



UNIVERSITY COLLEGE OF LONDON

GENETIC ALGORITHMS IN THE SEARCH FOR OPTIMAL KUHN POKER STRATEGIES

Dissertation Report - MSc in Computer Science

Author:

Paula LÓPEZ POZUELO

Supervisor:

Dr. Robin HIRSCH

September 2, 2015

This report is submitted as part requirement for the MSc Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This dissertation examines the search for intelligent computer strategies in the game of Kuhn Poker, a simplified Poker variant that is played between two players with a three-card deck.

A Genetic Algorithm (GA) was fully implemented to do this, as well as a Kuhn Poker game, and they were both combined to evolve strategies for the players. The effects of different types of population initialization, selection and crossover were studied.

The results show that Genetic Algorithms can be quite efficient at generating winning strategies against a particular player by modeling its behaviour. For instance, in a game against a standard player, the best strategies evolved made \$0.23 on average if playing first and \$0.32 if playing second (the second player generally has an advantage), in a game with \$1 bets.

Some of the computer player strategies produced displayed bluffing and sandbagging behaviours as a means to take advantage of the opponent's habits - highlighting the importance of deceptive game-play in Poker.

Through the use of co-evolution, in which two populations were trained against each other, strong strategies capable of winning against a multitude of different players were found. Additionally, a mixed approach between co-evolution and population breeding against fixed opponents was suggested, and successfully implemented, for the evolution of even fitter strategies.

More complex variants of the Kuhn Poker model were implemented and examined, in which players were allowed to place different amounts of bets, bet more than once or both.

Tournament and Rank selections were found to perform significantly better than Roulette-Wheel selection, converging faster towards better performing strategies (average wins of \$0.08 in 100 generations versus \$0.02 in 500 generations). A uniformly random initialization of the first population performs slightly better than a purely random one, while ensuring an equal distribution of values throughout the chromosomes. Finally, Uniform and One Point crossover performed similarly, however, the former was used due to a slightly improved performance against some models.

Contents

1	Introduction	1
1.1	Project Goals	1
1.2	Outline	2
1.3	Context	2
1.4	Implementation tools	3
2	Fundamentals of Poker	5
2.1	Underlying rules	5
2.2	Kuhn Poker	8
2.2.1	Nash Equilibrium	9
2.2.2	Adding complexity to Kuhn Poker	10
3	Genetic Algorithms	11
3.1	Terminology	11
3.2	Exploration and Exploitation	12
3.3	Chromosome encoding	12
3.4	Fitness	12
3.5	Initial population	13
3.6	Selection	13
3.7	Crossover	15
3.8	Mutation	16
3.9	Elitism	17
3.10	Co-evolution	17
3.11	Evolutionary forgetting and Halls of Fame	18
4	Implementation	19
4.1	Card games	19
4.2	Genetic Algorithms	20
4.3	Applying GAs to Kuhn Poker	25
4.3.1	Classic Kuhn Poker	25
4.3.2	Adding breadth to the betting tree	28
4.3.3	Adding depth to the betting tree	29
4.3.4	Bringing it all together	30

5	Results and Evaluation	31
5.1	Classic Kuhn Poker	31
5.1.1	Evolution against opponent models	31
5.1.2	Co-evolution	38
5.1.3	Mixed approach	40
5.2	Performance of different techniques	41
5.2.1	Selection	41
5.2.2	Randomization	42
5.2.3	Crossover	42
5.3	Complex forms of Kuhn Poker	43
5.3.1	Kuhn Poker with added breadth	43
5.3.2	Kuhn Poker with added depth	47
5.3.3	Kuhn Poker with added breadth and depth	50
5.3.4	Co-evolution	50
6	Conclusion	51
6.1	Conclusions	51
6.2	Further research	52
	Appendices	53
A	Additional models for classic Kuhn Poker testing	55
B	Additional tests for selection methods	57
C	Additional tests for crossover methods	60
D	Results for added breadth and depth variant	62
E	Results for co-evolution in complex variants	65
F	Code for Genetic Algorithm	68
G	Code for classic Kuhn Poker	75
H	Code for cardGame package	80
I	Code for geneticAlgo package	88

Chapter 1

Introduction

1.1 Project Goals

The main goal of this dissertation is to find intelligent strategies for computer Poker through the use of evolutionary computation.

My interest lied mainly on Texas Hold'em Poker. However, due to the large number of players, betting rounds and potential hands, as well as the lack of a limit on bets, it constituted an overly complex problem to approach as a first attempt at the implementation of evolutionary algorithms. An interesting alternative is Kuhn Poker, a simplified model that involves two players, a three-card deck, one-card hands and limited betting options and rounds.

Fun and Games by Ken Binmore [2] describes a number of simplified Poker models, which are studied from a Game Theoretical perspective, and discusses the importance of bluffing and sandbagging in Poker strategies, as they can offer the player an edge in certain situations. This served as an inspiration to investigate whether optimal strategies for Kuhn Poker would involve deceptive playing in order to maximize profit.

As a way to ease into the project, a game of Kuhn Poker was implemented in Java. The core of the project consists in the full implementation of a Genetic Algorithm (GA) and applying it to the exploration and optimization of Kuhn Poker strategies.

A Genetic Algorithm is a kind of evolutionary algorithm that is used to find solutions to optimization problems. It begins with a set of random solutions or strategies, that are improved recursively through the use of techniques inspired by natural evolution (selection, crossover and mutation).

After the full implementation of the GA, algorithms were written for the different kinds of techniques that can be used throughout the evolutionary process, with the objective of applying the ones with the best performance to the problem in question.

After the GA was working correctly with classic Kuhn Poker, more complex variants of this model were implemented and examined, in which players are allowed to place different amounts of bets or bet more than once, or even both.

1.2 Outline

The two chapters following this one offer a theoretical background of the topics examined throughout the dissertation. Chapter 2 explains the fundamental rules of Poker, that are common to all its forms, and the Kuhn Poker variant is carefully described, together with possibilities to make it more complex. After this, Chapter 3 thoroughly explains the theory behind Genetic Algorithms, including a selection of different techniques that are most commonly used nowadays.

Chapter 4 focuses on the implementation process: first, a brief outline of how Kuhn Poker was implemented, but mainly, the Genetic Algorithm, together with a detailed description of the different algorithms that were used for its development. This chapter also details how Kuhn Poker can be extended in order to increase its complexity and challenge the performance of the GA.

Chapter 5 outlines the tests that were run on the code and explains the results obtained from them, while offering an evaluation of how they prove the correct functioning of the algorithm. Winning strategies were found against fixed models as well as through the use of co-evolution. With the first method, some strategies evolved deceptive behaviours to trick a familiar opponent, while the strategies obtained from co-evolution played safely in order to face many unknown players with different techniques.

Finally, the last chapter draws a conclusion on the results obtained from the project, as well as an outline of possible future lines of research.

1.3 Context

Genetic algorithms were first described in the 1970s, and have been widely used for research in the field of Game Theory ever since.

There is a considerable amount of ongoing research into the different techniques that can be used for selection and crossover, as well as the choice of parameters, and how these might affect the performance of a Genetic Algorithm. However, no optimal choice of techniques and parameters have been found, but it is rather a problem-specific matter.

Poker is very popular as a test bed in computational problems because of its complexity. It is not just about statistics and computations, but there is a wide range of information that is not available to the players. Furthermore, the potential use of deceptive strategies renders it a prime game to analyze for opponent modeling and behaviour prediction. A department at the University of Alberta [12] is devoted to computer Poker research since the 1990's, and are leaders in the field.

1.4 Implementation tools

Every algorithm that was used for this project was written from scratch in Java language, using Sublime Text editor and command line prompts, as well as the Eclipse IDE. This report was typed using \LaTeX .

Chapter 2

Fundamentals of Poker

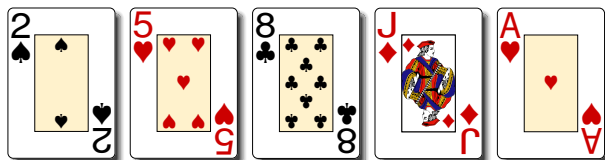
2.1 Underlying rules

Poker is, without a doubt, one of the most well-known card games in the world. Due to its popularity, there are multiple variants of Poker that constitute very different games. However, there is one thing that is common to all the variants: hands. Hands can refer to two different concepts in the context of poker,

1. The set of cards (generally 5) held by a player.
2. The set of betting rounds until a player wins the pot accumulated.

In this case, we refer to the first definition of the term. The hands that a player can obtain are divided into the following ten categories, listed in increasing order of importance:

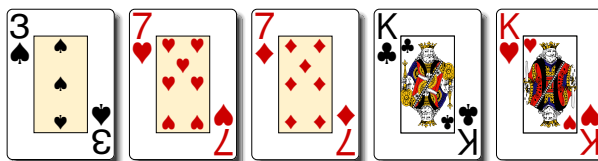
Bust or High Card



One Pair



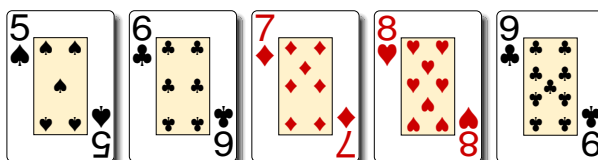
Two Pair



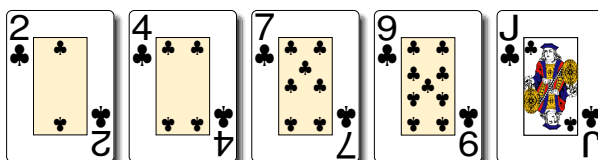
Three of a Kind



Straight



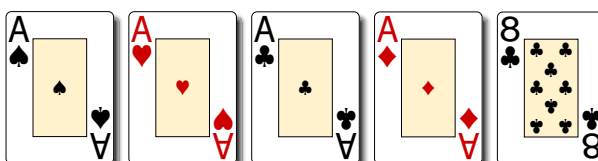
Flush



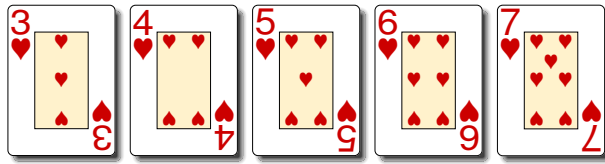
Fullhouse



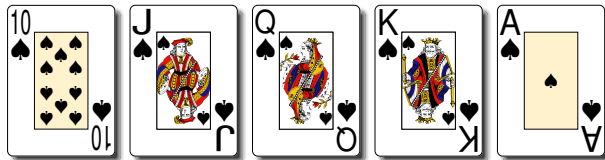
Four of a Kind



Straight Flush



Royal Flush



In the event that two or more players have hands that fall under the same category, this will be resolved on a highest card basis. For instance, a four of a kind of As $\spadesuit A \heartsuit A \clubsuit A \diamondsuit 8$, is better than a four of a kind of Ks $\spadesuit K \heartsuit K \clubsuit K \diamondsuit 8$, and a straight going onto a J $\spadesuit 7 \heartsuit 8 \clubsuit 9 \diamondsuit 10 \spadesuit J$, is better than one going onto a 7 $\spadesuit 3 \heartsuit 4 \clubsuit 5 \diamondsuit 6 \spadesuit 7$.

After the players are dealt their cards, betting begins. In the different varieties of poker, players will be forced to place *antes* or *blinds*, which are amounts of chips or money that must be placed into the pot before the hand begins. After this, players can see their cards and make different betting decisions:

- If a bet has been made previously, the player can:
 - Call: match the current bet by placing the same amount into the pot.
 - Raise: call the current bet and place an additional bet on top of it.
 - Fold: forfeit the current hand.
- If no bets have been made previously, the player can:
 - Check: pass.
 - Bet: place a bet.

If every player at the table folds except for one, this person will be the winner of the hand without the need for a *showdown* (showing and comparing the players' hands). Otherwise, when the betting rounds are over, there is a showdown and the player with the strongest

hand wins the total pot. Any draws are resolved by dividing the pot equally among the tied players.

Poker is said to be an *Imperfect Information* game because the players don't have access to all the information that would be relevant to make a decision. Examples of *Perfect Information* games are Chess and Tic Tac Toe. The fact that Poker is a game of Imperfect Information adds complexity to its resolution through the use of algorithms, since the information the computer needs in order to make accurate decisions is partly unavailable.

Another of the difficulties of poker lies on the fact that players will often show behaviours that don't correspond with their cards. When a player pretends to have better a hand than he actually has, he is said to be *bluffing*. On the contrary, *sandbagging* refers to when a player pretends to have a weaker hand than the one he has.

2.2 Kuhn Poker

Kuhn Poker is a simplified version of conventional Poker that was first described by game theorist Harold W. Kuhn in 1950. He introduced it as a means to simplify Poker enough to be able to properly analyze it from a Game Theory perspective.

In Kuhn Poker, there are only two players and a three-card deck, containing a Jack, a Queen and a King. Before the cards are dealt, each player places an ante of \$1, whose intention is to discourage excessive folding.

Each player is dealt one single card, and the third card is left upside down on the table, such that neither of the players will be able to see it. In this Poker variant the suit of the cards is not relevant, since the players only have one card and therefore can only play Bust hands. Once the cards are dealt, the two players take turns to make betting decisions. First, Player 1 has the option to place a \$1 bet or to check, after which the hand can unfold into several paths:

- Player 1 bets \$1:
 - Player 2 calls the bet by placing \$1 in the pot. The players show their cards and whoever has the strongest hand wins.
 - Player 2 folds and Player 1 wins the hand.

- Player 1 checks:
 - Player 2 places a \$1 bet.
 - * Player 1 calls the bet by placing \$1 in the pot. The players show their cards and whoever has the strongest hand wins.
 - * Player 1 folds and Player 2 wins the hand.
 - Player 2 checks. There is a showdown to determine who the winner is.

Since there is only one card of each kind, the two opponents will never draw. The potential wins (or losses) are displayed in the binary tree in Figure 2.1.

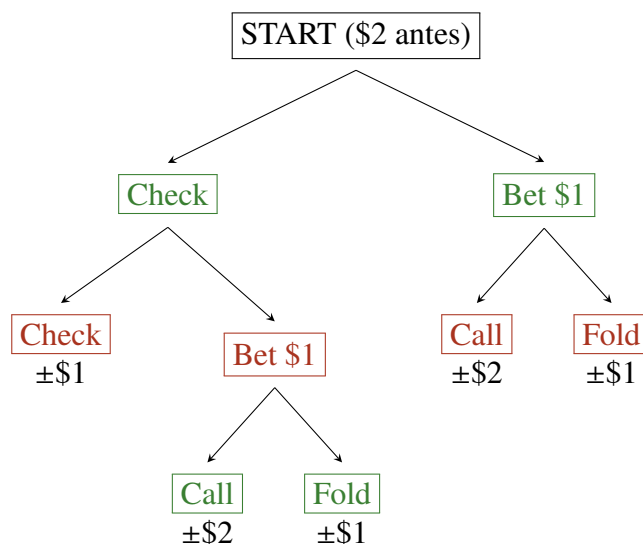


Fig. 2.1: Kuhn Poker betting tree

2.2.1 Nash Equilibrium

In Game Theory, a *Nash Equilibrium* occurs when, in any given game, every player makes the optimal strategic decisions in response to the other players' strategies [2].

A Nash Equilibrium can be reached in the game of Kuhn Poker. However, at equilibrium Player 1 loses at a rate of 1/18 of the betting amount. This means that Kuhn Poker is an unbalanced game, in which Player 1 can only expect to win by exploiting any game-play errors in Player 2's strategies, assuming that he is playing sub-optimally. This phenomenon is thoroughly described in [9].

2.2.2 Adding complexity to Kuhn Poker

It can be of interest to introduce some slight modifications into Kuhn Poker in order to test strategy-searching algorithms. Some of the potential adjustments include:

- Increasing tree breadth: it is possible to widen the betting tree by providing the players with more betting options, such as betting \$1, \$2 or \$10.
- Increasing tree depth: the betting tree can be deepened by allowing players to bet multiple times in one hand (in classic Kuhn Poker players can only bet once).
- Incorporating additional cards to the deck.
- Allowing more than two players in the game.

Chapter 3

Genetic Algorithms

Genetic Algorithms mimic the process of natural selection that takes place through the breeding of individuals and the evolution of their genes. They were first introduced in 1975 by John H. Holland, in his book *Adaptation in Natural and Artificial Systems* [11].

3.1 Terminology

In a Genetic Algorithm, there are groups, known as *populations*, of individuals or strategies. Any given individual in a population possesses a unique *chromosome* that is made of a number of *genes*, each of which determines a different characteristic of the individual. Genes can take any value within a set of *alleles* [1].

A first population is initialized at random, such that the chromosome's genes take values at random from the set of alleles. New, stronger *generations* are produced from each population, by selecting the fittest individuals and subjecting them to modifications.

Survival of the fittest takes an important role in Genetic Algorithms; fitter individuals will be more likely to be selected for reproduction (it is more likely that the weaker ones will be driven to extinction before they have a chance to reproduce) and pass on their genes onto subsequent generations. Reproduction leads to offspring being created from a *crossover* of the parent's chromosomes, with a possibility of *mutation* occurring (a gene being flipped randomly). External factors can also result in the mutation of an individual's chromosome.

A simple example can help illustrate the workings of Genetic Algorithms and explain their basic principles, and so the following sections outline the details of GAs while working through a simple example.

3.2 Exploration and Exploitation

Exploration and *Exploitation* are two vital concepts that must be understood when working with optimization algorithms. Exploration makes reference to finding new solutions outside of the current solution space, while exploitation alludes to the refinement of the existing solutions with an aim to optimize them.

In Genetic Algorithms, exploration is achieved through the randomization of the initial population and later on by mutation. On the other hand, exploitation is attained during the selection process and the crossover between the fittest chromosomes. Further details are given in the following sections.

3.3 Chromosome encoding

A potential solution to an optimization problem can be encoded in a unique chromosome, which is simply an array of values (or genes) that describe said solution.

Say the problem in question is to maximize the following function over the domain of integers $0 \leq x < 64$,

$$f(x) = x^2 - x.$$

The set of solutions can be encoded into 6-gene chromosomes, each gene representing a bit of the binary representation of x . The search space will be the binary alphabet $\{0, 1\}$, in other words, there are two alleles in this problem: 0 and 1. For example, the chromosome

1	0	0	1	1	1
---	---	---	---	---	---

represents the solution $x = 39$.

The choice of representation must make possible to **uniquely** encode and decode chromosomes, in order to avoid ambiguity.

3.4 Fitness

One of the most important stages of a Genetic Algorithm consists in the evaluation of a chromosome's fitness, which plays a crucial role in the selection process. A *fitness function* must be defined, that will provide each chromosome with an accurate measure of how well

it approximates the solution, such that it can be compared to the other chromosomes in its population.

In the problem explored previously, an appropriate and straightforward fitness function can be defined by

$$fitness = f(x),$$

since the algorithm is simply trying to maximize the value $f(x)$, so the larger $f(x)$, the fitter the chromosome.

3.5 Initial population

Every Genetic Algorithm starts from an initial population. One of the most common methods to initialize this population is by drawing random values from the defined search space.

However, this will yield a normal distribution of alleles in the genes of the population, and other methods exist to have alleles uniformly distributed throughout the population. One such method is described in [1, p. 29] as follows.

Given a Genetic Algorithm in which genes have x alleles, the size of the population n must be set to be a multiple of x . For every one of the n chromosomes, each gene is then assigned an allele from random permutations of $0, \dots, n - 1$ (modulo x). This results in a uniform distribution of values for each gene and hence a more thorough coverage of the search space.

For instance, in a problem with a binary search space, it would suffice to select an even population size, since there are only two possible alleles: 0 and 1.

Finally, in some occasions, initial populations are seeded with solutions that are known to be good, which can lead to a much faster convergence. However, this also entails a higher risk of premature convergence [1, p. 29].

3.6 Selection

The selection process in GAs builds on the idea of survival of the fittest: those chromosomes with a higher fitness value will be more likely to be selected (i.e. survive). This can be achieved through several different methods. In all of them, a given chromosome might be selected many times, or none at all.

Roulette Wheel Selection

In Roulette Wheel selection [1, 4], the relative fitness for each chromosome is calculated with the simple formula

$$\text{Relative fitness} = \frac{\text{Chromosome fitness}}{\text{Total Population fitness}}$$

Each chromosome is allocated a "slice" of the roulette wheel that is proportional to its relative fitness. The wheel is then spun and the chromosome on which it stops is selected from the population. This method of selection places special emphasis on exploitation, since the fitter individuals will gain control over the surface of the roulette, resulting in them being selected more often for subsequent generations.

Rank Selection

Rank selection, or Rank-based Wheel selection [1, 4, p. 32], is very similar to the previous method. Instead of using the chromosome's fitness to determine its weight in the roulette wheel, chromosomes in the population are ranked in order of fitness (such that the fittest has the highest rank) and the corresponding relative rank will be the weight in the roulette.

Contrary to Roulette Wheel selection, Rank selection performs exploration because it limits the leverage that the fittest individual can have over the wheel. It can also avoid premature convergence and it doesn't require fitness scaling. However, it is more computationally expensive because populations have to be sorted before the ranks can be determined [4]. Figure 3.1 displays how the weight distribution can compare between Roulette Wheel and Rank selections when performed on the same problem.

It is possible to blend Roulette Wheel and Rank selections to produce a selection algorithm that takes advantage of the exploitation generated by the first one as well as the exploratory essence of the second one. The process to do so is described in [5].

Tournament selection

Tournament selection [1, 4], is very effective and straightforward, which is one of the reasons why it has become a widely used method of selection in Genetic Algorithms.

In Tournament selection, n elements are chosen at random from the population (n is

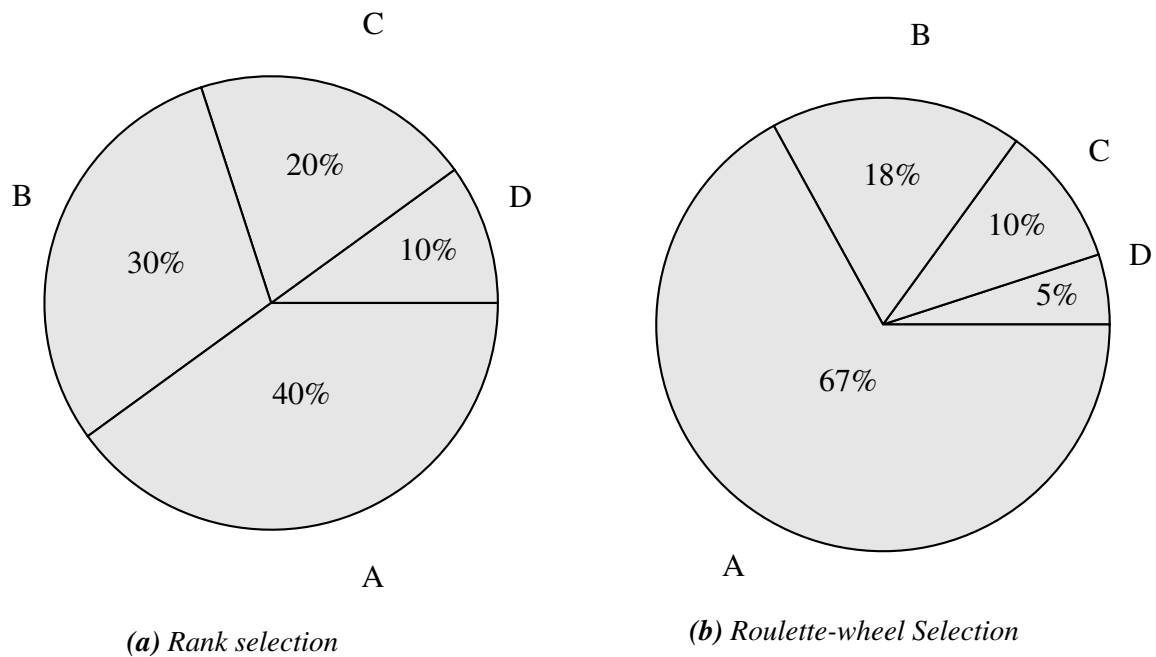


Fig. 3.1: Rank selection vs Roulette Wheel selection

referred to as the *tournament size*) to compete against each other. The fittest of the group is selected, and this is repeated until a new population is filled up. This method of selection helps maintain diversity in each new generation as long as the tournament size is kept relatively small (exploration); otherwise, it can quickly become greedy (exploitation) [6].

3.7 Crossover

Crossover takes place when two chromosomes reproduce, such that two parents merge producing two different children. Whether two selected chromosomes are reproduced or not will be subject to the chosen *crossover rate*. The crossover rate will generally be around 70%.

One-Point Crossover

Given a chromosome of length n , a *crossover point* is randomly generated (it must be an integer smaller than n). If 0 is selected as the crossover point, then the children produced will be clones of the parents. Therefore, if this wants to be avoided, only numbers starting from 1 should be allowed as valid crossover points.

Once the crossover point has been chosen, the crossover is carried out by interchanging

the parents' genes after said point. Figure 3.2 illustrates a One-Point crossover between two 8-gene chromosomes.

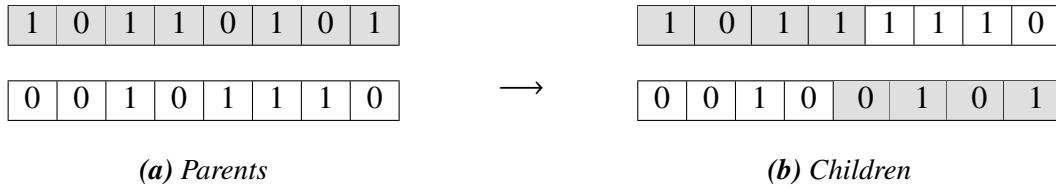


Fig. 3.2: One-Point crossover

Uniform Crossover

In Uniform crossover, children are produced by interchanging genes between the two parents, subject to a certain probability that specifies the degree of similarity to each parent. This can be a constant value, generally 50% such that each child will have half of the characteristics of each parent, or it can be randomized at every crossover. Uniform crossover is illustrated in Figure 3.3.

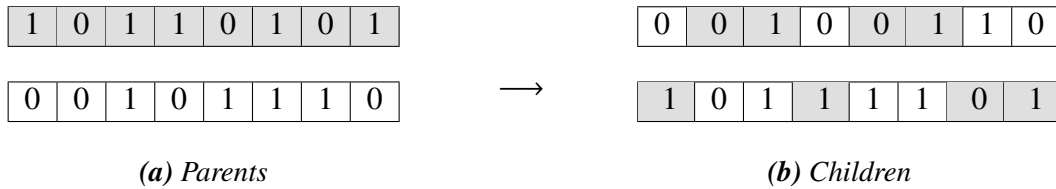


Fig. 3.3: Uniform crossover

Other types of crossover

There is a considerable amount of research into crossover techniques and their efficiency in different optimization problems. Alternative techniques have been proved to be more efficient in some scenarios, such as Selective Crossover [8].

3.8 Mutation

Mutation refers to the event in which a chromosome is altered by an external factor, unrelated to crossover. There are different schools on how mutation should interact with

crossover:

- Mutation can only occur during crossover.
- Mutation can only occur without crossover.
- Mutation can occur both during crossover or alone.

The *mutation rate* will generally be very small, usually below 1%. A higher mutation rate will favor exploration, however, it must be kept considerably small to avoid the loss of fit individuals.

3.9 Elitism

Elitism is sometimes used to prevent the strongest individuals from getting lost from one generation to the next. When a new generation is being evolved, a certain number of fittest individuals will be cloned from the previous generation and no crossover or mutation will be applied to them. This ensures that the strongest chromosomes will always survive, promoting exploitation; however, if the elitism number is too high, it can reduce the degree of exploration in the algorithm.

3.10 Co-evolution

The idea of *Co-evolution* is inspired by different species evolving in parallel in nature. Each species develops its characteristics in response to the surroundings, as well as other different species. The competing species will be evolving at the same time, and so the first species must keep evolving against the competition's new-bred strengths.

This can be imitated in Genetic Algorithms. Two populations are initialized, for two sets of strategies that are meant to compete against each other. Every time a new generation is bred, the fitness of the individuals of each population is evaluated by having them compete against all the individuals in the opposing population.

3.11 Evolutionary forgetting and Halls of Fame

The concept of *Hall of Fame* is used in [7] to overcome the difficulties posed by *evolutionary forgetting*. In non-transitive problems, the fact that A can defeat B and B can defeat C does not imply that A will be able to defeat C, which can lead to the algorithm getting stuck in a loop and not make any real progress - or even decline. This is what is commonly known as evolutionary forgetting.

The Hall of Fame stores the fittest individuals of each generation. They will not be reproduced to create new chromosomes, but the new chromosomes will compete against them, guaranteeing that the fittest individuals of new generations will defeat previous strong strategies.

Chapter 4

Implementation


A preliminary phase consisted on writing a reusable package to facilitate the implementation of any card game in Java. The core of the implementation process was the development of a full Genetic Algorithm, for which several crossover and selection methods were written, among others, to later compare their performance. Finally, the GA was combined with the card package, by having the strategies developed play Kuhn Poker games as a means to determine their fitness values.

4.1 Card games

A simple Java package, `CardGame`, was written to allow for a straightforward implementation of any game of cards. This package only includes three classes: `Card`, `Deck` and `Player`. A brief description of this package is given in this section, but the full code can be found in Appendix H.

Card

The class `Card` represents the cards that can be found in a classic Poker deck.

Each `Card` object has two private attributes: `suit` and `rank`, for which setter and getter methods exist. Cards can be compared on both of these attributes through the use of the methods `sameRank()` and `sameSuit()`. Finally, the `toString()` method was overridden, such that, for instance, the card  will return the string: "Jack of Spades".

Deck

The `Deck` class contains all the necessary methods to manage the distribution of cards within a game.

Its two private attributes are `ArrayLists` of `Card` objects, one holding the cards that are still in the deck and the other holding those that have been dealt. The constructor initializes the necessary cards and adds them to the corresponding `ArrayList`. Three methods help manage the deck: `shuffle()` (returns dealt cards to the deck and shuffles it), `add(Card c)` (adds a card to the deck) and `deal()` (removes and returns a card from the deck).

Player

The class `Player` was built to keep track of the players in a game. Every `Player` object has two private attributes: `stack`, which corresponds to the amount of money (or chips) the player has, and `hand`, an `ArrayList` of the cards that the player has been dealt in the current round.

The player's stack is managed through a getter and a setter method, as well as two other methods `win(int n)` and `bet(int n)` that allow to increment/reduce the stack by the value that is passed to them. The `ArrayList` holding the player's cards, `hand`, is manipulated similarly to a stack structure, with three main methods:

- `addCard(Card c)` - adds the card passed to it to the `ArrayList`.
- `showCard(int n)` - shows the card in the n th position.
- `popCard()` - removes and returns the first card in the hand. Can also pass it the particular `Card` object to be removed.

Finally, the method `handToString()` returns a `String` with the player's hand in square brackets, such as

```
"[Jack of Spades, Three of hearts, Ace of clubs]"
```

4.2 Genetic Algorithms

A second java package, `GeneticAlgo`, was created for the implementation of Genetic Algorithms, and it contains two classes: `Chromosome` and `Population`. The full code for

this package can be found in Appendix I.

These two classes contain all the necessary methods and sub-algorithms needed to implement the different stages of the Genetic Algorithm, and their workings are described in the next subsections.

Every Genetic Algorithm needs a terminating condition: in the case of Poker, there is no right solution that we are searching for, and so the terminating condition is set to be reached when a preset number of generations have been bred.

An initial population is generated and the fitness values of its chromosomes are evaluated. Fitness evaluation will depend on the problem that we are trying to optimise; each problem has its own fitness function. For each new generation, pairs of chromosomes are selected from the previous generation, crossover and/or mutation are performed on them (subject to the rates provided), and then they get added to the new generation until it reaches the selected population size. Once it is full, the population's fitness values are calculated and a new generation is bred following the same process.

Algorithm 1 Genetic Algorithm

```

1: procedure GENETICALGO
2:   Initialize first population
3:   Evaluate fitness of first population
4:   for  $i \leftarrow 0$  to number of generations do
5:     while new population not full do
6:       Select pair of chromosomes from old population
7:       Crossover selected chromosomes (subject to crossover rate)
8:       Mutate selected chromosomes (subject to mutation rate)
9:       Add resulting chromosomes to new population
10:    end while
11:    Evaluate fitness of new population
12:    Save fittest chromosome and average fitness
13:  end for

```

It is also important to obtain the values for the average fitness, as well as the best fitness, in every generation to be able to study the performance of the algorithm. If the algorithm is working well, the average fitness should increase with every new generation until convergence is achieved (if possible).

Population initialization

The initial population in this Genetic Algorithm is chosen at random. However, two different algorithms were implemented for the randomization of populations (section 3.5).

The first algorithm made use of pure randomization: every gene in every chromosome is assigned a randomly generated double value between 0 and 1 (alleles), producing a normal distribution.

Algorithm 2 Random Initialization

```

1: procedure RANDOMIZE
2:   for  $i \leftarrow 0$  to population size do
3:     Initilize chromosome with random genes
4:     Add chromosome to population
5:   end for

```

The second algorithm is slightly more complex, since it is aimed at producing chromosomes with uniformly distributed alleles. For every every gene in the chromosomes, a random permutation of the set of integers from 0 to population size is generated, and applied to said gene of each chromosome modulo the number of alleles (the population size chosen must be a multiple of the number of alleles). As a result, every gene will have the same number of occurrences of every allele throughout a population.

Algorithm 3 Uniform Random Initialization

```

1: procedure RANDOMIZEUNIFORMLY
2:   Initialize array with values of alleles
3:   for  $i \leftarrow 0$  to chromosome length do
4:     Shuffle array of alleles
5:     for  $j \leftarrow 0$  to population size do
6:       Set  $i$ -th gene of  $j$ -th chromosome to  $j \pmod{\text{number of alleles}}$ 
7:     end for
8:   end for

```

Selection

Three different methods of selection where implemented to test their performance with the Genetic Algorithm applied to Kuhn Poker, all of which were described in section 3.6.

Roulette Wheel selection can be implemented by calculating the total fitness of the population, and then generating a random number between 0 and this total fitness. Each

chromosome occupies a portion of the wheel that is proportional to its fitness, and the random value generated corresponds to the spot at which the wheel stops spinning. This form of selection has a complexity of $O(n)$, where n is the size of the population.

Algorithm 4 Roulette Wheel Selection

```

1: procedure SELECTROULETTEWHEEL
2:   Get total fitness of population
3:   Generate random double rand between 0 and total fitness
4:   for  $i \leftarrow 0$  to population size, with  $rand > 0$  do
5:     Decrease rand by fitness of i-th chromosome
6:   end for
7:   return i-th chromosome

```

Rank Wheel selection is very similar to the previous method, but the chromosomes' ranks are used instead of their fitness values. Complexity in this case is also $O(n)$, however, the need to sort the chromosomes by fitness in order to rank them results in a more computationally expensive algorithm.

Algorithm 5 Rank Wheel Selection

```

1: procedure SELECTRANKWHEEL
2:   Calculate sum of ranks
3:   Generate random double rand between 0 and sum of ranks
4:   Sort population by fitness value
5:   for  $i \leftarrow 0$  to population size, with  $rand > 0$  do
6:     Decrease rand by rank (i) of i-th chromosome
7:   end for
8:   return i-th chromosome

```

Tournament selection is widely popular because of its simplicity and straightforward implementation. A *tournament size* must be defined beforehand, corresponding to the number of chromosomes competing in each tournament, and is generally set as 2 (in which case it will be called *binary tournament*). This selection algorithm is the most computationally efficient of the three described, with a complexity of $O(\text{tournament size})$.

It must be noted that a smaller tournament size will be more inexpensive and promote diversity within new generations. Increasing the tournament size will promote exploitation, but if it is increased too much the algorithm can become greedy and loose exploration qualities.

Algorithm 6 Tournament Selection

```

1: procedure SELECTTOURNAMENT
2:   for  $i \leftarrow 0$  to tournament size do
3:     Generate random integer  $r$ 
4:     Compare  $r$ -th chromosome to best chromosome selected so far
5:   end for
6:   return best chromosome in tournament

```

Crossover

For research purposes, two different kinds of crossover were implemented: One-Point crossover and Uniform crossover (section 3.7). The algorithms are fairly similar, and the pseudo-code for Uniform crossover can be found in Algorithm 7.

In One-Point crossover, a random integer is generated as the crossover point, and then every gene from this point and until the end of the chromosome is exchanged between the parents to produce the two offspring. If the random number produced can take the value 0, then this will produce children that are identical to their parents (if needed, this is easily avoidable by limiting random integers to values equal or larger than 1).

In the case of Uniform crossover, a rate of resemblance to each parent has to be defined. This can either be predefined as a constant, say 0.5 such that children are approximately half of each parent, or it can be randomized in every crossover. Then for every gene in the chromosome, a new random double is generated, such that if it is smaller than the rate of resemblance, said gene will be exchanged between the parents.

Algorithm 7 Uniform Crossover

```

1: procedure UNIFORMCROSSOVER
2:   Create copies of parent chromosomes
3:   Initialize  $\delta$   $\triangleright \delta$ : rate of resemblance to each parent
4:   for  $i \leftarrow 0$  to chromosome length do
5:     Generate random double rand
6:     if  $rand < \delta$  then
7:       Exchange  $i$ th gene in copies
8:     end if
9:   end for
10:  return copies (children)

```

Mutation

The implementation of the mutation stage described in section 3.8 is quite elementary. Subject to the mutation rate, the selected chromosome will be mutated by selecting at random one of its genes and changing its value. The value might be changed by replacing it with a different random value, or by flipping the gene (in chromosomes with binary alleles).

Algorithm 8 Mutation

```

1: procedure MUTATE
2:   Generate random integer  $i$  ( $0 \leftarrow \text{chromosome length}$ )
3:   Generate random double  $r$  ( $0 \leftarrow 1$ )
4:   Replace  $i$ -th gene with  $r$ 
5:   return mutated chromosome

```

4.3 Applying GAs to Kuhn Poker

4.3.1 Classic Kuhn Poker

The first implementation of Genetic Algorithms together with Kuhn Poker was done on it's classic variant. The full code can be found in Appendices F and G.

Strategy encoding

In the study of Kuhn Poker, the goal is to find optimal strategies to play against certain players. For that reason, the chromosomes in our Genetic Algorithm will have to hold the information that defines a Kuhn Poker strategy.

In the most simple version of Kuhn Poker, with two players and three cards, each player will have to make at most two decisions, which will each depend on the card that he has in his possession. This can be summarized into six parameters, where the first three parameters represent the probabilities of betting with card X in a check/bet situation (B_X), and the last three the probabilities of calling in a call/fold situation (C_X). As a result, each chromosome will contain six genes, with values corresponding to doubles between 0 and 1 (both included), as follows:

B_J	B_Q	B_K	C_J	C_Q	C_K
-------	-------	-------	-------	-------	-------

Two different sets of strategies were evolved, one for each of the two players in the game. This is because a strategy that is considered optimal for player 1, might not necessarily be a good strategy for Player 2.

The game tree in section 2.2 contains four binary decisions, two corresponding to each player. Figure 4.1 shows how each of these decision nodes relates to the genes in each player's chromosome.

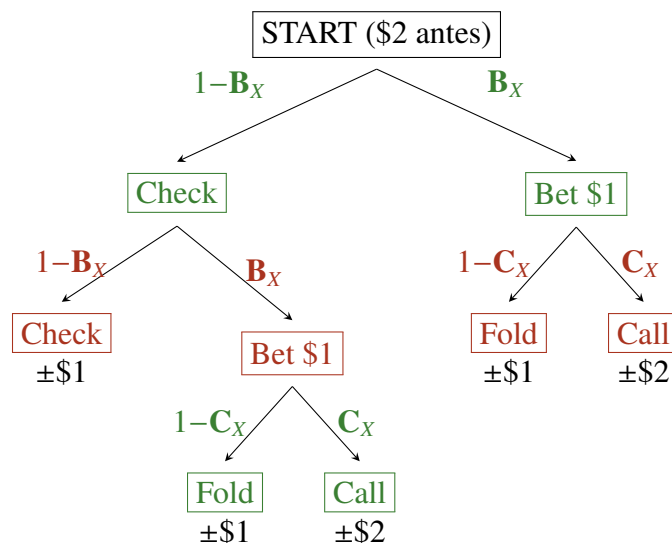


Fig. 4.1: Kuhn Poker betting tree with corresponding genes

Strategies, or chromosomes, can be evolved in two different ways. The first one consists in evolving strategies for a player against predefined opponent strategies. The second one is to evolve in parallel strategies for both players 1 and 2 through the use of co-evolution.

Genetic Algorithm

The structure of the Genetic Algorithm applied to Kuhn Poker will be exactly the same as that described in Algorithm 1 on page 21.

A set of Kuhn Poker games were used in the fitness evaluation stage of the algorithm. A chromosome's fitness was calculated by playing the corresponding strategy against a certain number of opponents.

In the case where a single player is evolved against a fixed opposing strategy, each chromosome in the population of size N was played N times against the fixed strategy.

On the other hand, in the case of co-evolution, each chromosome in the first population

(formed by Player 1 strategies) is played against every chromosome in the second population (Player 2 strategies). Both populations must have the same size.

Algorithm 9 Fitness Evaluation

```

1: procedure EVALFITNESS
2:   for  $i \leftarrow 0$  to population size do
3:     for  $j \leftarrow 0$  to population size do
4:       Play  $i$ -th chromosome in pop1 against  $j$ -th chromosome in pop2
5:     end for
6:   end for
7:   Set each chromosome's fitness to average of its wins

```

After all the games are played, the fitness assigned to each chromosome corresponds to the average of wins across all the games played. However, in the case of classic Kuhn Poker, this value will be between -2 and 2 . For simplicity when performing Roulette Wheel Selection, this value can be made positive by adding 2 to it, such that the fitness value will be between 0 and 4 . As a result, fitness values above 2 will be indicative of good strategies that are profitable.

Fair play

A special evaluation algorithm was implemented, such that for every two players that compete against each other, six games are played (one for each of the possible card combinations that can be dealt). This ensures that the results obtained are not affected by the luck of card dealing, however, it is computationally expensive since it multiplies the total computational cost by six.

Additional cards

Kuhn Poker as described by Harold Kuhn is played with only three cards, Jack, Queen and King. However, adding other cards from the Poker deck will help increase the complexity of the game and the decisions the players must make. For every card that gets added to the Kuhn Poker game, 3 extra genes will have to be added to the chromosome. A chromosome in a Kuhn Poker game with all thirteen possible ranks of a Poker deck will have 26 genes:

B ₂	B ₃	B ₄	B ₅	B ₆	B ₇	B ₈	B ₉	B ₁₀	B _J	B _Q	B _K	B _A
C ₂	C ₃	C ₄	C ₅	C ₆	C ₇	C ₈	C ₉	C ₁₀	C _J	C _Q	C _K	C _A

4.3.2 Adding breadth to the betting tree

The first step towards increasing the complexity of the betting tree for Kuhn Poker consisted in splitting the bets into three different options:



where betting \$10 is representative of an "all in" move. The new betting systems was implemented in two phases: first there was only a choice between betting \$1 or \$2, and after running some tests, the \$10 bet was incorporated. After the three betting options were implemented, the game tree looked like the one in Figure 4.2, where the breadth has been increased and as a result we are no longer working with a binary tree.

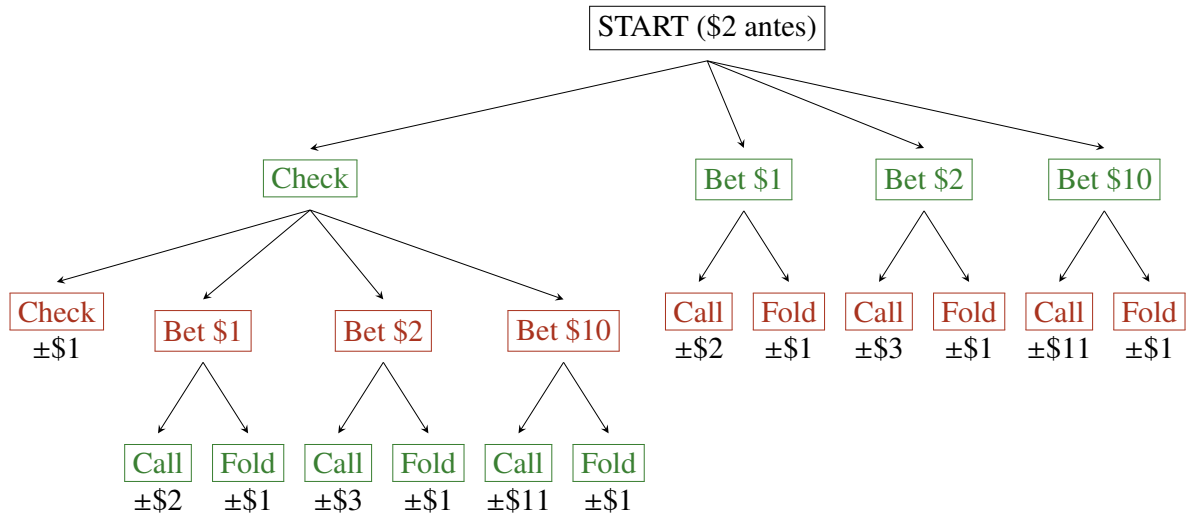


Fig. 4.2: Kuhn Poker betting tree with added breadth

These changes required an alteration in the Genetic Algorithm, by increasing the length of the chromosomes, and adding the extra betting choices to the Kuhn Poker game that was played between chromosomes. Since the chromosomes were now of length 18 and harder to interpret, they were reorganized into a matrix instead of a one-dimensional array, as shown in Figure 4.3.

Each column in the chromosome corresponds to the probabilities for each card the player can be dealt, and rows represent the different decision nodes in the tree.

The first row, B_X , is the probability that the player will make a bet given card X . The second row, $B_X^{>1}$, is the conditional probability that the player will place a bet of more than

B_J	B_Q	B_K
$B_J^{>1}$	$B_Q^{>1}$	$B_K^{>1}$
B_J^{10}	B_Q^{10}	B_K^{10}
C_J^1	C_Q^1	C_K^1
C_J^2	C_Q^2	C_K^2
C_J^{10}	C_Q^{10}	C_K^{10}

Fig. 4.3: Chromosome for added breadth Kuhn Poker.

\$1 given that he is placing a bet. The third row, B_X^{10} , is the conditional probability that the player will place a bet of \$10, given that he is placing a bet larger than \$1. The parameters C_X^n in the last three rows stand for the probability that the player will call a bet of \$ n .

4.3.3 Adding depth to the betting tree

The betting tree for Kuhn Poker can be deepened by giving the players the option to raise an extra \$1 after a bet has been placed by the opponent. This will add an extra layer to the tree, that will now look as the one in Figure 4.4.

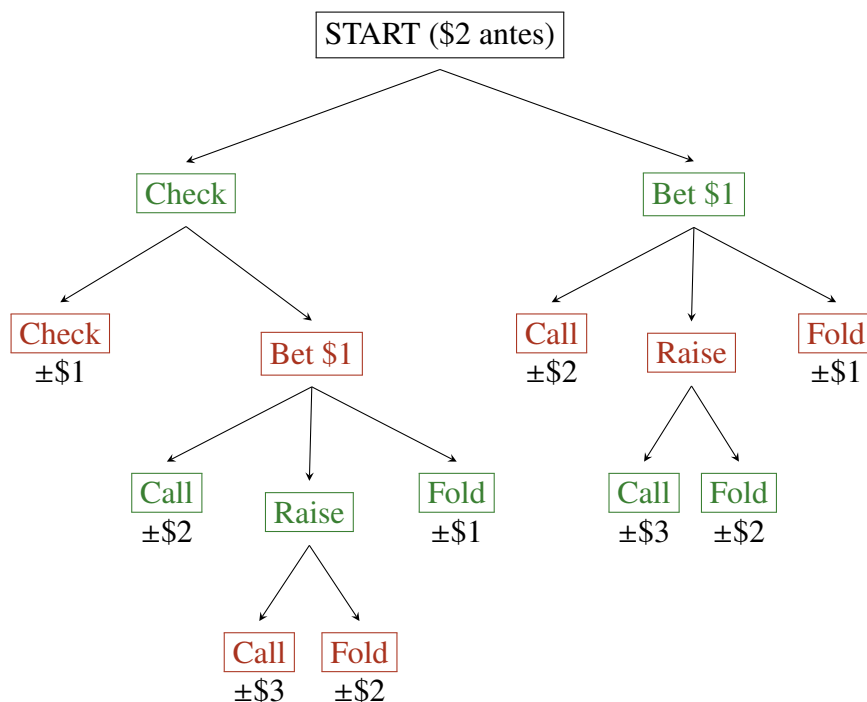


Fig. 4.4: Kuhn Poker Betting Tree with added depth

The chromosome for the extended bets variation of Kuhn Poker has 12 genes, arranged into a matrix as was done in the previous section:

B_J	B_Q	B_K
NF_J	NF_Q	NF_K
R_J	R_Q	R_K
C_J	C_Q	C_K

B_X stands for the probability of placing a bet, NF_X for the probability of not folding after the opponent places a bet, R_X for the conditional probability of raising after an opponent places a bet (given that the player decides to not fold) and, finally, C_X stands for the probability of calling a raise from the opponent.

4.3.4 Bringing it all together

Kuhn Poker can be complicated further by adding depth and breadth to the betting tree at the same time; that is, by allowing players to place bets with different values as well as raising a previous bet. For the sake of simplicity, players will only be able to raise by the amount that was bet by the opponent in the first place (double would be a more appropriate term, however, raise is used for consistency). The chromosomes for this variant of Kuhn Poker have 36 genes, that are arranged into a matrix as follows:

B_J	B_Q	B_K
$B_J^{>1}$	$B_Q^{>1}$	$B_K^{>1}$
B_J^{10}	B_Q^{10}	B_K^{10}
NF_J^1	NF_Q^1	NF_K^1
NF_J^2	NF_Q^2	NF_K^2
NF_J^{10}	NF_Q^{10}	NF_K^{10}
R_J^1	R_Q^1	R_K^1
R_J^2	R_Q^2	R_K^2
R_J^{10}	R_Q^{10}	R_K^{10}
C_J^1	C_Q^1	C_K^1
C_J^2	C_Q^2	C_K^2
C_J^{10}	C_Q^{10}	C_K^{10}

Chapter 5

Results and Evaluation

5.1 Classic Kuhn Poker

Getting the Genetic Algorithm to work for the first time proved to be one of the most challenging and time consuming tasks in the project.

Once the algorithm was implemented for the classic version of Kuhn Poker, some initial tests were run to make sure it was working as expected, and to test the different algorithms implemented for selection and crossover.

5.1.1 Evolution against opponent models

In order to evolve strategies for a single player, opponent strategies had to be established to train the populations against. Recall the structure of a chromosome for a strategy in classic Kuhn Poker, as defined in section 4.3.1:

B_J	B_Q	B_K	C_J	C_Q	C_K
-------	-------	-------	-------	-------	-------

B_X represents the probability of betting with card X , while C_J represents the probability of calling a bet. The betting tree corresponding to the probabilities in a strategy can be found on page 26, and might be of assistance in the upcoming sections.

Models were established to simulate real life players and train strategies against them. Four different models were defined and translated into the following four chromosomes, and can be used indistinctively for both Player 1 and Player 2:

0	0	0	0	0	1	0	0	1	0	0	1
(a) <i>Shy</i>						(b) <i>Safe</i>					
0	.5	1	0	.5	1	.5	.7	1	0	.7	1
(c) <i>Rational</i>						(d) <i>Bluffer</i>					

The relationship between these four models can be examined in the graph in Figure 5.2. The direction of the edges represents a winner \rightarrow loser relationship, where the weight is the average amount gained by the winner, after having them play 1000 games in each position. This figure highlights the complexity of finding such thing as an optimal strategy that will win against every different kind of player, as well as it proves that Poker is a non-transitive game; for example, the *Safe* player defeats the *Rational* on average, who defeats the *Bluffer*, however, the *Safe* does not defeat the *Bluffer*; and therefore the relationship is intransitive.

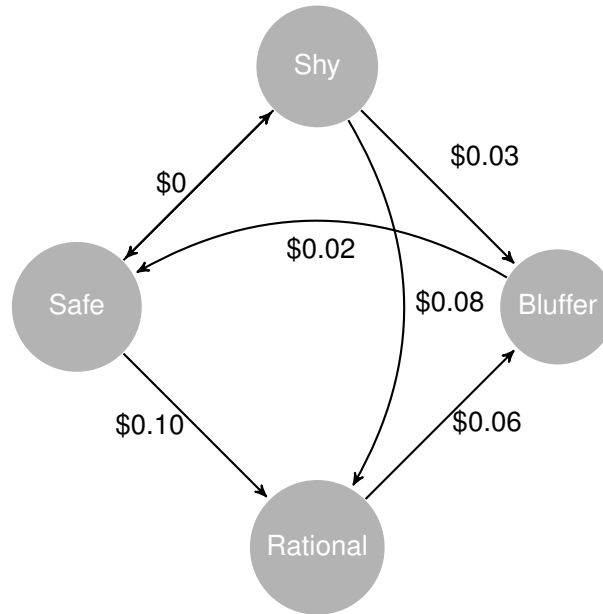


Fig. 5.2: Graph representing the relationships between player models.

Two other models were used for testing, a *Folder* and a *Caller*, but were left out of the report due to the fact that no real player, regardless of their level of expertise, would ever adopt such behaviours. Additional information about these models can be found in Appendix A.

For the preliminary work against the four static players, Tournament selection and Uniform crossover were used, the initial population was uniformly randomly initialized and the

algorithm's parameters were kept always the same, with the following values:

Generations: 100

Population size: 1010

Crossover rate: 70%

Mutation rate: 1%

Elitism: 0

Shy

The *Shy* player always checks when he is faced with a check/bet choice, and will only call the opponent's bets if he has a King.

The counter-strategy for a *Shy* player is the same regardless of what position he is playing in. At a first glance, the results obtained seem to be off: only the first two genes in the chromosomes converge towards a value, while the others fluctuate randomly, such that the fittest chromosomes in the later generations are of the form:

1	0	~	~	~	~
---	---	---	---	---	---

where ~ stands for a value that does not converge.

One might wonder why the strategy evolved always makes bets with a Jack but never with a Queen, while making random moves in the other 4 cases. However, this is exactly the strategy that will take advantage of the opponent's game-play and maximize the player's winnings in the game.

Suppose Player 1 is playing against a *Shy* Player 2. The gene corresponding to betting with a Jack, B_J , converges to 1 because if Player 1 has a Jack and checks, he will lose regardless of whether Player 2 bets or checks. On the other hand, betting will give him a win whenever Player 2 has a Queen (since he will fold on the bet), which, probabilistically speaking, will happen half of the time. Therefore the counter-strategy developed for Player 1 against a *Shy* player displays a **bluffing** behaviour.

In the case that Player 1 has a Queen, it is safer for him to check ($B_Q \rightarrow 0$) and let the showdown determine who wins because the *Shy* player will only call a bet if he has a King, losing Player 1 \$2.

The value of gene B_K (probability of betting with a King) fluctuates, since regardless of the action he takes, Player 1 will always win \$1. Finally, the three genes corresponding

to the call/fold decision don't converge because the corresponding node is never reached when playing against a *Shy* player, because he never places bets.

The exact same reasoning can be used to explain this strategy for a Player 2 against a *Shy* first player; since the latter never makes bets, the betting tree traversal is analogous to that of Player 1.

An individual who uses this strategy against a *Shy* player will win roughly \$0.34 on average, regardless of what position he plays in.

Safe

The *Safe* player always bets or calls when he has a King, and always checks or folds otherwise.

The strategies evolved against this player are slightly different depending on the position in which he is playing. Player 1 strategies to beat this player converge towards the values:

1	0	~	~	0	~
---	---	---	---	---	---

The strategy when having a Jack is again to **bluff** by always betting, and when having a Queen to always check, whereas with a King it is indifferent. However, in the event that Player 1 has a Queen and checks, if the *Safe* player bets, this implies he must have a King, so folding in this case will incur in a smaller loss (\$1 instead of \$2). This is why gene C_Q converges towards 0.

Genes C_J and C_K don't converge because they are never used against the *Safe* player, and regardless of its value, B_K will always result in a \$1 win.

Player 2 strategies against a *Shy* player who plays first are slightly different, converging towards the values:

1	~	~	0	0	~
---	---	---	---	---	---

In a game in which a *Safe* player has the first turn, he will bet whenever he has a King and check otherwise. For that reason, the values of C_J and C_Q converge to 0: folding will only generate a \$1 loss, while calling the bet would always lose \$2 for Player 2. The value of C_K doesn't converge because this case is never reached in a game against a *Shy* player.

For any of the cases B_J , B_Q and B_K to be reached, the *Shy* Player will have to check on his first turn; which he will only do if he has a Jack or a Queen. Let us examine the three

possible scenarios from Player 2's perspective:

- Player 2 has a Jack: the *Safe* player has a Queen and will fold on a bet, so Player 2 must **bluff** ($B_J \rightarrow 0$) to avoid losing at showdown.
- Player 2 has a Queen: the *Safe* player must have a Jack (he would have bet with a King), and Player 2 will gain \$1 regardless of whether he checks or bets, which is why B_Q doesn't converge.
- Player 2 has a King: if he bets, the *Safe* player will fold (+\$1) and if he checks, he will win at showdown (+\$1). Therefore the value of B_K is irrelevant, which is why it doesn't converge.

The fittest strategy obtained across all generations wins \$0.34 on average when playing first, while the fittest strategy when playing last wins \$0.47. There is an obvious advantage when playing last against a *Safe* player.

Rational

The problem of Kuhn Poker becomes more complex with the introduction of the *Rational* player, because, unlike the previous models, his strategy is not purely deterministic. We recall that this player will always bet or call with a King and always check or fold with a Jack. In the case that he has a Queen, he will bet or call half of the times, and check or fold the other half.

Again, the strategies evolved to beat this player differed depending on the position in which they were playing. Player 1 strategies show no signs of deceptive behaviour, converging towards the chromosome:

0	0	1	0	0	1
---	---	---	---	---	---

Note that optimal strategies to defeat a *Rational* player converge towards the *Safe* player's strategy. Genes B_J and C_J both converge to 0, such that when Player 1 has a Jack, he will always check and, in the event that Player 2 makes a bet, he will fold; this ensures that he will lose \$1 instead of \$2. He could try to bluff; however, the bluff will be called by Player 2 75% of the time, resulting in a \$2 loss.

The values for B_Q and C_Q converge towards 0 as well. At first sight, one might think that these values would converge towards 0.5. However, from the betting tree we can calculate the average earnings for the different deterministic strategies:

- $B_Q = 1 \Rightarrow$ Average earnings: $-\$0.5$
- $B_Q = 0$ and $C_Q = 1 \Rightarrow$ Average earnings: $-\$0.5$
- $B_Q = 0$ and $C_Q = 0 \Rightarrow$ Average earnings: $\$0$

and conclude that those strategies in which B_Q and C_Q are closer to 0 will minimize Player 1's losses.

Genes B_K and C_K converge to 1: if Player 1 has a King, he will always make a bet hoping that the *Rational* player will call it and win him $\$2$. If he calls a bet from Player 2, Player 1 will also win $\$2$.

When the *Rational* player gets to play first, the counter-strategy for the second player converges towards the values:

1	\sim	1	0	0	1
---	--------	---	---	---	---

The call or fold genes, C_J , C_Q and C_K , take the same values as they did when the individuals played in the reversed positions. However, the strategy for checking or betting changes, since it follows a previous check or bet decision from the *Rational* player, in which he will always bet if he has a King and half of the times when he has a Queen. Therefore the check or bet node for Player 2 will only be reached when the *Rational* player has a Jack or half of the times when he has a Queen. There are three scenarios:

- Player 2 has a Jack: Player 1 has a Queen, and will fold on a bet half of the time. Player 2 loses $\$1$ if he checks, but only loses $\$0.5$ on average if he bets. The best strategy in this case is to **bluff** ($B_J \rightarrow 1$).
- Player 2 has a Queen: in this case, Player 1 has a Jack and will fold every time, so it is indifferent whether Player 2 bets or checks since he will win $\$1$ regardless.
- Player 2 has a King: if he checks, he will always win $\$1$ at showdown, however, if he bets, Player 1 will call $1/6$ of the time earning him $\$2$. So betting with a King will earn him more on average ($B_K \rightarrow 1$).

The fittest strategy obtained across all generations wins \$0.23 on average when playing first, and \$0.32 when playing second. Once again, playing last gives a big edge to a player who is up against the *Rational* model, since he has access to additional information and can base his decisions on the first player's moves.

Bluffer

Finally, the *Bluffer* player always bets or calls whenever he has a King, if he has a Queen he will bet or call with a 70% chance, and if he has a Jack, he will bet with a 50% chance and he will never call a bet.

The strategies evolved by the Genetic Algorithm to compete against the *Bluffer* when playing first converged towards the values

0	0	0	0	1	1
---	---	---	---	---	---

Let us take a closer look at why Player 1 decides to always check when he has a King, since this might seem counter-intuitive. For the different three strategies he can adopt, the average earnings will be:

- $B_K = 1 \Rightarrow$ Average wins: +\$1.35
- $B_K = 0$ and $C_Q = 1 \Rightarrow$ Average wins: -\$0.2
- $B_K = 0$ and $C_Q = 1 \Rightarrow$ Average wins: +\$1.6

The optimal strategy makes use of **sandbagging** in order to trick the *Bluffer*. It can also be worked out that the strategies for when the player has a Jack or a Queen result in average losses (-\$1 with a Jack and -\$0.25 with a Queen). Fortunately, these losses are compensated by the large average of wins when Player 1 is dealt a King.

For an individual who plays last against a *Bluffer*, the probabilities of checking or betting vary, with convergence towards the values:

↓	~	1	0	1	1
---	---	---	---	---	---

where ↓ means that there is no clear convergence but the corresponding gene takes small values. In this case, sandbagging does not give Player 2 an advantage. Let us examine the three possibilities:

- Player 2 has a Jack: Player 1 has a Queen, because if he had a King he would have made a bet. Player 2 will always lose \$1 if he checks, and an average of \$1.1 if he makes a bet. However, the value of B_J doesn't converge to 0, although it tends to take small values. A possible explanation for the value not converging is that the *Bluffer* strategy is non-deterministic and the outcomes depend on the luck of the game, so there will always be some fluctuation in the results.
- Player 2 has a Queen: Player 1 must have a Jack, with which he will fold any bet, such that Player 2 will win \$1 regardless of whether he chooses to check or bet. For this reason, the value of B_Q doesn't converge.
- Player 2 has a King: if he checks, he will always win \$1. However, if he bets, the bet will be called by the *Rational* player 70% of the times that he has a Queen, yielding average winnings of \$1.16 for Player 2. Betting maximizes his earnings, and so $B_Q \rightarrow 1$.

When playing against a *Bluffer*, there is no clear advantage of one position over the other: when playing first, the strategies evolved made \$0.33 on average, while the winnings were of \$0.35 when playing last. Since the strategies are non-deterministic and there is a luck factor, this slight difference could be attributed to them and doesn't prove that the first player is at a disadvantage.

5.1.2 Co-evolution

The alternative way of evolving player strategies consists in making use of co-evolution, a technique that was described in section 3.10.

Co-evolution is achieved by evolving two populations at the same time, one for each player. Both populations start out fairly weak, but with each new generation they learn to play against many new kinds of players. The average fitness of each population will not be as good as it was when evolving strategies against a fixed opponent; it might even decrease with time, since the opposing strategy is also becoming fitter. Nonetheless, the fittest individuals will be stronger in the sense that they are prepared to play against several more advanced players with stronger strategies.

Figure 5.3 shows how the average fitness of each population for every new generation that is created. It can be observed that, after roughly 800 generations, the average fitness

gets stabilized, corresponding to a Nash Equilibrium. It must be noted that the average fitness for Player 1 is negative, $-\$0.055$, which corresponds to an average loss of $1/18$ of the bet amount, exactly what is predicted by theoretical calculations [9]. The plots for the populations are symmetric across the x-axis because Kuhn Poker is a zero-sum game (the payoffs will always sum to zero).

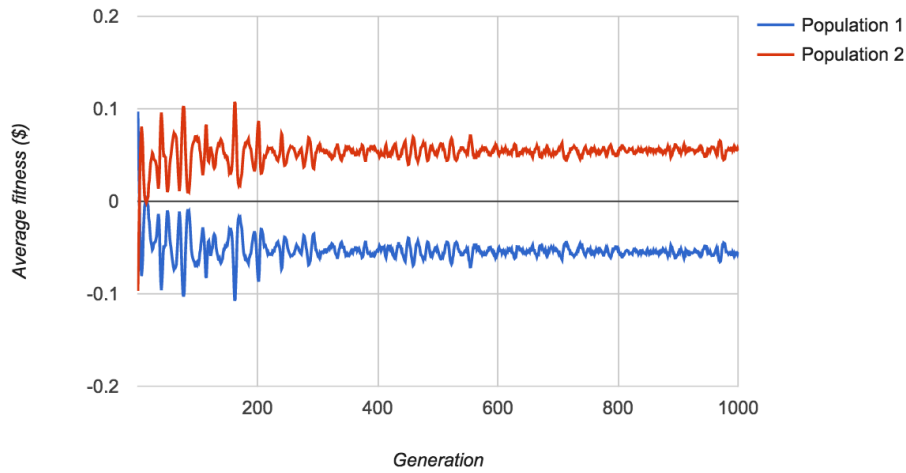


Fig. 5.3: Average fitness of the populations throughout the co-evolutionary process.

In co-evolution, the fittest strategy across all generations will not necessarily be the fittest strategy overall, since the opponents are evolving as well, and a later strategy might earn less but against a more experienced player. For this reason, we will look at the fittest strategies in the later generations, in which an equilibrium has been reached. The fittest chromosomes in the last generation of both populations have the parameters:

.1	0	.43	.01	.56	1
----	---	-----	-----	-----	---

Player 1

.44	0	1	.0027	.24	1
-----	---	---	-------	-----	---

Player 2

To test the fittest Player 1 strategy evolved with this method, it can be played against the models that were used in the previous section to get an idea of how it performs. These are the average wins of the Player 1 chromosome above, when played 1000 times against each model:

- Shy: $+\$0.02$ on average, up to $+\$0.14$.
- Safe: $-\$0.07$ on average, up to $+\$0.07$.

- Rational: $-\$0.02$ on average, up to $+\$0.14$.
- Bluffer: $+\$0.03$ on average, up to $+\$0.20$.

Player 1 wins against two of the four predefined models, and is unsuccessful against the *Safe* and *Rational* players. On the contrary, Player 2 wins against the four modeled players, with the following earnings:

- Shy: $+\$0.07$ on average, up to $+\$0.22$.
- Safe: $+\$0.10$ on average, up to $+\$0.25$.
- Rational: $+\$0.10$ on average, up to $+\$0.28$.
- Bluffer: $+\$0.08$ on average, up to $+\$0.26$.

This, once again, confirms the fact that the second player has an advantage over the first one, since the choices of the first player are also a factor that can be taken into account when making his own decisions.

5.1.3 Mixed approach

The reason why the Player 1 strategies developed with co-evolution lose against certain kinds of players is that the second population doesn't develop strategies corresponding to the behaviours of these players.

This led to the idea of a mixed approach between co-evolution and static opponent models. The approach consists in evolving the players by evaluating their fitness values against another evolving population and static models at the same time.

From this method, when training Player 1 strategies against both evolving Player 2's and *Rational* players, one of the fittest later strategies was:

.00046	0	.063	0	.49	1
--------	---	------	---	-----	---

for which the earnings against each Player 2 model were:

- Shy: $+\$0.015$ on average, up to $+\$0.15$.
- Safe: $-\$0.02$ on average, up to $+\$0.12$.

- Rational: +\$0.03 on average, up to +\$0.2.
- Bluffer: +\$0.03 on average, up to +\$0.23.

This resulted in a strategy that can win against more different kinds of players, and that performs better on average. The chart in Figure 5.5 shows the average fitness of Populations 1 and 2 across their evolution throughout 1000 generations. At equilibrium, Population 1 is at an average loss of \$0.015, which is a considerable improvement from the standard co-evolutionary method described in the previous section. With the mixed approach, Player 1 is expected to lose on average 1/70 of the betting amount instead of 1/18. However, this method doubles the complexity of the algorithm, so it has a higher computational cost.

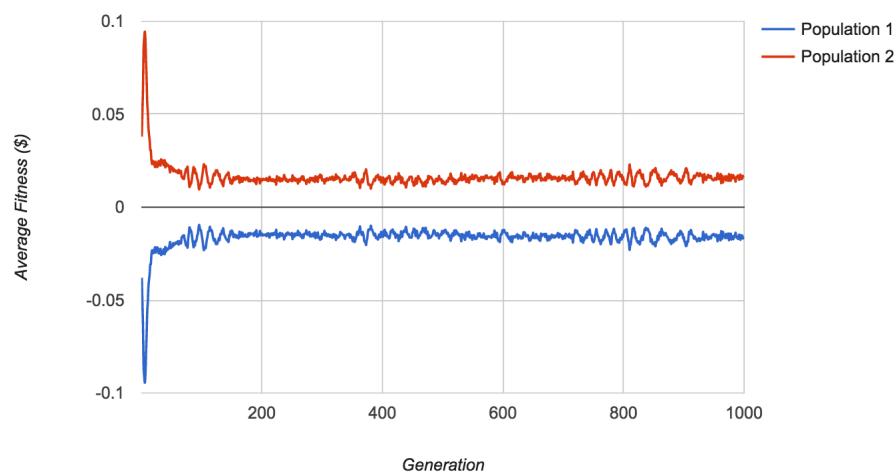


Fig. 5.5: Average fitness of the populations throughout the evolutionary process with the mixed approach.

5.2 Performance of different techniques

5.2.1 Selection

The performance of the three kinds of selection was tested on the four different models across 500 generations, and the average and best fitness results plotted for each of the methods and models. The full graphs for the four models showing both average and best fitness can be found in Appendix B.

Tournament and Rank selections perform equally well, whereas the Roulette Wheel method needs more generations to converge, and after 500 generations does not reach values

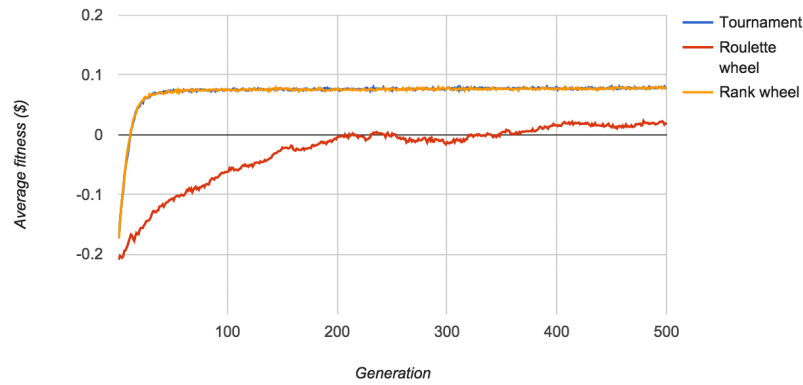


Fig. 5.6: Average fitness of the population throughout 500 generations for each selection method, playing against the *Rational* player.

as good as the other two reach in just a hundred. This happened across all four models, and can be seen Figure 5.6 for the *Rational* player.

Tournament selection was chosen over Rank selection because it is less computationally expensive, since it eliminates the need to sort the population by fitness.

5.2.2 Randomization

As mentioned in section 4.2, the population initialization was implemented using two different randomization methods: a purely random one (yielding a normal distribution), and a uniformly random one (yielding a uniform distribution).

These two methods performed quite similarly when tested against the four player models, showing a comparably paced convergence and achieving similar best fitness results. In some cases, the uniform randomization method showed signs of a slightly faster convergence (with the *Bluffer* player, Figure 5.7) or arriving at a higher average fitness (with the *Safe* player, Figure 5.8).

This slight difference in average fitness is not necessarily enough to prove that the uniformly random method performs better in general, however, this method does ensure a uniform distribution of alleles in the initial population, and is therefore more reliable than a purely random approach.

5.2.3 Crossover

Strategies were trained against each of the four player models, first using Uniform crossover and then One-Point crossover. Most of these graphs revealed a very balanced performance

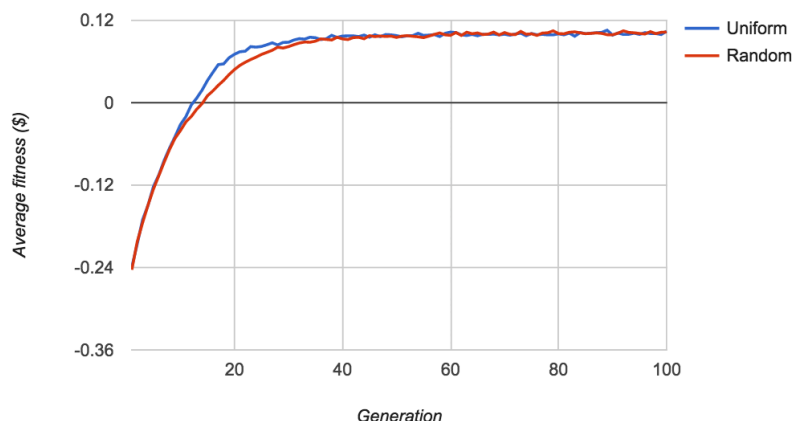


Fig. 5.7: Average fitness against the *Bluffer* for the each randomization method.

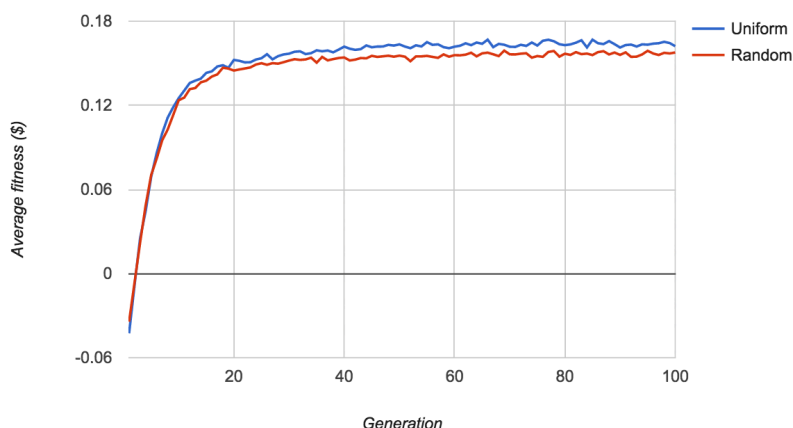


Fig. 5.8: Average fitness against the *Safe* player for the each randomization method.

between these two methods, and can be found in Appendix C. However, in the case of the *Rational* player, a slightly better performance of the Uniform crossover method was observed (Figure 5.9).

5.3 Complex forms of Kuhn Poker

5.3.1 Kuhn Poker with added breadth

After several tests were ran on the Genetic Algorithm for classic Kuhn Poker and its correct performance was confirmed, the added breadth form of the game, with multiple betting options, was implemented as described in Section 4.3.2 (page 28). Tests were ran against

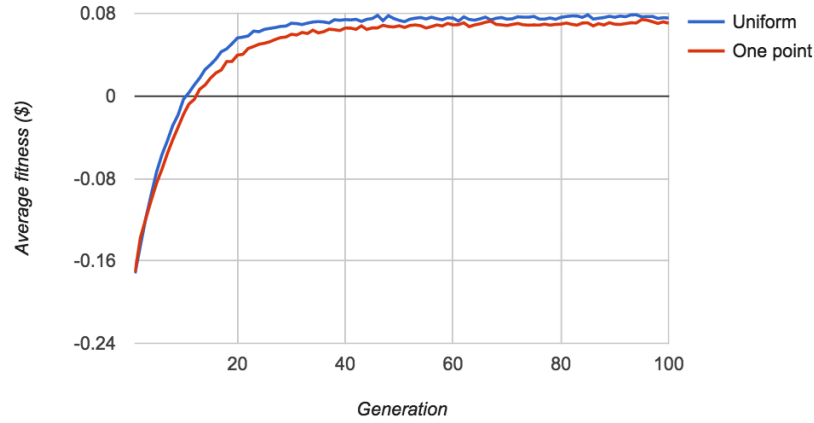


Fig. 5.9: Average fitness for each crossover method, playing against the Rational player.

fixed models analogous to those used for classic Kuhn Poker.

We remind that the structure of chromosomes in this form of Kuhn Poker with extra betting choices is the following:

B_J	B_Q	B_K
$B_J^{>1}$	$B_Q^{>1}$	$B_K^{>1}$
B_J^{10}	B_Q^{10}	B_K^{10}
C_J^1	C_Q^1	C_K^1
C_J^2	C_Q^2	C_K^2
C_J^{10}	C_Q^{10}	C_K^{10}

and the corresponding betting tree can be found in section 4.3.2, and will be a useful reference in the subsequent sections. Recall that $B_X^{>1}$ and B_X^{10} are conditional probabilities, such that $B_X^{>1}$ is the probability of betting more than \$1 given that the player bets, and B_X^{10} is the probability of betting \$10 given that the player bets more than \$1.

Shy

The *Shy* player will always check, and always fold except for when he has a King, in which case he will call any bet from the opponent regardless of its value (Figure 5.10a). This behaviour alters the strategies evolved such that the parameters converge towards the values in Figure 5.10, where \downarrow stands for parameters that tend to take very small values but don't entirely converge to 0, and \sim for values that don't converge.

0	0	0
0	0	0
0	0	0
0	0	1
0	0	1
0	0	1

(a) *Shy player*

1	0	~
0	↓	~
0	↓	~
~	~	~
~	~	~
~	~	~

(b) *Player 1 counter-strategy*

1	0	↓
0	↓	↑
0	0	~
~	~	~
↑	~	~
↓	~	~

(c) *Player 2 counter-strategy*

Fig. 5.10: Chromosome for the *Shy* player and counter-strategies evolved against it.

The strategy for an individual facing a *Shy* player consists in **bluffing** when having a Jack, such that the *Shy* player will fold with a Queen, and call the bet if he has a King, in which case the loss will be equal to what it would have been had he checked in the first place. A small bet of \$1 will suffice to make the opponent fold with a Queen.

In the event that a Queen is dealt, the player will check and let the showdown determine the winner, since the *Shy* player will only call a bet with a King, which would result in a greater loss.

The fittest chromosomes across the one hundred generations that were bred had average winnings of \$0.33, regardless of the position of the players, so there is no advantage for either of the two players.

Safe

The *Safe* player (Figure 5.11a) only bets or calls a bet when he is certain that he will win, that is, when he has a King. He will always bet \$10 in this case, since he knows that he cannot lose the hand. He will always check or fold if he has something other than a King.

The counter-strategies evolved to compete against the *Safe* player converged towards the chromosomes in Figure 5.11, whose behaviour is similar to the strategies against the *Shy* player.

When playing first, C_Q^{10} converges towards 0. This is because if the *Safe* player bets \$10, this is a clear sign that he has a King and therefore folding is the obvious response to this. The values of C_Q^1 and C_Q^2 are random because the *Safe* player never bets less than \$10.

When playing second, C_J^{10} and C_Q^2 converges to 0, such that player 2 will fold if the *Safe* player makes a bet.

The strongest strategy for Player 1 across all generations wins \$0.33 on average when

0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1

(a) *Safe player*

1	0	~
0	↓	~
0	↓	~
~	~	~
↓	~	~
↓	0	~

(b) *Player 1 counter-strategy*

1	~	↑
↓	~	~
~	~	~
~	↓	~
~	0	↑
0	~	↓

(c) *Player 2 counter-strategy***Fig. 5.11:** Chromosome for the *Safe* player and counter-strategies evolved against it.

playing against the *Safe* player. The fittest Player 2 wins \$0.46 on average.

Rational

The *Rational* player always checks or folds when he has a Jack, and always bets \$10 and calls bets when he has a King. If he has a Queen, half the times he will bet \$1 and the other half he will bet \$2, but never more than that. With a Queen he will call all \$1 bets, half of the \$2 bets and none of the \$10 ones. This behaviour can be translated into the chromosome in Figure 5.12a.

0	1	1
0	.5	1
0	0	1
0	1	1
0	.5	1
0	0	1

(a) *Rational player*

0	0	1
0	0	1
~	~	1
0	~	~
0	~	~
0	0	~

(b) *Player 1 counter-strategy*

~	~	~
~	~	~
~	~	~
0	↓	1
0	0	1
~	~	~

(c) *Player 2 counter-strategy***Fig. 5.12:** Chromosome for the *Rational* player and counter-strategies evolved against it.

The strategies that were evolved in response to this player showed safe behaviours in both cases, only calling or making bets when having a King, as can be seen in Figure 5.12.

The strongest strategy for Player 1 across all generations wins \$1.24 on average when playing against the *Rational* player, while the fittest Player 2 chromosome earned only \$0.47. In this case, there is a clear advantage when playing first against a *Rational* player, which can be due to the fact that the first player's check/bet decision affects the performance of the second player.

Bluffer

The behaviour of the *Bluffer* is displayed in the chromosome in Figure 5.13a. When he has a Jack, the *Bluffer* will bet \$1 50% of the time, \$2 35% and \$10 15% of the time; but he will never call a bet since this would guarantee a loss. If he has a Queen, he bets \$2 half of the time and \$10 the other half, and calls all \$1 bets, half of the \$2 bets and none of the all-ins. Finally, if he has a King, he will call every bet and bet \$10 100% of the time.

1	1	1	0	0	0	~	~	~
.5	1	1	0	0	~	~	~	~
.3	.5	1	~	~	~	~	~	~
0	1	1	~	1	1	~	1	1
0	.5	1	0	1	1	0	0	1
0	0	1	0	0	1	~	~	~

(a) *Bluffer player* (b) *Player 1 counter-strategy* (c) *Player 2 counter-strategy*

Fig. 5.13: Chromosome for the *Bluffer* player and counter-strategies evolved against it.

An individual who plays first against a *Bluffer* will make use of **sandbagging**, to give the opponent a feeling of self-confidence and trick him into making bets. The opponent learns that he should always see bets with a King and never with a Jack, while he should see \$1 and \$2 bets with a Queen, but never an all-in.

When playing after a *Bluffer*, check/bet values are irrelevant since he will be the one making those decisions. However, Player 2 will cautiously call bets with a King or a Queen.

The strongest strategy for Player 1 across all generations wins \$2.06 on average when playing against the *Bluffer*, while the strongest Player 2 wins \$1.89. The advantage when playing first is due to the fact that the individual can be the one to make the betting decisions, whereas when playing second the *Bluffer* gets to make them.

5.3.2 Kuhn Poker with added depth

Remember the structure of chromosomes after adding a betting round to the game, as described in section 4.3.3:

Recall that NF_X is the probability that the player will not fold with card X , while R_X is the conditional probability that he will raise, given that he did not fold.

B_J	B_Q	B_K
NF_J	NF_Q	NF_K
R_J	R_Q	R_K
C_J	C_Q	C_K

Shy

The *Shy* player never makes bets of his own, but he will call any bet made by the opponent if he has a King.

0	0	0
0	0	1
0	0	0
0	0	1

(a) *Shy player*

1	0	~
~	~	~
~	~	~
~	~	~

(b) *Player 1 counter-strategy*

1	0	~
~	~	~
~	~	~
~	~	~

(c) *Player 2 counter-strategy*

Fig. 5.14: Chromosome for the *Shy* player and counter-strategies evolved against it.

The counter-strategy for the *Shy* player is the same regardless of what position the individual is competing from, and consists in **bluffing** with a \$1 bet when having a Jack, and checking with a Queen. This strategy wins \$0.32 on average in both positions.

Safe

When a *Safe* player has a King, he will raise bets if he has the option and call them otherwise, and will always place a bet. If he has a Jack or Queen he will always either check or fold.

0	0	1
0	0	1
0	0	1
0	0	1

(a) *Safe player*

1	0	~
↓	0	~
~	~	~
0	↓	~

(b) *Player 1 counter-strategy*

1	~	~
0	0	~
↓	↓	~
~	~	~

(c) *Player 2 counter-strategy*

Fig. 5.15: Chromosome for the *Safe* player and counter-strategies evolved against it.

As we saw in the previous variants of Kuhn Poker, the strategies against the *Safe* player involve **bluffing** with a Jack by placing a \$1 bet. The best strategies across all generations win \$0.34 on average for Player 1 and \$0.46 for Player 2.

Rational

The *Rational* player will always call, raise or bet if he has a King and will always fold if he has a Jack. In the event that he is dealt a Queen, he will call or raise half of the bets, and will make his own bets half of the time:

0	.5	1
0	.5	1
0	.5	1
0	.5	1

(a) *Rational player*

0	0	1
0	0	1
~	~	↑
↓	~	1

(b) *Player 1 counter-strategy*

1	~	1
0	0	1
↓	↓	1
0	~	1

(c) *Player 2 counter-strategy*

Fig. 5.16: Chromosome for the *Rational* player and counter-strategies evolved against it.

The strategy adopted by the player when playing first will be, like in previous cases, that of a *Safe* player. However, if he is playing second against the *Rational* player, he will **bluff** with a Jack ($B_J \rightarrow 1$). To reach this decision, the *Rational* player must have checked, which means that he has a Queen. Two things can happen:

- Player 2 checks and loses \$1 in the showdown.
- Player 2 bets and loses between \$0.5 and \$0.75 on average, depending on the subsequent betting choices.

The best strategy across all generations for a first player wins \$0.28 on average, and \$0.40 for the second player. This advantage is due to the fact that when playing second, it is possible to infer what card the *Rational* player has from the choices he makes.

Bluffer

The *Bluffer* will always call, raise or bet with a King, and take more risks with Jacks and Queens than all the other previous players. In the case of a Jack, he will either bet or raise, or fold, but never call a bet because this guarantees a loss.

The strategies against the *Bluffer* are based on safe game-play: betting and calling with a King, and folding and checking with a Jack. However, Player 2 strategies are more prone to risk since conclusions can be drawn from the choices of the *Bluffer*. This leads to an advantage when playing last, such that the best strategy across all generations for Player 1 wins an average of \$0.46 and the best strategy for Player 2 wins \$0.63.

<table border="1"> <tr><td>.5</td><td>.7</td><td>1</td></tr> <tr><td>.5</td><td>.7</td><td>1</td></tr> <tr><td>1</td><td>.5</td><td>1</td></tr> <tr><td>0</td><td>.7</td><td>1</td></tr> </table>	.5	.7	1	.5	.7	1	1	.5	1	0	.7	1	<table border="1"> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>~</td><td>0</td><td>~</td></tr> <tr><td>↓</td><td>↑</td><td>1</td></tr> </table>	0	0	1	0	1	1	~	0	~	↓	↑	1	<table border="1"> <tr><td>↓</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>~</td><td>0</td><td>1</td></tr> <tr><td>↓</td><td>1</td><td>1</td></tr> </table>	↓	1	1	0	1	1	~	0	1	↓	1	1
.5	.7	1																																				
.5	.7	1																																				
1	.5	1																																				
0	.7	1																																				
0	0	1																																				
0	1	1																																				
~	0	~																																				
↓	↑	1																																				
↓	1	1																																				
0	1	1																																				
~	0	1																																				
↓	1	1																																				
(a) <i>Bluffer player</i>	(b) <i>Player 1 counter-strategy</i>	(c) <i>Player 2 counter-strategy</i>																																				

Fig. 5.17: Chromosome for the *Bluffer* player and counter-strategies evolved against it.

5.3.3 Kuhn Poker with added breadth and depth

The Kuhn Poker variant with added breadth and depth (extra betting round and betting options) involved the use of 36-gene chromosomes, which resulted in unstable solutions, with less signs of convergence and therefore very complex to interpret.

Strategies against the *Rational* player involved **sandbagging** behaviours when playing in the first position, and counter-strategies for the *Bluffer* displayed **bluffing** regardless of the playing position. Full results for the four models, including the ones displaying these deceptive behaviours, can be found in Appendix D.

5.3.4 Co-evolution

Co-evolution offers similar results in all complex versions of Kuhn Poker. It is quite unstable, never reaching an equilibrium between the two players as it does in the classic version.

There is, in the three complex forms, an advantage for the second player, having in general higher average fitness values, but it is not predominant. This can be observed in the three co-evolution graphs in Appendix E, in all of which the average fitness of Population 1 sometimes prevails over that of Population 2.

Regarding the strategies evolved in co-evolution, these always corresponded to very safe strategies, as it happened in classic Kuhn Poker. Again, this shows that, when the kind of game-play of an opponent is unknown, it might be less risky to adopt a safe or shy attitude instead of using deceptive strategies.

Chapter 6

Conclusion

6.1 Conclusions

Genetic Algorithms can be very effective in the search for intelligent strategies, as long as the game in question doesn't involve an overly large betting tree. In this case, the algorithms can become extremely complex and take hours of computation. The variant with additional betting options and rounds proved to be very challenging when it came to interpreting the results, and did not achieve results as accurate as the classic version due to a highly increased computational complexity.

In the study of Kuhn Poker, Tournament and Rank-Wheel both proved to be powerful methods of selection, successfully converging in under 100 generations to winning strategies. Tournament selection is a less complex algorithm, since it does not require the population to be sorted, as does Rank selection. Roulette-Wheel selection showed very slow convergence, not achieving average earnings as fit in over 500 generations.

The accuracy of the co-evolution method can be backed up by the fact that the opposing populations reached an equilibrium in classic Kuhn Poker, in which the population representing the first player lost on average $1/18$ of the betting value, as was proved in [9].

Deceptive game-play was very efficient when playing against a particular player, since his moves can be anticipated and hypothesis can be made on what cards he has from his betting choices. However, the strategies evolved from co-evolution did not show signs of bluffing or sandbagging, which suggests that when playing against unfamiliar opponents it can be optimal to adopt a safe strategy.

6.2 Further research

There are several directions from which future research could be approached in this project. The most obvious one is to explore more complex forms of Kuhn Poker, for example by adding extra bets or rounds, as well as additional.

Regarding the techniques used in the Genetic Algorithm, it could be of interest to implement and test additional ones. A method of crossover called Selective crossover was proved to be more efficient than more common methods in [8], and could potentially improve the performance of the algorithm.

The last research option could result in the problem taking a different path, and becoming much more complex, by tackling Texas Hold'em. This variant of Poker has too many factors to take into account to be approached exclusively with GAs, but an alternative method using evolutionary Neural Networks is described in [7].

Appendices

Appendix A

Additional models for classic Kuhn Poker testing

Folder

The *Folder* always checks in a Check/Bet situation and always folds on Call/Fold, as portrayed by the chromosome:

0	0	0	0	0	0
---	---	---	---	---	---

At a first glance, the results obtained seem to be off: only the two first genes in the chromosomes converge, while the others fluctuate randomly, such that the fittest chromosomes in the later generations are of the form:

1	1	~	~	~	~
---	---	---	---	---	---

One might wonder why would Player 1 always bet with a Jack and a Queen, and make random moves in the other 4 cases. However, this is exactly the strategy that will maximize his wins in the game. Since Player 2 will always fold or check, only two branches of the tree will ever be traversed:

- Player 1 bets and Player 2 folds ($\pm \$1$)
- Player 1 and Player 2 check ($\pm \$1$)

If Player 1 has a King, he will always win the game regardless of his decision. If he has a Jack or a Queen and he bets, Player 2 will fold, avoiding a showdown and granting the pot directly to Player 1.

The strategies evolved are therefore optimal, and have an average earning of \$1, which is the maximum pot that can ever be earned by Player 1 when playing against the Folder.

Caller

The *Caller* player always bets/calls regardless of his cards, such that every gene in the chromosome takes a value of 1:

1	1	1	1	1	1
---	---	---	---	---	---

and Player 1 strategies evolved to beat this player converge towards the chromosome:

0	1	1	0	1	1
---	---	---	---	---	---

The fittest strategy obtained across all generations wins \$0.59 on average and has the following parameters:

.01	.98	.97	0	.87	1
-----	-----	-----	---	-----	---

Appendix B

Additional tests for selection methods

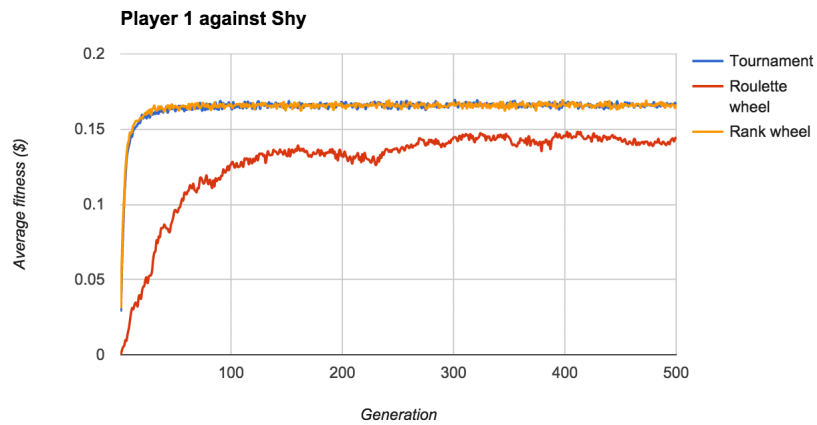


Fig. B.1: Average fitness for each selection method, playing against the Shy player.

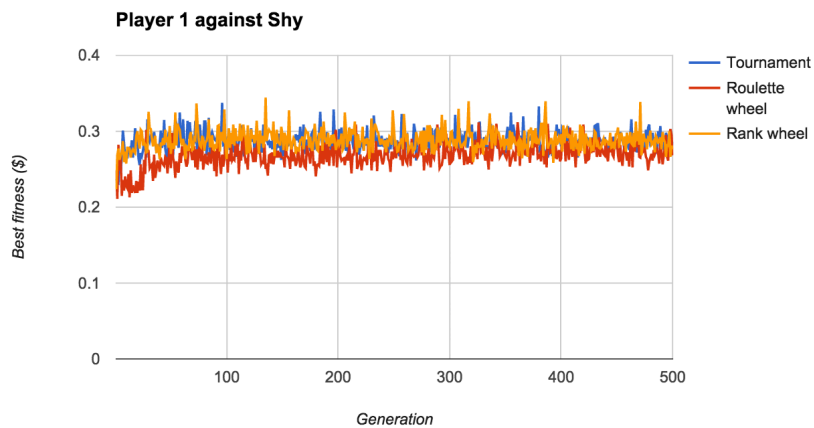


Fig. B.2: Best fitness for each selection method, playing against the Shy player.

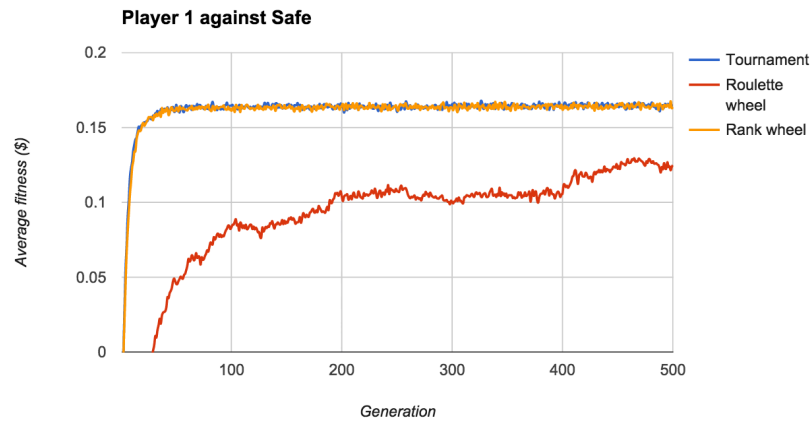


Fig. B.3: Average fitness for each selection method, playing against the Safe player.

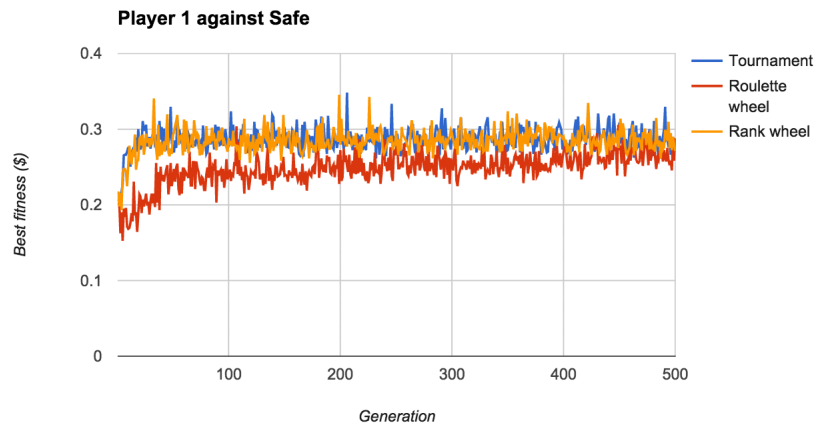


Fig. B.4: Best fitness for each selection method, playing against the Safe player.

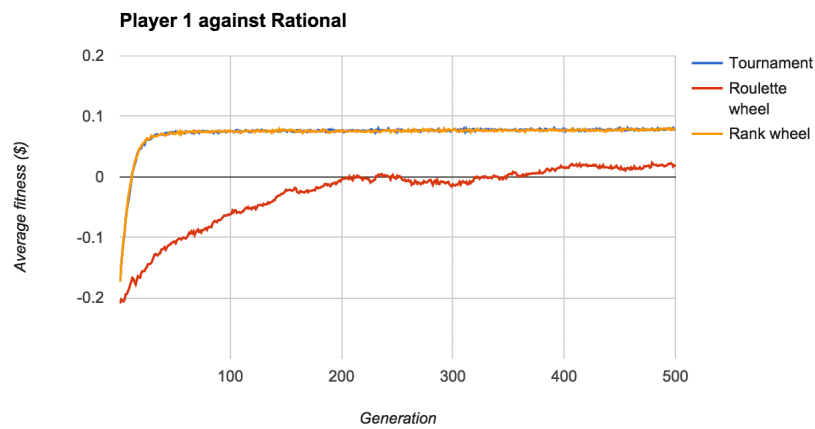


Fig. B.5: Average fitness for each selection method, playing against the Rational player.

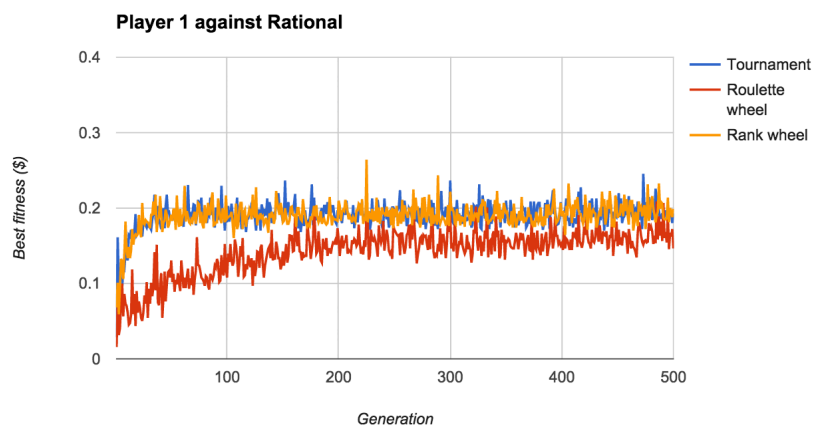


Fig. B.6: Best fitness for each selection method, playing against the Rational player.

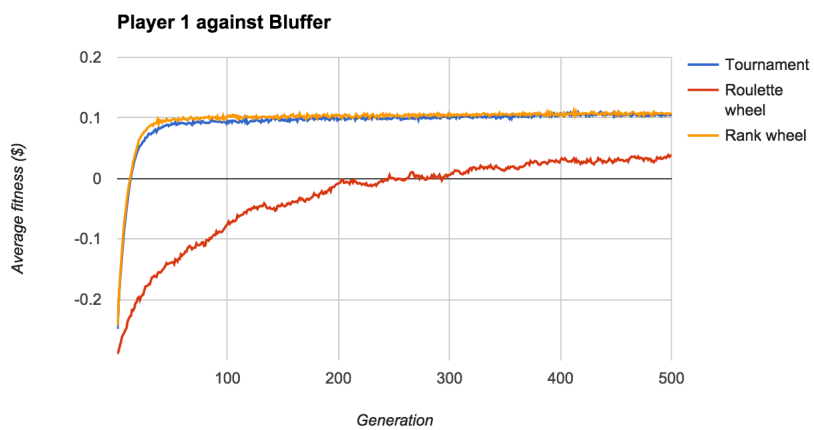


Fig. B.7: Average fitness for each selection method, playing against the Bluffer player.

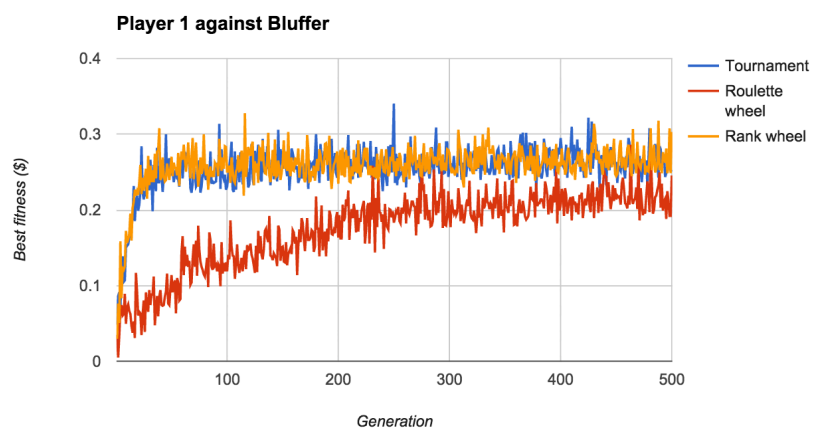


Fig. B.8: Best fitness for each selection method, playing against the Bluffer player.

Appendix C

Additional tests for crossover methods

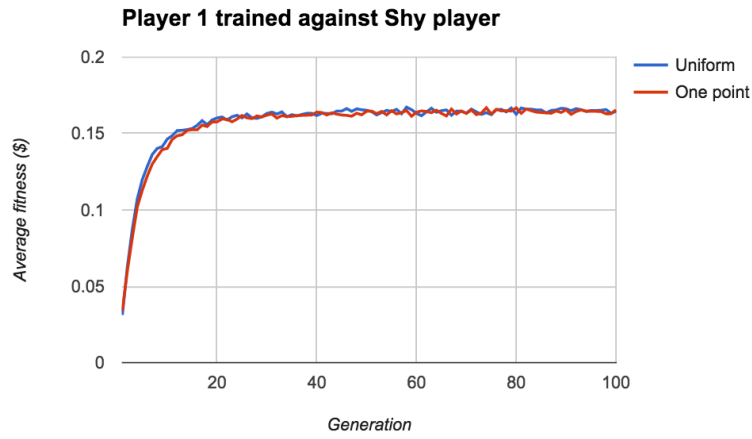


Fig. C.1: Average fitness for each crossover method, playing against the Shy player.

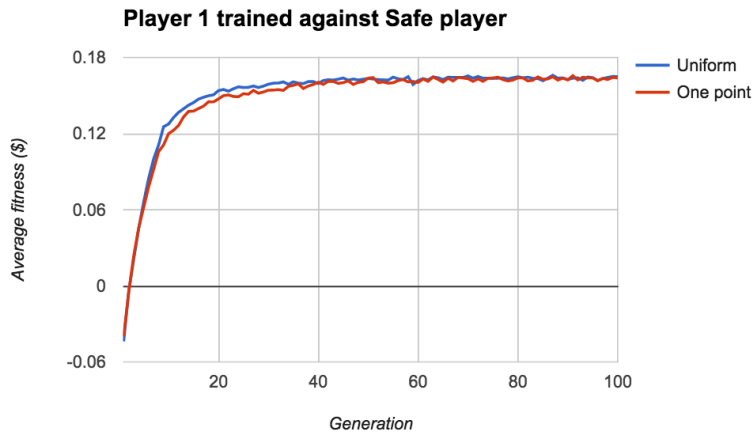


Fig. C.2: Average fitness for each crossover method, playing against the Safe player.

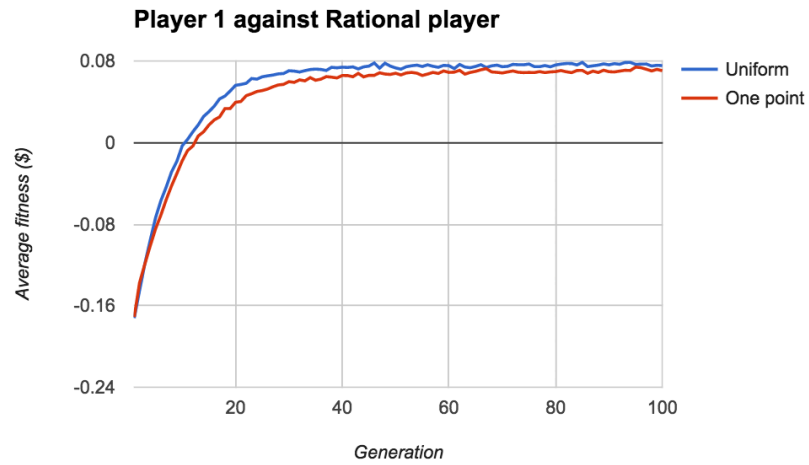


Fig. C.3: Average fitness for each crossover method, playing against the Rational player.

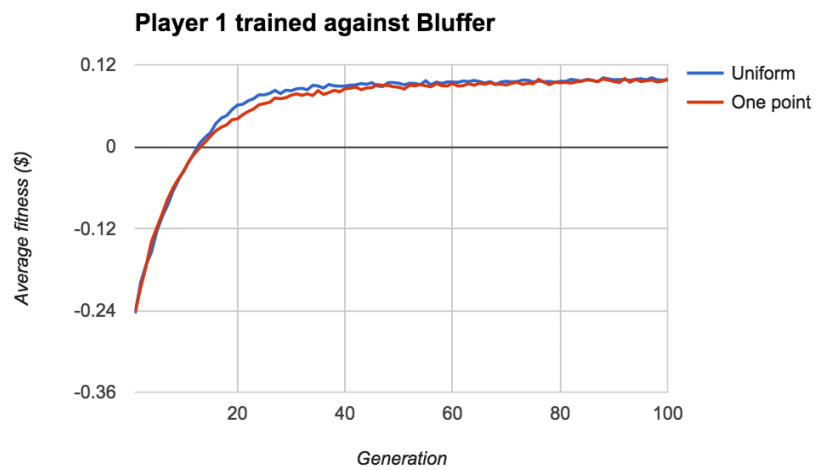


Fig. C.4: Average fitness for each crossover method, playing against the Bluffer.

Appendix D

Results for added breadth and depth variant

Shy

Average winnings regardless of the position of the *Shy* player are approximately \$0.33, with strategies converging towards:

0	0	0
0	0	0
0	0	0
0	0	1
0	0	1
0	0	1
0	0	0
0	0	0
0	0	0
0	0	1
0	0	1
0	0	1

(a) *Shy player*

1	0	~
0	↓	~
0	↓	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~

(b) *Player 1 counter-strategy*

1	0	~
0	↓	~
0	↓	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~

(c) *Player 2 counter-strategy*

Fig. D.1: Chromosome for the *Shy* player and counter-strategies evolved against it.

Safe

When playing against a *Safe* player first, the average winnings were of \$0.33, while they were \$0.44 when playing last, with strategies converging to the chromosomes:

0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1

(a) *Safe player*

1	0	↓
0	↓	.6
0	0	~
~	↓	~
~	~	~
0	0	~
~	~	↓
↑	~	↓
↓	~	↑
0	~	~
~	↓	~
↓	~	~

(b) *Player 1 counter-strategy*

1	↓	~
↑	~	↑
~	~	~
↓	~	~
.5	.5	.5
0	0	~
~	0	~
~	~	~
~	~	~
~	0	~
~	~	~
↓	~	~

(c) *Player 2 counter-strategy***Fig. D.2:** Chromosome for the Safe player and counter-strategies evolved against it.**Rational**

Average earnings when playing first against the *Rational* player are \$0.57, and \$0.59 when playing second. Solutions converge to the following values, where some **sandbagging** can be observed for a first player:

0	1	1
0	.5	1
0	0	1
0	1	1
0	.5	1
0	0	1
0	1	1
0	.5	1
0	0	1
0	1	1
0	.5	1
0	0	1

(a) *Rational player*

0	0	0
0	↓	0
0	0	0
0	~	1
0	~	1
0	0	~
~	~	1
1	~	1
~	0	~
0	~	1
0	~	1
~	~	~

(b) *Player 1 counter-strategy*

~	~	~
~	~	~
~	~	~
0	~	1
0	~	1
0	0	~
~	~	~
1	~	1
~	~	~
~	~	~
~	~	~
~	~	~

(c) *Player 2 counter-strategy***Fig. D.3:** Chromosome for the Rational player and counter-strategies evolved against it.

Bluffer

The strategies respond to a *Bluffer* by **bluffing** with a small bet of \$1 when having a Jack. Average winnings are \$2.82 when playing first and \$3.82 when playing second.

1	1	1
.5	1	1
.5	.5	1
.5	1	1
.5	.7	1
.5	.5	1
1	1	1
1	.7	1
1	.5	1
0	1	1
0	.5	1
0	0	1

(a) *Bluffer player*

1	0	1
0	0	1
↓	↓	1
~	~	~
~	~	~
↓	0	~
~	~	~
~	~	~
~	~	~
~	~	~
0	~	~
~	~	~
~	~	1

(b) *Player 1 counter-strategy*

1	1	1
0	1	1
↓	1	1
~	~	~
~	~	~
0	0	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	~
~	~	1

(c) *Player 2 counter-strategy*

Fig. D.4: Chromosome for the *Bluffer* player and counter-strategies evolved against it.

Appendix E

Results for co-evolution in complex variants

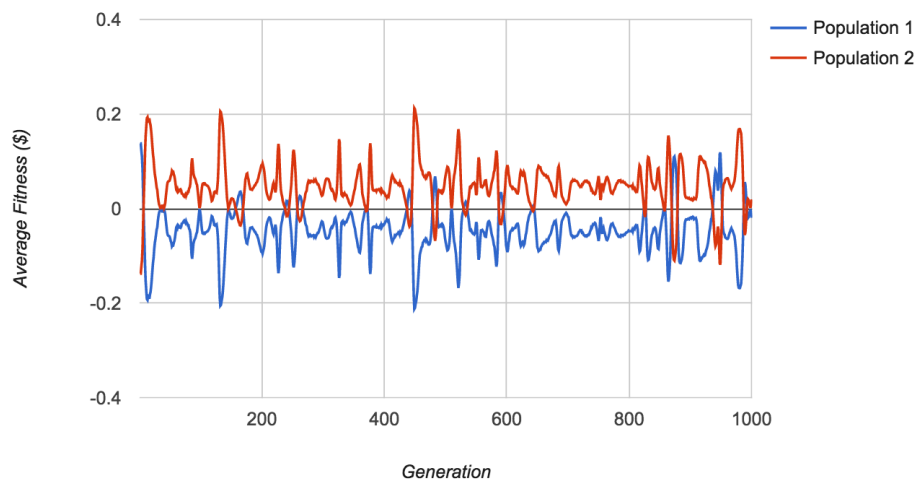


Fig. E.1: Average fitness of the populations throughout the co-evolutionary process in added breadth Kuhn Poker.

0	0	~
~	0	↓
0	~	1
0	↑	1
0	~	1
0	0	1

(a) Player 1

↑	0	1
0	0	0
0	~	~
0	~	1
0	0	1
~	↓	~

(b) Player 2

Fig. E.2: Chromosome's convergence values obtained from co-evolution in added breadth Kuhn Poker.

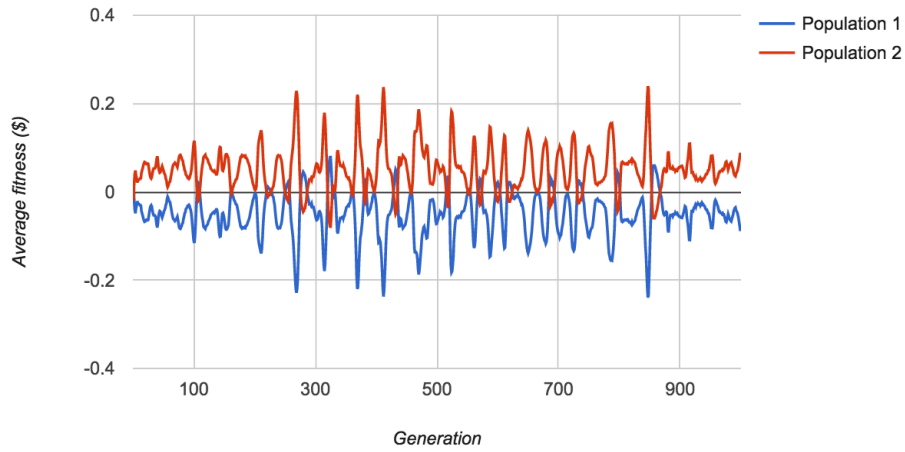


Fig. E.3: Average fitness of the populations throughout the co-evolutionary process in added depth Kuhn Poker.

~	0	↑
0	~	1
~	0	1
0	↓	1

(a) Player 1

~	0	1
0	~	1
↓	0	1
0	~	1

(b) Player 2

Fig. E.4: Chromosome's convergence values obtained from co-evolution in added depth Kuhn Poker.

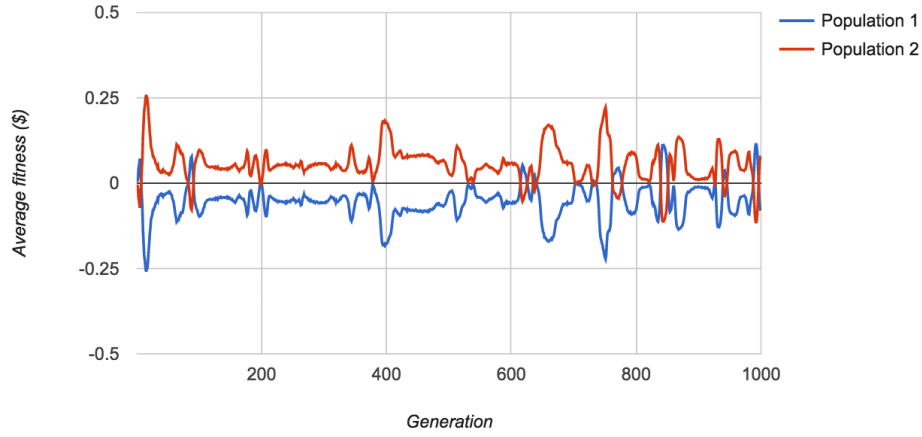


Fig. E.5: Average fitness of the populations throughout the co-evolutionary process in added breadth and depth Kuhn Poker.

↓	0	↑
↓	0	↑
0	0	0
0	1	0
0	0	1
0	0	1
~	0	~
~	0	1
↓	~	↑
0	↓	1
0	~	1
0	~	1

(a) Player 1

↓	0	↑
↓	0	↑
0	0	0
0	~	0
0	0	1
0	0	1
~	~	~
~	0	1
↓	~	↑
0	1	1
0	~	1
↓	~	1

(b) Player 2

Fig. E.6: Chromosome's convergence values obtained from co-evolution in added breadth and depth Kuhn Poker.

Appendix F

Code for Genetic Algorithm

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> GeneticAlgo.java
 *
 * <dt> Description:
 * <dd> This file consists on a genetic algorithm that can be used
 * to solve optimization problems. In this cases, it attempts to find
 * strategies for a Kuhn Poker game.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */

import geneticAlgo.*;

public class GeneticAlgo {
    public final static int GENERATIONS = 100;
    public final static int SIZE = 1010;
    public final static double CROSS_RATE = 0.7;
    public final static double MUTATE_RATE = 0.01;

    public final static int ELITE_COUNT = 0;

    // CHANGE VALUES DEPENDING ON VARIANT OF KUHN POKER TO BE USED
    // Set chromo length for classic Kuhn Poker
    public final static int CHROMO_LENGTH = 6;

    // Set chromo length for added breadth Kuhn Poker
    //public final static int CHROMO_LENGTH = 18;

    // Set chromo length for added depth Kuhn Poker
    //public final static int CHROMO_LENGTH = 12;

    // Set chromo length for added breadth and depth Kuhn Poker
    //public final static int CHROMO_LENGTH = 36;
```

```

/*
 * Evaluate fitness of two populations through co-evolution.
 */
public static void evalFitness(Population p1, Population p2) {

    // Set all fitness values to zero in both populations
    for (int i = 0; i < SIZE; i++) {
        p1.getChromo(i).resetFitness();
        p2.getChromo(i).resetFitness();
    }

    // Play every chromosome in every population against each other,
    // adding wins of gameplays to their fitnesses
    double[][] wins = new double[2][SIZE];
    int[] gameResult = new int[2];

    // Loop through chromosomes of population 1
    for (int i = 0; i < SIZE; i++) {
        // Loop through chromosomes of population 2
        for (int j = 0; j < SIZE; j++) {
            switch(CHROMO_LENGTH) {
                case 6:
                    gameResult = ClassicKuhnPoker.chromoGamePlay(p1.getChromo(i)
, p2.getChromo(j));
                    break;
                case 18:
                    gameResult = BreadthKuhnPoker.chromoGamePlay(p1.getChromo(i)
, p2.getChromo(j));
                    break;
                case 12:
                    gameResult = DepthKuhnPoker.chromoGamePlay(p1.getChromo(i),
p2.getChromo(j));
                    break;
                case 36:
                    gameResult = ComplexKuhnPoker.chromoGamePlay(p1.getChromo(i)
, p2.getChromo(j));
                    break;
            }

            wins[0][i] += gameResult[0];
            wins[1][j] += gameResult[1];

            // Mixed approach (with rational player)
            // gameResult = KuhnPoker.chromoGamePlay(p1.getChromo(i),
rational);
            // wins[0][i] += gameResult[0];
            // wins[1][j] += gameResult[1];
            // wins[0][i] /= 2;
            // wins[1][j] /= 2;
        }
    }

    // Divide into number of games to get average win
    for (int i = 0; i < SIZE; i++) {
        p1.getChromo(i).setFitness(Math.pow(wins[0][i]/(SIZE), 1));
    }
}

```

```

        p2.getChromo(i).setFitness(Math.pow(wins[1][i]/(SIZE), 1));
    }
}

/*
 * Evaluate fitness of population against strategy as a first player.
 */
public static void evalFitness(Population p1, Chromosome strategy) {

    // Set all fitness values to zero in both populations
    for (int i = 0; i < SIZE; i++) {
        p1.getChromo(i).resetFitness();
    }

    // Play every chromosome in every population against each other,
    // adding wins of gameplays to their fitnesses
    double[] wins = new double[SIZE];
    int[] gameResult = new int[2];

    // Loop through chromosomes of population 1
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            switch(CHROMO_LENGTH) {
                case 6:
                    gameResult = ClassicKuhnPoker.chromoGamePlay(p1.getChromo(i)
, strategy);
                    break;
                case 18:
                    gameResult = BreadthKuhnPoker.chromoGamePlay(p1.getChromo(i)
, strategy);
                    break;
                case 12:
                    gameResult = DepthKuhnPoker.chromoGamePlay(p1.getChromo(i),
strategy);
                    break;
                case 36:
                    gameResult = ComplexKuhnPoker.chromoGamePlay(p1.getChromo(i)
, strategy);
                    break;
            }
            wins[i] += gameResult[0];
        }
    }

    // Divide into number of games to get average win
    for (int i = 0; i < SIZE; i++) {
        p1.getChromo(i).setFitness(Math.pow(wins[i]/(SIZE), 1));
    }
}

/*
 * Evaluate fitness of population against strategy as a second player.
 */
public static void evalFitness(Chromosome strategy, Population p2) {

```

```

// Set all fitness values to zero in both populations
for (int i = 0; i < SIZE; i++) {
    p2.getChromo(i).resetFitness();
}

// Play every chromosome in every population against each other,
// adding wins of gameplays to their fitnesses
double[] wins = new double[SIZE];
int[] gameResult = new int[2];

// Play each chromosome against the given strategy
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        switch(CHROMO_LENGTH) {
            case 6:
                gameResult = ClassicKuhnPoker.chromoGamePlay(strategy, p2.
getChromo(i));
                break;
            case 18:
                gameResult = BreadthKuhnPoker.chromoGamePlay(strategy, p2.
getChromo(i));
                break;
            case 12:
                gameResult = DepthKuhnPoker.chromoGamePlay(strategy, p2.
getChromo(i));
                break;
            case 36:
                gameResult = ComplexKuhnPoker.chromoGamePlay(strategy, p2.
getChromo(i));
                break;
        }
        wins[i] += gameResult[0];
    }
}

// Divide into number of games to get average win
for (int i = 0; i < SIZE; i++) {
    p2.getChromo(i).setFitness(Math.pow(wins[i]/(SIZE), 1));
}

}

/*
 * Breed and return new generation from an old one.
 */
public static Population breedGeneration(Population oldPop) {
    // Create new population
    Population newPop = new Population(SIZE, CHROMO_LENGTH, CROSS_RATE,
MUTATE_RATE);

    // Generation counter
    int count = 0;

    // Sort population when there is elitism bigger than zero
    if (ELITE_COUNT > 0) {

```

```

    oldPop.sort();
}

while ( count < SIZE ) {
    Chromosome[] chromo = new Chromosome[2];

    // Take best chromosomes and reinsert them unaltered.
    if (count < ELITE_COUNT) {
        chromo[0] = new Chromosome(oldPop.getChromo(count));
        chromo[1] = new Chromosome(oldPop.getChromo(count+1));
        // For the rest, take chromosomes using Roulette wheel selection
and
        // then crossover/mutate them.
    } else {
        // Select two chromosomes with tournament selection
        chromo[0] = new Chromosome(oldPop.selectTournament(2));
        chromo[1] = new Chromosome(oldPop.selectTournament(2));

        // Select two chromosomes with roulette wheel
        //chromo[0] = new Chromosome(oldPop.selectRouletteWheel());
        //chromo[1] = new Chromosome(oldPop.selectRouletteWheel());

        // Select two chromosomes with rank wheel
        //chromo[0] = new Chromosome(oldPop.selectRankWheel());
        //chromo[1] = new Chromosome(oldPop.selectRankWheel());

        // Crossover chromosomes with Uniform crossover
        chromo = Population.uniformCrossover(chromo);

        // Crossover chromosomes with One-Point crossover
        //chromo = Population.onePointCrossover(chromo);

        // Mutate chromosomes
        chromo[0] = new Chromosome(Population.mutate(chromo[0]));
        chromo[1] = new Chromosome(Population.mutate(chromo[1]));
    }

    // Add resulting chromosomes to new population, making sure that
they are
    // not clones of other members of population
    if (!newPop.contains(chromo[0], count)) {
        newPop.setChromo(count, chromo[0]);
        count += 1;
    }
    if (count < SIZE && !newPop.contains(chromo[1], count)) {
        newPop.setChromo(count, chromo[1]);
        count += 1;
    }
}

return newPop;
}

```

```
/*
 * Main method that runs GA
 */
public static void main(String[] args) {
    // Rational player strategy
    Chromosome rational = new Chromosome(CHROMO_LENGTH);
    rational.setGene(0, 0);
    rational.setGene(1, 0.5);
    rational.setGene(2, 1);
    rational.setGene(3, 0);
    rational.setGene(4, 0.5);
    rational.setGene(5, 1);

    // Print the parameters used
    System.out.println("Generations: " + GENERATIONS);
    System.out.println("Population size: " + SIZE);
    System.out.println("Crossover rate: " + CROSS_RATE);
    System.out.println("Mutation rate: " + MUTATE_RATE);
    System.out.println("Elitism: " + ELITE_COUNT);

    // String to hold output for csv file
    String csvFile = "";

    // Array to hold fittest chromosomes in populations.
    Chromosome[] fittestChromo = new Chromosome[2];
    fittestChromo[0] = new Chromosome(CHROMO_LENGTH);
    fittestChromo[1] = new Chromosome(CHROMO_LENGTH);

    // Create two initial populations (p1 and p2).
    Population[] oldPopulation = new Population[2];
    oldPopulation[0] = new Population(SIZE, CHROMO_LENGTH, CROSS_RATE,
    MUTATE_RATE);
    oldPopulation[1] = new Population(SIZE, CHROMO_LENGTH, CROSS_RATE,
    MUTATE_RATE);

    // Randomize uniformly initial population
    oldPopulation[0].randomizeUniformly();
    oldPopulation[1].randomizeUniformly();

    // Randomize uniformly initial population
    //oldPopulation[0].randomize();
    //oldPopulation[1].randomize();

    // Evaluate fitness rates of chromosomes in initial population
    evalFitness(oldPopulation[0], oldPopulation[1]);
    //evalFitness(oldPopulation[0], rational);
    //evalFitness(rational, oldPopulation[1]);

    // Create array to hold two new populations
    Population[] newPopulation = new Population[2];
    newPopulation[0] = new Population(SIZE, CHROMO_LENGTH, CROSS_RATE,
    MUTATE_RATE);
    newPopulation[1] = new Population(SIZE, CHROMO_LENGTH, CROSS_RATE,
    MUTATE_RATE);
```

```

for (int i = 0; i < GENERATIONS; i++) {

    // Breed new generations from previous ones.
    newPopulation[0] = new Population(breedGeneration(oldPopulation[0]))
    ;
    newPopulation[1] = new Population(breedGeneration(oldPopulation[1]))
    ;

    // Evaluate the fitness of the populations through co-evolution
    evalFitness(newPopulation[0], newPopulation[1]);

    // Evaluate the fitness population 1 against rational player
    //evalFitness(newPopulation[0], rational);

    // Evaluate the fitness of population 2 against rational player
    //evalFitness(rational, newPopulation[1]);

    // Save outcome to string to generate a csv in the end
    csvFile += (i+1) + ", " + (newPopulation[0].avgFitness()) + ", " + (
    newPopulation[0].getFittest().getFitness()) + "\n";

    // Keep track of fittest chromosome in Population 1
    if (fittestChromo[0].getFitness() < newPopulation[0].getFittest().
    getFitness()){
        fittestChromo[0] = new Chromosome(newPopulation[0].getFittest());
    }

    // Keep track of fittest chromosome in Population 2
    if (fittestChromo[1].getFitness() < newPopulation[1].getFittest().
    getFitness()){
        fittestChromo[1] = new Chromosome(newPopulation[1].getFittest());
    }

    // New population becomes old population to be used for next
    generation breeding
    oldPopulation[0] = new Population(newPopulation[0]);
    oldPopulation[1] = new Population(newPopulation[1]);
    }

    // Generate csv file with results
    GenerateCsv.generateCsvFile("test.csv", csvFile);
}
}

```


Appendix G

Code for classic Kuhn Poker

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> ClassicKuhnPoker.java
 *
 * <dt> Description:
 * <dd> This program consists of a classic Kuhn Poker game for two
 *       players.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */

import cardGame.*;
import geneticAlgo.*;
import java.util.Random;

public class ClassicKuhnPoker {

    // Variable to hold the pot in game.
    public static int pot;

    /**
     * Returns the winning player by comparing their hands.
     */
    public static Player checkCards(Player p1, Player p2) {
        int r1 = p1.popCard().getRank().ordinal();
        int r2 = p2.popCard().getRank().ordinal();
        if ( r1 > r2 ) {
            return p1;
        } else {
            return p2;
        }
    }
}
```

```

/*
 * Play a game given two chromosomes, and return the amount won by
 * each player.
 */
public static int[] chromoGamePlay(Chromosome ch1, Chromosome ch2) {

    // Players initialized
    Player p1 = new Player();
    Player p2 = new Player();

    // Ante (each player puts 1 in pot)
    pot = 2;
    p1.bet(1);
    p2.bet(1);

    // Deck is shuffled and one card is dealt to each player
    Deck deck = new Deck();
    deck.shuffle();

    // Deal cards (will return exception if deck is empty)
    try {
        p1.addCard(deck.deal());
        p2.addCard(deck.deal());
    } catch (Exception e) {
        System.out.println(e);
    }

    // Play hand and give pot to winning player.
    Player winner = chromoHandPlay(p1, p2, ch1, ch2);

    // Print out result and give pot to winner.
    winner.win(pot);
    pot = 0;

    // Store wins in an array and return it.
    int[] wins = new int[2];
    wins[0] = p1.getStack();
    wins[1] = p2.getStack();

    return wins;
}

/*
 * Play 6 games between two chromosomes, one for each possible
 * combination of cards, and return the average winnings of all of
 * them.
 */
public static int[] chromoFairPlay(Chromosome ch1, Chromosome ch2) {

    // Players initialized
    Player p1 = new Player();
    Player p2 = new Player();

    // Deck is shuffled and one card is dealt to each player
    Deck deck = new Deck();

```

```

    deck.shuffle();

    Card card1 = new Card(Card.Rank.JACK, Card.Suit.SPADES);
    Card card2 = new Card(Card.Rank.QUEEN, Card.Suit.SPADES);
    Card card3 = new Card(Card.Rank.QUEEN, Card.Suit.SPADES);

    // Store wins in an array.
    int[] wins1 = gamePlay(ch1, card1, ch2, card2);
    int[] wins2 = gamePlay(ch1, card1, ch2, card3);
    int[] wins3 = gamePlay(ch1, card2, ch2, card1);
    int[] wins4 = gamePlay(ch1, card2, ch2, card3);
    int[] wins5 = gamePlay(ch1, card3, ch2, card1);
    int[] wins6 = gamePlay(ch1, card3, ch2, card2);

    int[] wins = new int[2];
    wins[0] = (wins1[0] + wins2[0] + wins3[0] + wins4[0] + wins5[0]
+ wins6[0])/6;
    wins[1] = (wins1[1] + wins2[1] + wins3[1] + wins4[1] + wins5[1]
+ wins6[1])/6;

    return wins;
}

/*
 * Return wins from a game between two strategies with the given cards.
 */
public static int[] gamePlay(Chromosome ch1, Card c1, Chromosome ch2
, Card c2) {
    // Players initialized
    Player p1 = new Player();
    Player p2 = new Player();

    // Deal cards passed in function
    p1.addCard(c1);
    p2.addCard(c2);

    // Ante (each player puts 1 in pot)
    pot = 2;
    p1.bet(1);
    p2.bet(1);

    // Play hand and give pot to winning player.
    Player winner = chromoHandPlay(p1, p2, ch1, ch2);

    // Print out result and give pot to winner.
    winner.win(pot);
    pot = 0;

    // Store wins in an array.
    int[] wins = new int[2];
    wins[0] = p1.getStack();
    wins[1] = p2.getStack();

    return wins;
}

```

```

/*
 * Return whether a player checks or bets from the chromosome passed
 * to it.
 */
public static char checkOrBet(Player p, Chromosome ch) {
    // Get ordinal of player's card J = 9, Q = 10, K = 11, A=12
    int card = p.showCard(0).getRank().ordinal();

    // Get index of strategy corresponding to card within chromosome
    int indStrategy = card - 9;

    // Generate random double between 0 and 1
    Random rand = new Random();
    double n = rand.nextDouble();

    // Determine strategy from corresponding parameter
    if (n <= ch.getGene(indStrategy)) {
        return 'b';
    } else {
        return 'c';
    }
}

/*
 * Return whether a player calls or folds from the chromosome passed
 * to it.
 */
public static char callOrFold(Player p, Chromosome ch) {
    // Get ordinal of player's card J = 9, Q = 10, K = 11
    int card = p.showCard(0).getRank().ordinal();

    // Get index of strategy corresponding to card within chromosome
    int indStrategy = card - 6;

    // Generate random double between 0 and 1.
    Random rand = new Random();
    double n = rand.nextDouble();

    // Determine strategy from corresponding parameter
    if (n <= ch.getGene(indStrategy)) {
        return 'c';
    } else {
        return 'f';
    }
}

/*
 * Plays a hand between two players using the given strategies and
 * returns the winning Player.
 */
public static Player chromoHandPlay(Player p1, Player p2, Chromosome
ch1, Chromosome ch2) {

    // P1 calls/folds
    char action = checkOrBet(p1, ch1);

```

```

if ( action == 'c' ) {
    // P2 checks/bets
    action = checkOrBet(p2, ch2);
    if ( action == 'b' ) {
        // P2 bets $1 and gets added to pot
        p2.bet(1);
        pot += 1;
        // P1 calls/folds
        action = callOrFold(p1, ch1);
        if ( action == 'c' ) {
            // P1 bets $1 and gets added to pot
            p1.bet(1);
            pot += 1;
            // Showdown: check cards and return winner
            return checkCards(p1, p2);
        } else {
            // P1 has folded so P2 wins
            return p2;
        }
    } else {
        // Showdown: check cards and return winner
        return checkCards(p1, p2);
    }
} else {
    // P1 bets $1 and gets added to pot
    p1.bet(1);
    pot += 1;
    action = callOrFold(p2, ch2);
    if ( action == 'c' ) {
        // P2 bets $1 and gets added to pot
        p2.bet(1);
        pot += 1;
        // Showdown: check cards and return winner
        return checkCards(p1, p2);
    } else {
        // P2 has folded so P1 wins
        return p1;
    }
}
}
}

```

Appendix H

Code for cardGame package

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> Player.java
 *
 * <dt> Description:
 * <dd> This class represents a player in a game of cards.
 * </dd>
 *
 * @author Paula Lopez Pozuelo
 */
package cardGame;
import java.util.ArrayList;

public class Player {

    // Player's money
    private int stack;

    // Player's hand
    private ArrayList <Card> hand;

    /**
     * Default constructor (players have 0 money by default)
     */
    public Player() {
        stack = 0;
        hand = new ArrayList<Card>();
    }

    /**
     * Constructor for creating a player with an initial amount of money.
     */
    public Player(int st) {
        stack = st;
        hand = new ArrayList<Card>();
    }
}
```

```
/*
 * Getter method that returns the value of the player's stack.
 */
public int getStack() {
    return stack;
}

/*
 * Setter method to modify the value of the player's stack.
 */
public void setStack(int s) {
    stack = s;
}

/*
 * Method to add a card to the player's hand.
 */
public void addCard(Card newCard) {
    hand.add(newCard);
}

/*
 * Method to show a specific card from a player's hand.
 */
public Card showCard(int ind) {
    return hand.get(ind);
}

/*
 * Method that takes away the first card in the player's hand.
 */
public Card popCard() {
    Card tempCard = hand.get(0);
    hand.remove(0);
    return tempCard;
}

/*
 * Method that takes away a specific card from the player's hand.
 */
public Card popCard(Card oldCard) {
    Card tempCard = oldCard;
    hand.remove(oldCard);
    return tempCard;
}

/*
 * Method that decreases stack by the value passed to it.
 */
public void bet(int n) {
    stack -= n;
}
```

```
/*
 * Method that increases stack by the value passed to it.
 */
public void win(int n) {
    stack += n;
}

/*
 * Method that returns a String with the player's current hand.
 */
public String handToString() {
    String str = "[";
    for (Card card : hand) {
        str += card.toString() + ", ";
    }
    str += "]";
    return str;
}
}
```



```
/**
 * <dl>
 * <dt> File Name:
 * <dd> Card.java
 *
 * <dt> Description:
 * <dd> This class represents a card in a game.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */
package cardGame;

public class Card {

    public enum Suit {
        SPADES, HEARTS, DIAMONDS, CLUBS;
    }

    public enum Rank {
        TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
        NINE, TEN, JACK, QUEEN, KING, ACE
    }

    private Suit suit;
    private Rank rank;

    /**
     * Constructor that takes rank and suit of the card to initialise.
     */
    public Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    /**
     * Getter method that returns the card's suit.
     */
    public Suit getSuit() {
        return suit;
    }

    /**
     * Setter method to modify the card's suit.
     */
    public void setSuit(Suit s) {
        this.suit = s;
    }

    /**
     * Getter method that returns the card's rank.
     */
    public Rank getRank() {
        return rank;
    }
}
```

```
/*
 * Setter method to modify the card's rank.
 */
public void setRank(Rank r) {
    this.rank = r;
}

/*
 * Method that returns true if the card passed to it has the same rank
 */
public boolean sameRank(Card otherCard) {
    return (rank == otherCard.getRank());
}

/*
 * Method that returns true if the card passed to it has the same suit
 */
public boolean sameSuit(Card otherCard) {
    return (this.suit == otherCard.getSuit());
}

/*
 * Method that returns a string to represent the current card.
 */
public String toString() {
    String cardString = "";
    switch (rank) {
        case ACE:
            cardString += "Ace";
            break;
        case TWO:
            cardString += "Two";
            break;
        case THREE:
            cardString += "Three";
            break;
        case FOUR:
            cardString += "Four";
            break;
        case FIVE:
            cardString += "Five";
            break;
        case SIX:
            cardString += "Six";
            break;
        case SEVEN:
            cardString += "Seven";
            break;
        case EIGHT:
            cardString += "Eight";
            break;
        case NINE:
            cardString += "Nince";
            break;
        case TEN:
            cardString += "Ten";
            break;
    }
}
```

```
        cardString += "Ten";
        break;
    case JACK:
        cardString += "Jack";
        break;
    case QUEEN:
        cardString += "Queen";
        break;
    case KING:
        cardString += "King";
        break;
    }
    cardString += " of ";
    switch (suit) {
        case SPADES:
            cardString += "Spades";
            break;
        case HEARTS:
            cardString += "Hearts";
            break;
        case DIAMONDS:
            cardString += "Diamonds";
            break;
        case CLUBS:
            cardString += "Clubs";
            break;
    }
    return cardString;
}
}
```

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> Deck.java
 *
 * <dt> Description:
 * <dd> This class represents a deck in a game of cards.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */

package cardGame;

import java.util.*;

public class Deck {

    // ArrayList to hold all the cards that are currently in the deck.
    private ArrayList <Card> cards = new ArrayList<Card>();

    // ArrayList to hold all the cards that have been dealt.
    private ArrayList <Card> dealtCards = new ArrayList<Card>();

    /**
     * Constructor of deck of three cards
     */
    public Deck() {
        cards.add(new Card(Card.Rank.JACK, Card.Suit.SPADES));
        cards.add(new Card(Card.Rank.QUEEN, Card.Suit.SPADES));
        cards.add(new Card(Card.Rank.KING, Card.Suit.SPADES));
    }

    /**
     * Constructor of a full deck of cards
     */
    public Deck(boolean b) {
        for (Card.Rank rank : Card.Rank.values()) {
            for (Card.Suit suit : Card.Suit.values()) {
                cards.add(new Card(rank, suit));
            }
        }
    }

    /**
     * Method that adds all dealt cards back to deck
     * and shuffles it randomly.
     */
    public void shuffle() {
        cards.addAll(dealtCards);
        dealtCards.clear();
        Collections.shuffle(cards);
    }
}
```

```
/*
 * Method that deals a card out of deck.
 */
public Card deal() throws EmptyDeckException {
    if (!cards.isEmpty()) {
        Card dealtCard = cards.get(0);
        dealtCards.add(dealtCard);
        cards.remove(0);
        return dealtCard;
    } else {
        throw new EmptyDeckException();
    }
}

/*
 * Method that returns a card to deck.
 */
public void add(Card card) {
    cards.add(new Card(card.getRank(), card.getSuit()));
}

/*
 * Exception for an empty Deck.
 */
class EmptyDeckException extends Exception {
    EmptyDeckException() {
        super();
    }
    EmptyDeckException(String s) {
        super(s);
    }
}
```

Appendix I

Code for geneticAlgo package

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> Chromosome.java
 *
 * <dt> Description:
 * <dd> Class that represents a chromosome in a Genetic Algorithm.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */

package geneticAlgo;

import java.util.Random;
import java.util.Arrays;

public class Chromosome implements Comparable<Chromosome> {

    // Fitness rate for chromosome, array holding genes
    // (which are random doubles between 0 and 1) and length
    private double fitness;
    private double[] genes;
    private static int length;

    // Random variable
    private static Random rand = new Random();

    /**
     * Chromosome constructor
     */
    public Chromosome(int l) {
        length = l;
        fitness = 0;
        genes = new double[length];
    }
}
```

```
/*
 * Chromosome clone constructor
 */
public Chromosome(Chromosome clone) {
    this.length = clone.getLength();
    this.fitness = clone.fitness;
    genes = new double[length];
    for (int i = 0; i < length; i++) {
        this.setGene(i, clone.getGene(i));
    }
}

/*
 * Method that randomizes gene values in chromosome.
 */
public void randomize() {
    for (int i=0; i<length; i++) {
        genes[i] = (double) rand.nextDouble();
    }
}

/*
 * Getter method for gene value
 */
public double getGene(int ind) {
    return genes[ind];
}

/*
 * Setter method for gene value
 */
public void setGene(int ind, double value) {
    genes[ind] = value;
}

/*
 * Getter method for chromosome length
 */
public int getLength() {
    return length;
}

/*
 * Getter method for fitness value
 */
public double getFitness() {
    return fitness;
}

/*
 * Setter method for fitness value
 */
public void setFitness(double f) {
    fitness = f;
}
```

```
/*
 * Method to reset fitness value to zero
 */
public void resetFitness() {
    fitness = 0;
}

/*
 * Method that returns true if the chromosome passe to it
 * has the same gene values.
 */
public boolean sameGenes(Chromosome compareChromo) {
    return (Arrays.equals(this.genes, compareChromo.genes));
}

/*
 * Method that returns offspring produced from One-Point crossover of
 * the
 * two chromosomes passed to it.
 */
public static Chromosome[] onePointCrossover(Chromosome ch1,
Chromosome ch2) {
    // Random number between 1 and length (both included, not 0 because
    it
    // would result in no crossover)
    int ind = rand.nextInt(length-1) + 1;

    // Create a two chromosome array to hold the children produced
    // from crossover.
    Chromosome[] newChromo = new Chromosome[2];
    newChromo[0] = new Chromosome(ch1);
    newChromo[1] = new Chromosome(ch2);

    for (int i = ind; i < length; i++) {
        newChromo[0].setGene(i, ch2.getGene(i));
        newChromo[1].setGene(i, ch1.getGene(i));
    }

    return newChromo;
}

/*
 * Method that returns offspring produced from Uniform crossover of
 * the
 * two chromosomes passed to it.
 */
public static Chromosome[] uniformCrossover(Chromosome ch1, Chromosome
ch2) {
    // Variable that determines similarity to each parent
    double n = rand.nextDouble();
    n = 0.5;

    // Make copies of parents
    Chromosome[] newChromo = new Chromosome[2];
    newChromo[0] = new Chromosome(ch1);
```



```

        newChromo[1] = new Chromosome(ch2);

        for (int i = 0; i < length; i++) {
            double r = rand.nextDouble();
            if (r < n) {
                newChromo[0].setGene(i, ch2.getGene(i));
                newChromo[1].setGene(i, ch1.getGene(i));
            }
        }

        return newChromo;
    }

    /**
     * Method that returns mutation of chromosome.
     */
    public Chromosome mutate() {
        int ind = rand.nextInt(length);
        Chromosome mutatedChromo = new Chromosome(this);
        mutatedChromo.setGene(ind, rand.nextDouble());
        return mutatedChromo;
    }

    /**
     * Method that returns String with gene values of chromosome.
     */
    public String toString() {
        String chromo = "[";
        for (int i=0; i<(length-1); i++) {
            chromo += String.format("%.2g", getGene(i));
            chromo += ", ";
        }
        chromo += String.format("%.2g", getGene(length-1));
        chromo += "]";
        return chromo;
    }

    @Override
    public int compareTo(Chromosome compareChromo) {
        double compareFitness = compareChromo.getFitness();
        if (this.getFitness() > compareFitness) {
            return -1;
        } else if (this.getFitness() == compareFitness) {
            return 0;
        } else {
            return 1;
        }
    }
}

```

```
/**
 * <dl>
 * <dt> File Name:
 * <dd> Population.java
 *
 * <dt> Description:
 * <dd> Class that represents a Population in a Genetic Algorithm.
 * </dl>
 *
 * @author Paula Lopez Pozuelo
 */

package geneticAlgo;

import java.util.Random;
import java.util.Arrays;
import java.util.*;

public class Population {

    // Population parameters
    private int size;
    private Chromosome[] chromos;
    private double fitness;
    private int chromoLength;

    // Rates for population
    private static double crossRate;
    private static double mutateRate;

    /**
     * Population constructor
     */
    public Population(int s, int cL, double c, double m) {
        size = s;
        chromoLength = cL;
        crossRate = c;
        mutateRate = m;
        chromos = new Chromosome[size];
    }

    /**
     * Population clone constructor.
     */
    public Population(Population clone) {
        this.size = clone.size;
        this.chromos = clone.chromos;
        this.chromos = new Chromosome[size];
        for (int i = 0; i < size; i++) {
            this.setChromo(i, new Chromosome(clone.getChromo(i)));
        }
        this.fitness = clone.fitness;
        this.crossRate = clone.crossRate;
        this.mutateRate = clone.mutateRate;
    }
}
```

```
/*
 * Generates a population of distinct random chromosomes
 */
public void randomize() {
    for (int i = 0 ; i < size; i++) {
        Chromosome randomChromo = new Chromosome(chromoLength);
        randomChromo.randomize();
        while (this.contains(randomChromo, i)) {
            randomChromo.randomize();
        }
        chromos[i] = new Chromosome(randomChromo);
    }
}

/*
 * Generates a population of distinct chromosomes with uniformly
 * distributed values.
 */
public void randomizeUniformly() {
    ArrayList<Integer> values = new ArrayList<Integer>();
    for (int i = 0; i < size; i++) {
        values.add(i);
    }
    for (int i = 0; i < size; i++) {
        chromos[i] = new Chromosome(chromoLength);
    }
    for (int i = 0 ; i < chromoLength; i++) {
        Collections.shuffle(values);
        for (int j=0; j < size; j++) {
            chromos[j].setGene(i, (double) (values.get(j)%101)/100);
        }
    }
}

/*
 * Returns true if population contains given chromosome.
 */
public boolean contains(Chromosome chromo, int topLimit) {
    for (int i=0; i<topLimit; i++) {
        if (chromo.sameGenes(chromos[i])) {
            return true;
        }
    }
    return false;
}

/*
 * Evaluate total fitness of population.
 */
public void evalTotalFitness() {
    fitness = 0;
    for (int i = 0 ; i < size; i++) {
        fitness += chromos[i].getFitness();
    }
}
```

```
/*
 * Returns average fitness of population.
 */
public double avgFitness() {
    evalTotalFitness();
    return fitness/size;
}

/*
 * Select and return chromosome using Roulette-Wheel selection.
 */
public Chromosome selectRouletteWheel() {
    evalTotalFitness();
    Random rand = new Random();
    double randFitness = rand.nextDouble() * (fitness);
    int i;
    for (i = 0; (randFitness > 0) && (i < size); i++) {
        randFitness -= (chromos[i].getFitness());
    }
    return chromos[i-1];
}

/*
 * Select and return chromosome using Rank selection.
 */
public Chromosome selectRankWheel() {
    int rankTotal = 0;
    for (int i = 0; i < size; i++) {
        rankTotal += i;
    }
    this.sort();
    Random rand = new Random();
    double randRank = rand.nextDouble() * rankTotal;
    int i;
    for (i = 0; (randRank > 0) && (i < size); i++) {
        randRank -= i;
    }
    return chromos[size-(i-1)];
}

/*
 * Select and return chromosome using Tournament selection.
 */
public Chromosome selectTournament(int tourSize) {
    Random rand = new Random();
    double bestFitness = -1;
    int bestInd = 0;
    for (int i = 0; i < tourSize; i++) {
        int randInd = rand.nextInt(size);
        if (chromos[randInd].getFitness() > bestFitness) {
            bestFitness = chromos[randInd].getFitness();
            bestInd = randInd;
        }
    }
    return chromos[bestInd];
}
```

```
}

/*
 * Find and return fittest chromosome.
 */
public Chromosome getFittest() {
    Chromosome fittest = chromos[0];
    for (int i = 1 ; i < size; i++) {
        if (fittest.getFitness() < chromos[i].getFitness()) {
            fittest = chromos[i];
        }
    }
    return fittest;
}

/*
 * Sort population.
 */
public void sort() {
    Arrays.sort(chromos);
}

/*
 * Return specified chromosome from population.
 */
public Chromosome getChromo(int ind) {
    return chromos[ind];
}

/*
 * Set chromosome in population.
 */
public void setChromo(int ind, Chromosome ch) {
    chromos[ind] = ch;
}

/*
 * Returns mutated copy of chromosome (subject to mutation rate)
 */
public static Chromosome mutate(Chromosome ch) {
    Random rand = new Random();
    double prob = rand.nextDouble();
    if ( prob < mutateRate ) {
        return ch.mutate();
    } else {
        return ch;
    }
}

/*
 * Method that returns offspring produced from Uniform crossover,
 * subject to the crossover rate
 */
public static Chromosome[] uniformCrossover(Chromosome[] ch) {
    Random rand = new Random();
```

```
double prob = rand.nextDouble();
if (prob < crossRate) {
    return Chromosome.uniformCrossover(ch[0], ch[1]);
} else {
    return ch;
}
}

/*
 * Method that returns offspring produced from One-Point crossover,
 * subject to the crossover rate
 */
public static Chromosome[] onePointCrossover(Chromosome[] ch) {
    Random rand = new Random();
    double prob = rand.nextDouble();
    if (prob < crossRate) {
        return Chromosome.onePointCrossover(ch[0], ch[1]);
    } else {
        return ch;
    }
}
}
```

Bibliography

- [1] Collin R. Reeves, Jonathan E. Rowe. *Genetic Algorithms - Principles and perspectives*. Kluwer Academic Publishers, 2003.
- [2] Ken G. Binmore *Fun and Games: A Text on Game Theory*.
- [3] Darse Billings, Aaron Davidson, Jonathan Schaeffer, Duane Szafron. *The challenge of Poker*. Artificial Intelligence 134 (1-2) pp 201-240.
- [4] Noraini Mohd Razali, John Geraghty. *Genetic Algorithm Performance with Different Selection Strategies in Solving TSP*. Proceedings of the World Congress on Engineering 2011, Vol II.
- [5] Rakesh Kumar and Jyotishree. *Blending Roulette Wheel Selection and Rank Selection in Genetic Algorithms*. International Journal of Machine Learning and Computing, Vol. 2, No. 4, August 2012.
- [6] Tobias Blickle and Lothar Thiele. *A Comparison of Selection Schemes used in Genetic Algorithms* TIK-Report, No. 11, December 1995, Version 2.
- [7] Garrett Nicolai, Robert Hilderman. *Algorithms for evolving no-limit Texas Hold'em poker playing agents*. CIG'09 Proceedings of the 5th international conference on Computational Intelligence and Games, pp 125-131.
- [8] Kanta Vekaria and Chris Clack. *Selective Crossover in Genetic Algorithms: An Empirical Study*. Lecture Notes in Computer Science Volume 1498, 1998, pp 438-447.
- [9] Breth Hoehn, Finnegan Southey, Robert C. Holte, Valeriy Bulitko. *Effective Short-Term Opponent Exploitation in Simplified Poker* Machine Learning. February 2009, Volume 74, Issue 2, pp 159-189.

- [10] Lawrence Davis. *Handbook of Genetic Algorithms*
- [11] John H. Holland. *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*
- [12] University of Alberta. Computer Poker Research Group. <http://poker.cs.ualberta.ca/>