Profir-Petru Partachi

Deck building in Hearthstone

Computer Science Tripos – Part II

King's College

April 16, 2016

Proforma

Name:	**Profir-Petru Partachi**
College:	**King's College**
Project Title:	**Deck building in Hearthstone**
Examination:	**Computer Science Tripos – Part II, July 2016**
Word Count:	**9163**
Project Originator:	Profir-Petru Partachi
Supervisor:	Dr Sean Holden

Original Aims of the Project

Create an algorithm capable of constructing sets of cards that are valid within the rule of the game Hearthstone, such that when tested against a random baseline it will win more than 50% of the matches with confidence 99%.

Work Completed

An algorithm based on a genetic algorithm that searches for solutions against a choice of random or given a baseline and will output the best solution found when at least one passes the required quality threshold.

Special Difficulties

The main difficulties of the project were finding a library that both simulates the game of Hearthstone or the match aspect of it, and provides a game playing AI implementation. Once such a library was found, the next difficulty that shaped the course of the project was integrating a Java library and the main code-base that was written in Haskell. The latter issue was solved via means of an additional library (Apache Thrift) which allowed communication between the two parts of the project.

Declaration

I, Profir-Petru Partachi of King's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

Strategy card games provide an interesting platform for exploring AI related questions within the domain of imperfect information games and strategy elaboration for pre-game match scenarios as well as in-game scenarios. The usual game from this genre that is of research interest was, and arguably still is, "Magic: The Gathering", which exists digitally as well as a hard-copy. The game studied here is, however, Hearthstone, mostly due to its pure digital format as well as more simple and restrictive rules as we will later see.

This project aims to solve the problem of choosing a set of cards according to the rules of this chosen game such that, disregarding skill disparity, the odds of winning are more than 50% with 99% confidence. I have managed to create an algorithm capable of probing the search space for satisfactory solutions via means of a genetic algorithm. The search can be geared towards using a choice of random or user provided baselines to evaluate against or evolving the populations dynamically by testing solutions against each other. The choice of the search strategy can be used to answer different questions. The core objective is implemented by using the search algorithm with a random baseline while the user provided baseline implements one of the extensions.

Additionally, an extra option of mixed populations has been implemented. The use of this option is mostly of predictive nature, anticipating meta-game[1] shifts in the ladder environment of Hearthstone. It may be of further interest if paired with a Monte Carlo Tree Search based AI for playing the game matches.

1.1 The game studied: Hearthstone

Hearthstone represents a Collectible Card Game and is sometimes referred to as a Strategy Card Game as well. It has several main aspects to it:

1. A card collection, which you must slowly build through a variety of means;

2. A deck builder, which is a method of selecting a set of card out of your collection according to a set of rules;

[1]In this context: what sets of cards are chosen by most players.

3. A game match, an environment, with a set of associated match rules, where you play your chosen set of cards against an opponent who also chose a set of cards according to the same rules;

The aspect of interest to us is the second one, the deck-building aspect, as such let us explore the rules that govern it in more detail. When building a deck one must adhere to the following restrictions:

1. A deck must have a class associated with it, this restricts the allowed deck cards to neutral and the chosen class.
2. A deck must have exactly 30 cards.
3. Cards may be used at most twice in a deck, except for legendary cards which may be used at most once.

A deck respecting these restrictions is considered valid.

It is precisely these fairly restrictive rules for deck-building that made this problem for Hearthstone more tractable as opposed to Magic: The Gathering where decks may be of variable length.

1.2 Other similar games

As already mentioned, the staple of the genre is Magic: The Gathering, however, that is not the only game that can be identified as a Collectable Card Game. Other games of similar nature that are worth mentioning are the Yu-Gi-Oh card game or the Pokémon card game. Similar to Magic, they offer variable length decks and hence add complexity to the problem of interest.

If we canvas the digital scene, other games of interest would be Scrolls or Hex. In this case, the difficulty would be the lack of simulating libraries due to reduced popularity of the games. Another reason for the lack of such libraries could potentially be the difficulty of simulating these games as the game match aspect is considerably more complex, containing a more advanced notion of game phases as well as the option to interrupt opponent actions that cannot be easily resolved in the same manner as Magic, i.e. an event stack.

On the other hand, each of these games comes with official and fairly extensive rule books detailing the resolution of most if not all game events that could occur. An advanced rulebook for Hearthstone is being maintained by the community, but it is an unofficial endeavour. What this means for this project is that the simulation may create event resolutions that cannot occur in the actual game and cause a disparity between the performance of a deck in simulations and the performance in the actual game. I shall consider the disparity to be insignificant on the basis that such scenarios are tagged as bugs for the simulating frameworks and are being actively patched.

Finally, do note that there are quite a few other games not mentioned here in both digital or physical format due to a spike in the popularity of such games in the mid-90s, however, they have not properly survived the test of time. With regards to newer ones not mentioned, especially those that are on phones and tablets, they are either experimental in style and/or features or have other issues that are beyond the scope of this project.

Chapter 2

Preparation

This chapter will focus on how players attempt to solve the problem at hand, as well as ideas that were considered as possible implementable solutions along with the reason why they were or were not taken forward. Finally, it will consider early design decisions that set the stage forward for implementation.

2.1 The players and their approach to the problem

Before attempting to solve the problem, it is reasonable to observe how players usually approach the same issue. By analysing tournament interviews and videos that are publicly available where players discuss deck building and strategies whilst thinking out loud a few patterns can be observed:

1. Choose a theme, a core mechanic within the game that the deck will focus on
2. Add support cards to ensure the thematic choices work
3. Add ”tech” choices that have the purpose of counteracting specific opponent cards

Alternatively:

1. Choose a game finishing strategy or combination of cards
2. Add draw mechanic cards to speed up the acquisition of the combination of cards
3. Add cards to control the board state and survive until the combination is in hand
4. Optionally add ”tech” choice cards

However, regardless of which approach was chosen for the initial deck, the process by which they improve it is mostly identical. That is they play games with the purpose of identifying dead cards, cards that sit in the hand and cannot be played optimally. Once they are found, they are swapped with a different choice, usually depending on opponents encountered, and the process repeats. The decision could also be biassed by players noting down when they wish they had a different card in place of the one in hand if this occurs sufficiently often. Sadly, this is not something that could easily be translated into code as this would require a concept of when an arbitrary card is good given a board state.

2.2 The chosen approach: A genetic algorithm

The style in which players perform deck-building as well as a study in identifying opponent decks based on the early stages of the match[1] pointed towards two main approaches that would be tractable within a reasonable time-frame.

The first approach would be to learn in a similar fashion to predicting opponent choices, sets of cards, bi- and trigrams, that counter specific choices from an opponent and an algorithm for merging such choices while respecting deck building restrictions. This would mimic the initial building of a deck along with the refinement phase by simply trying to counter choose cards.

The algorithm would then proceed to attempt to spot suboptimal cards. Such cards would then be replaced by other uni/bi/trigrams that are flagged as useful given the opponents' cards.

This approach appears appealing until the amount of data required to make it work reasonably is assessed. An additional issue would be the correct handling of user data when collecting game replays which would be necessary so that I could mine the correct choices. The process is fairly straightforward but intrusive to the users. As such, onto the second approach, a genetic algorithm approach.

A genetic algorithm performs a search for one or more good enough candidates within a solution space using the following steps:

1. Generate an initial population and set time to 0
2. Evaluate the fitness or quality of the population
3. Determine if there are sufficiently many fit enough candidates in the population or enough time has elapsed
4. Generate a new population by performing genetic operations (selection and crossover, mutation, optionally deletion or insertion)
5. Progress time by 1 and go to 2

Usually, such an algorithm uses a binary string encoding for candidates for the solution as to simplify the genetic operations, but there is no easy way to encode the decks as binary strings. On the other hand, there is the option of making the selection and mutation operators mimic human behaviour towards the problem with, as was later seen, favourable results.

2.3 Type and Operator design

The next step was choosing an appropriate level of encoding granularity or expressivity of the structures representing candidate solutions. We want to be expressive enough to easily translate into a format understood by the simulation harness while not bothering with too many details that are game specific. This will ensure searching the solution space can be done in a sufficiently intelligent manner while avoiding cluttering and complicating the implementation of such an algorithm. It could be argued that adding more details would improve our ability to choose intelligently, however, the true difficulty lies in finding the balance for just the right amount of information.

The design took shape in several iterations of adding and discarding information. Since cards can only be found reliably by the simulation harness by their specific card id, I was locked into either using them directly or having a look-up table that would have to be used prior to simulation calls. Additionally, I needed to ensure that deck building rules are respected, that is we don't use more than the allowed number of cards for each card. This pointed towards the first Card type being simply a pair of card id and card copy limit with the assumption that I would maintain a per-class card list.

While still in the design phase, it was found that this will be sufficient only for populations of candidates of a predefined in-game class, and as such will not generalise nicely to the mixed populations case. I wanted to avoid restrictive design decisions if possible. The Card type was therefore refined to a triplet of card class, card id and card copies limit.

At some later point, card resource cost was being considered as an additional entry. This would allow us to better construct the initial population. The issue was that to do so, we would need to employ a good deck archetype classifier that can distinguish aggressive, control, tempo-based and combination-based strategies. This could be a small side-project in its own right, and could have been considered as an extension. The main issue was the lack of a good corpus to train such a classifier and the idea was discarded. As such, the card notion took the following form:

```
type Card = (String, String, Integer)
```

Having a notion of Card, I needed to decide on a structure for the "Chromosome" or candidate solutions. I am trying to build a deck, or in other words a selection of an in-game class along with 30 cards following certain rules. This lead to defining a Deck type as a pair of Hero (in-game class) of type string and DeckCards as a synonym for a list of pairs of cards and copies used. Building DeckCards is not exposed to the end-user, and the deck-building rules are used as post-conditions for any function returning DeckCards. Regarding the in-game hero, it was necessary to include as to generate correct calls for the simulation harness, however, this choice also ensured that the type could persist unchanged once mixed populations were added. As a final note on type choices, the population was modelled simply as pairs of Deck and an optional Double score.

```
type Deck = (String, DeckCards)
type DeckCards = [(Card, Integer)]
type ScoredDeck = (Deck, Maybe Double)
```

With these in hand, *selection* can be implemented using elitism. This choice can be argued based on the fact that within the game ecosystem, the usual propagation of a good deck is done by copying it directly from the best players who provide them publicly. Web sites such as "www.hearthpwn.com" are both a proof and a vehicle of this phenomenon. Therefore, this operator must obey the following statements:

For the candidates that persist to the next generation:

$$\lfloor e * len(population) \rfloor = len(fst\ select_e(population))$$
and
$$\forall i \leq \lfloor e * len(population) \rfloor . get\ i\ population = get\ i\ (fst\ select_e(population))$$

For the candidates that go to crossover:

$$\forall (candidateA, candidateB) \in (snd\ select_e(population)).$$
$$candidateA \in (fst\ select_e(population)) \wedge candidateB \in (fst\ select_e(population))$$
and
$$len(population) - \lfloor e * len(population) \rfloor = len(snd\ select_e(population))$$

Mutation can be used to mimic the process of refinement that players perform where they identify bad cards and attempt to replace them. A notion of granular utility can be used to mimic player card preference. This, although already in the implementation phase, triggered a further refinement of the DeckCards type to allow for a per card utility score.

Mutation is meant to work on at most one card at a time, therefore, the result should be a valid deck and may differ from the input by at most one card.

Crossover sadly does not have a natural equivalent, but it is the main force guiding the search with the help of selection. This is done, from a high-level perspective, by propagating choices that were successful. The complexity of crossover did increase due to non-binary encodings, mostly due to trying to reduce the time complexity of the operation.

This operator provides a third valid candidate solution that is related to two candidate solutions that are given as input. To ensure this, a reasonable definition of crossover within this solution-space had to be defined.

The initial intuition was to merge all cards together, sorted by score, and pick 30 at random biassed by the card position. This idea was later scrapped in favour for a slightly different one due to performance concerns and the inability to generalise the first attempt to mixed populations.

The current definition works differently on cards that are in both decks and cards that are only in one or the other. The exact workings will be further discussed in the chapter to follow.

One final thing to note within the context of genetic algorithms is that selection and crossover work towards reducing diversity and converging on a solution, while mutation acts similar to a perturbation factor, that is, works against convergence. The reason why we want to reduce the rate of convergence is to avoid local maxima, in which we might get stuck if we were to only use the operators for selection and crossover. The trade-off is that we might have to wait longer for a solution.

2.4 Language of choice and working environment

The language of choice to code up the implementation was Haskell. This is to allow a rapid prototyping style of development and a modular approach to writing and testing the operators individually. Testing was to be done by, at least for pure code, stating properties of correct input and properties that should be respected by output along with asserting invariants within the implementation. Each prototype was then assessed for correctness using QuickCheck[2]. It works by using the preconditions and the type system to deduce and generate random valid input and checks if the output satisfies the provided post-conditions.

QuickCheck enabled me to catch data corruption early if and when it occurred. Later in the project, the assertions have been changed to recovery attempts as to tolerate occasional recoverable corruption, such as more than 30 cards in a deck after a crossover operation.

Prior to any project related coding, "An introduction to genetic algorithms"[3] was read and several smaller problems presented as exercises for the reader have been solved using a genetic algorithm. Amongst these "The iterated prisoner's dilemma" and evolving polynomials that approximate $\sin(x)$. This was done as to get accustomed to the style, mindset and assumptions required when solving a problem in such a fashion as well as interpreting results from such a system.

Finally, a working environment and back-up system had to be set-up for both code as well as write-up related files. Git was used to manage code changes for both Java and Haskell and SVN for LaTeX. The repositories themselves were backed-up and cloned across all physical working environments using Dropbox. This allowed different granularity of history for different files while maintaining access to the work in progress private.

2.5 A canvas of libraries for simulating Hearthstone

For the purpose of evaluating a deck, we need to determine how well it performs against a baseline or against other candidate solutions. This requires simulating Hearthstone games efficiently and using an AI capable of playing a variety of decks. Luckily there is a community dedicated to studying Hearthstone and simulating the game accurately[4] which also provides links to GitHub repositories representing projects that aim to allow simulating Hearthstone games as well as AI implementations.

Sadly, Fireplace[5] was an inactive project at the time of preparation containing bugs that would have interfered with the project, although it appears to have been resurrected by the time of writing. The next library considered, mostly due to being written in Haskell: HearthShroud[6], does not provide an AI implementation. Implementing an AI for the game was considered too great an endeavour for the time-frame of this project.

An alternative has been found, MetaStone[7]. It seemed to be the best available option, in terms of AI implementation diversity, the performance of simulations and cards implemented at the time of preparation. The downside is that it is written in Java. Quite obviously, the MetaStone project is not geared towards being used as an evaluation function within a genetic algorithm nor was written with compatibility with Haskell in mind. With minimal changes, it was possible to re-purpose it as such, at the cost of an additional library dependency to handle the communication between the two different languages. The exact details of this will be discussed later.

2.6 Related Work

There is little literature with regards to deck building within a CCG, however, there is sufficient work that attempts to solve the other two problems. It can be argued that we need an efficient simulator, and as such a solution to the 3rd problem (game playing AI), to attempt deck building and until recently there were no solutions that were sufficiently fast. Another reason for why this might be the case is that until Hearthstone, decks were not restricted to exactly n cards for some known n, rather they were allowed to fluctuate and this makes the problem considerably harder as it increases the solution space, i.e. adds an additional degree of freedom.

Nevertheless, we can see that there has been work done towards the game playing AI[8][9], where the respective authors explore different ways of deciding moves within a match and compare different approaches, including supervised and unsupervised learning and different methods of game tree exploration arguing why Monte Carlo Tree Search might be slow within the context of Hearthstone. This approach is not unique to Hearthstone and has even been applied to Magic: The Gathering in the past with some success[10] and game specific improvements were later found[11], which might lead one to believe that similar improvements could be ported over to Hearthstone as well.

Another article considered the meta-game concept in different games including Hearthstone[12]. This works is more directly related to the problem I am tackling as one of the inputs is, for all intents and purposes, a meta-game snapshot. For the purpose of this project, I consider this information to be known to the user as there are sites and communities preoccupied with gathering, maintaining and presenting this data.

Overall, the current literature provides ways of improving the input as well as improving the game match simulations that are required for this project. What this project attempts to do, is to unite the results from previous work into a coherent structure to solve a different problem and fill a gap in the current status quo. Hopefully, it will serve as a staging ground for more interesting projects in the future.

Chapter 3

Implementation

In this chapter, we will explore the implementation of the proposed solution along with the decisions that guided and shaped it. For each major operators and functions I will state pre- and post-conditions which were checked using QuickCheck during development for the purpose of exposing and fixing bugs. Encountered bugs will not be discussed unless they forced a major re-factoring of the code.

3.1 Main algorithm

Here we will mostly focus on the algorithm and operators as they were implemented for populations containing candidates only for the same in-game class.

3.1.1 Representing game specific structures

Armed with the chosen type representation of game data, those types still had to be populated by concrete data that is valid within the context of the game. The first step was a manual generation of a Catalogue (or rather a catalogue of catalogues). This hard-coded variable contained exactly 10 lists within it. Each list represented cards that had a certain class restriction to them, including the no-class restriction. This ensures that the algorithm chooses cards that actually exist in the game while also respecting class restrictions.

Additionally, this allows specialising the algorithm to build candidate solutions from a restricted set of potential cards.[1]

Finally, and mostly for the mixed population scenario, a hard-coded index containing all valid in-game classes has been manually crafted. It has the purpose of simplifying the generation of an initial mixed-population with minimal user input as well as an easier generation of the class global score for crossover as will later be discussed.

[1]As a side note, the cards used by Haskell and Java are not necessarily the same, and that can be used to search for solutions against opponents having access to more cards. This is not within the scope of this project, but the curious could easily adapt the solution to this new scenario if they so desire.

3.1.2 Initial Population Generation

The problem of generating an initial population has been broken down into generating a single valid candidate at random and aggregating results into a list. This approach also extends nicely to mixed populations as we can call the single candidate generator with different restrictions each time.

To generate a single candidate we:

1. Generate a reduced catalogue that contains two lists. The first is class-neutral cards. The second is class-specific cards.
2. Initialise the deck as a pair of input in-game class and the empty list.
3. Flip an unbiased coin to choose whether the card to be added is class-neutral or class-specific.
4. Pick an index uniformly at random for the chosen card type.
5. Attempt to add the card to the deck we have built thus far. This could fail if adding it would invalidate a deck-building restriction.[2]
6. Count the number of cards, return to 3. if there are fewer than 30, otherwise return the deck

Precondition: The input represents a String which is a valid representation of an in-game class[3] and a random number generator.
Postcondition: The output is a valid deck paired with a random number generator that has a different state than the input one.

Now for population generation we have:
Precondition: An integer seed, a positive integer representing the number of candidates to generate(let us call it n) and a String that represents an in-game class.
Postcondition: The output is a list of length n, containing only valid decks and each deck must have the Hero field equal to the provided String input.

3.1.3 Selection

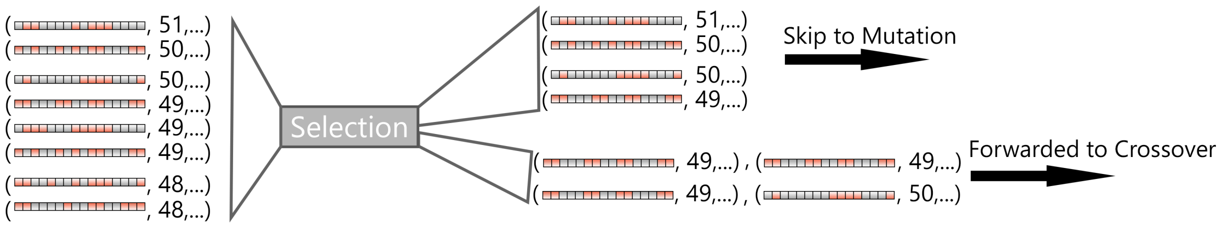


Figure 3.1: A high-level view of the elitism selection operator

[2]This is ensured by making the operators that add or remove cards to/from a deck check that card level restrictions remain valid and that the only deck restriction we might ignore at this point is that we must have exactly 30 cards.

[3]Valid values are ensured by providing the strings from the pre-built class index

Selection is implemented in a fairly straightforward way. Since it is elitism, selection works by taking the top p percent of the population and then picking uniformly at random with replacement two decks to be crossed-over. It generates sufficiently many pairs to regenerate the (1-p) fraction of the population that is being discarded.

Precondition: A sorted by score list representing the candidate solutions (of length n) and an elitism parameter in $(0;1]$

Postcondition: Output is two lists. The first list is of length $\lfloor e*n \rfloor$ and agrees with the input population on each index. The second list is of length $n - \lfloor e*n \rfloor$, and each element is a pair of candidates. Both candidates are from the first output list.

3.1.4 Mutation

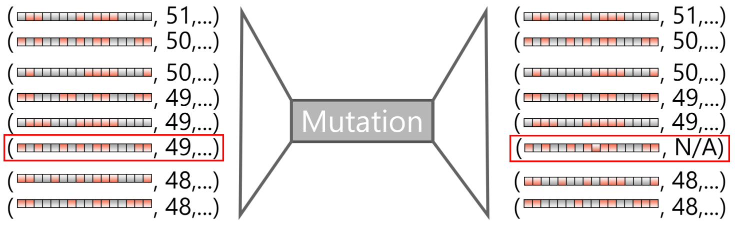


Figure 3.2: A high-level view of the mutation operator

Mutation is intended to mimic how players choose what they perceive as bad cards and replace them with a different card. Originally, Mutation did not take into account a card level utility, however, it was later added as the convergence rate was sub-par without it. We shall consider the details of the final implementation only.

Given a candidate, we:

1. Flip a biassed coin, the odds of getting head is the same as the mutation probability input.
2. Return the original input unchanged if we get Tails, otherwise, we continue.
3. Choose an index for the card to change from a triangular distribution.
4. Remove the card at the chosen index in a list of cards that has been sorted in descending order by card utility score.[4]
5. Choose a valid[5]card completely at random and attempt to add it to the deck.
6. Return the new candidate if we have exactly 30 cards, otherwise go back to 5.

Precondition: We are provided with a random number generator, a mutation probability in $(0, 1]$ and a candidate that has been evaluated.

[4]The chosen triangular distribution has a PDF with a positive slope. Hence, we choose later indices with higher likelihood and as such need to sort in descending order to ensure that lower scoring cards are more likely to be replaced.

Postcondition: The expected output represents a pair of a valid candidate and a random number generator. The state of the random number generator must be different from the input one. If the output candidate has a score, it must equal the input candidate. Otherwise, they must differ by exactly one card.

3.1.5 Crossover

The crossover operator takes in a pair of decks and will output a single deck out. It thus acts to reduce the diversity of the population and force it to home into better candidates. Our implementation works in the following manner:

1. Flip a coin, the odds of getting Head is the crossover probability.
2. If we roll Tails, we then flip a new coin, the probability of Head is proportional to the first input deck score, and Tails respectively to the second.
3. If we get Heads, we return the first input, otherwise we return the second input.
4. If the first coin was Heads, i.e. we perform crossover, we compute the intersection and differences of the two inputs.
5. The result automatically contains all the cards that are in both candidates.
6. For the cards in the differences we:
    (a) Pick uniformly at random without replacement two cards, one from each set.
    (b) We toss a coin, the probability of Heads is proportional to the first card score, Tails to the second.
    (c) Depending on the coin toss we add one or the other card to our return list.
    (d) If either set becomes empty before the other, we copy over the remaining cards into the returned list.

    Until we have depleted both sets.
7. We concatenate the cards, check if we have 30 cards in the obtained deck, and return if so.
8. If we have more than 30, we sort them by score and trim to 30.
9. If we have less than 30, we start adding random valid[5] cards until we have 30.

Precondition: The input is two valid decks that have the same in-game class and a random number generator along with a crossover probability in $(0; 1]$
Postcondition: The output is:

[5]Valid according to all but the number of copies in the deck rule.

Either one of the inputs and a random number generator that has had its state changed if the output has a score;
Or a deck that contains all of the cards that are in the intersection of the inputs and a random number generator with a changed state.[6]

3.1.6 Evaluating fitness

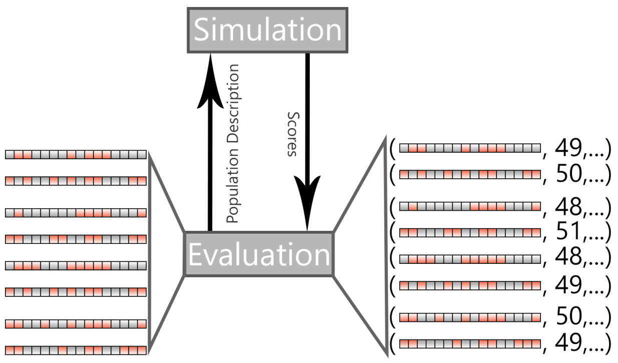


Figure 3.3: A high-level view of the evaluation operator

Fitness evaluation of candidates is performed in Java by running simulations where the same AI plays candidate decks against baseline decks or other candidates.

In order to do this, we need to:

1. Create a connection to the Java server that will run simulations on our behalf.
2. Perform an RPC with the required simulation parameters that we generate at run-time according to the current population.
3. Once we receive a result from the simulations, we parse it to deduce deck fitness, and for each deck – card utility and update the values accordingly.
4. Finally, we sort the population in descending order according to fitness.

Precondition: The input is a list of valid candidates, some may be scored.
Postcondition: The output represents a list of candidates, each having a score associated with it and sorted in descending order according to score.

Additionally, an optimisation was considered for the scenarios where we search for a solution against a baseline. That is to evaluate the fitness only for candidates that have been modified by either mutation or crossover. Since both operators erase the score, we do not need to implement a new mechanism for tagging the candidates and instead would only need to implement a method to filter out the candidates that do not have a score value. To determine if this is worthwhile consider the following:

[6]Note that I make no claims about cards from the differences of decks as we might have a random fill stage where cards may come from the outside. We can think of this as a burst mutation due to insufficient building block material.

Given a population of size n and hyper-parameters e,c,m for elitism, crossover probability and mutation probability respectively, we can determine the expected number of candidates to be modified.

Crossover: On average $c \times (1 - e) \times n$ are subject to crossover.

Mutation: This operator works on the whole population and hence we expect $m \times n$ candidates to be mutated.

Both: To avoid double counting we subtract the value of candidates acted on by both operators which we expect to be $m \times c \times (1 - e) \times n$

Modified: Finally, we conclude that $(1 - m) \times c \times (1 - e) \times n + m \times n$ candidates are expected to be changed.

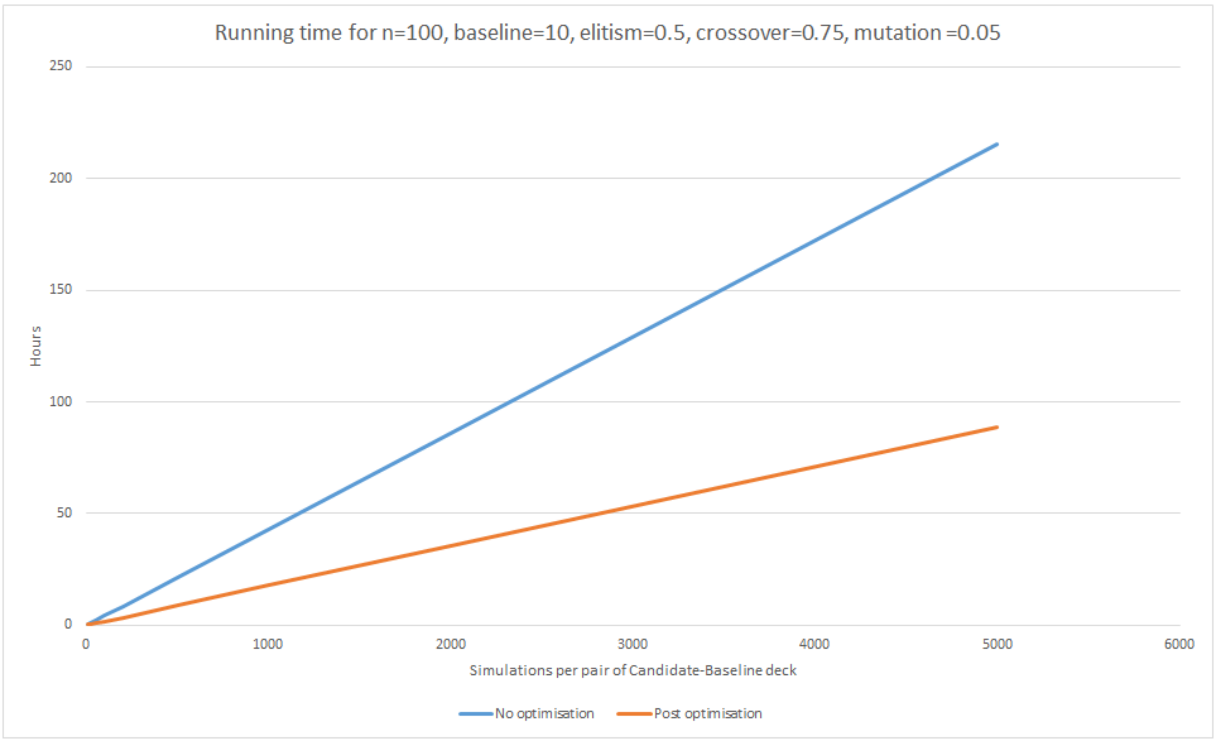


Figure 3.4: A comparison of expected runtime if we re-evaluate the whole population vs only modified candidates.

From just the formula, it is unclear if it is worthwhile to pursue this optimisation. By estimating the time taken on a single generation and computing the values for a given set of hyperparameters we can find the following (fig. 3.4): For a representative choice of hyper-parameters, if we choose to simulate 200 games per candidate-baseline pair, the estimated gain is considerable, i.e. on the scale of hours.

3.1.7 Altering the Java library

The Java library used provided an easy way to queue arbitrary amounts of simulations given candidate decks, the AI that should play them and how many times should it play a pairing. The main issue was the GUI and game-logic code being tangled together. Thankfully, the library is open-source which allowed me to modify it to work without the GUI. The downside was the lack of code documentation which resulted in a bit of reverse-engineering and some re-implementation of features according to the community-maintained advanced rulebook.

The first step was to deduce how it queues up games for a simulation batch[7]. By examining the GUI code for the Battle of Decks mode it was possible to reverse-engineer how the library performed the equivalent of dynamic landscape evaluation, playing every deck against every other deck a given number of times.

The original code created a Task which represents playing one game between two players using a given deck each. It then proceeded to navigate the input list of decks and queue in a Work Stealing Thread Pool the required number of games for each deck pairing.

This works out-of-the-box for dynamic landscape evaluation, however, for static evaluation against a baseline, we require to queue games against the decks in the baseline. Therefore, the first modification was to create a new method, that along with the candidate decks and game configuration(AI type, number of games per pairing) took a baseline as well. Then a trivial modification to the for-loop allowed me to queue the tasks that play candidates against the baseline.

This trivial solution had the unexpected result. It polluted the output statistics regarding the performance of candidates. The solution to this is more of a hack in nature. Each deck in the library has a name field that can be populated as we wish via a setter. I have used the name field to tag candidates as 'candidate' and baseline decks as 'baseline'. Before returning results to Haskell, I perform a small post-processing where I filter out the statistics related to baseline decks.

Next, for the random baseline case, we want to generate random decks to test against. The library already had a random deck generator hidden within it, and the extension was simply to uniformly at random pick an in-game class for which we will generate a random deck to include in the baseline.

Later in the project, I wished to include card level utility. The library already kept track of how often each card is played. The issue with using this directly is that some cards, due to being randomly drawn, might end up in the hand of the player more often than others. Since cards that do not end up in the player's hands cannot be played, we need to normalise the number of times each card was played by the number of times the card was drawn into the player's hand. By adding a statistic to keep track of cards drawn and a slight alteration to the return value, this new utility was included into the library.

This notion of card utility works in a similar fashion to the concept described by players as dead cards. Cards that are dead are cards that are in the hand and are not being played. If this happens, the value of times drawn will be incremented while the value of times played remains unchanged, therefore reducing the card utility.

Next, the library had to be adapted to be a server that can accept Apache Thrift RPC calls. The server itself is a trivial implementation that opens a port on the localhost.

[7]Playing the same two decks against each other using the same AI for a given number of times

When it receives an RPC call, it delegates it to a handler. So to be able to call Java from Haskell to have simulations run for us, I needed to move the logic into the Handler.

Finally, I had to decide on a contract between Haskell in Java so that I may pass the arguments easily. To avoid complications due to some features of Apache Thrift not being fully implemented in Haskell, I decided to encode the data between them as Strings in a Nested CSV format. Due to this choice of format, parsing and conversion on both sides to the respective correct data types was fairly straightforward. The chosen grammar for the communication of extra arguments is as follows:

$< class >= ?all\ in-game\ classes?;$

$< id >= ?all\ cards\ in\ the\ card\ catalogue?;$

$< deck >=< class >, ",", (< id >, ",")^{30};$

$< int >= ?all\ positive\ integers?$

$< extraArgs >= (< deck >, ",")^{+}, (< int > |(< deck >, ",")^{+})^{?}$

Here in the extra arguments, the first part is the description of the population and the second part which is optionally included provides details regarding the baseline when this is necessary. The implementation can be seen in Appendix A (SimulatedGameHandler.java), method parseListOfDecks(...).

Now with regards to results, the format is thus:

$< float >= ?double\ precision\ float\ numbers\ \in [0,1]?$

$< result >= ((< id >, "|", < float >, ",")^{1-30}, ":", < float >, ",")^{+}$

The implementation of score updates can be seen in Appendix A (EvolutionCodeForHS.hs) function updateScores, in particular the cardScoreParse code.

3.1.8 Glueing things together

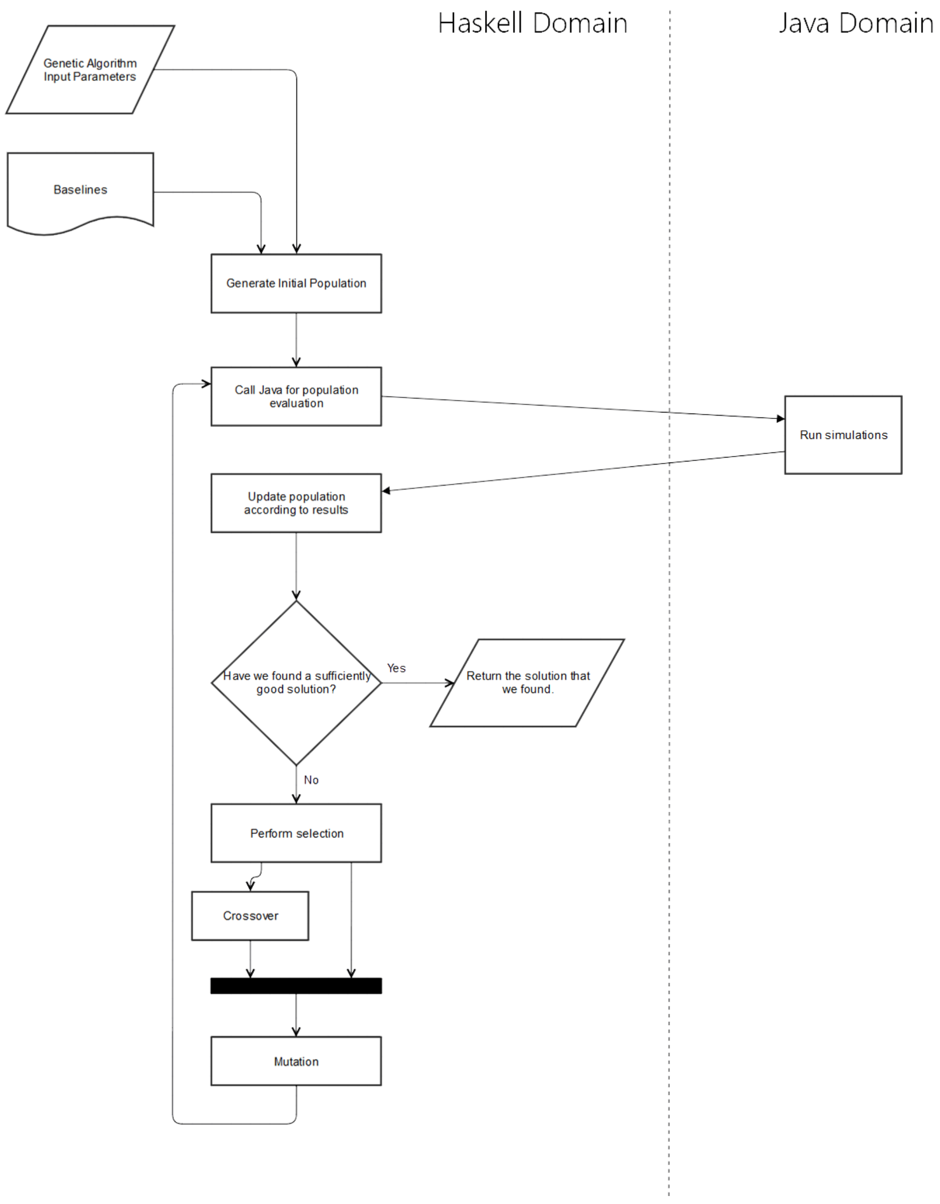

Haskell Domain
Java Domain
Genetic Algorithm Input Parameters
Baselines
Generate Initial Population
Call Java for population evaluation
Run simulations
Update population according to results
Have we found a sufficiently good solution?
Yes
Return the solution that we found.
No
Perform selection
Crossover
Mutation

Figure 3.5: A high-level overview of the Control Flow within the implemented solution.

Now that we have all of the building blocks, we want to chain them in a reasonable way so that we may progress from one population to the next and find a solution.

The first step is to generate a correct initial population. It might be the case that this random search generated a sufficiently good solution. Additionally, the requirement for the input to selection is that the population is already evaluated, so our next step is scoring.

To perform scoring, there is a small overhead related to translating the current population into a format suitable for Java according to the grammar previously defined. Then we create an instance of an Apache Thrift client that can connect to the Java server, and perform an RPC with the translated representation. We then wait for a return value for the RPC and parse it to update our population. We sort our output before returning it.

Having parsed the results of the simulations, we check if we have found a sufficiently good solution by peeking at the score of the head value in the population, returning the found solution if that is the case. Note that this check also includes an error estimation at 99% confidence.

If we have not returned, the scored and sorted population is fed into selection. We generate the first version of the next population by calling crossover for each selected pair and concatenating the selected candidates and the offspring we have generated for them by crossover.

We now have a population formed from the top candidates from the previous time-step along with their offspring. We now map mutation across the population to avoid premature convergence.

The result is fed back to evaluation to complete the cycle.

3.2 Extending to mixed populations

Extending to support the evolution of candidates that may be different in-game classes, apart from some book-keeping related changes, forced only two major refactorings of the single-class implementation. I needed to support a method of creating a population that contains different classes and I needed to make sure that the only operator that worked with more than one candidate at a time, i.e. crossover, could cope with generating, in a reasonable way, valid offspring decks for two input parent decks.

3.2.1 Initial Population

The initial population generation has simply gained one level of indirection. To obtain a mixed population containing candidates for each in-game class, we need to call the generator for a single class once for each class. To avoid having to input the classes, we can have a class index containing the possible valid values, hence:

```
genMixedPopulation :: [Int] -> [Int] -> [Deck]
genMixedPopulation ns seeds =
assert ((length ns) == 9)
(assert ((length seeds) == 9)
(genMixedPopulation' ns seeds (tail heroClassIndex)))

genMixedPopulation' :: [Int] -> [Int] -> [String] -> [Deck]
genMixedPopulation' [] [] [] = []
genMixedPopulation' ns seeds heroindex =
(genPopulation (head ns) (head heroindex) (head seeds))
++
(genMixedPopulation' (tail ns) (tail seeds) (tail heroindex))
```

Note that the genPopulation is the previously defined method that generates a population for a single in-game class.

3.2.2 Crossover

The generalisation of crossover to allow mixed populations was not as straight-forward as the initial population. We needed a way of choosing an in-game class for the offspring, and as such a notion of a class dominating at crossover over a different class. To perform this a new utility was computed, that of a class. This new utility was defined as the average fitness of a candidate of that in-game class. We do not need to worry that this utility is undefined when there are no candidates from a specific in-game class as we only need it for crossover. If in the future this becomes an issue, it can be smoothed by a uniform distribution.

Armed with this new utility we can define what it means to choose a class for an offspring in crossover and we can formulate the operator.

To perform crossover in a population containing candidates for different in-game classes we:

1. Check if the pair to be crossed over is for parents of different in-game classes.
2. If they are for the same in-game class, we can use the previously defined crossover operator.
3. If they are for different in-game classes, we flip a coin biassed by the class scores to choose the class of the offspring.
4. We separate the parents into class-specific and class-neutral cards.
5. We add the class specific cards for the parent that we determined with the coin flip.
6. For the class-neutral cards, we perform crossover as previously defined.
7. We add the neutral crossed-over cards to the class specific.
8. If we have exactly 30 cards we return the resulting offspring.
9. If we have more than 30 cards, we sort them by score and trim to 30 by dropping cards from the end.
10. If we have less than 30 cards, we add valid cards at random until we have exactly 30.

The reason why we don't add the class-specific cards to the neutral of the chosen parent and crossover those for a first approximation of the offspring is to induce a class-card bias in the algorithm as class cards are generally more powerful. This decision is, of course, a heuristic.

Precondition: The input is two valid decks and a random number generator along with a crossover probability in $(0; 1]$

Postcondition: The output is:

Either one of the inputs and a random number generator that has had its state changed if the output has a score;
Or a deck that contains all of the class-cards of the parent that shares an in-game class with the offspring along with all of the neutral cards that are in the intersection of the neutral cards of the inputs and a random number generator with a changed state.

Chapter 4

Evaluation

In this chapter, we will explore the results of the algorithm in terms of ability to find a solution, time required to find a solution, numerical quantification of optimisations performed. We will also look at the impact of the genetic algorithm hyperparameters (here elitism, mutation and crossover probabilities), however, we will not attempt to find the optimal ones as that generally requires an additional algorithm on top of the system that is able to learn the optimal choices.

4.1 Random baseline

For a randomly generated baseline, the results are for the following configuration:

200 game match simulations per pair for fitness estimation;

60 % Elitism, 10 % Mutation probability, 80 % Crossover probability

Homogeneous populations of 50 candidates and 20 runs per in-game class

Heterogeneous populations of 12 candidates per class (108 total) and 20 runs

The algorithm was allowed to run for at most 100 generations

As such 200 runs of the algorithm were performed. Each one found a solution within a single generation. The average quality of the best candidate in the first generation was found to be $71.7\% \pm 13.6\%$, where the quality cut-off is 59.1% to determine that the found candidate is good enough when running 200 simulations per pair.

This result is equivalent to stating that with high likelihood a solution to the core problem can be found by random search guided by the following heuristic:

When constructing a random deck of cards first choose with equal probability whether the card to be added is class-specific or not.

4.2 Predefined baseline

When performing a search for good enough candidates against a user-provided baseline, the following configuration was used:

200 game match simulations per pair for fitness estimation;

40 % Elitism, 10 % Mutation probability, 80 % Crossover probability

Homogeneous populations of 100 candidates and 20 runs per in-game class

Heterogeneous populations of 12 candidates per class (108 total) and 20 runs

The algorithm was allowed to run for at most 100 generations

The baseline was constructed to represent the most popular card choices in the month of December according to [13].

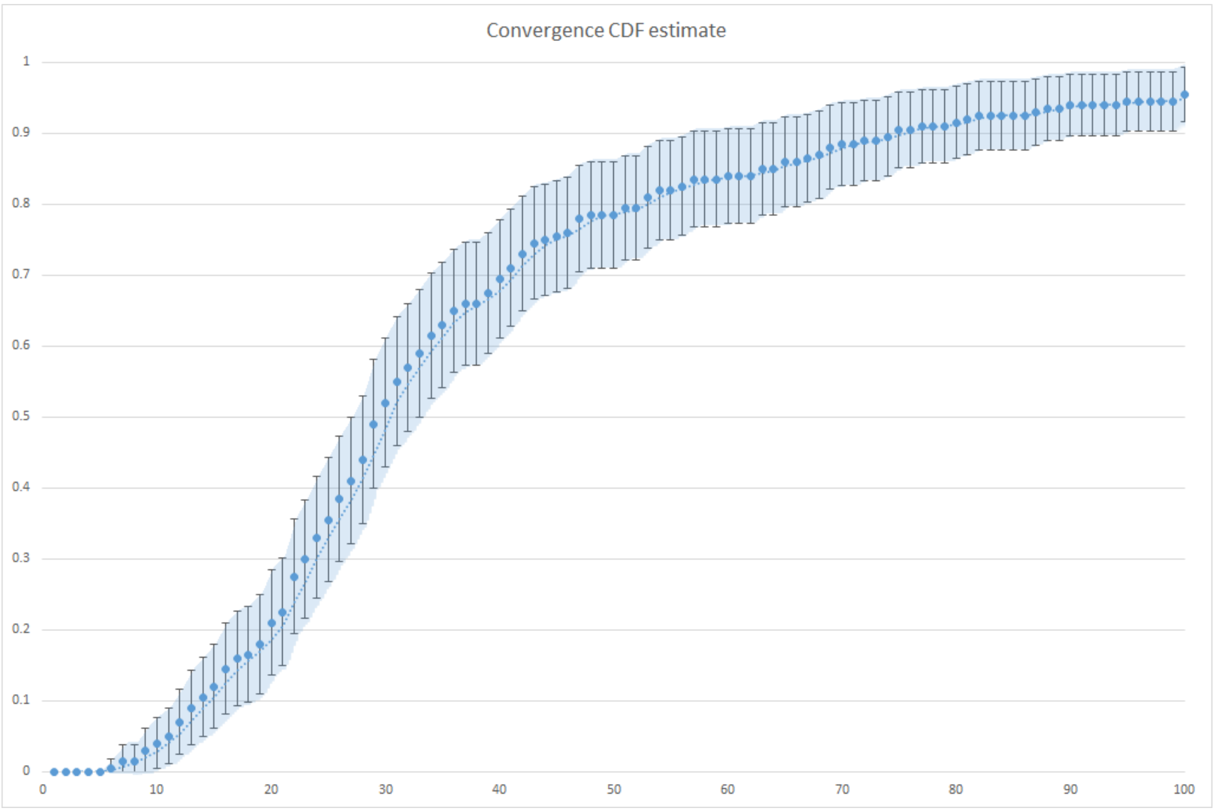


Figure 4.1: An estimation of the CDF of the convergence of the algorithm against a user baseline with hyperparameters: $e = 0.4, m = 0.1, c = 0.8$

During these runs, I have found that the algorithm converges, i.e. finds a solution according to the quality criterion, at the 100th generation with $95.5\% \pm 3.8\%$ likelihood, and it took on average 37.4 ± 60.7 generations to converge.

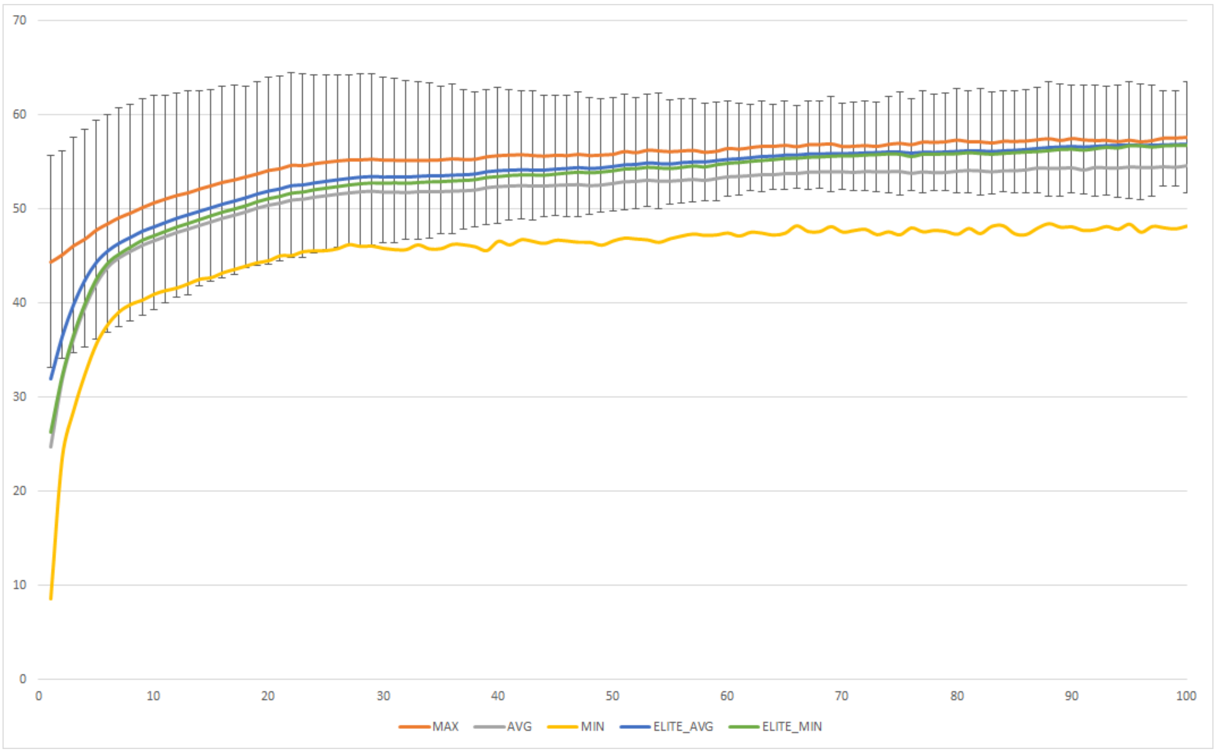


Figure 4.2: An estimate of candidate fitness evolution against the number of generations. Since the data was generated with 200 simulations per candidate-baseline pair, the quality cut-off is 59.1%. Note that the errors are due to a combination of variable number of sample points per generation and the natural variance of the values we are measuring.

From (fig. 4.2) we can see that most of the quality increase were within the first 10 generations, and the maximum fitness value stops increasing by much towards the 60th generation. This logarithmic behaviour is expected and is telling towards the fact that a higher quality threshold than the one attempted here might be exponentially harder to achieve.

Additionally, an issue with the approach was found in the form of stale populations. If we look at the last 10 generations in (fig. 4.2) we can see that there is little variance between the elite average, elite minimum and the population maximum (the missing error bars are not relevant for this comparison as the algorithm treats non-maximum scores as if they were known perfectly). When there is little variance within the elite section of the population, crossover creates only clones due to how it was implemented. Making progress is then only attainable via mutation. In these scenarios when only selection and mutation work, we are performing a slower version of hill-climbing within the search space. A potential solution that was not explored was to attempt using a different selection operator or bundling like candidates, so that multiple copies of the same candidate are considered only once.

Finally, to validate the optimisation choice, I have measured run-time of the first generation, where we always must evaluate the whole population, to use as an estimate for the non-optimised code, and I have measured the run-time of all post-first generations to quantify the optimisation. The results can be seen in (fig. 4.3) and are as expected.

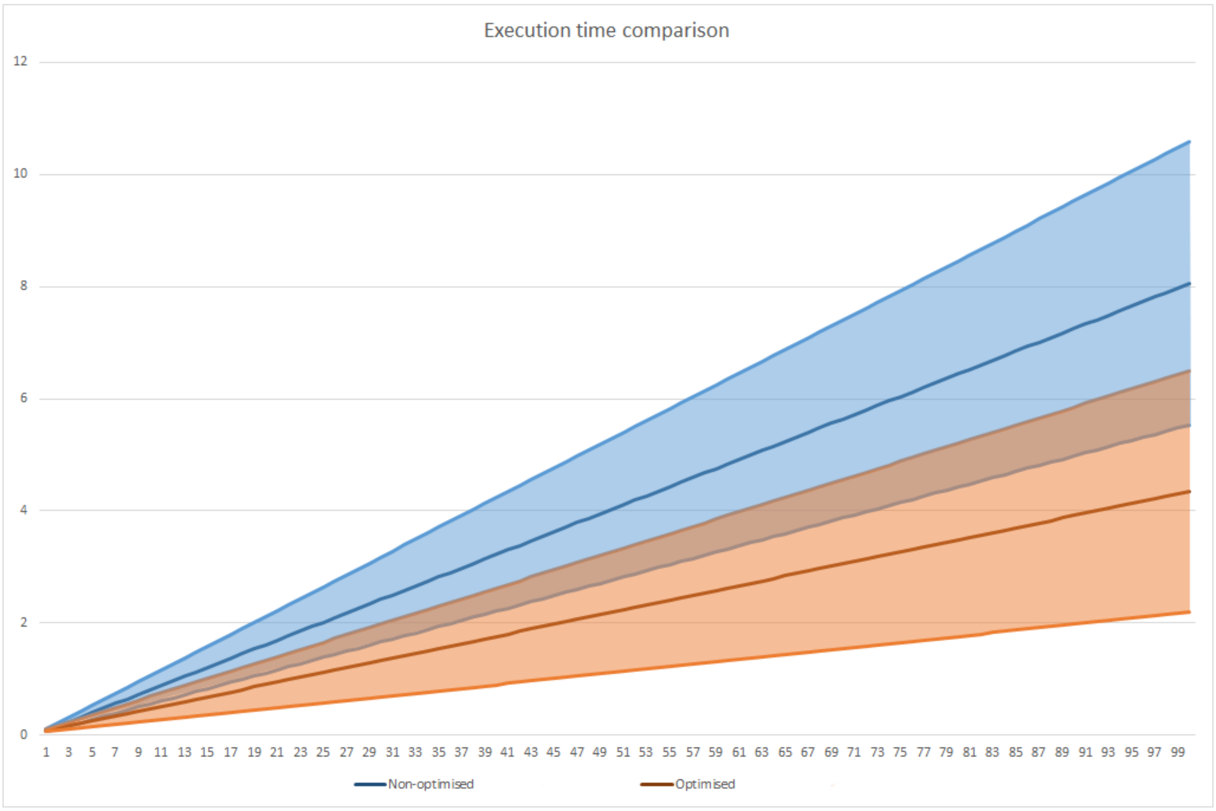


Figure 4.3: A comparison of optimised and unoptimised run-time lengths. The coloured regions represent the respective confidence intervals.

4.3 Dynamic landscape

For dynamic landscape evolution the following configuration was used:

100 game match simulations per pair for fitness estimation;

60 % Elitism, 10 % Mutation probability, 80 % Crossover probability

Heterogeneous populations of 12 candidates per class (108 total) and 20 runs

The algorithm was allowed to run for at most 20 generations

The reduced number of maximum generations was due to time constraints. In real-world time, 20 generations for this configuration translates to an expected maximum of 24 hours per run.

As expected at a maximum of 20 generations the convergence rate was rather small at 15% $\pm$ 20.5%. This was, nevertheless, a worthwhile computation as we can still observe what in-game classes survive within the simulation. These results can be seen in the following figure (fig. 4.4).

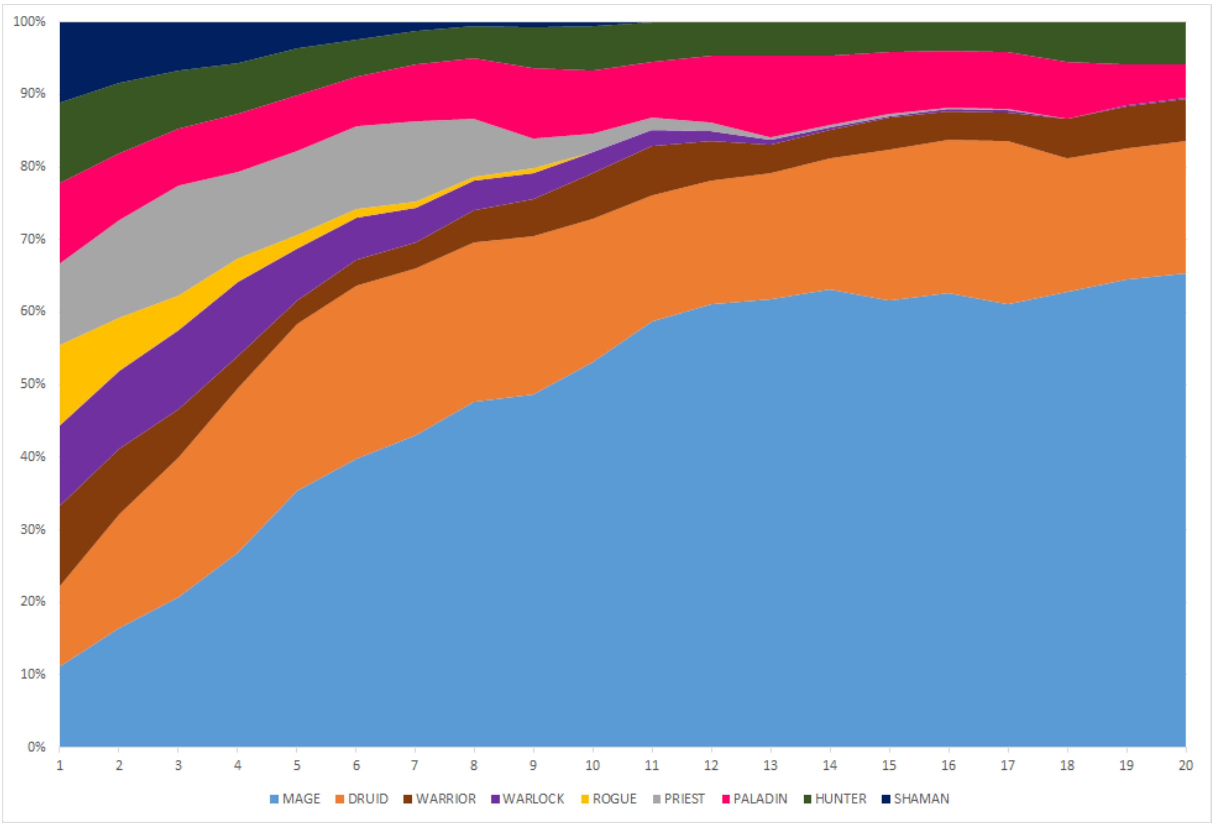


Figure 4.4: The population dynamic as generations progress coloured as a function of the candidate's in-game class. Provides a perspective in what classes are easier to play more efficiently for a greedy Board State Value AI.

From (fig. 4.4) we can see that the Mage in-game class quickly starts to dominate populations. This phenomenon is expected as with regards to basic cards, cards provided for free to all players, the best methods to control the board state is given to the Mage class via cards that can be used both offensively as well as defensibly. This provides an advantage to the type of game playing AI that was used as part of the project. Of future interest might be the impact on card choices and in-game class that manifest themselves in dynamic evolution when controlling for different AI types.

Additionally, the behaviour of candidate fitness is as expected, a slow convergence towards a population that has candidate fitness clustered at 50% as can be seen in (fig 4.5)

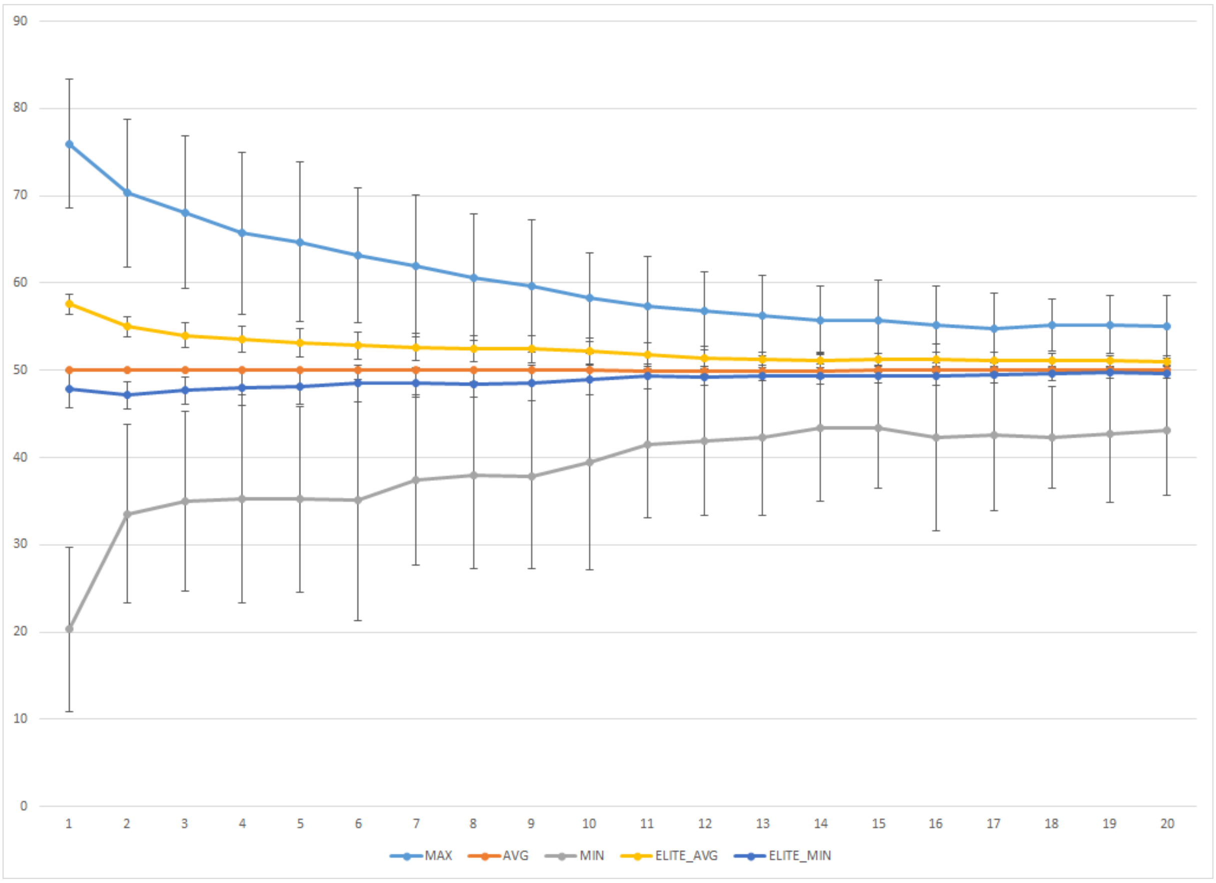


Figure 4.5: The evolution of candidate fitness in dynamic evolution exhibits a tendency towards an equi-fit population.

4.4 Exploring the impact of hyper-parameters

For the purpose of determining the impact of hyper-parameter choice on how fast the algorithm finds a solution the following configuration has been used:

200 game match simulations per pair for fitness estimation;

40 % Elitism, 10 % Mutation probability, 80 % Crossover probability, when the respective hyper-parameter is not the current control one.

Homogeneous populations of 100 candidates, in-game class Mage, 20 runs per control value of hyper-parameter

The algorithm was allowed to run for at most 100 generations

The baseline was constructed to represent the most popular card choices in the month of December according to [13].

Due to the reduced dataset, comparing the convergence of the algorithm under different sets of crossover, mutation and elitism directly was not an option. To work around this, the following method was used. The CDF of the convergence of the algorithm exhibits a logistic S-curve shape. We can use this to our advantage and compare how the slope of the S-curve varies between control variables for the section where the curve goes from 0 to 1 for a notion of speed of convergence together with the average number of generations until convergence for a notion of location. The later value has to be scaled by the optimisation factor, the gain we obtain from our reduced re-evaluation under a given set of hyper-parameters.

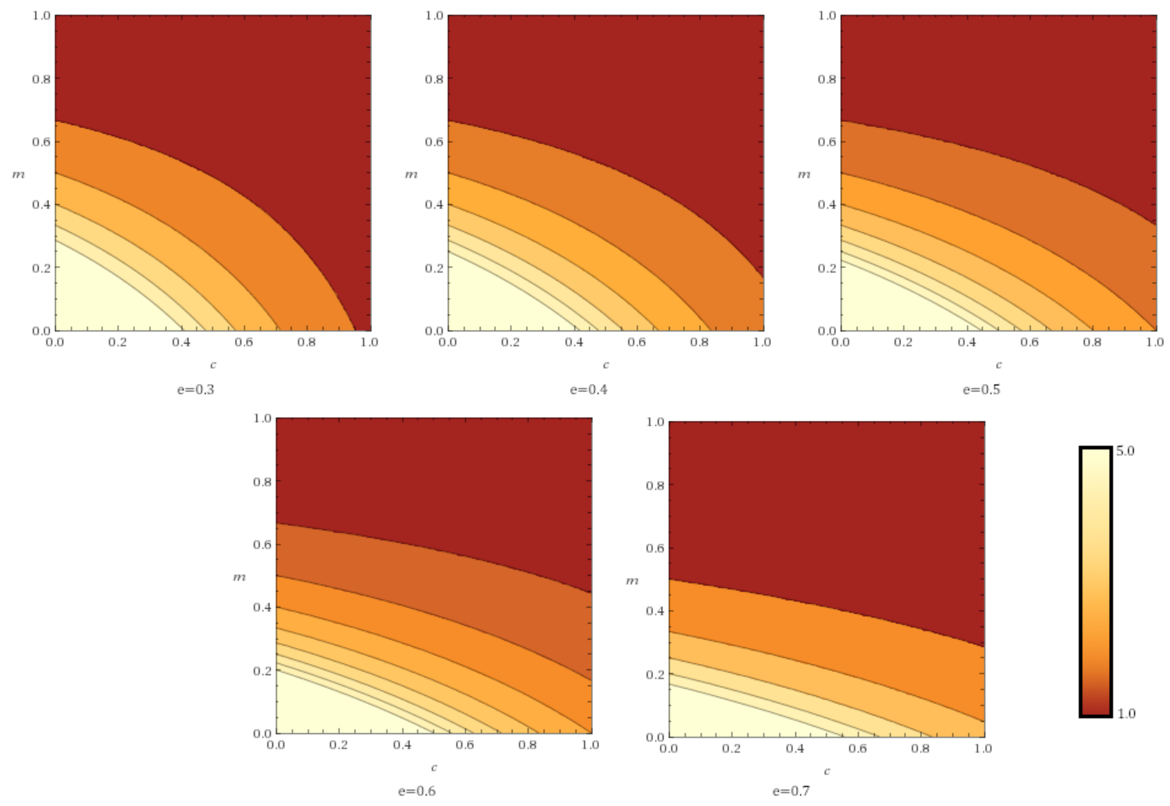
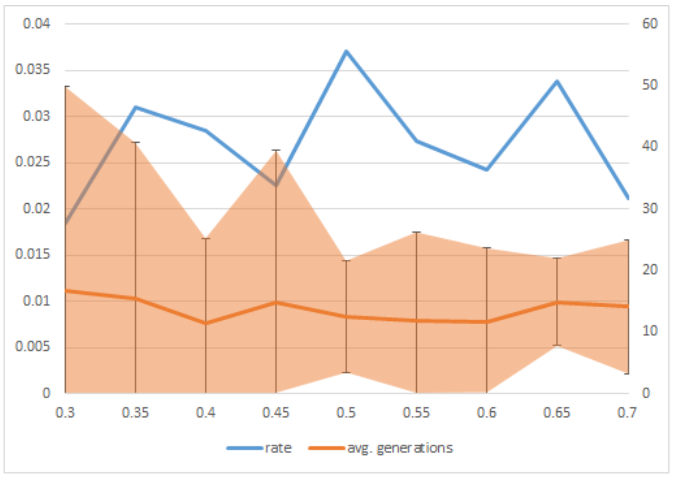


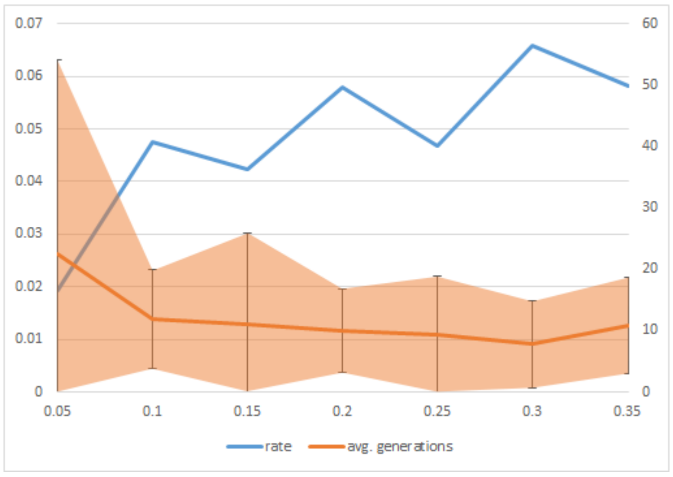
Figure 4.6: Speed-up factor due to evaluating only changed candidates under different choices of crossover, mutation and elitism.

From this (fig. 4.6) we can immediately see that we might want to keep mutation and crossover low while we keep elitism high to maximise the gain from the optimisation. This might, however, hinder the ability of the algorithm to find a solution, since such settings would reduce the number of fresh candidates we create at each generation. In the following figure(fig. 4.7) we can see how the slope and location of the jump within the S-curve vary with different values of the control variables.

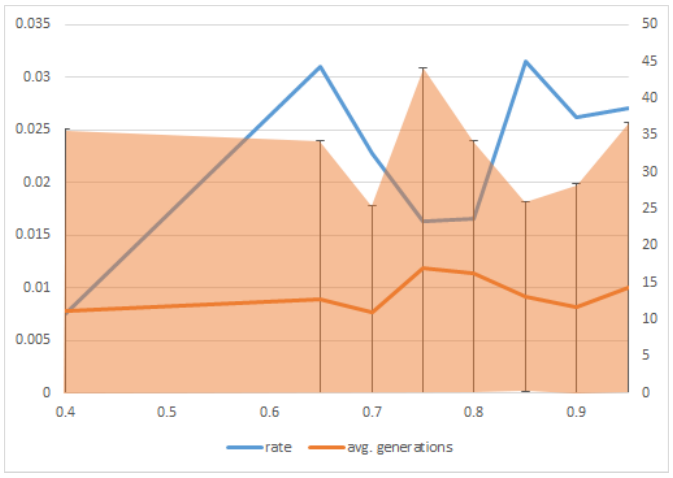
While the perspective here is fairly restricted, we can still observe the impact in broad terms. Mutation, which in the context of being used with selection, is effectively hill-climbing. A higher mutation is synonymic with performing hill-climbing for more candidates, the price is, of course, a reduced speed-up from the optimisation, however, we can see from (fig. 4.7) that the reduced speed-up is less than the gain from a larger mutation converging faster in terms of a number of generations.



(a) Elitism



(b) Mutation



(c) Crossover

Figure 4.7: The impact of elitism, mutation and crossover in that order on the slope(blue) and location of the jump(orange, shaded region represents the confidence interval) of the S-curve shaped CDF of convergence.

Crossover is tightly coupled with selection and hence with elitism within this project due to implementation as it only affects regenerated candidates. A low crossover means we will clone more of the good candidates to regenerate the missing population while a higher crossover will generate more new candidates. As such we wish a higher crossover to ensure we do not degenerate into stale populations as previously discussed in this chapter. This is loosely confirmed by both a higher slope as well as a closer centre at $c = 0.85$.

Elitism provides a free ticket for good candidates to persist across generations. The smaller the elitist generation, the more impact from the few that are within this section. On the other hand, the higher the elitist portion, the fewer candidates are generated by crossover, and hence, we might explore the search space too slowly. This nature can be observed as well from the multi-peaked behaviour of both the slope and centre of the convergence CDFs when varying elitism.

This provides an intuition regarding the trade-offs we have to take when choosing these parameters. Of future interest might be generalising these observations regarding speed-up factor, convergence rate and average number of generations until convergence into a fitness function. This would allow us to run a more traditional genetic algorithm implementation to determine the optimal choice.

Chapter 5

Conclusion

This project explored solving the problem of deck building using a genetic algorithm that was seeded with a random initial population. This represents a single-goal optimisation problem within a space that did not easily admit a binary string encoding of potential solutions without too much loss of information. However, tricks and arguably small hacks were possible due to the availability of domain knowledge.

One such trick is using domain knowledge to improve the initial guess. The first population did not originally have a bias to favour class cards, and since neutral cards out-number them, using a uniform distribution favoured choosing neutral cards. This resulted in sub-par initial populations and slow convergence. However, it is known that class cards are made to be more powerful on purpose by the game designers. A simple heuristic to choose the card type first improved initial population quality and as such convergence rate. Since the genetic algorithm is a method of refining found solutions to a certain degree, work can be done towards further improving initial populations. These improvements can take one of the several forms.

The first improvement would be to add a bias via a card score at the catalogue level. This could be done using statistics from sites such as "www.hearthpwn.com" with regards to how often a card is used. The scores might have to be manually rebalanced, though.

A different method would be to employ a deck type classifier and a soft clustering algorithm to partition the catalogue by deck type. The assumption is that cards within the same cluster interact in positive rather than negative ways with each other. We can then build candidates randomly only up to a point, at which we try to classify them and bias the following cards depending on the tag. Effectively trying to create a form of self-fulfilling prophecy in the hope of card-level consistency in choices.

The final method of improving the initial population that I propose is using the other idea mentioned in Preparation as a potential solution to the problem. That is, we learn card combinations that work well together, and card combinations that work well together against a specific opponent deck. We then use this algorithm to build the initial candidates by choosing pairs and triplets of cards, potentially combining with the previous two suggestions.

Do note that the third suggestion might provide the deck type classifier as a side-product.

The other shortfall of the project is the use of a Greedy Board State Value AI. The decision was taken to ensure that sufficient simulations can be run. As such, another improvement would be to create an optimised Monte Carlo Tree Search based AI for Hearthstone and use this new AI within the context of this project. This would remove the bias that a Greedy AI brings to the table. More specifically, a Greedy AI is unable to look more than one board state step ahead and decks that rely on counter-intuitive strategies such as Handlock where we wish to take damage, something seen as a negative utility by the Greedy AI. A better AI would allow us to find decks that make use of long-range strategies that work well against a proposed baseline.

Additionally, a better AI could provide better meta-game shifts predictions when used together with dynamic landscape evolution. Sadly, as was previously seen, this type of evolution is considerably more time consuming as the number of pairs to be estimates grows as n^2 in the size of the population, and could remain intractable until improvements are done towards the time-efficiency of game-playing AI.

Another point of interest could be the hyperparameters and operators. While we explored the impact of the hyperparameters on the convergence and convergence rate of the solution, we did not attempt to optimise these choices. The intuition provided here coupled with an additional optimisation refinement algorithm could be used to learn the best choices of mutation, crossover and elitism. Of course, this could also be paired with an additional dimension of choosing the selection operator type as well. This latter suggestion is mostly as an idea of combating the issue of stale populations that can occur with elitist selection.

Finally, the project provided an interesting setting to unite algorithmic and domain knowledge to obtain results under time-constraints and provided interesting insight into the game structure itself. Some of the results that we did not focus on during this paper aligned fairly well with community predictions or complaints, such as some of the most used cards by the AI being those that people complain are 'broken' or 'over powered'. It would be interesting to see if such insight could be coupled with and used during the Quality Assurance process of the game development cycle.

Bibliography

[1] C. B. Elie Bursztein, “Predicting hearthstone opponent deck using machine learning.” https://www.elie.net/publication/i-am-a-legend, 2014.

[2] K. Claessen, “Quickcheck.” https://hackage.haskell.org/package/QuickCheck.

[3] M. Mitchell, *An introduction to genetic algorithms.* Complex adaptive systems, Cambridge (Mass.): MIT press, 1996. A Bradford book.

[4] “Hearthstone simulation & ai.” http://hearthsim.info/.

[5] J. Leclanche, “Fireplace.” https://github.com/jleclanche/fireplace.

[6] T. Eding, “Hearthshroud.” https://github.com/thomaseding/hearthshroud.

[7] “Metastone.” https://github.com/demilich1/metastone.

[8] M. Zopf, “A comparison between the usage of flat and structured game trees for move evaluation in hearthstone,” 2015.

[9] D. Taralla, “Learning artificial intelligence in large-scale video games: A first case study with hearthstone: Heroes of warcraft,” 2015.

[10] C. Ward and P. Cowling, “Monte carlo search applied to card selection in magic: The gathering,” in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pp. 9–16, Sept 2009.

[11] P. Cowling, C. Ward, and E. Powley, “Ensemble determinization in monte carlo tree search for the imperfect information card game magic: The gathering,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, pp. 241–257, Dec 2012.

[12] S. Donaldson, “Towards a typology of metagames,” in *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW ’16, (New York, NY, USA), pp. 73:1–73:4, ACM, 2016.

[13] “Tempostorm metasnapshot.” https://tempostorm.com/hearthstone/meta-snapshot/meta-snapshot-38-there-can-be-only-one.

Appendix A

Source Code

A.1 EvolutionCodeForHS.hs

```
-- Trimmed for Apendix --
-- This represents only part of the source code that may be referenced from the dissertation text --
-- Helper functions to load the cards, create random decks
-- The type signatures for the game speciffic data, aka. Chromosome structure
type Card = (String,String,Integer)
type Deck = (String, DeckCards)
type DeckCards = [((Card,Integer),Double)]
type Catalogue =[[Card]]
type ScoredDeck = (Deck, Maybe Double)
type ScoredClassIndex = [(String, Double)]
type ScoredClassIndexUnnormed = [(String, Double, Int)]

-- Trimmed --

-- Random generation of a deck
genEntity :: (RandomGen g) => String -> g -> (Deck,g)
-- On average I am offering 10 retires per card,
-- this is in case we get a rndGen that will become stale and try to add the same card endlessly
genEntity heroClass rnd = genEntity' rnd (filterByClass heroClass catalogue) (heroClass,[]) 0 300

genEntity' :: (RandomGen g) => g -> Catalogue -> Deck -> Integer -> Integer -> (Deck, g)
-- In case we run out of tries, we reset, also, I know I am hacking based on my own data structs
genEntity' rnd catalog _ _ 0 = genEntity' rnd catalog ((\(x,_,_) -> x) (head ((catalog !! 1))),[]) 0 300
genEntity' rnd _ deck 30 _ = (deck, rnd)
genEntity' rnd catalog deck count tries =
    genEntity' rnd'' catalog deck' count' (tries-1)
    where
        (cardToAdd, rnd')  = getRandomCard rnd catalog
        (copiesToAdd, rnd'') = randomR (1,2) rnd'
        deck' = if (copiesToAdd == (2::Int))
        then (fst deck,(addToDeck cardToAdd (addToDeck cardToAdd (snd deck))))
        else (fst deck,(addToDeck cardToAdd (snd deck)))
        count' = cardCount deck'

-- Trimmed --

-- For mutation choose a card to mutate from a triangular distribution,
-- assuming cards sorted by score, and replace with high likelihood a low scoring card.
mutation :: (RandomGen g) => Double -> g -> ScoredDeck -> (ScoredDeck, g)
mutation mutProb rnd candidate =
    if (roll < mutProb)
    then assert ((cardCount (fst resulting)) == 30) (resulting, rnd''')
    else (candidate, rnd')
    where
        scorelessDeck = fst candidate
        hero = fst scorelessDeck
        cards = mergesort (\(_,x) (_,y) -> x >= y) (snd scorelessDeck)
        catalog = filterByClass hero catalogue
        (roll, rnd') = randomR (0.0,1.0) rnd
        --We want to transform a uniform distribution on (0,1)
        -- into one that ramps up to the maximum value, we can still get 0 from the discritization!
        n = (fromIntegral ((length cards) - 1))::Double
        (fValue, rnd'') = randomR (0.0, 1.0)  rnd'
        mutatedCardIndex = floor(n * sqrt(fValue))
        mutatedCard = (fst . fst) $ cards !! mutatedCardIndex
        (finalCards, rnd''') = fillTo30 rnd'' hero (removeFromDeck mutatedCard cards)
        resulting = ((hero, finalCards),Nothing)

-- For crossover, copy over all overlapping cards, for the remaining ones, flip a biased-by-card-score coin and choose.
-- For mixed populations the hero toss is biased by hero score.
crossover :: (RandomGen g) => g -> Double -> ScoredDeck -> ScoredDeck -> ScoredClassIndex -> (ScoredDeck, g)
crossover rnd crossProb parentA parentB index =
    if (((fst.fst) parentA) == ((fst.fst) parentB))
    then (crossoverMono rnd crossProb parentA parentB)
    else (crossoverMixed rnd crossProb parentA parentB index)

crossoverMixed :: (RandomGen g) => g -> Double -> ScoredDeck -> ScoredDeck -> ScoredClassIndex -> (ScoredDeck, g)
crossoverMixed rnd crossProb parentA parentB index =
    if (roll < crossProb) then
        (let
            (hero,rnd'') = selectRndOutOfTwo rnd' heroA heroB (getClassScore heroA index) (getClassScore heroB index)
            firstTry = if (hero == heroA)
            then (deckConcat (deckConcat overlap classA) mixed)
            else (deckConcat (deckConcat overlap classB) mixed)
            (mixed, rnd''') = mixRndCards rnd'' neutralJustA neutralJustB
            howMany = (cardCount (hero,firstTry))
        in
        if (howMany == 30) then (((hero,firstTry), Nothing), rnd''') else
            (if (howMany > 30) then (((hero, trimTo30 firstTry), Nothing), rnd''') else
                let
                    (filled, rnd'''') = fillTo30 rnd''' hero firstTry
                in
                    (((hero, filled), Nothing), rnd'''')
        ))
    else
        if (parentRoll < (fromMaybe 0 (snd parentA))) then (parentA, rnd'') else (parentB, rnd'')
    where
    (classA,neutralA) = filterClassCards (fst parentA)
    (classB,neutralB) = filterClassCards (fst parentB)
    (overlap,neutralJustA,neutralJustB) = findOverlap neutralA neutralB
    heroA = (fst.fst) parentA
    heroB = (fst.fst) parentB
    (roll, rnd') = randomR (0.0, 1.0) rnd
    (parentRoll, rnd'') = randomR (0.0,(fromMaybe 0 (snd parentA)) + (fromMaybe 0 (snd parentB)))  rnd'

-- Trimmed --

-- The old crossover implementation for homogeneous populations
crossoverMono :: (RandomGen g) => g -> Double -> ScoredDeck -> ScoredDeck -> (ScoredDeck, g)
crossoverMono rnd crossProb parentA parentB =
    if (roll < crossProb) then
        let
            (overlap,mixA,mixB) = findOverlap cardsA cardsB
            (mix,rnd''') = (mixRndCards rnd'' mixA mixB)
            result = deckConcat overlap mix
        in
            assert ((cardCount (hero,result)) == 30) (((hero,result),Nothing), rnd''')
    else
        if (parentRoll < (fromMaybe 0 (snd parentA))) then (parentA, rnd'') else (parentB, rnd'')
    where
    hero = fst (fst parentA)
    cardsA = snd (fst parentA)
    cardsB = snd (fst parentB)
    (roll, rnd') = randomR (0.0, 1.0) rnd
    (parentRoll, rnd'') = randomR (0.0,(fromMaybe 0 (snd parentA)) + (fromMaybe 0 (snd parentB)))  rnd'

-- Trimmed --

-- evolution step: select -> crossover/mutate -> return next generation
-- We assume the population to have been sorted before this is called
evolutionStep :: Int -> [ScoredDeck] -> Double -> Double -> Double -> [ScoredDeck]
evolutionStep seed population elitism crossProb mutProb =
    assert ((length mutated) == size) mutated
    where
    rnd = mkStdGen seed
    size = length population
    toTake = (floor(elitism * (fromIntegral size)))
    -- First selection result is here
    workingPop = take toTake population
    -- And this is the result of piping the second result of selection into crossover
    (crossed, rnd') = crossGenerate rnd crossProb (size - toTake) (genClassScores population) workingPop
    (mutated, rnd'') = mapWithRandom rnd' (mutation mutProb) (workingPop++crossed)

-- For scoring we call the Java server for the data.
-- We only forward non-scored decks when running against a baseline.
score :: I.Int32 -> Operation -> String -> [ScoredDeck] -> IO [ScoredDeck]
score nrGames op args decks = do
    let (unscored, scored) = filterUnscored decks
    newScored <- case (unscored) of
        [] -> return []
        _ -> (do
        updatesPacked <- clientFunc op nrGames (LT.pack ((printPopulation unscored)++" "++args)) -- Java call
        let updates = LT.unpack updatesPacked
        let freshScored = (updateScores updates unscored)
        return (assert ((length freshScored) == (length unscored)) freshScored))
    return (mergesort (\(_,x) (_,y) -> x >= y) (newScored++scored))

-- Trimmed --

-- Function that takes the result from the Java call and updates the Haskell structure accordingly
updateScores :: String -> [ScoredDeck] -> [ScoredDeck]
updateScores scores decks = updateScores' (map read (snd partialParse)) cardScoreParse decks []
  where
  partialParse = (debraid (map (splitOn (==':')) (splitOn (==',') scores)))
  cardScoreParse =
    (map
        (map (\(x, y) -> (x, read y::Double)))
        (map
            (map tuplify2)
            (map
                (map removeEmpty)
                (map
                    (map (splitOn (=='|')))
                    (map
                        (splitOn (==';'))
                        (fst partialParse)
                    )
                )
            )
        )
    )

-- This function updates the deck score
updateScores' :: [Double] -> [[(String, Double)]] -> [ScoredDeck] -> [ScoredDeck] -> [ScoredDeck]
updateScores' [] [] [] updated = updated
updateScores' [] [] _ updated = updated
updateScores' _ [] [] updated = updated
updateScores' (s:xs) (cs:zs) (deck:ys) updated =
    updateScores' xs zs ys ((updateCardScores cs (fst deck), Just (s * 100.0)):updated)
```

```
-- This function recursevly finds each decks cards and updates the scores
updateCardScores :: [(String, Double)] -> Deck -> Deck
updateCardScores [] deck = deck
updateCardScores (cardScore:xs) (hero,cards) = updateCardScores xs (hero,(updateSingleCard cardScore cards))

-- This function updates a card score given a card id
updateSingleCard :: (String, Double) -> DeckCards -> DeckCards
updateSingleCard _ [] = []
updateSingleCard (id, score) (((card,copies),score'):deck) =
    if (id' == id)
    then (((card,copies),score):deck)
    else (((card,copies),score'):(updateSingleCard (id, score) deck))
    where id' = ((\(_,y,_) -> y) card)

-- Trimmed --

-- Function used to determine if we found a good enough candidate/entity,
-- it uses an estimate for the error margin at 99% confidence for
-- a given population size and updates the required min score.
-- To be used as "let isPerfect = isSufficient <some n>"
-- Maybe bring the 50 up to be changeable, or increase it? evolution is usually only 1-2 steps with 50%

isSufficient :: Integer -> Double -> Bool
isSufficient n score =
    if (score > criterion)
    then True
    else False
    where criterion = 50.0 + ((2.5759 * sqrt(score * (1-score) / (fromIntegral n))) * 100.0)
-- From central limit theorem, 2.5759 is the Z number for 99% confidence

-- Used as a stop criterion when running dynamic evolution, it is saying that the minimum and maximum
-- fitness are close enough to be considered the same.
deltaMinMaxIsSmall :: Integer -> Double -> Bool
deltaMinMaxIsSmall n delta =
    if (delta < criterion)
    then True
    else False
    where criterion = fromRational ((2.5759 / (2.0 * (toRational $ 2 * sqrt (fromIntegral n)))) * 100.0)

-- Function to know when to stop
stopCriterion :: Integer -> (Double -> Bool) -> Double -> Integer -> Bool
stopCriterion maxGeneration isPerfect score generation =
    if ((isPerfect score)||(generation >= maxGeneration))
    then True
    else False

-- Trimmed --
```

A.2 SimulateGameHandler.java

```java
// Trimmed to include only interesting functions referenced from the dissertation
/**
* Parse a string into a list of decks. Assumes a CSV format.
*
* @param mode The string used to tag the parsed decks. We use this to distinguish
*             Baseline and Candidate decks.
* @param listAsString The CSV formated string that includes the description of the decks.
*/
private static List<Deck> parseListOfDecks(String mode, String listAsString) {
    List<Deck> collection = new ArrayList<>();
    int j = 0;
    for (String deckDesc : listAsString.split(";")) {
        String[] deckAsList = deckDesc.split(",");
        // The first thing in a deck description is the hero.
        Deck deck = new Deck(HeroClass.valueOf(deckAsList[0]));
        // The name of the deck is just the order in which we provide them.
        deck.setName(mode + Integer.toString(j++));
        // To add cards, we request a reference to the collection and mutate it.
        CardCollection deckCardCollection = deck.getCards();
        for (int i = 1; i < deckAsList.length; i++) {
            // We assume to be provided valid descriptions so we do not perform validation.
            Card cardToAdd = CardCatalogue.getCardById(deckAsList[i]);
            if (cardToAdd != null) {
                deckCardCollection.add(cardToAdd);
            } else {
                logger.error("Card not found for id: " + deckAsList[i]);
            }
        }
        collection.add(deck);
    }
    return collection;
}

/**
 * Run evaluation of fitness of decks given a baseline to test against.
 *
 * @param battleConfig The configuration of the simulation.
 * @param baseline     The decks that we optimise against.
 */
protected String runTestAgainstBaseline(BattleConfig battleConfig, List<Deck> baseline) {
    result = new BattleResult(battleConfig.getNumberOfGames());
    logger.info("Battle of Decks started");
    ExecutorService executor = Executors.newWorkStealingPool();
    List<Future<Void>> futures = new ArrayList<>();

    //Queue up tournament play
    for (Deck deck1 : battleConfig.getDecks()) {
        for (Deck deck2 : baseline) {
            for (int i = 0; i < battleConfig.getNumberOfGames(); i++) {
                PlayGameTask task = new PlayGameTask(deck1, deck2, battleConfig.getBehaviour());
                Future<Void> future = executor.submit(task);
                futures.add(future);
            }
        }
    }

    //Queue up a shutdown event to collect results
    //Removed boiler-plate code for apendix
    logger.info("Battle of Decks finished");
    return printToString();
}

/**
 * Method that runs a full tournament play-off to determine absolute fitness.
 *
 * @param battleConfig The configuration we are testing
 */
protected String runFullTournament(BattleConfig battleConfig) {
    result = new BattleResult(battleConfig.getNumberOfGames());
    logger.info("Battle of Decks started");
    ExecutorService executor = Executors.newWorkStealingPool();
    List<Future<Void>> futures = new ArrayList<>();
    HashSet<Deck> processedDecks = new HashSet<>();

    //Queue up tournament play
    for (Deck deck1 : battleConfig.getDecks()) {
        processedDecks.add(deck1);
        for (Deck deck2 : battleConfig.getDecks()) {
            if (processedDecks.contains(deck2)) {
                continue;
            }

            for (int i = 0; i < battleConfig.getNumberOfGames(); i++) {
                PlayGameTask task = new PlayGameTask(deck1, deck2, battleConfig.getBehaviour());
                Future<Void> future = executor.submit(task);
                futures.add(future);
            }
        }
    }

    //Queue up a shutdown event to collect results
    //Removed boiler-plate code for apendix

    logger.info("Battle of Decks finished");
    return printToString();
}

/**
 * Method used to print to System.out the results of the simulation.
 */
private String printToString() {
    StringBuilder stringBuilder = new StringBuilder();
    result.getDeckResults().stream()
            .filter(candidate -> !candidate.getDeckName().contains("baseline"))
            .sorted((p1, p2) ->
                (Integer.parseInt(p1.getDeckName().substring(9)) > Integer.parseInt(p2.getDeckName().substring(9))) ? 1 :
                (Integer.parseInt(p1.getDeckName().substring(9)) == Integer.parseInt(p2.getDeckName().substring(9))) ? 0 :
                -1)
            .forEach(candidate -> {
                List<Card> deck = result.getDecks().get(candidate.getDeckName()).toList();
                if ((deck != null) &&  (deck.size() > 0)) {
                    for (Card card : deck) {
                        String id = card.getCardId();
                        double score = ((double) candidate.getDeckStatistics().getCardsPlayed().getOrDefault(id, 0))
                            / ((double) candidate.getDeckStatistics().getCardsDrawn().getOrDefault(id, 1));
                        stringBuilder.append(id).append("|").append(score).append(";");
                    }
                    if (stringBuilder.length() > 0) stringBuilder.setLength(stringBuilder.length() - 1);
                }
                stringBuilder.append(":").append(candidate.getWinRate()).append(",");
            });
    String scores = stringBuilder.substring(0, stringBuilder.length() - 1);
    logger.info("Scores: " + scores);
    return scores;
}
```

Appendix B

Project Proposal

Profir-Petru Partachi
King's College
ppp23

Part II in Computer Science Project Proposal

Deck building in Hearthstone

15 Oct 2015

Project Originator: Profir-Petru Partachi
Project Supervisor: Sean Holden
Director of Studies: Tim Griffin
Overseers: Richard Gibbens and Larry Paulson

Introduction and Description of the Work

A strategy card game is a category of games that have several features in common:

1. It is a game usually between two parties, each of which has a set of cards, referred to as deck;
2. There is a known way of obtaining a resource which is used to play cards;
3. The cards have a cost and text associated with them;
4. Unless otherwise specified, the text of a card is interpreted when the card is played.

Usually when tackling strategy card games, it is attempted to solve the problem of creating an algorithm that, given a deck, learns to play optimally against different opponents. This project, however, aims to tackle the less considered aspect of strategy card games, the process of selecting and optimising the selection of cards to play.

Since strategy card games differ in rules of both how to play and how to build a deck that can be used in matches, the main focus will be on one of the more simple games, Hearthstone. For the purpose of this introduction, we shall consider the game matches themselves as black-boxes and focus on the deck-building aspect.

The deck-building aspect of the game consists of a few choice points.

Firstly, a class must be picked out of 9 existing ones. This choice limits the cards that we can use, and as a consequence the combination of cards that we can chain together. To reduce the complexity of the problem, the initial attempt will focus on the situations where the class is provided as input. If time allows, the situation where the class choice can also be optimised will be tackled as an extension.

Next, once a class is chosen, 30 cards are selected out of a set of class specific and neutral cards (that can be used by all 9 classes). The rule governing card choices is:
Cards can be chosen at most twice, except if they are of legendary rarity, which can be used at most once.
Therefore for our purposes it will suffice to consider only two types of cards: legendary and non-legendary.

After choosing the 30 cards, the deck is complete and can be used to play against opponents. The deck might be suboptimal at this point and we might choose to refine it at a later point after observing how it plays out in matches.

Focusing on how players at different skill levels describe building a deck, a pattern can be observed:

1. The building of a deck starts with an idea, a seed, usually with either a theme or known win-condition, a state of the world such that the game is won on the next game turn.

2. The deck is played for a few rounds against other opponents. At this stage, suboptimal card choices are identified (too slow/costly, too situational).

3. Using the insight gained by playing the deck, cards determined to be suboptimal are replaced, go back to 2.

4. If we are satisfied with the success of the deck we stop optimising.

This process looks very similar to the high-level description of Evolutionary Algorithms:

1. Generate the initial population of individuals randomly - the first generation

2. Evaluate the fitness of each individual in that population

3. Select, optionally crossover and optionally mutate individuals to create the next generation.

4. Repeat on this generation until termination (time limit, sufficient fitness achieved, etc.)

Therefore, the main approach considered to solve the deck building problem is via the use of Genetic Algorithms and searching for possible solutions in a manner that would emulate how players usually tackle this problem.

The result of this project will be that a user can provide the program with a set of decks and their frequencies to test against and the program should output an optimised deck such that the expected likelihood of winning with this deck against an opponent of comparable skill is more than 50 %.

The reason of choosing 50 % as the sufficient criterion is due to the fact that the competitive ladder of the game can be modelled as a random walk, with the win probability matching up with the probability of going up, the aim of the ladder being to climb up.

Work has already been done with regards to predicting what decks opponents are playing from decisions at the start of the game as well as towards publicly available simulating frameworks for the game logic and AI. This work can be used as a starting point for this project in terms of how data could be handled and what data structures can be useful to encode the state of the world.

Additionally, since players like sharing ideas of how to build decks, these are provided on different game-specific websites, sometimes accompanied by a small description of the logic behind the creation of the deck and in very rare occasions data in the format of games won/lost against what class and at what rank. This information can be compiled into a suitable format to be used by the solution. More specifically, this will allow the user to input the state of the world in high-level, domain-specific terms rather than having to describe all card choices.

Links to previous work or useful frameworks

Predicting Hearthstone opponent deck using machine learning

https://www.elie.net/blog/hearthstone/predicting-hearthstone-opponent-deck-using-machine-learning

Simulator libraries and advanced rulebook

http://hearthsim.info/

Resources Required

I Overwolf SDK for the purpose of gathering data from the game, alternatively writing a log parser for the game.

II Fireplace or equivalent simulator for the game for the purpose of testing and validating as well as for the learning process of the algorithm. (http://hearthsim.info/)

III Personal computers will be used for the purpose of this project:
A laptop used for primary work (Quad-core 3.5Ghz, 32GB RAM, (0.5 + 1)TB storage space(SSD+HDD), Nvidia GTX-980M GPU with 8GB VRAM).
A tower used for offline computations (Hexa-core 3.8Ghz, 32GB RAM, 20TB storage space(HDD), Nvidia GTX-770 with 2GB VRAM).
Work will be synchronised between the two machines via Dropbox and OneDrive.
Source code will be revision controlled via a private repository on GitHub.

Starting Point

At the start of the project, a list of utilities that can be used for the purpose of simulation or data gathering has been compiled and I have familiarised myself with the interface employed by these.

The Artificial Intelligence I course provides a starting point for the algorithms that may be considered when solving the proposed problem, however, additional reading will have to be done as to adapt them to the specific use-case.

Domain specific knowledge from playing and watching tournaments and analysing professional style of play is compiled for reference during the project.

An informal interview with a few professional players regarding "How they build decks for tournaments" as well as an analysis of similar questions from interviews conducted by others (usually as part of a tournament broadcast) has been done to narrow down possible algorithms for a solution.

Software and Interface Design and Software Engineering provide a basis for how the work and code management and design might be done.

Substance and Structure of the Project

The aim of this project is to create a program capable of optimising a deck for the game Hearthstone to be viable against a distribution of opponent decks.

As such, this problem appears to be a good candidate to be tackled via a Genetic Algorithm. In particular:

1. Generate random initial decks.

2. Simulate games where an AI plays against the same type of AI, and:
   One uses a deck from the ones provided as input and the other plays a candidate (algorithm with a static landscape)
   Or
   Both play candidate decks (algorithm with a dynamic landscape).
   The probability of winning as determined by simulating games is used as a fitness function for the individual.

3. Create a fitness/usefulness function for cards and treat them as chromosomes, the bag of chromosomes will represent the genotype.

4. Create offspring in one of two ways:
   By cloning
   Or
   By performing crossover between the parents.
   This step might result in a deck that does not respect the deck building rules, so details about it will be discussed a bit further along.

5. Evaluate the fitness of the new offspring if a recombination happened.

6. Iterate until sufficient fitness is achieved or a time limit is exceeded.

When performing crossover, it might be the case that we end up with 2+ legendary cards or 3+ non-legendary cards. If that is the case, we replace the excess cards with random ones chosen uniformly from the cards that are not used in the deck so far and are valid for the considered class.

A different concern is that post-crossover the number of cards in a deck is not 30.

In this scenario, if we have less than 30, we choose random cards until we have enough.

If we have more than 30, then we start removing cards that have duplicates, least fit first. If there are no duplicates, then we simply start removing the least fit cards until we comply with the deck size requirement.

In the event that this approach does not yield the desired results and this cannot be fixed by changing the crossover or mutation probabilities the first course of action will be to change the crossover algorithm to work on card combinations rather than cards.

If the problem persists, a different approach to genetic algorithms will be considered. This will consist of:

1. Create a notion of usefulness of combination of cards. This measure will be a function of how effectively they counter card combinations played by the opponent.

2. Run simulation for the purpose of estimating the usefulness of different card combinations.

3. Build deck variations as a function of the card combinations we identify within the decks against which we optimise.
   Note that we will prioritise good choices against the more frequent decks.

4. Attempt to validate. If we fail, determine from the history of the game the least useful combination and consider a deck variation where we replace it.

Success Criterion

To demonstrate, through well-chosen examples, that the project is capable of providing a viable deck(choice of cards) when provided with a state of the world(meta-game snapshot), where viable means that the expected likelihood of winning against opponents of comparable skill that are using a random deck is more than 50% with confidence more than 99%.

Note that the harness used to validate the result will be almost identical to the one used to evaluate the fitness of a candidate during the search for a solution. However, it would be insufficient to have the search stop when it has statistically determined that the fitness is above 50% as we might want to validate against fresh random decks that were not necessarily used during the evolution process.

Extensions

If time allows, the following extensions can be considered and tackled:

1. Extend the solution to be able to optimise the deck that is generated against a target collection of decks, such that the expected win rate of the created deck is more than 50% with confidence at least 99% when playing against an opponent of comparable skill using a deck that is in the target collection.

2. Extend the crossover/deck merging algorithm to be able to handle generating a valid deck from parent decks that are from different classes. Use this to remove the requirement of choosing a class at the start of the iteration and as such allow the change of class as part of the optimisation. This might require the implementation of a notion equivalent to class fitness.

3. Create an interface for the game, that using the statistics and fitness results from the simulations and data from a live game makes suggestions to the player for possible improvements that can be made to their deck. The assumption here is that sufficient simulations have already been run offline to know approximate fitnesses of cards in a meta-game context (what opponents choose to play).

Timetable and Milestones

Weeks 1 and 2 (26 Oct 2015 – 8 Nov 2015)

Create a data store in a suitable format (JSON) for the cards that exist in Hearthstone. Create a data-store of the most popular decks (this varies from month to month but is never more than 10).

Start reading about Evolutionary Algorithms (An Introduction to Genetic Algorithms by Melanie Mitchell), sketches and pseudo-code for how to apply them to this particular problem, discussions with the supervisor to ensure a good choice of algorithm.

Milestones: A small implementation of a Genetic Algorithm searching for optimal solutions to a solved problem (Iterated Prisoner's Dilemma). The expected result will be strategies that resemble Tit for Tat.

Weeks 3 and 4 (9 Nov 2015 – 22 Nov 2015)

Write a small program that interfaces with the game and can react to game events(cards being played, player turn starting, etc.) and at the end of a match shows who much did the observed opponent cards overlapped with the popular decks.

Milestones: A small program that interfaces with the game and can find within the data-store of cards the ones being played by the opponent as well as find how much overlap there is with the popular decks.

Weeks 5 to 7 (23 Nov 2015 – 13 Dec 2015)

Write code that will later be used as part of the fitness function, i.e. code to identify cards that are effectively dead, situational cards that did not have their criteria met, or cards that were useful and attribute them a meaningful score.

Create a framework that can be used on live games so that the fitness function can be validated before it is used without a GUI as part of the evolution process.

Create the data structure that will hold enough information about a deck for the purpose of this project.

Milestones: A overlay/application that can track and identify good and bad cards given a fitness function; a valid fitness function to be used later within this project.

Weeks 8 and 9 (14 Dec 2015 – 27 Dec 2015)

Implement a framework that can use a simulator for the game to play two AI against each other with given decks. Create an interface that can read a deck data structure and interpret it as a Hearthstone deck. Create an application that takes two decks as input and provides back the history of a simulated game. Use the previously created fitness function to determine the fitness of cards in a deck.

Milestones: Ability to simulate a game of Hearthstone and compute the fitness of the cards from the history of the game.

Week 10 (28 Dec 2015 – 3 Jan 2016)

Winter holidays/break; will be used to catch up with any missed milestones if necessary.

Weeks 11 and 12 (4 Jan 2016 – 17 Jan 2016)

Implement the core algorithm that runs the framework from the previous milestone to determine the fitness of decks. Implement the code for deck crossover.

Milestones: A non-optimised version of the algorithm with the ability to run game simulations and crossover candidates until a termination criterion is met.

Weeks 13 and 14 (18 Jan 2016 – 31 Jan 2016)

Polish up the algorithm, run simulations to determine any parameters that have to be set, such as the crossover or mutation probabilities, improve the crossover step, make the simulations in parallel, generate population-wide statistics and aggregate them.

Write up a progress report for the project so far. Attempt to have statistical data before the write up to provide some empirical results along with it.

Start writing the Preparation and Implementation sections, including any data generated so far that will be relevant for Evaluation.

Milestones: The algorithm should require less time at the cost of using more resources, termination should be now within reasonable time, i.e. not required to be left to run overnight, progress report.

Weeks 15 and 16 (1 Feb 2016 – 14 Feb 2016)

Run the algorithm in different modes (optimising against a target set and optimising against other candidates). Create a validation framework that can test if the goal is met for both the core aim and the first extension, i.e. make sure the validation framework is able to generate valid random decks.

Prepare a presentation of the project for my peers.

Finish writing the Preparation section and attempt to finish the Implementation section (conditional on the validation framework being done as well).

Milestone: A framework that is able to demonstrate if the core aim is met and if the first extension is met. A presentation of the project has been given. Finished Preparation and Implementation.

Weeks 17 to 19 (15 Feb 2016 – 6 Mar 2016)

Run simulations for random seeds. Generate enough data for it to be statistically significant. Compute metrics such as runtime until termination, win probability at termination, the number of generations/cycles required. Aggregate the data and validate the algorithm, amend and retest if necessary.

Write the Evaluation section, aggregate the simulation and validation data and statistics into presentable formats.

Milestones: Solution is statistically proven to work, data provided with recreation steps, dissertation effectively written.

Weeks 20 to 27 (7 Mar 2016 – 1 May 2016)

Easter break, time will be used to catch up on missed milestones. If amendments to the parameters of the algorithm have been done, this time will be used to make sure enough data is generated to ensure proper validation. Proof-reading of the dissertation alongside peer proof-reading.

Week 28 (2 May 2016 – 8 May 2016)

Finish Dissertation, additional proof-reading, polish-up, add diagrams to better represent the data alongside the raw data. (where possible; due to the size of the data, in some cases only the graphical representation might be feasible)

Milestone: Hard-copies and electronic copies of the Dissertation are ready for submission,

Week 29 (8 May 2016 – 13 May 2016)

Milestone: Submission of Dissertation.