have provided a winning strategy. Opting to summon River Crocolisk instead of Blowgill Sniper raises some issues with the AI's scoring method, as it appeared to score its choice based on the card stats instead of the potential to remove minions.

The second game had a much better outcome. In this match, the Warlock deck won against the midrange Hunter deck. Table 11 below outlines the turns that were taken.

Turn Number	Card(s) Played	Notes
1	Nothing	N/A
2	Mad Bomber	An average card that attacks targets chosen at random. It did manage to remove one minion from the board.
3	Sunfury Protector	An average move by using it to turn the Mad Bomber into a Taunt.
4	Wild Pyromancer + Loot Hoarder	Wild Pyromancer is an average board clearing card, however the lack of spells means its ability is useless. Opponent summoned 'Scavenging Hyena' by this turn.
5	Silver Hand Knight	A decent play that gets two minions onto the board for the cost of one card.
6	Hero Power	By this point, the AI sacrificed its Silver Hand Knight to remove a potentially dangerous minion, 'Rat Pack'.
7	Arcane Anomaly	Another useless card due to late game usage and lack of spell cards. By this point, the Hyena was boosted to an attack value of 12.
8	Twilight Drake	In a strange turn of events, the opponent used its Hyena to attack the Twilight Drake.
9	Baron Geddon	A strong minion that damages the entire board every turn. Managed to clear the opponent's entire board.
10	Faceless Manipulator	A good card if used on a strong minion. It was used on Baron Geddon.
11	N/A	Won match due to opponent having no board clear/Taunt.

Table 11 – Turns taken by the AI player using the evolved Warlock deck.

As shown in Table 11, the Warlock deck made some very good decisions during the late game stage. Using Baron Geddon to remove all of the minions on the opponent side, along with using Faceless Manipulator to gain an exact copy, allowed the deck to secure the win.

4.1.5. Deck 3 – Mage

The third deck was generated with a static mutation probability value that did not decrease over time, with a maximum generation count of 10. As these decks were generated with a variating probability value when selecting neutral or class cards, by the 10th generation, the final deck that was generated included a varied amount of neutral and class type minion cards, along with a small amount of spell cards. Table 12 below shows the deck list for the Mage deck with the fitness value of 33.

Card Name	Rarity	Mana Cost	Card Ability
Arcane Anomaly (N)	Common	1	Increment Attribute
Ancient Brewmaster (N)	Common	4	Battlecry
Ancient Brewmaster (N)	Common	4	Battlecry
Ancient Mage (N)	Rare	4	Battlecry
Argent Squire (N)	Common	1	N/A
Blood Knight (N)	Epic	3	Battlecry + Increment Attribute
Bomb Squad (N)	Rare	5	Battlecry + Deathrattle
Bluegill Warrior (N)	Common	2	Charge
Dread Corsair (N)	Common	4	Taunt
Dark Iron Dwarf (N)	Common	4	Battlecry
Doppelgangster (N)	Rare	5	Battlecry + Summon
Dirty Rat (N)	Epic	2	Taunt + Put Onto Battlefield
Emperor Cobra (N)	Rare	2	Poisonous
Gadgetzan Auctioneer (N)	Rare	6	Card Draw
Mad Bomber (N)	Common	2	Battlecry + Deal Damage
Ice Barrier (C)	Common	3	Secret
Master Swordsmith (N)	Rare	2	Increment Attribute
Murloc Tidecaller (N)	Rare	1	Increment Attribute
Naga Corsair (N)	Common	4	Battlecry
Prince Malchezaar (N)	Legendary	5	Battlecry + Shuffle into Deck
Polymorph (C)	Common	4	N/A
Pompous Thespian (N)	Common	2	Taunt
Runic Egg (N)	Common	1	Deathrattle
Sorcerer’s Apprentice (C)	Common	2	Set Attribute
Sorcerer’s Apprentice (C)	Common	2	Set Attribute
Secretkeeper (N)	Rare	1	Increment Attribute
Shattered Sun Cleric (N)	Common	3	Increment Attribute
The Beast (N)	Legendary	9	Deathrattle + Summon
Wind-Up Burglebot	Epic	6	Card Draw

(N)			
Water Elemental (C)	Common	4	Freeze

Table 12 – Mage deck final card list. Neutral cards are denoted by (N), and class cards are denoted by (C).

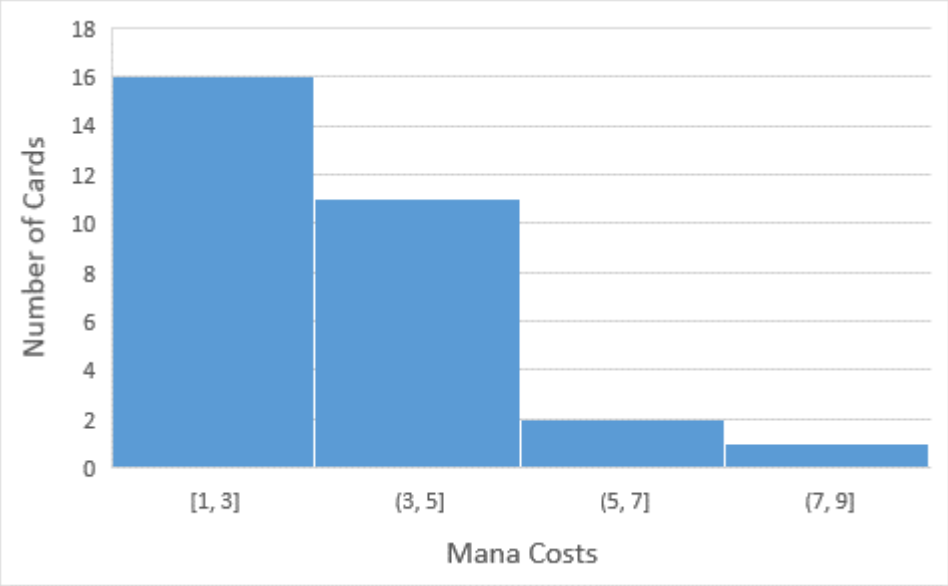

Figure 20 – Mana curve of the evolved Mage deck.

This deck structure is completely different when compared to the last two decks we discussed, with the total number of neutral and class cards being 25 and 5, respectively. As seen in Figure 20, the mana curve again has a high focus on low and medium cost cards, suggesting that the deck leans towards more *midrange* and *zoo* deck types. However, Mage decks are not known to fit well into these categories, as Mage decks are more focused on spell cards and higher cost cards to gain control of the board.

The inclusion of spells immediately places the quality of this deck above the other two. Ice Barrier provides extra health, which helps with survivability, and also provides synergy with Secretkeeper which depends on 'Secret' cards to be viable. Polymorph is also a very viable card, providing removal of a minion which could be deemed a high threat. There is also a direct lack of damage dealing spells, which can decrease the chance of maintaining control of the board.

There are cards that provide decent synergy with the included spell cards. Gadgetzan Auctioneer, Sorcerer's Apprentice and Arcane Anomaly all work well with spell cards, however the inclusion of only two spell cards lessens the impact of these cards.

As seen with the previous decks, there is the inclusion of cards that are deemed useless, either by their ability or by the card being poor overall. Cards such as Bluegill Warrior and Ancient Mage are poor choices due to the lack of 'Murloc' cards and damage dealing spell cards, respectively.

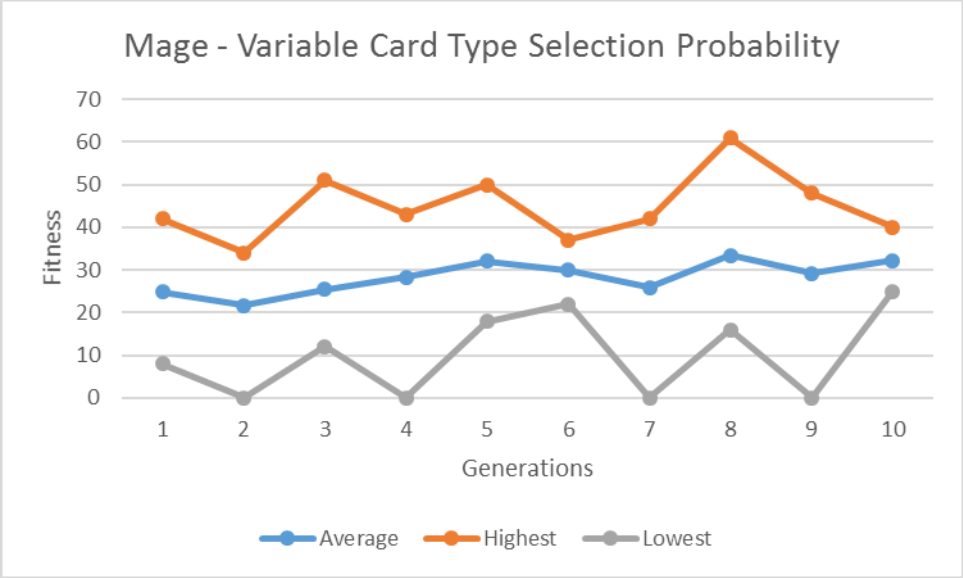

Figure 21 – Evolution of the Mage Deck across Ten Generations.

The generation timeline for the Mage deck shows a steady increase in average fitness overall. The steady increase is only for the 10 generations we performed, however, and we can only assume what the generation pattern would become if the maximum number of generations were increased.

4.1.6. Deck 3 – Gameplay Analysis

As shown with the previous two decks, the Mage deck had similar outcomes and playstyles, in that there were a few questionable choices being made by the AI. Table 13 below shows the turns taken by the deck against the aggro Warrior deck.

Turn Number	Card(s) Played	Notes
1	Secretkeeper	A good card to play if the player is holding any secret cards. The player was holding Ice Barrier at this point.
2	Hero Power	A typical action to take when nothing else is available.
3	Dirty Rat	A very poor choice, especially when up against an aggro deck which relies on minions. An interesting highlight is that Ice Barrier was not played, leading to Secretkeeper not being boosted, and eventually destroyed.
4	Dread Corsair	A decent play, if only to protect the hero from all the enemy minions. Another interesting note is that the player did not use Polymorph, which would have removed one of the enemy minions with the highest threat.
5	Hero Power	By this point, the player lost all minions, and was facing a nearly full board of enemy minions.
6	N/A	Lost the Game.

Table 13 – Turns taken by the AI player using the evolved Mage deck.

As Table 13 shows, we are beginning to view a similar pattern of a mixture of both normal and strange choices from the AI. Not opting to play Ice Barrier when Secretkeeper is on the board is a poor play, along with not using Polymorph to get rid of direct threats.

The Mage deck also did not do very well in the second match. Table 14 below describes what happens on each turn against the midrange Hunter.

Turn Number	Card(s) Played	Notes
1	'The Coin' + Pompous Thespian	A decent turn, which places a small taunt onto the board, providing early game defences.
2	Hero Power	A typical action to take when nothing else is available.
3	Hero Power	Opponent gained board control by this point.
4	Sorcerer's Apprentice	Not a very good play due to the small amount of available spell cards.
5	Bomb Squad	Was able to remove an enemy minion of average threat, but was quickly killed by others.
6	Naga Corsair	A completely useless card, due to lack of weapons.
6	N/A	Lost the Game.

Table 14 – Turns taken by the AI player using the evolved Mage deck.

As Table 14 shows, nothing of note really occurred, with the AI playing cards at random towards the midgame, almost as to try and keep minions on the board.

4.1.7. Deck 4 – Paladin

The fourth deck was generated with a static mutation probability value that did not decrease over time, with a maximum generation count of 10. This deck was generated with the varying card type probability from the previous Mage deck, but also has a varying probability for when selecting a minion or spell card. Table 15 below shows the deck list for the Paladin deck with the fitness value of 40.

Card Name	Rarity	Mana Cost	Card Ability
Argent Commander (N)	Rare	6	Charge + Divine Shield
Aldor Peacekeeper (C)	Rare	3	Battlecry + Set Attribute
Blubber Baron (N)	Epic	3	Increment Attribute
Blessed Champion (C)	Rare	5	Spell + Double Attribute
Baron Geddon (N)	Legendary	7	Deal Damage
Captain Greenskin (N)	Legendary	5	Battlecry + Increment Attribute
Cult Master (N)	Common	4	Draw Card
Cult Master (N)	Common	4	Draw Card
Grimscale Chum (C)	Common	1	Battlecry + Increment Attribute
Guardian of Kings (C)	Common	7	Battlecry + Restore Health

Ironfur Grizzly (N)	Common	3	Taunt
Ivory Knight (C)	Rare	6	Battlecry + Discover + Restore Health
Ivory Knight (C)	Rare	6	Battlecry + Discover + Restore Health
Consecration (C)	Common	4	Deal Damage
Kooky Chemist (N)	Common	4	Battlecry + Set Attribute
Nightblade (N)	Common	5	Battlecry + Deal Damage
Priestess of Elune (N)	Common	6	Battlecry + Restore Health
Priestess of Elune (N)	Common	6	Battlecry + Restore Health
Repentance (C)	Common	1	Secret
Scarlet Crusader (N)	Common	3	Divine Shield
Silvermoon Guardian (N)	Common	4	Divine Shield
Stormwind Knight (N)	Common	4	Charge
Sunfury Protector (N)	Rare	2	Battlecry + Taunt
Secretkeeper (N)	Rare	1	Increment Attribute
Secretkeeper (N)	Rare	1	Increment Attribute
Spiteful Smith (N)	Common	5	Enrage + Set Attribute
Spiteful Smith (N)	Common	5	Enrage + Set Attribute
Truesilver Champion (C)	Common	4	Weapon + Restore Health
Tirion Fordring (C)	Legendary	8	Divine Shield + Taunt + Deathrattle + Equip Weapon
Wolfrider (N)	Common	3	Charge

Table 15 – Paladin deck final card list. Neutral cards are denoted by (N), and class cards are denoted by (C).

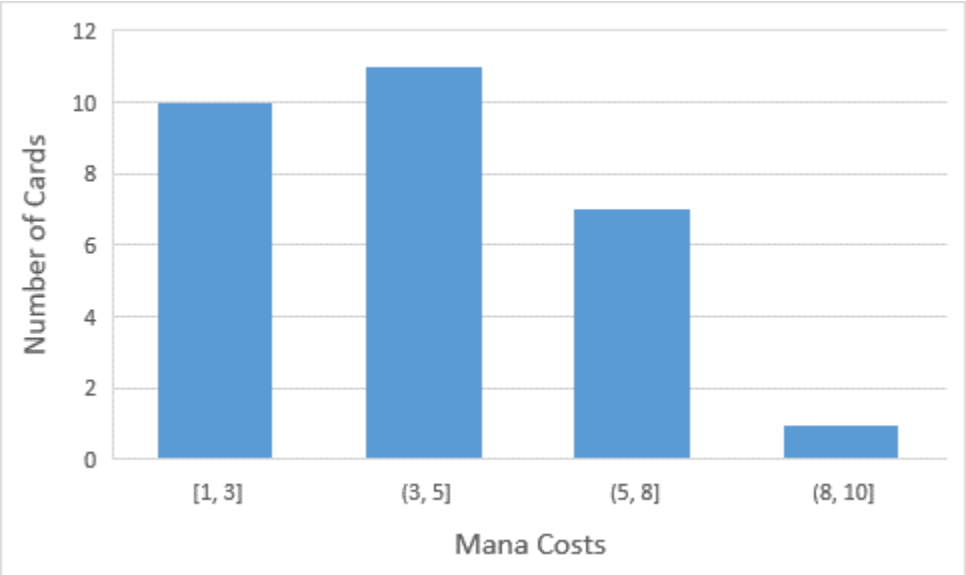

Figure 22 – Mana curve of the evolved Paladin Deck.

The overall quality of the deck is much higher than the previous decks we discussed. As seen in Table 6, there is a varied number of neutral and minion cards, along with minion, spell and weapon cards. Figure 22 shows that there is large number of low to medium cost cards, suggesting a potential leaning towards *aggro* and *midrange* archetypes, as there are a number of strong medium cost cards, such as Truesilver Champion and Consecration, and a low number of expensive but strong high cost cards, such as Tirion Fordring and Baron Geddon.

There are a number of cards that are considered ‘good’ in this deck, either by their ability, or by potential card synergies. The main card that has the highest value is Blubber Baron, which gains increased attack and health every time a minion with Battlecry is played. This can lead to the gain of an extremely dangerous minion, which can provide a quick win for the player.

Aldor Peacekeeper, Repentance and Kooky Chemist provide good soft removal of potential high threat minions by lowering of health or attack values, which can be taken advantage of by the various minions with Charge, and the spell card Consecration.

In addition to the previous cards mentioned, there are a decent number of health restoring cards within this deck. Ivory Knight provides a variable amount of healing, along with a spell/secret card, which can further synergise with Secretkeeper. Priestess of Elune and Guardian of Kings both provide a decent amount of healing, whilst also adding strong minions to the board.

The inclusion of weapon cards provides good combinations with some of the minions included in the deck. Truesilver Champion is a staple card of Paladin decks, providing a medium cost, average damage weapon along with a small number of health restoration. Tirion Fordring also equips the player with a weapon after it is destroyed. These weapons can be combined with Captain Greenskin and Spiteful Smith to gain increased attack values.

The only inclusion that can be considered poor is the Grimscale Chum card, which gains the most benefit by having more ‘Murloc’ cards in the deck. As there are no other ‘Murloc’ cards, Grimscale Chum loses some relevance to the deck. However, it does have the Battlecry ability, so it can still have some usage with Blubber Baron.

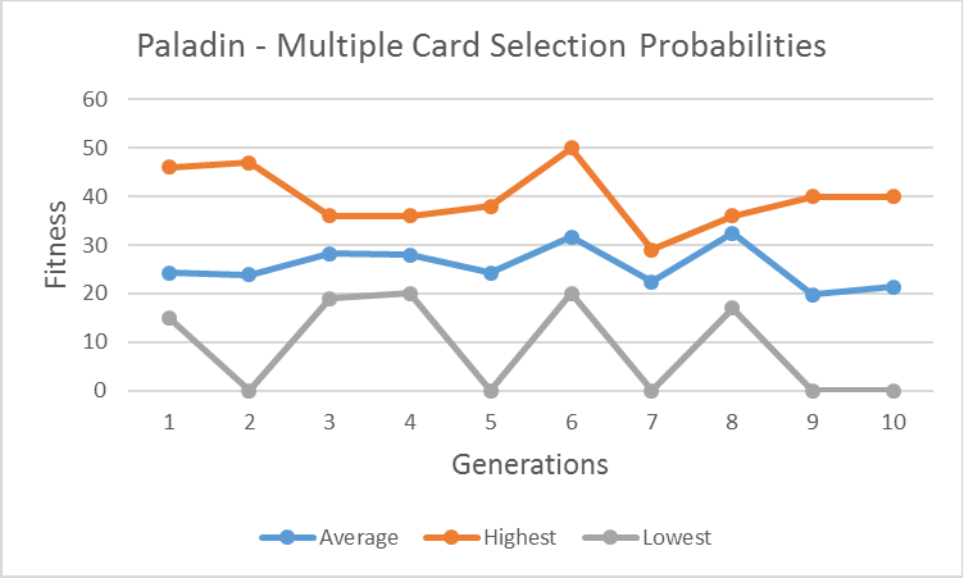

Figure 23 – Evolution of the Paladin Deck over Ten Generations.

The evolution process shares a similar pattern to the Mage deck discussed earlier. The average fitness flucuates slightly over time with only minor increases and decreases. As also discussed previously, we can only assume whether the algorithm was converging on a high fitness solution, or towards a local minima or maxima.

4.1.8. Deck 4 – Gameplay Analysis

The Paladin deck had the most promising outcomes from its test matches. In the first match, the deck narrowly lost, managing to regain control of the board by the late game, but the lack of taunt minions caused the loss. Table 16 below outlines the turns taken by the AI against the aggro Warrior.

Turn Number	Card(s) Played	Notes
1	'The Coin' + Sunfury Protector	A bad first turn play, it would be better to save the Sunfury Protector for later turns where it may see more usefulness.
2	Hero Power	A typical action to take when nothing else is available.
3	Aldor Peacekeeper	A good card to play when a high damage minion is on the board, as it can reduce its attack value to 0.
4	Consecration	By this point, the AI used its minions to lower all the enemy minion's health to 2 or below. This along with consecration was a very optimal move to take.
5	Nightblade	By this turn, the enemy had a boosted weapon thanks to Captain Greenskin.
6	Ironfur Grizzly + Scarlet Crusader	Although played rather late, using a taunt minion saves some health from being lost on the hero.
7	Secretkeeper + Ivory Knight	The AI chose to play secretkeeper first, instead of Ivory Knight; to wait and see what spell is chosen.
8	Guardian of	A good play by this point due to the low amount of health

	Kings	on the hero.
8	N/A	AI lost the game due to loss of board control and taunt minions.

Table 16 – Turns taken by the AI player using the evolved Paladin deck.

The Paladin deck showed the more promising playstyle, just barely losing by the end by a lack of taunt minions, which would have allowed the AI to use its stronger minions to clear the opponent's.

In addition to the performance of the previous game, the deck won the second match against the midrange Hunter. Table 17 shows the actions taken by the AI.

Turn Number	Card(s) Played	Notes
1	'The Coin' + Hero Power	A fairly poor action to take, as it would be much better to save The Coin for a minion or spell.
2	Hero Power	A typical action to take when nothing else is available.
3	Scarlet Crusader	AI had minor board control at this point.
4	Grimscale Oracle + Wolfrider	By this point, the opponent had a 6 damage minion on the board, which the AI did not attack with the Wolfrider. This could be seen as a strange move, as you would normally use a Charge minion to attack minions.
5	Captain Greenskin	Nothing of note beyond the player having no weapon equipped.
6	Argent Commander	By this point, opponent had regained board control, which makes this card a decent play to remove minions.
7	Spiteful Smith + Hero Power	Again, this is a fairly average move, Spiteful Smith's ability only activates once it takes damage.
8	Blubber Baron + Stormwind Knight	Not the most optimal move to take, as Blubber Baron had only been boosted once by Captain Greenskin. Stormwind Knight is an average choice, allowing some damage against minions.
9	Consecration + Hero Power	A very good move, as minions were already weak/weakened by previous charge minions.
10	Ivory Knight + Cult Master	Ivory Knight is a good play, as it heals and provides a free spell card. Cult Master is average, providing card draw when another minion dies.
11	Priestess of Elune + Truesilver Champion	As the opponent still had board control at this point, the AI focused for health restoration.
12	Solemn Vigil + Ivory Knight	Solemn Vigil was retrieved from the previous Ivory Knight. Although not an awful move, but the AI decided to use the spell first, instead of Ivory Knight.
13	Tirion Fordring + Hero Power	The deck's most powerful card, this card is most likely the reason the deck did not lose this match.
14	Repentance + Blessed Champion	As the opponent was unable to destroy Tirion Fordring, AI used Blessed Champion to boost its attack to 12, allowing for a win.

14	N/A	Won the game.

Table 17 – Turns taken by the AI player using the evolved Paladin deck.

The outcome of this match was much more positive than the previous matches by all the other decks. The AI made moves that human players would also make, although there are some glaring exceptions. The AI wasting the 'Coin' card is a sub-optimal move, as it could have been used on a more expensive card to use it a turn early. As well as this, the AI summoning Blubber Baron early could be seen as a poor play, as it had only been boosted once. However, the AI made 'smart' decisions such as clearing the board with consecration, and using Solemn Vigil to attempt to gain more cards.

4.2. Analysis

Based on the results that we retrieved, we can come to a few interesting conclusions.

The first conclusion is that changing the overall algorithm to become more stochastic resulted in an overall declining fitness in both the Shaman and Warlock decks. It also resulted in both of the decks becoming entirely a deck with neutral cards, as shown in Sections 4.1.1 and 4.1.3. This brings up questions about the implementation of the genetic operators, in that there may have been errors with the mutation function for when choosing to swap a card with a neutral or class card. It also could be caused by the higher mutation rates at the start of the generation process, which when combined with crossover, could have led to the neutral heavy decks.

The Shaman deck's mana curve pointed it towards the aggro and midrange archetypes, which are both common types for the Shaman. However, the lack of spell and weapon cards severely impacts the potential strategies of this deck. The Warlock deck suffers in the same way, as although its mana curve points towards the zoo archetype, the complete of spell cards limits the strategies for the deck beyond playing cards at random. Both the Shaman and Warlock decks suffered from lowering win rates over time, ending with fitness values of 26 and 24, respectively.

In contrast to this, adding adaptive selection probabilities for both the initial generation and mutation functions designed decks that have a more even spread of neutral, class, minion and spell cards. The Mage deck showed an increase in the quality of the type of cards in the deck, but still suffered with an overall majority of the cards being neutral. This is not necessarily a bad outcome; however, most decks succeed with a varied mixture. The Paladin deck was a further improvement upon the Mage deck, showing a more varied spread of neutral and class cards, along with spell and weapon cards. As Section 4.1.7 shows, adding more adaptive probabilities based on card structure allowed the algorithm to generate more even decks that could be viewed as human created. These decks had higher win rates than the previous decks, with fitness values of 33 and 40.

One point to note about the Mage and Paladin decks is that they were generated with medium and high selection pressure. While this leads to a potential quick increase in fitness, it can also lead to premature convergence on local maxima due to a lack of population diversity, which could have become the outcome if the number of generations were increased.

In terms of the AI having a potential impact during the fitness evaluation process, there are a few glaring points to mention. In all of the test matches, the AI did perform a few sub-optimal moves, especially during the early game turns. As seen in all the test matches, the AI played useless cards that a human player would never do. We can attribute this to the AI not knowing what cards it has in its deck, so it therefore cannot make an accurate move. Another point is that the AI would sometimes not focus its attention onto the obvious minion threats on the board, which leads to loss of board control. Even with these issues, the AI did manage to win two games, with the Warlock and Paladin decks. However, the first win can easily be attributed to a poor move by the enemy AI, which decided to sacrifice its strongest minion on another. The second Paladin match had much better overall moves, making good use of minions and spells to keep control of the board, until it eventually won.

With these points in mind, and with the overall playstyles of the lower quality decks, we can assume that the AI would not have a huge impact on the outcome of fitness evaluation, and that the more major influences would be the decks themselves.

Another issue that revealed itself is a potential problem with the crossover and mutation operators. As seen in all of the generation timelines, there were a number of individuals with a fitness of 0, which means their decks were invalid. This points out a problem with the implementation of crossover and potentially mutation, which may be accidentally producing invalid children.

5. Conclusion

5.1. Conclusions

5.1.1. Success of Objectives

Our original objectives were defined in Section 1.5.

1. We performed an extensive literature review within Section 2, consisting of an overview into the area of genetic algorithms, followed by an analysis of the application of procedural content generation algorithms within video game environments, and finally into the implementations of genetic algorithms within video games, with academic and real life uses.
2. A C# program was developed that handles all deck and card representation and construction, along with all utility functions previously discussed. We also made changes to Metastone to allow us to receive our required statistics, along with the automation of the generation process.
3. A genetic algorithm is designed and implemented that is able to randomly generate a number of valid decks, before performing the required fitness evaluation, parent selection, crossover and mutation throughout each generation.
4. We make use of Metastone to simulate games for our candidates, which also returns us the required data for our fitness calculations.
5. We have evaluated the outcome of the testing we performed, analysing card choices within the four decks we created, along with the overall generation process.
6. We also use Metastone to handle testing of the built-in AI, to determine the potential influences on the fitness evaluation process.

5.1.2. Success of Aim

The aim of this project, as outlined in Section 1.5, was:

To design and develop a genetic algorithm that has the capability to generate decks of varying archetypes, with the hope that they can gain effective win rates over human designed decks.

A genetic algorithm was successfully implemented, which, along with the Winforms program, is capable of generating valid Hearthstone decks for the various hero classes in the game. We then performed an in depth analysis into the generated decks, as outlined in Section 3.8, with the resulting data being evaluated in Section 4.2.

The evaluation of the data determined that the more stochastically generated decks cannot obtain effective win rates over human created decks consistently. However, adding more levels of control to algorithm increased the overall win rates of the decks. The results also showed that the algorithm created decks that adhered to some of the deck archetypes of Hearthstone, however, the decks that were generated with higher levels of control designed decks that were more refined and adhere more to particular archetypes.

These improvements show that the algorithm has the capability to create decks that gain effective win rates, but more work would be needed to make sure that the decks continue to

improve over time, and that they continue to adhere to the guidelines of having a varied mixture of neutral, class, minion and spell cards.

5.1.3. Success of Hypothesis

The hypotheses that was originally defined Section 1.5 is as follows:

Can a genetic algorithm generate decks for the game Hearthstone; that can effectively beat human created decks?

As previously viewed in Section 4.1, the results showed that a genetic algorithm is fully capable of generating decks that are capable of winning matches against human created decks. However, the decks struggle to obtain *effective* win rates consistently. Therefore, we can conclude that our hypothesis was partially met, and that a new hypothesis is needed, which requires further refinement to include a *consistent* win rate.

5.2. Limitations

5.2.1. Genetic Operators

A potential downside of our implementation are the genetic operators we chose. As shown in Section 4.1, our mutation and crossover operators require some future work, as they both have the potential of accidentally producing invalid decks.

5.2.2. Population Size

Another limitation is our population size, which is set to 10. As mentioned in Section 2.4, previous implementations of genetic algorithms ran into issues when a population is set to a low value. With a size of 10, we may have run into issues of not enough exploration of the search space during the start of the generation process.

5.2.3. Overall Generation Time

The total time spent generating the decks took a very long time to finish, with the Shaman and Warlock decks taking almost a week to complete, and the Mage and Paladin decks taking around two days. This downside meant that more testing could not be completed, due to the long timeframes spent testing.

5.3. Future Work

5.3.1. More Testing

As shown in Section 4.1.5 and 4.1.7, it would be interesting to see if increasing the number of generations on the decks with an adaptive probability would increase in fitness, whilst keeping a normal card structure.

5.3.2. Increased Level of Control

As shown in Section 4.1, the decks with an increased level of control placed upon them performed better than the more stochastic decks.

This means that time could be further spent developing extra parameters that could affect deck structures, such as having a scoring value on each card in the deck that synergise well with others, which impacts the probability of that particular card being mutated. We could

also add selection probabilities based on card abilities, for example, if a deck had a lack of Taunt minions but an abundance of Battlecry minions, we could have a probability value that heavily weights it towards choosing Taunt minions.

5.4. Summary

Our genetic algorithm implementation shows that it is fully capable for a genetic algorithm to produce valid decks that can adhere to common deck archetypes in Hearthstone, and ones that can beat human designed decks inconsistently. Our implementation, along with past research, shows that this is still an unsovled problem, as there are still issues relating to algorithm-specific problems such as fitness calculation, and more complex problems such as the playstyle of the AI.

The usage of genetic algorithms within the deckbuilding aspect of DCCGs is not a fully realised area of research, with only a few past examples. Because of this, we can safely say that this study has added to this relatively untouched research area in new ways.

6. Bibliography

Alabsi, F. and Naoum, R. (2012). Comparison of Selection Methods and Crossover Operations using Steady State Genetic Based Intrusion Detection System. *Journal of Emerging Trends in Computing and Information Sciences*, 3(7).

Bjørke, S. and Fludal, K. (2017). *Deckbuilding in Magic: The Gathering Using a Genetic Algorithm*. Msc. Norwegian University of Science and Technology.

Demilich1. (2015). *demilich1/metastone*. [online] Available at: https://github.com/demilich1/metastone [Accessed 19 Apr. 2018].

Cole, N., Louis, S. and Miles, C. (2004). Using a genetic algorithm to tune first-person shooter bots. *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*.

Counter-Strike. (2000). [Game] Valve Corporation.

Garcia-Sanchez, P., Tonda, A., Squillero, G., Mora, A. and Merelo, J. (2016). Evolutionary deckbuilding in hearthstone. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. 1st ed. Boston [u.a.]: Addison-Wesley.

Hearthstone: Heroes of Warcraft. (2014). [Game] Blizzard Entertainment.

Holland, J. (1992). *Adaptation in natural and artificial systems*. Cambridge, Mass.: MIT Press.

invAIders. (2011). [Game] Martian Face Games.

Isocpp.org. (n.d). *Serialization and Unserialization*. [online] Available at: http://isocpp.org/wiki/faq/serialization [Accessed 16 Apr. 2018].

Li, S. (2016). *Local Maxima*. [image] Available at: http://blog.justsophie.com/teaching-cars-to-drive-with-genetic-algorithms/ [Accessed 16 Apr. 2018].

Louis, S. and Li, G. (1997). Combining robot control strategies using genetic algorithms with memory. *Evolutionary Programming VI*.

Magic: The Gathering. (1993). [Game] Wizards of the Coast.

Martin, M. (2011). *Using a Genetic Algorithm to Create Adaptive Enemy AI*. [online] Gamasutra.com. Available at: https://www.gamasutra.com/blogs/MichaelMartin/20110830/90109/Using_a_Genetic_Algorithm_to_Create_Adaptive_Enemy_AI.php [Accessed 27 Apr. 2018].

McCullock, J. (2012). *Parent Crossover*. [image] Available at: http://mnemstudio.org/genetic-algorithms-recombination.htm [Accessed 16 Apr. 2018].

Msdn.microsoft.com. (n.d.). *Implementing Singleton in C#*. [online] Available at: https://msdn.microsoft.com/en-us/library/ff650316.aspx [Accessed 16 Apr. 2018].

Newtonsoft.com. (2018). *Json.NET - Newtonsoft*. [online] Available at: https://www.newtonsoft.com/json [Accessed 19 Apr. 2018].

Sourceforge.net. (2016). *MicroGP / Wiki / Population Settings*. [online] Available at: https://sourceforge.net/p/ugp3/wiki/Population%20Settings/ [Accessed 25 Apr. 2018].

Togelius, J., Shaker, N. and Nelson, M. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Turing, A. (1950). I.—Computing Machinery and Intelligence. *Mind*, LIX(236), p.19.

www.tutorialspoint.com. (2018). *Genetic Algorithms Parent Selection*. [image] Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm [Accessed 16 Apr. 2018].

TNO. (n.d.). *SketchaWorld: from sketch to virtual world | TNO*. [online] Available at: https://www.tno.nl/en/focus-areas/defence-safety-security/roadmaps/operations-human-factors/sketchaworld-from-sketch-to-virtual-world/ [Accessed 23 Apr. 2018].

Xie, H. and Zhang, M. (2013). Parent Selection Pressure Auto-Tuning for Tournament Selection in Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 17(1), pp.1-2.

7. Appendices

7.1. Potential Deck Archetypes in Hearthstone

Deck Archetype	Overview
Aggro/Aggressive	High focus on playing aggressively by dealing damage to the opponent as quickly as possible, through a large number of low cost cards.
Control	A main focus on keeping control of the game board, with the main goal of winning towards the end of the game with higher costing cards.
Midrange	A combination between aggro and control, focusing on controlling the game board early, and then becoming aggressive during the midgame.
Combo	Revolves around playing a specific combination of cards, to produce an overwhelming advantage over the opponent.
Zoo	Focuses on a large amount of low cost minions to gain board control early.
Minion Based	A deck that has focus around a particular type of minion card, with the particular cards providing benefits when played together.
Tempo	A combination of Aggro and Combo, with the goal of gaining early board control and then finishing with large amounts of damage in a small number of turns.
Quest	Revolves around 'Quest' cards which provide a strong card reward after meeting their completion criteria.
Mill	A high focus on cards that draw extra cards, with the goal of stalling the game and causing the opponent to lose via too large of a hand/loss of cards from the deck.

7.2. Card Abilities in Hearthstone

Card Ability	Description	Neutral	Class
Battlecry	Triggers an effect when the card is played.	Yes	Yes
Charge	Allows the minion to attack as soon as it is played.	Yes	Yes
Choose One	Let's the player choose from 2 unique selections that trigger an event.	No	Yes
Combo	Triggers an event based on whether a card was played beforehand.	No	Yes
Deathrattle	Triggers an event when the minion is destroyed.	Yes	Yes
Discover	Allows the player to choose one of three randomly generated cards, and draw it.	Yes	Yes
Divine Shield	Causes the first instance of damage to be nullified, which also removes the shield.	Yes	Yes
Freeze	Freezes the target for one turn, making them unable to attack.	Yes	Yes
Immune	Prevents all damage from all sources	No	Yes
Overload	Causes the player to have X less amount of mana	No	Yes

	next turn. This amount stacks cumulatively.		
Poisonous	Causes the target minion to be instantly destroyed.	Yes	Yes
Secret	Triggers an event when a certain criteria is met.	No	Yes
Silence	Removes all current stat increases and active abilities on the target.	Yes	Yes
Stealth	Cannot be attacked or targeted by spells until they attack.	Yes	Yes
Spell Damage	Increases spell damage output by X amount, which increases cumulatively.	Yes	Yes
Taunt	Causes all hero and minion attacks to hit this minion first.	Yes	Yes
Windfury	Allows two attacks in the same turn.	Yes	Yes
Card Draw	Draws X amount of cards from the deck.	Yes	Yes
Cast Spell	Casts a specific spell when played.	Yes	No
Deal Damage	Deals damage to a target by a specified amount.	Yes	Yes
Destroy	Immediately destroys X amount of minions.	Yes	Yes
Discard	Makes the player remove X amount of cards from their hand.	No	Yes
Enrage	Increases damage output of minion once it has taken damage.	Yes	Yes
Equip	Equips the player hero with the stated weapon.	No	Yes
Forgetful	The minion has 50% chance of attacking the wrong target.	Yes	Yes
Gain Armour	Gives the player X number of armour.	No	Yes
Generate	Creates and adds a new card to the player's hand.	Yes	Yes
Increment Attribute	Increases or decreases a value from target's health, attack, cost or durability.	Yes	Yes
Mind Control	Gives ownership of target to the other hero.	Yes	Yes
Multiply Attribute	Multiplies the target's health, attack or cost by X value.	No	Yes
Put onto Battlefield	Places a minion from the deck onto the battlefield, ignoring any battlecry effects.	Yes	No
Restore Health	Restores target health by X amount.	Yes	Yes
Return	Returns the target minion to the owner's hand.	Yes	Yes
Set Attribute	Sets the target's attack, health or cost to a set number.	Yes	Yes
Shuffle into Deck	Places a card into the player's deck.	Yes	Yes
Summon	Creates the specified minion card on the battlefield.	Yes	Yes
Transform	Changes a target into a new card permanently.	Yes	Yes
Can't Attack	Minion is unable to choose a target to attack, but can still damage targets that attack it in return.	Yes	No