UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Using grammatical evolution to create blackjack players

Dean Maloney
14140306

Supervisor:
Dr. Conor Ryan

Course:
LM110 BSc in Computer Games Development

Summary

This project aims to research the effectiveness of using grammatical evolution to create generations of AI players capable of making decisions while playing blackjack, and to learn about the process of genetic evolution and how it is researched and developed.

Blackjack is an immensely popular casino game in which players are trying to beat the dealer by getting their card values as close to 21 as possible, without going over. Each game requires the player to decide whether to keep drawing new cards, or “hitting”, until they either decide to stay or they have cards equal to or above 21. Each card is assigned a value equal to their numerical value, with face cards getting a value of 10 and aces being either 1 or 11. Along with hitting or staying, players can also place extra bets by other means, such as doubling down or splitting.

The aim of the project is to research the process of genetically evolving players using grammatical evolution that can learn to make correct decisions regarding whether to hit or stay, along with taking doubling down and splitting into account. These players will maintain ‘genes’ which will determine how the player will react to a specific scenario. A fitness function will be used to determine which players are performing the best, in order to create a new generation of players with superior genes. Data will be collected from these players as they progress, allowing for analysis of how the players improved over time.

The language used to code the blackjack system was Java, with additional frameworks being used to implement grammatical evolution. Log4J was used to produce logs containing data regarding the system in action, which were then parsed by Logstash and sent to an Amazon Web Services Elasticsearch cluster, to be viewed in dashboards in Kibana.

Introduction

Overview

Grammatical evolution is a method of genetic evolution pioneered by Conor Ryan, J.J. Collins and Michael O'Neill in 1998. It is defined as "an evolutionary algorithm (EA) that can evolve complete programs in an arbitrary language using a variable-length binary string" (Conor Ryan, Michael O'Neill, 2003). The goal of the project is to explore how grammatical evolution can be applied to the game of blackjack and the decisions that are required by a player to succeed and win as often as possible.

Blackjack is a staple at casinos worldwide as it is easy to learn yet hard to master. In a game of blackjack, players are trying to beat the dealer by getting their card values as close to 21 as possible, without going over. Each card is assigned a value equal to their numerical value, with face cards getting a value of 10 and aces being either 1 or 11.

Once initial bets are placed, both the dealer and the player are dealt two cards; both the player's cards are revealed while only one of the dealer's cards is face up. As long as neither person has cards equalling values of 21, the game progresses. The player may choose to draw more cards by "hitting", or pass priority to the dealer by "staying". If a player hits and gets an overall value of 21 or higher, the dealer takes their turn. The dealer then hits until they either go above 21 or reach a certain threshold set by the casino; a typical threshold is a "soft-17", or a hand consisting of an ace counting as an 11 and other cards which add up to 17 total. If the player had a better hand than the dealer, they win the bet, otherwise they either get their bet returned in the case of a draw or taken by the dealer after a loss.

In addition to the above, there are extra decisions the player can make before hitting or staying. After you've been dealt your first two cards, you have the choice of doubling your initial bet — but you get only one additional card. This is only advantageous in certain situations, as you'll often want to draw more than one card to get closer to 21. Players can also split: when they are dealt a pair, where both cards are the same, they have the option to 'split' them into two new hands. They are then dealt two more cards, one for each new hand, and their bet is doubled. They play each hand normally and get two chances to beat the dealer. Again, this is quite risky, as splitting two 10's is often more beneficial than splitting two 1's. It takes experience to know when the best time to double down or split is during a session of blackjack.

The aim of the project is to research the process of genetically evolving players using grammatical evolution that can learn to make correct decisions regarding whether to hit or stay, along with taking doubling down and splitting into account. These players will maintain 'genes' which will determine how the player will react to a specific scenario. A fitness function will be used to determine which players are performing the best, in order to create a new generation of players with superior genes. Data will be collected from these players as they progress, allowing for analysis of how the players improved over time.

The language used to code the blackjack system was Java, as it co-operates with many of the tools used elsewhere in the project along with being a strong object-oriented programming language. Additional frameworks were studied for use to implement a genetic evolution solution, such as ECJ. Log4J was used to produce logs containing data regarding the system in action and how players improved throughout the evolutionary process. Logstash reads these logs, parses them and outputs them to an Elasticsearch cluster by using JSON. Amazon Web Services Elasticsearch was chosen as the endpoint

for Logstash, which comes with the Kibana dashboard interface to represent the data obtained.

Motivation

This project idea appealed to me as I was interested in seeing how evolutionary computation works from a back-end perspective. Seeing it applied to the game of blackjack was also of interest, since it is a very popular game worldwide and has a layer of strategy behind it. I personally enjoy studying statistics and understanding the significance of different values and how they affect the system, and this project allows for such observation. This is also an opportunity for me to learn more about grammatical evolution from one of its innovators, Conor Ryan, and to see how it can be applied to a real-life scenario.

Objectives

I plan to improve my knowledge on grammatical evolution, genetic evolution and artificial intelligence as a whole and how it can be used to great effect in a situation such as creating blackjack players. I also plan to improve my analysis skills by constantly optimizing my system through use of frameworks and experimenting with different fitness functions, and observing the results through logs and player data.

Overview of Research

Core Research

Blackjack

Understanding the game of blackjack and all of its intricacies is vital to creating accurate simulations and results. While the core game is reasonably simple, there is a lot of extra features and house rules that must be taken into account to ensure that the casino always maintains the edge against the player.

In the game of blackjack, there are four possible player decisions: hit, stay, double down and split. The player can hit or stay no matter what, unless the player or dealer has accumulated a points total of 21. When it comes to doubling down, you can only do so after being dealt your first two cards, and you are limited to drawing one more card with the advantage of doubling your current bet. Splitting can only be done if both cards that you have been dealt carry the same value (most casinos consider face cards to be 10's in this scenario), and it causes the player to separate the two cards into two hands and place a new bet on the newly created hand.

There are a lot of rules surrounding splitting due to how advantageous it can be for the player. Splitting a pair of aces is highly regulated, to the point where you can't hit on either hand after splitting. Most casinos also limit the amount of times a player can split in a single game to four. A player can still double down after a split, but it must be immediately afterwards before hitting.

There are also some aspects of the game that can sway the edge towards the casino without the player knowing it. It has been mathematically proven that a fewer number of decks in play leads to more blackjacks (a 10 value card and an ace opening hand) being dealt out, which benefits the player since the casino offers a bonus rate for blackjacks (Ken Smith, 2016). Along with this, some casinos pay out less for blackjacks than others, with 3 to 2 being the most common payout while some offer 6 to 5 instead. This can have a huge effect on the player's winnings, so it's something they need to look at before they sit down and play.

To ensure I fully understood the game, I played in multiple simulations online that used a set of scenarios designed to test a seasoned blackjack player, in order to ensure I was aware of all of blackjack's smaller rules.

Genetic Programming & Grammatical Evolution

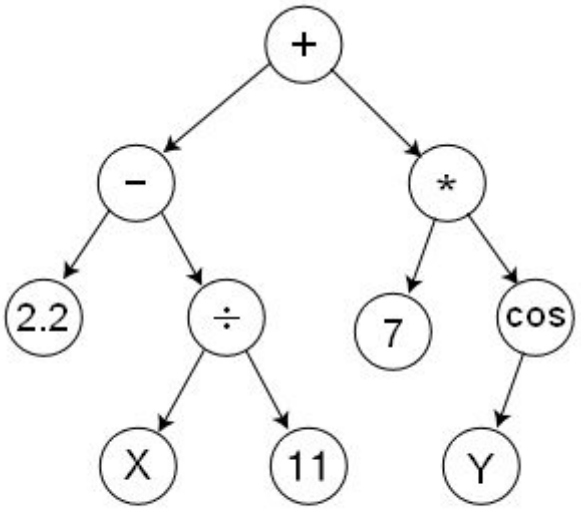
I built upon my knowledge of genetic evolution that I learned from developing my product by researching genetic programming and how grammatical evolution builds upon it.

Genetic programming has been a topic of interest since the early 1950's. Italian-Norwegian mathematician Nils Aall Barricelli worked on one of the earliest examples of genetic simulation in the Institute of Advanced Study in Princeton, New Jersey. According to an Nautilus article on his innovations, "Barricelli programmed some of the earliest computer algorithms that resemble real-life processes: a subdivision of what we now call "artificial life," which seeks to simulate living systems—evolution, adaptation, ecology—in computers." (Robert Hackett, 2014). He introduced many basic concepts such as genes, mutation and reproduction which took inspiration from the evolution of humans ourselves. He established the idea of an individual

possessing genes, and those who perform better than others are used in future generations and mutated to provide variance.

Work continued in the decades that followed, however many programmers in the field found that it was too computationally intensive to use genetic programming in more demanding scenarios. Improvements in CPU power and GP technology did improve its standings in recent years, however, and it finds uses in quantum computing, electronic design, game playing, cyber-terrorism prevention, sorting, and searching.

Genetic programming differs from regular genetic evolution techniques as it evolves programs, represented in tree structures. Typically in genetic programming examples, the Lisp programming language is used, as it naturally embodies tree structures. These tree structures represent functions, with each parent node being an operator and each terminal node being a variable.



An example of a function tree used in genetic programming (Wikipedia, 2018)

Grammatical evolution takes the ideas of genetic programming and adds the concept of a Formal Grammatical Structure (Peter Adam, 2017). This is a set of rules that defines “what makes sense” for the program to realize and learn during computation. The process of grammatical evolution is as follows:

1. Define a Backus Naur Form (BNF) representation of the grammar;
2. Randomly generate linear chromosomes to encode information;
3. Evaluate the fitness of chromosomes by mapping the grammar to the chromosome and testing against the fitness function;
4. Use Genetic Operators such as Crossover, Mutation and Reproduction to generate next generation of chromosomes;
5. Repeat steps 3 and 4 for a certain number of generations.

A BNF grammar typically uses four categories:

- T: The terminal set
- N: The non-terminal set
- P: Set of production rules
- S: Start symbol (which is a member of N)

The terminal set is a list of variables and operators that will be used in the program. If the grammar were to contain all four basic operators and variables x and y, set would look like:

```
T = {+, -, /, *, x, y}
```

The non-terminal set is the set of all production activities that the program can undertake. They are encased in chevrons, and an example is shown below, with e for expressions, o for operators and v for variables:

```
N = {<e>, <o>, <v>}
```

The production rules takes the terminal and non-terminal sets and defines what should replace each instance of a non-terminal element when encountered in the function tree. A non-terminal element can have multiple terminal or non-terminal elements assigned to it, each split by a |. An example is shown below:

```
<e> ::= <e><o><e> | <v>

<o> ::= + | - | / | *

<v> ::= x | y
```

The start symbol simply defines what element of the non-terminal set appears first. An example is shown below, with e as the chosen start symbol:

```
S = <e>
```

Chromosomes, or genes, are used by grammatical evolution to determine which option defined in the production rules should be chosen in a given scenario. A chromosome is an integer array filled with positive values, with an example shown below:

```
C: [10, 63, 25, 4, 87, 6]
```

Given the start symbol of <e>, the system decides whether to use <e><o><e> or <v> by determining the number of possible values, getting the modulus of the value in the chromosome when divided by this number and then making the decision. In this case, 10 mod 2 would give 0, which would mean that the chosen option was <e><o><e>. The function is then checked again and again until all non-terminal values are removed from the function by iterating through the chromosome.

Once the function is formed, it can be put to the test through a fitness function and a fitness value can be attained. Genetic programming generally uses a Koza fitness, where the ideal fitness is as close to 0 as possible. Once this fitness has been computed for all individuals in a generation, a selection process occurs and a mixture of crossover and mutation of chromosomes is completed before a new generation is produced using these new chromosomes as a base.

Existing Products

During my time researching projects that used general genetic evolution in real-life scenarios, I came across a project similar to mine that aimed to create an optimal strategy for blackjack using genetic evolution utilizing Java and the Watchmaker framework (Brian Benchoff, 2013). While his project didn't take into account being able to split, it was a good starting point towards being able to put my project into perspective. It suggested a fitness function to use, along with explaining how certain aspects of evolutionary computation can be abstracted using Java objects. In the end I opted for a completely differently-structured blackjack system compared to the one used here.

I also found an example of grammatical evolution being used to play the game Mrs. Pac Man optimally, which was supervised by innovator Michael O'Neill (Edgar Galvan-Lopez, John Mark Swafford, Michael O'Neill, Anthony Brabazon, 2010). This was an interesting read to see how grammatical evolution can be used in a game with much more complex decisions, as it is required to take into account the nearest pill, the location of each ghost and where in the maze Mrs. Pac Man is located, while also using score as a base for the fitness value. The system was also written in Java, however they opted to use GEVA instead of ECJ.

Relevant Academic Research

As part of a project for my Intelligent Systems module, which focused on the mechanisms of various artificial intelligence systems and their applications in real life, I was required to create a simulated annealing system which plotted an adjacency matrix onto a circle, with each point being as close as possible to each other. Of note was the fact that it utilized a fitness function to determine when to stop determining the distance between points in the matrix. I revisited this project to recall how the fitness function fits into the system, as I would be required to use one as part of my project.

I was also invited to the BDS Group mailing list for people in the university who are actively involved in innovating grammatical evolution. The BDS Group is a gathering of individuals from University of Limerick who have an interest in genetic programming and its applications. These include PhD students and lecturers within the CSIS department, along with Conor Ryan himself. They organize meetings on a weekly basis to discuss papers published by global organizations about their findings on genetic programming in given scenarios. I was able to attend some of these meetings and observe the process of reading through these papers to find aspects of note and commenting on them. Though I wasn't able to

contribute consistently myself with my limited knowledge on the topic, it was still interesting to be a part of.

I have also researched many papers and webpages that relate to the topic of grammatical evolution. Excerpts from books such as Grammatical Evolution by Conor Ryan and Michael O'Neill are publically available online, and I had the chance to read through them to gain a better understanding of how grammatical evolution compares to other evolutionary computation practices. I met with Conor Ryan, my supervisor, on numerous occasions to further discuss the finer aspects of grammatical evolution which will tie into my research.

Implementation Technologies

ECJ

Evolutionary Computation Java, or ECJ, is a Java framework that provides a variety of tools for genetic programmers to use to create and breed individuals for their specified problem. It is written in Java and uses a parameter file written by the user to determine its settings at runtime. It is available under the George Mason University Department of Computer Science website for the Evolutionary Computation Laboratory, with the latest version being Version 25.

An advantage of being written in Java means that all of its classes can be extended to further suit the programmer's needs, as well as the fact that each segment of the genetic process is condensed into an object. The framework can be implemented with relative ease into a Java project, especially when using an integrated development environment, or IDE, such as IntelliJ.

The core elements of ECJ are shown below:

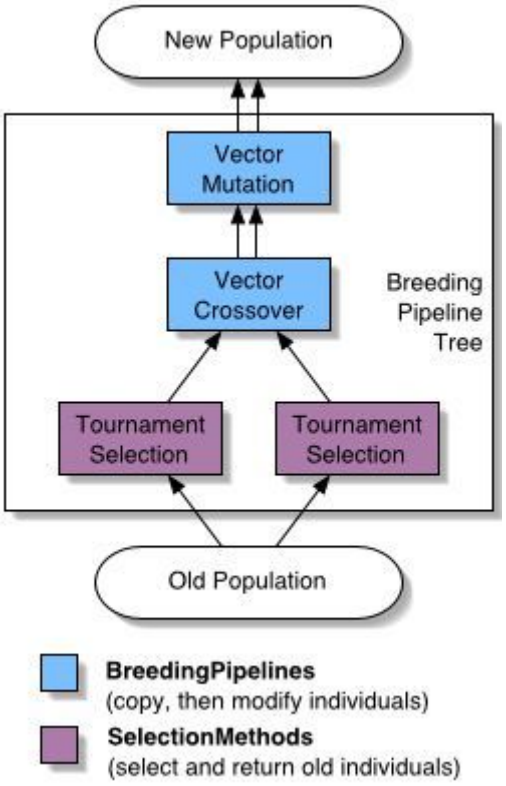
- Initializer, responsible for creating the initial population.
- Population, created initially by the Initializer. A Population stores an array of Subpopulations. Each Subpopulation stores an array of Individuals, plus a Species which specifies how the Individuals are to be created and bred.
- Evaluator, responsible for evaluating individuals.
- Breeder, responsible for breeding individuals.
- Exchanger, responsible for exchanging individuals among subpopulations or among different processes.
- Finisher , responsible for cleaning up when the system is about to quit.

ECJ follows a process for creating and evolving Individuals, as shown below:

1. Call the Initializer to create a Population.
2. Call the Evaluator on the Population, replacing the old Population with the result.
3. If the Evaluator found an ideal Individual, and if we're quitting when we find an ideal individual, then go to Step 9.
4. Else if we've run out of generations, go to Step 9.
5. Call the Exchanger on the Population (asking for a Pre-breeding Exchange), replacing the old Population with the result.
6. Call the Breeder on the Population, replacing the old Population with the result.
7. Call the Exchanger on the Population (asking for a Post-breeding Exchange), replacing the old Population with the result.
8. Increment the generation count, and go to Step 2.
9. Call the Finisher on the population, then quit.

ECJ determines the type of class that should be used in the user's system based on a supplied parameter file. The user specifies which of ECJ's suite of classes they wish to use for each aspect of the process. For example, the user could choose a BitVectorIndividual as their desired Individual if they wanted an array of booleans, or an IntegerVectorIndividual if they wanted an array of integers instead.

One of the key aspects of ECJ that must be taken into account while using it is the breeding procedure. This is the process of selecting specific individuals from a generation after their fitness has been calculated, mutating them together and using this spliced individual as a basis for the next generation. ECJ allows for a large amount of customization when defining the breeding procedure; a Tournament Selection can be used to find the fittest individual in a specified sample of the generation, or the Fit Proportionate Selection can be used to get the fittest individual regardless of sample size. A custom pipeline can also be implemented for more unusual scenarios, such as finding the individual with the worst fitness.



An example of an ECJ breeding pipeline. (George Mason University Department of Computer Science)

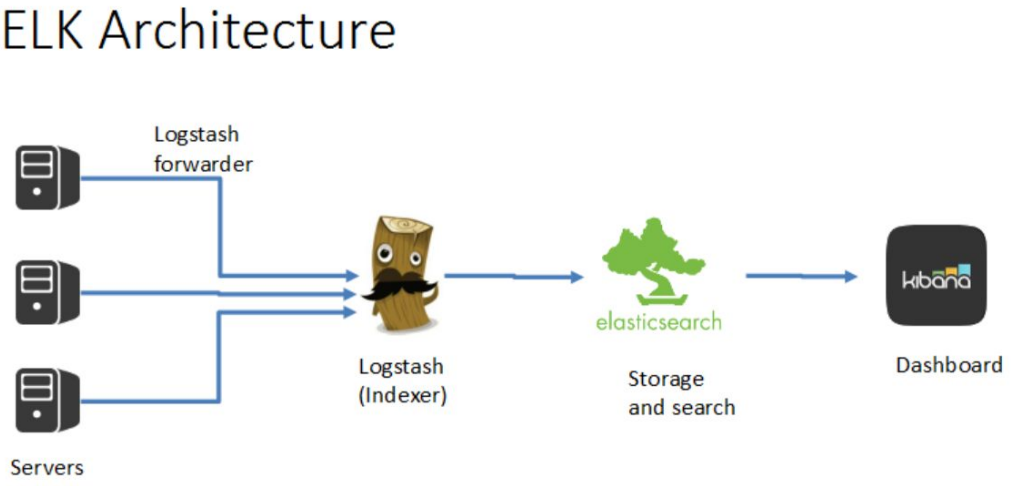
My research required me to understand how the entire system works so I can customize it for my product's specific needs. The ECJ website contains a series of tutorials which go into depth on each aspect of its system while implementing a genetic solution to a given problem. Tutorial 1 and 2 were specifically useful in the implementation of my product as they explain the basics of ECJ while also showing the process of using BitVectorIndividuals and IntegerVectorIndividuals, which were used in various iterations of the product.

Though it was not implemented in the product, ECJ comes built-in with support for genetic programming and grammatical evolution (Sean Luke, 2015). The GEIndividual class is a version of the IntegerVectorIndividual that contains an integer array to be used alongside a supplied grammar file. The GESpecies class then follows standard grammatical evolution procedures to

create a typical genetic programming function tree, which is stored in a GPIndividual and provided to the fitness function to determine its fitness. Multiple grammar files are required for each tree that the GPIndividual is expected to hold.

Elasticsearch-Logstash-Kibana (ELK)

Elasticsearch-Logstash-Kibana (ELK) is a combination of three systems focused on the storage, parsing and representation of log data respectively. When a log is appended while Logstash is running, it will read the new data, parse it and convert it into JSON. This JSON data will then be sent to an Elasticsearch cluster specified within its configuration, where it is stored under an index. Most versions of Elasticsearch comes built-in with the Kibana dashboard, which allows the user to view the data stored in Elasticsearch in dashboards and graphs.



A diagram showing the flow of how log data is handled in the ELK system (Arun Prasath, 2016)

The main focus at the start of the project schedule was organizing the ELK system along with the Log4J framework, as it would reduce time debugging the overall system once it was in place. I needed to determine what the

headers for a player's data would be when they were logged. I also needed to decide what plugins Logstash would use when reading, parsing and outputting the data, along with learning the syntax for each. I chose the 'kv' and 'mutate' filter plugin as it allows headers and values to be easily extracted from a single-line log while also allowing for values such as StartingBank and CurrentBank, which represent money, to be converted to integers.

While deciding on an output plugin, I had to determine how I wanted to implement an Elasticsearch cluster. Elasticsearch typically comes bundled with Kibana, which accesses data gathered by the Elasticsearch cluster and creates graphs and dashboards based on this data. Maintaining a local Elasticsearch interface on my laptop was an option, however during testing, I realized it was not a very portable solution since I was restricted to using Kibana on only my laptop. I chose Amazon Web Services (AWS) Elasticsearch as it allowed me to access Kibana anywhere to view my dashboards once I configured the security settings. I created an AWS account, provisioned a host which would contain the Elasticsearch cluster and followed guides to determine the settings I needed to maintain the cluster.

Once I had the Elasticsearch cluster set up, I then researched ways to connect it with Logstash. The best solution seemed to be the 'amazon-es' output plugin, however I faced the issue of a lack of detailed and up-to-date documentation regarding how to integrate it into Logstash's configuration file. After some trial and error, I successfully installed the plugin and discovered that the plugin required specific access keys produced by AWS's IAM user security settings, along with the region name that the host is located in and an index to identify the log data. Once I updated the config file, I created a sample single-line log and supplied it to Logstash, which it successfully sent to the AWS Elasticsearch cluster.

The next step was researching how to use Kibana and create a dashboard. I

learned how to create bar and pie charts using the data stored in the Elasticsearch cluster and decided on what meaningful data should be represented in the dashboard.

Log4J

When ELK was ready, the next step was researching the Log4J framework to create logs that Logstash could read. I downloaded the Log4J 2.9 library and integrated it into IntelliJ, my IDE of choice for the project. I created an IntelliJ project and created a log4j.properties file in the resources folder. I then researched the syntax for the properties file so I could configure it to behave in the way I desired, as I needed the logs to be in single-line format.

I also researched the ECJ and Watchmaker frameworks which are used to implement evolutionary computation in Java. ECJ was recommended by my supervisor, while Watchmaker was used in the blackjack example mentioned earlier. I plan on researching ECJ further once I'm more comfortable with the idea of grammatical evolution.

IntellliJ

IntelliJ is an integrated development environment, or IDE, developed by JetBrains, and was first released in 2001. It has become one of the top IDE's in the industry in recent years and is used by technological companies over competitors such as Eclipse and NetBeans for its clean user interface, keyboard shortcuts which help save time and other ease-of-life features to help a developer in a wide array of languages.

I chose IntelliJ as my IDE to use during the course of the project as its user interface is the most pleasant compared to other IDE's I have tried in the past and it allows for smooth debugging and testing. External libraries are also a key element of the product due to ECJ and Log4J , and IntelliJ is capable of easily adding libraries and building the project using them.

GitHub

GitHub is a version control and source code management tool available online and through a downloadable client. It uses git as a base and allows users to create a repository, upload their code in batches known as "commits" and offers a wide number of tools to keep track of which files were edited when and to allow for easy downloading, or "pulling" of the entire repository.

I decided to use GitHub to maintain version control over my project as it lets me access my project online and also lets me revert a commit to a working commit if a bug arises. This proved especially useful when IntelliJ couldn't recognize my project and I needed to pull all of my most recent data from the repository.

Prototypes and Exploratory Programs

Initial Blackjack System

My first prototype of the product was creating a game of blackjack in Java where the only two possible decisions were "hit" and "stay". This would take

decisions from user input through the console and kept track of wins and losses. This initial prototype set the precedent for future iterations, as a project structure was established that would be maintained throughout development of the product. Cards were represented by integer values, a Rank enum and a Suit enum and were deemed GameObjects, while the Hand object contained an ArrayList of Card objects that both the player and dealer have been dealt, and was deemed a PlayerObject.

ECJ Tutorials

I followed the first two tutorials available under the ECJ website and learned how to set parameters for ECJ and the significance of setting certain values in different instances.

The first tutorial was a simple problem that used BitVectorIndividuals, a type of individual that uses an array of booleans as a genome, and seeked to find an individual with a genome filled entirely with “true” statements. This explored basic concepts such as the breeding pipeline using Tournament Selection, the fitness function at its core and how to create a Problem class that ECJ can use. It showed the process of running ECJ from the command line and specifying the params file to be used.

The second tutorial expanded upon the first tutorial by using IntegerVectorIndividuals, which would prove to be much more useful in the implementation of the product. Instead of an array of booleans, these individuals contained integers, which allowed for a much wider variety of decisions that an individual can make. The tutorial introduced a different type of breeding pipeline that used Fit Proportionate Selection in conjunction with Tournament Selection and also implemented its own custom mutator pipeline, which flips the minus-sign of any integer that is mutatated at a 0.05 probability per gene.

Blackjack System Incorporating ECJ

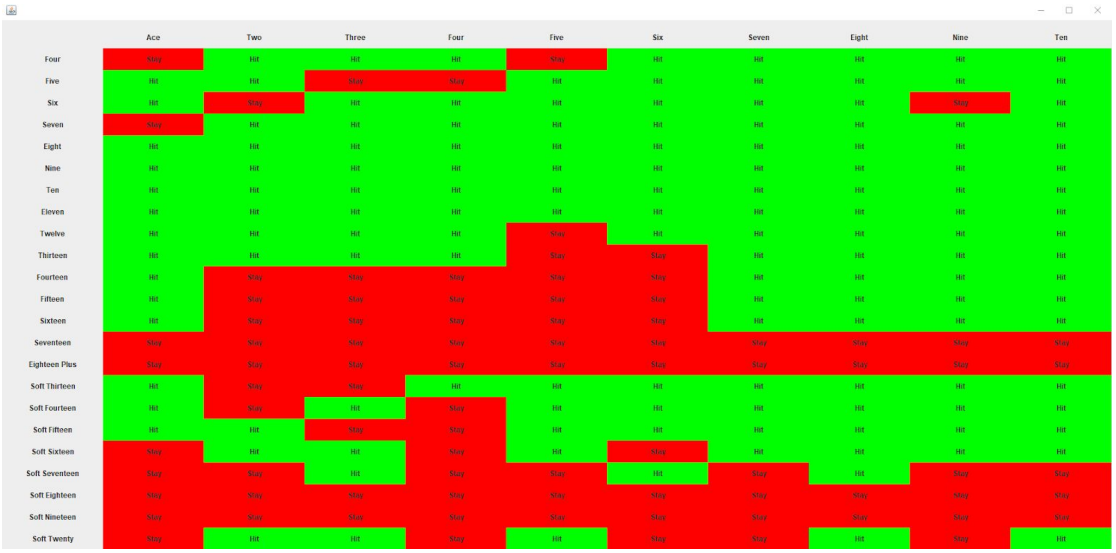
After building a basic understanding of how ECJ works, it was incorporated into the existing blackjack system with BitVectorIndividuals. Initially the genome array was of length 240, with 24 possible player hands (4 to 18+, and soft 12 to soft 20) and 10 possible dealer hands (ace to 10). Each gene was either “true” or “false”, to signify if the player should hit or stay. A Player object was created which contains a BitVectorIndividual along with a Hand object, and code was written to determine where in the genome array the decision for the specific scenario was stored.

The initial fitness function was simply the funds the player had left over after playing a set number of games. Players would bet €2 in each game, and once the set number of games was over, the individual would be assigned the value of their leftover funds.

To begin writing logs for ELK to parse, Log4J was implemented into the build. After each player has played their games, their tallies and their fitness were logged to a “logs.txt” file in a single-line form. Once the entire job was complete, the genome of the fittest individual was logged in the “jobData.txt” file along with data relating to the job itself. Logstash’s configuration was then adapted to the changes made to the system, and time was spent configuring Kibana to represent this new data in graphs and dashboards.

While the ELK system is great for analyzing data from a long range in time, it doesn’t have a clean solution for representing the genome of the best individual of a run. In order to properly analyze the genome of the best individual besides comparing fitness values to other runs, a user interface was created to view each individual gene as represented by the decision a

player could make. This was designed using JFrame and consisted of a grid of JPanels showing either green for “hit” or red for “stay.



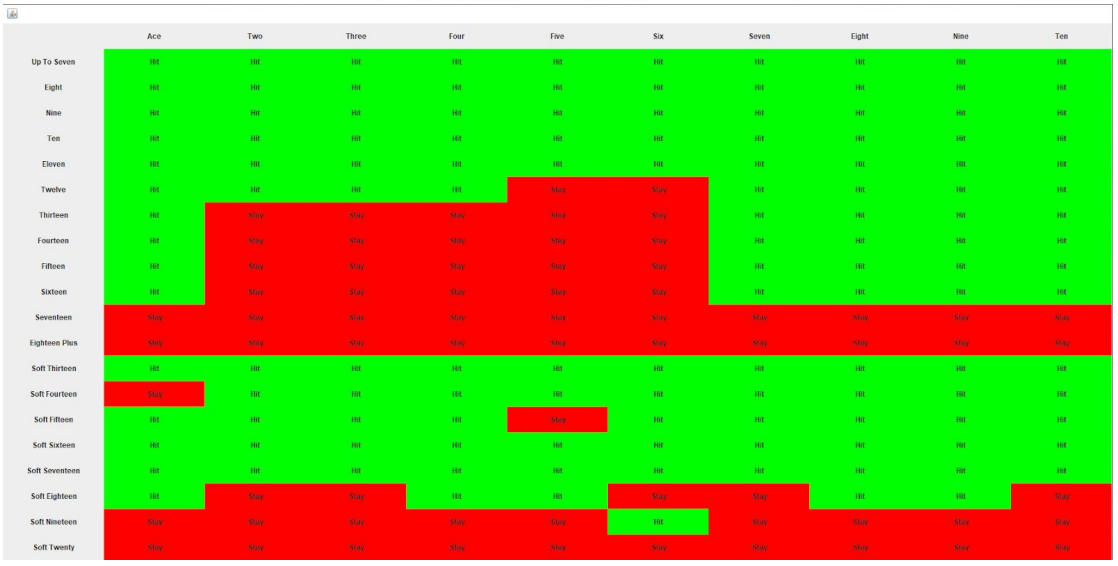
An early example of a blackjack player represented in a JFrame grid, with each column representing the dealer’s revealed card and each row representing each possible hand a player can have. This specific player lost €5017 over 100,000 games while betting €2 each game, meaning it lost roughly €0.05 per game. Not the red “stay” panels in the top-left, which is making the player stay on hands with values four, five, six and seven.

Optimizing The Fitness Function

As shown in the picture above, the program in its current form had some problems. Runs without a very large number of games per player would often produce players that have bad genes, represented by the player choosing to stay on hands from 4 to 7, which is generally a very poor play since it leaves the player with very little chances of winning and they have nothing to lose by hitting. The sheer lack of probability of a player getting certain hands meant that these situations weren’t actively being taken into account.

After a long amount of testing, two measures were taken to fix this issue. Firstly, player hands from four to seven shared the same gene, meaning the probability of these genomes being tested was greatly increased. Secondly, the fitness function was altered to take into account how far away the player was from 21 every game. The idea was to punish a player for staying when they should have otherwise hit, and a tally was maintained of the distance from 21 after every game a player played. This was then multiplied by 0.01 and subtracted from the final funds to create the final fitness value.

The results were immediate, and it was found that better players were generated much faster throughout the early generations as a result. Luck became less of a factor as even if the dealer went bust and a player technically “won” for staying on a low value, they were rightfully punished for doing so.

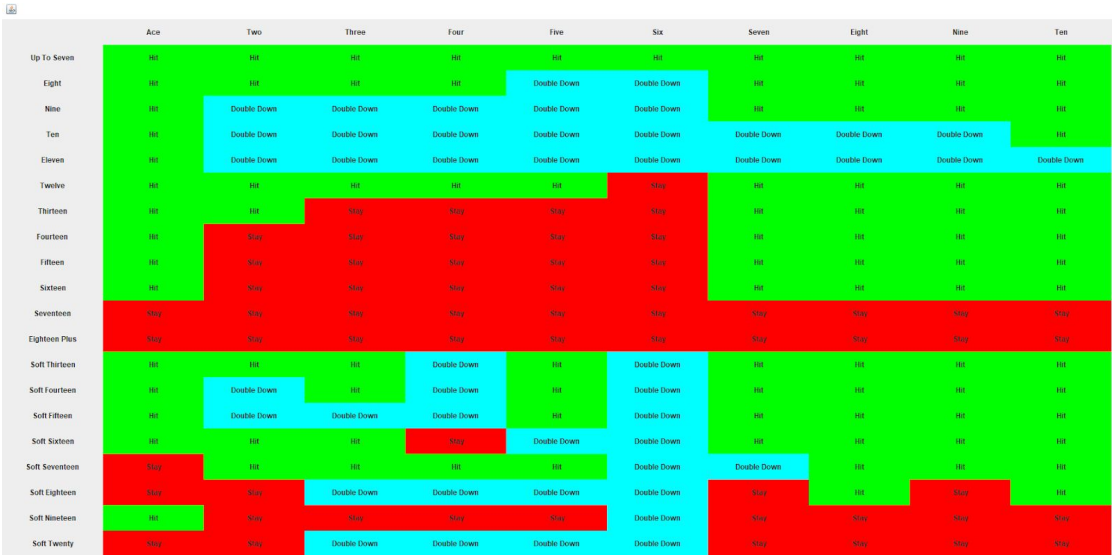


An example of the fittest blackjack player generated after 500 generations with the new changes to the fitness function and genome. This player lost €2826 over 100,000 games, meaning it lost roughly €0.03 per game. The presence of genes causing the player to stay on low values has been completely eliminated.

Implementing The Double Down

The next objective was implementing the decision of doubling down into the system. This wasn't overly complicated and slotted nicely into the existing system. When a player chose to double down, they would be given one more card, a multiplier would be set to 2 to show that their bet has been doubled, and they would not be allowed to make any further decisions.

The addition of doubling down did not have a huge impact on the system's performance generating players. A clear pattern formed of when to double down and when not to after generating a variety of players.



	Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Up To Seven	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit
Eight	Hit	Hit	Hit	Hit	Double Down	Double Down	Hit	Hit	Hit	Hit
Nine	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit
Ten	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit
Eleven	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down
Twelve	Hit	Hit	Hit	Hit	Hit	Stay	Hit	Hit	Hit	Hit
Thirteen	Hit	Hit	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fourteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fifteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Sixteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Seventeen	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Eighteen Plus	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Soft Thirteen	Hit	Hit	Hit	Double Down	Hit	Double Down	Hit	Hit	Hit	Hit
Soft Fourteen	Hit	Double Down	Hit	Double Down	Hit	Double Down	Hit	Hit	Hit	Hit
Soft Fifteen	Hit	Double Down	Double Down	Double Down	Hit	Double Down	Hit	Hit	Hit	Hit
Soft Sixteen	Hit	Hit	Hit	Stay	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Seventeen	Stay	Hit	Hit	Hit	Hit	Double Down	Double Down	Hit	Hit	Hit
Soft Eighteen	Stay	Stay	Double Down	Double Down	Double Down	Double Down	Stay	Hit	Stay	Hit
Soft Nineteen	Hit	Stay	Stay	Stay	Stay	Double Down	Stay	Stay	Stay	Stay
Soft Twenty	Stay	Stay	Double Down	Double Down	Double Down	Double Down	Stay	Stay	Stay	Stay

An example of the fittest blackjack player generated after 500 generations with the new decision of doubling down. This player actually made a profit of €1080 after 100,000 games, meaning it made roughly €0.01 per game, however there is still the element of luck to take into account.

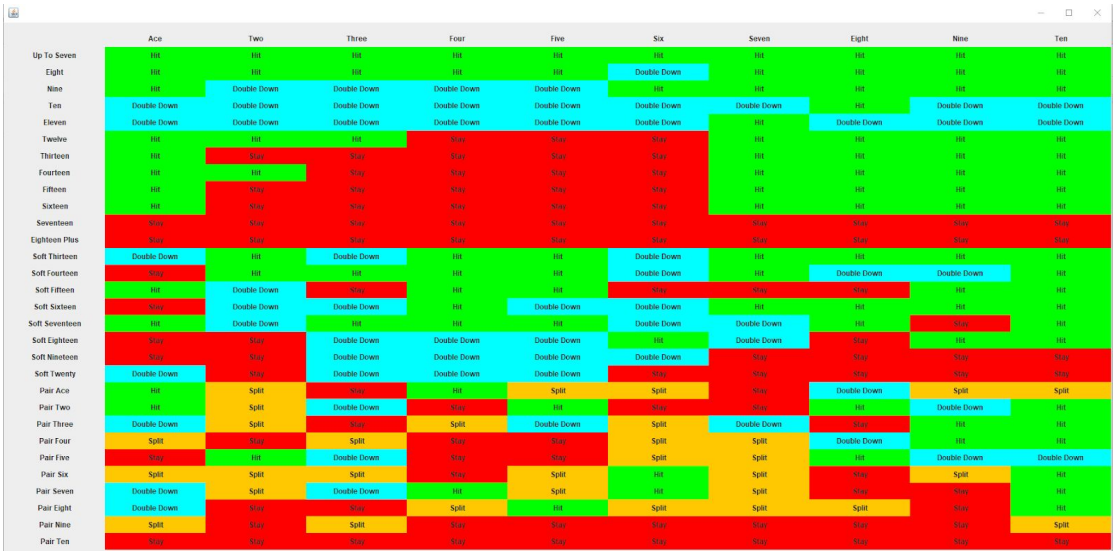
Implementing Splitting

The final player decision to be implemented was the split. This challenged how the system was built up to this point: only certain hands can be split and not others. There needed to be a way to ensure that the genome was “legal” in the sense that a player couldn’t decide to split a non-splittable hand. To do this, a BlackjackEvaluator class was created so ECJ can check the genome, see if any split values are in illegal genes and remove them and replace them with randomly-assigned value. This class was declared in the ECJ parameters file so ECJ would know to run it.

The challenge that followed was the idea of a player maintaining multiple hands at once. The entire system needed to be overhauled to accommodate this. The BlackjackPlayer was designed to hold an ArrayList of Hand objects, and all methods that handled a player’s hand now took the hand number as a parameter. Recursion was implemented so the player can make decisions for different hands while still being in the same game. The ‘final’ hand would let the dealer draw their cards, while any other games will take into account what the dealer has already drawn. A player can only split a hand four times, so a check had to be implemented to ensure they do not exceed this number of hands. If they had reached the limit, they would have to make a decision based on the value of the cards and not whether they’re capable of splitting or not. Furthermore, a player cannot decide on a hand of split aces, meaning special code had to be written for this scenario.

The system’s performance in the early generations took a hit as a result of implementing splitting. In a run of 500 generations, individuals from early generations were found to split as high as 7% of all games, which eventually decreased to 0.7% after roughly 100 generations. This number slowly increased back up to 1.2% after 200 generations and was around 1.5% after 400 generations. The system basically needed runs with more generations

due to the increased genome size and the need to ‘confirm’ which hands are worth splitting and which aren’t. A period of experimentation occurred to figure out the optimal way to generate an ideal individual in a shorter period of time, including changing the rate of genes being mutated when a new generation is being populated.



	Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Up To Seven	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit
Eight	Hit	Hit	Hit	Hit	Hit	Double Down	Hit	Hit	Hit	Hit
Nine	Hit	Double Down	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit	Hit
Ten	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit	Double Down	Double Down
Eleven	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit	Double Down	Double Down	Double Down
Twelve	Hit	Hit	Hit	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Thirteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fourteen	Hit	Hit	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fifteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Sixteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Seventeen	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Eighteen Plus	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Soft Thirteen	Double Down	Hit	Double Down	Hit	Hit	Double Down	Hit	Hit	Hit	Hit
Soft Fourteen	Stay	Hit	Hit	Hit	Hit	Double Down	Hit	Double Down	Double Down	Hit
Soft Fifteen	Hit	Double Down	Stay	Hit	Hit	Stay	Stay	Stay	Hit	Hit
Soft Sixteen	Stay	Double Down	Double Down	Hit	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Seventeen	Hit	Double Down	Hit	Hit	Hit	Double Down	Double Down	Hit	Stay	Hit
Soft Eighteen	Stay	Stay	Double Down	Double Down	Double Down	Hit	Double Down	Stay	Hit	Hit
Soft Nineteen	Stay	Stay	Double Down	Double Down	Double Down	Double Down	Stay	Stay	Stay	Stay
Soft Twenty	Double Down	Stay	Double Down	Double Down	Double Down	Stay	Stay	Stay	Stay	Stay
Pair Ace	Hit	Split	Stay	Hit	Split	Split	Stay	Double Down	Split	Split
Pair Two	Hit	Split	Double Down	Stay	Hit	Stay	Stay	Hit	Double Down	Hit
Pair Three	Double Down	Split	Stay	Split	Double Down	Split	Double Down	Stay	Hit	Hit
Pair Four	Split	Stay	Split	Stay	Stay	Split	Split	Double Down	Hit	Hit
Pair Five	Stay	Hit	Double Down	Stay	Stay	Split	Split	Hit	Double Down	Double Down
Pair Six	Split	Split	Split	Stay	Split	Hit	Split	Stay	Split	Hit
Pair Seven	Double Down	Split	Double Down	Hit	Split	Hit	Split	Stay	Stay	Hit
Pair Eight	Double Down	Stay	Stay	Split	Hit	Split	Split	Split	Stay	Hit
Pair Nine	Split	Stay	Split	Stay	Stay	Stay	Stay	Stay	Stay	Split
Pair Ten	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay

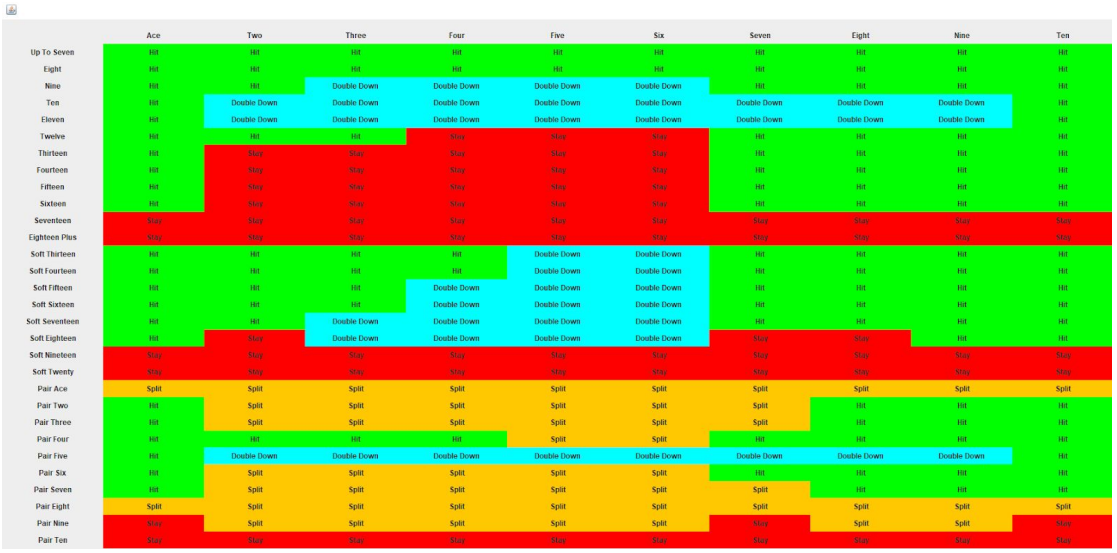
An example of the fittest blackjack player generated after 500 generations with the new decision of splitting. When compared to earlier genomes, there is less confirmation in some areas due to the added amount of possible decisions. A larger amount of generations is required for more accurate results. This player lost €1516 after 100,000 games.

JUnit Testing

With all four decisions implemented, it was time to ensure that the system was bug-free. The JUnit framework was added to the project to add an integration test that takes a hard-coded genome, assigns it to an IntegerVectorIndividual and passes it to the BlackjackProblem class to get it

to play games, form a fitness and log the results.

An optimal blackjack strategy was found (David Labowsky, 2016) and it was converted into an integer array. This was then put to the test in a number of games to see how it performed. This helped with providing a normal for what the ideal individual should look like, along with discovering small bugs that were born due to how splitting was implemented.



	Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten
Up To Seven	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit
Eight	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit	Hit
Nine	Hit	Hit	Double Down	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit
Ten	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit
Eleven	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit
Twelve	Hit	Hit	Hit	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Thirteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fourteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Fifteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Sixteen	Hit	Stay	Stay	Stay	Stay	Stay	Hit	Hit	Hit	Hit
Seventeen	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Eighteen Plus	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Soft Thirteen	Hit	Hit	Hit	Hit	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Fourteen	Hit	Hit	Hit	Hit	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Fifteen	Hit	Hit	Hit	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Sixteen	Hit	Hit	Hit	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Seventeen	Hit	Hit	Double Down	Double Down	Double Down	Double Down	Hit	Hit	Hit	Hit
Soft Eighteen	Hit	Stay	Double Down	Double Down	Double Down	Double Down	Stay	Stay	Hit	Hit
Soft Nineteen	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Soft Twenty	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay
Pair Ace	Split	Split	Split	Split	Split	Split	Split	Split	Split	Split
Pair Two	Hit	Split	Split	Split	Split	Split	Split	Hit	Hit	Hit
Pair Three	Hit	Split	Split	Split	Split	Split	Split	Hit	Hit	Hit
Pair Four	Hit	Hit	Hit	Hit	Split	Split	Hit	Hit	Hit	Hit
Pair Five	Hit	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Double Down	Hit
Pair Six	Hit	Split	Split	Split	Split	Split	Hit	Hit	Hit	Hit
Pair Seven	Hit	Split	Split	Split	Split	Split	Split	Hit	Hit	Hit
Pair Eight	Split	Split	Split	Split	Split	Split	Split	Split	Split	Split
Pair Nine	Stay	Split	Split	Split	Split	Split	Stay	Split	Split	Stay
Pair Ten	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay	Stay

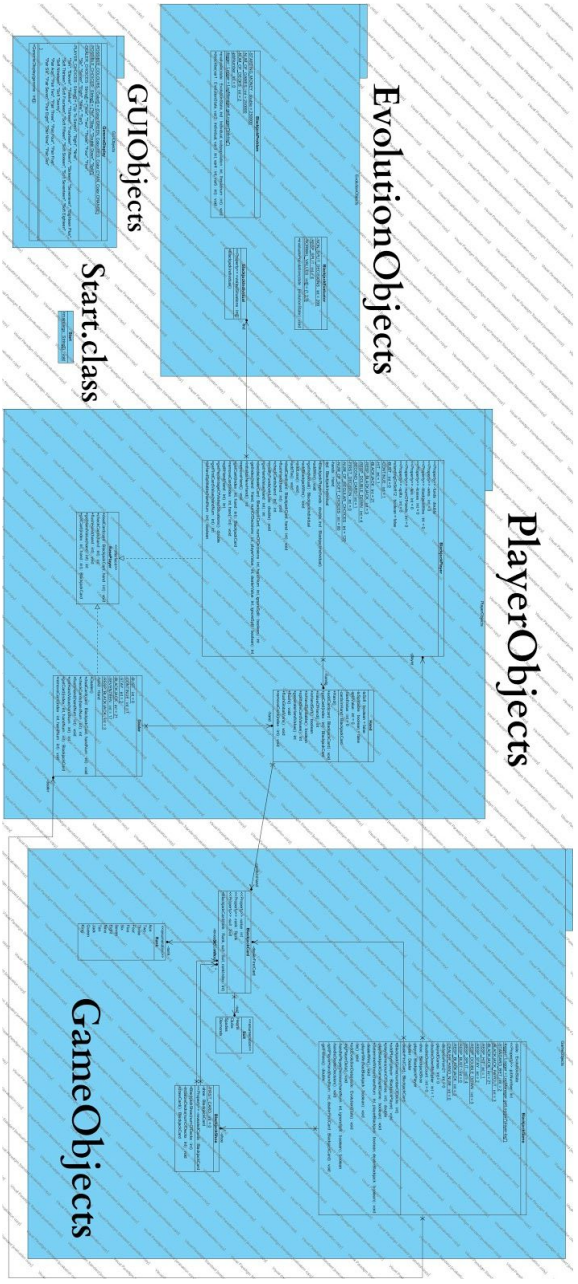
The optimal blackjack strategy represented in the JFrame. This strategy makes a profit in roughly 1 in 8 sets of 100,000 games, meaning the house edge still prevails even with an ideal individual.

Introduction of Product

The final product is a complete package of my experiences learning how research and evolve blackjack players.

ECJ & Blackjack System

The system is split into four packages: EvolutionObjects (classes that ECJ calls directly or classes that extend ECJ classes), GameObjects (classes that directly involve the game of blackjack), PlayerObjects (classes that involve the player, dealer and how they handle the game of blackjack) and GUIObjects (classes that represent aspects of the system in a graphical manner).



A UML diagram of the blackjack system.

To run the program, Start.class is run, which evokes ECJ's "Evolve.main" method. It also supplies the location of the parameters file for ECJ to use. ECJ will then start by creating the Population, a Subpopulation within it and fill it with the first generation of Individuals.

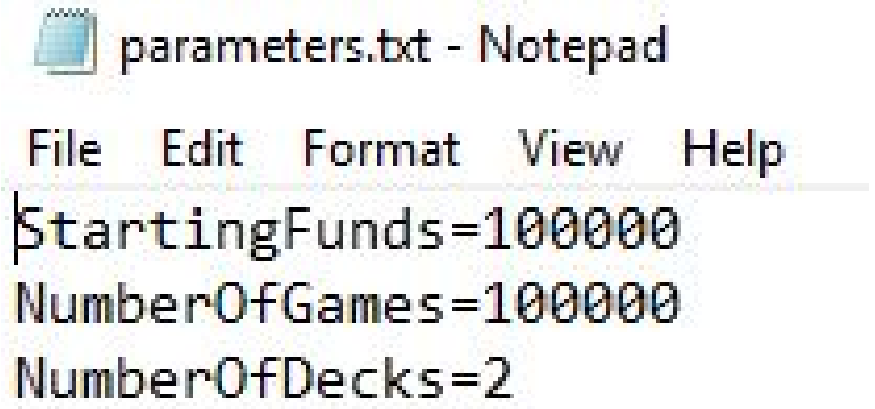
The parameters file outlines all the desired classes for the system and what values should be used for specific parts of the job. Most of these are built into the ECJ framework, however some were created by myself to add extra features. For example, the "eval" class used is the BlackjackEvaluator class, which checks each individual in the population to see if the genome is 'legal' ie. players can only make the "split" decision (the value of 4) in the area of the genome that is dedicated to splittable hands. Also outlined here is the number of generations, which can be changed between runs depending on how much time the job should last. The length of the genome is declared, along with the possible values that can show in each gene (in this case, 300, and with values 1 to 4). The BlackjackIndividual extends IntegerVectorIndividual except it has code allowing for experimentation of keeping track of how many times each gene was accessed.

The chosen pipeline for selecting the best individuals from a generation is the VectorMutationPipeline. This takes individuals from two sources, mutates them together by selecting genes at random from each parent and uses these new individuals as a base for the next generation. One individual is the fittest individual from the generation (FitProportionateSelection) and the other is the best individual from a random sample from the generation (TournamentSelection). When creating new individuals, each gene has a 1% chance of resetting into another value. The problem class is defined, and ECJ will run it providing an individual to be evaluated and assigned a fitness.

```
parent.0        = simple.params

breedthreads = 1
evalthreads = 1

pop.subpop.0.size = 50
generations = 800

eval = main.java.EvolutionObjects.BlackjackEvaluator

pop.subpop.0.species        = ec.vector.IntegerVectorSpecies
pop.subpop.0.species.ind    = main.java.EvolutionObjects.BlackjackIndividual
pop.subpop.0.species.fitness    = ec.simple.SimpleFitness
pop.subpop.0.species.min-gene   = 1
pop.subpop.0.species.max-gene   = 4
pop.subpop.0.species.genome-size    = 320
pop.subpop.0.species.crossover-type = two
pop.subpop.0.species.mutation-type  = reset
pop.subpop.0.species.mutation-prob  = 0.01

pop.subpop.0.species.pipe           = ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0      = ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0 = ec.select.FitProportionateSelection
pop.subpop.0.species.pipe.source.0.source.1 = ec.select.TournamentSelection
select.tournament.size = 25

eval.problem            = main.java.EvolutionObjects.BlackjackProblem
```

The "blackjack.params" ECJ parameters file. In this case, the number of generations is set to 800, each generation will have 50 BlackjackIndividuals and the random sample of the TournamentSelection will be 25.

The BlackjackProblem class extends the Problem class and implements the SimpleProblem interface, and it is the class that sets up a number of games and has an individual assigned to a BlackjackPlayer, an implementation of the BasePlayer interface. The BlackjackPlayer class contains the methods for making decisions based on the individual's genome along with objects that a player would usually possess in a game of blackjack, such as Hand objects stored in an ArrayList and tallies of wins, losses, ties etc. BlackjackProblem gets its values for the number of games, the starting funds and the amount of decks to use in a run from a parameters file located in the C drive.



The parameters file used to define the amount of funds every player starts with, the number of games and how many decks will be used in the game's shoe.

The BlackjackGame enters a loop of running games until the limit has been reached. Each game is the process of the following:

1. The dealer and player are dealt two cards from the randomized shoe.
2. Both hands are checked for blackjack (a 10 and an ace). If one player has it, immediately go to step 5.
3. Get the player's decision for each hand until they choose to stay or double down. If they split, resolve all split games before moving to the next step.
4. The dealer hits until their cards value 17 or more. If their cards are already of value 17 or higher, skip this step.
5. Determine the winner based on each player's card values.
6. Flush both players' hands.

The game uses BlackjackCards, which are assigned a value, rank and suit when initialized by the BlackjackShoe. The BlackjackShoe is told how many decks to generate and randomizes them all into one large shoe, which is reshuffled when it runs out.

The Dealer object also implements the BasePlayer interface and has altered methods to take into account the fact that it cannot hit past 17, along with only possessing a single Hand. The BlackjackPlayer instead has an ArrayList of Hand objects due to the possibility of splitting. The Hand object has methods for placing a BlackjackCard in the hand and calculating the value of the hand, which also determines if the hand is soft (contains an ace counting as an 11) or splittable (has two cards of the same value). Once a split is over, all extra Hand objects are removed, leaving only the first Hand.

The game calls the BlackjackPlayer for making decisions, who in turn figures out the location of the gene in the genome that relates to the given scenario based on the player's hand and the dealer's revealed card, then finds the integer value stored within. This value is returned to the BlackjackGame, which parses it and handles its decision in the game.

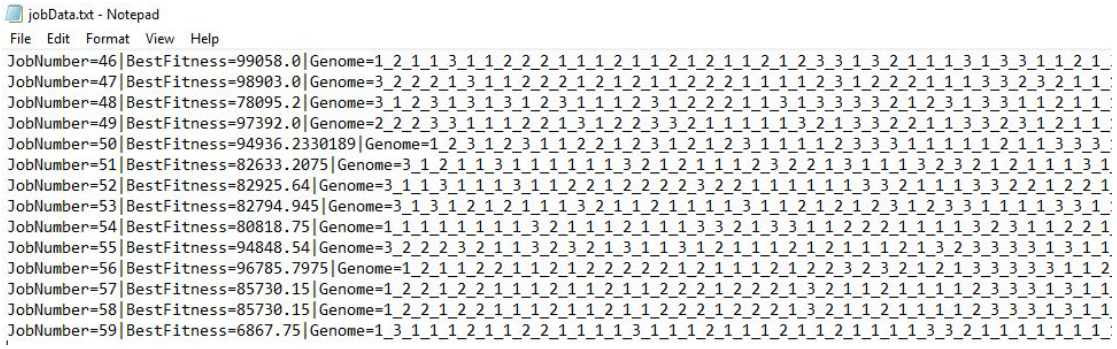
Once the loop of games is complete, Log4J is used to log the player's data to the "logs.txt" file located in the C drive. The fitness is then created with the following function:

Funds - (Funds * (Distance From 21 / Number Of Games) * 0.01)

"Distance From 21" is a tally of how far away the player was from 21 after each game. If a player was 10 or more away from 21 (meaning they stayed on an 11 or smaller, which is pointless since you don't lose anything by hitting), the distance for that game is multiplied by 3.

The system keeps handling new individuals until the limit of generations has been reached. ECJ keeps track of the fittest individual throughout the entire run, and their genome is represented in a JFrame to be visualized once the run is over as ECJ calls BlackjackProblem's "describe" method. The genome is

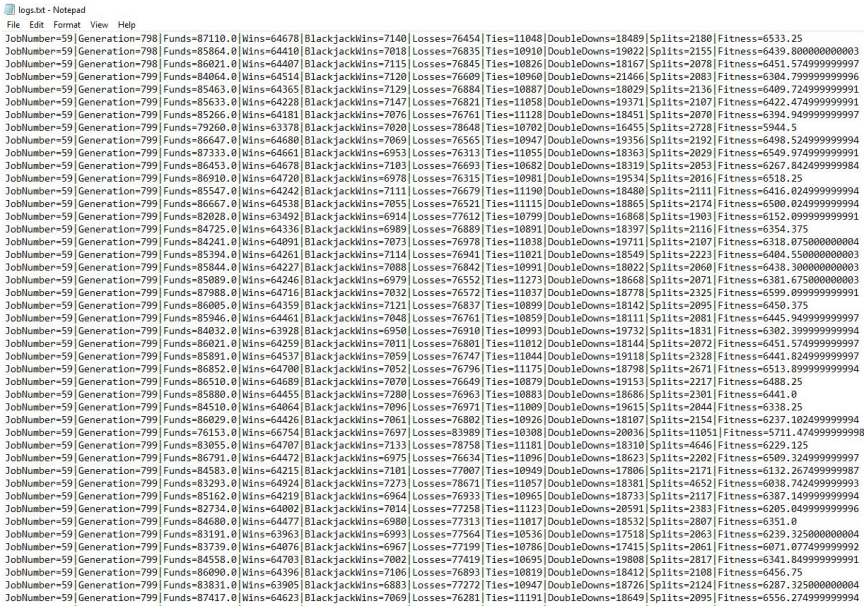
also logged using Log4J in the “jobData.txt” file along with data such as the job number and the individual’s fitness.



The “jobData.txt” log file, showing the genome of the best individual in each run along with their fitness.

Elasticsearch-Logstash-Kibana (ELK)

Logstash uses in a configuration file which contains the location of the file it should read (in this case, we want to read the “logs.txt” file since it contains each player’s data), the headers and values it should extract and where it should output this data to. I created an Elasticsearch cluster through Amazon Web Services (AWS) and obtained all the necessary credentials for Logstash to contact the cluster and send data to it. I also created a sample log file containing player data that Logstash can read.



logs.txt - Notepad

File Edit Format View Help

```
JobNumber=59|Generation=798|Funds=87110.0|Wins=64678|BlackjackWins=7140|Losses=76454|Ties=11048|DoubleDowns=18489|Splits=2180|Fitness=6533.25
JobNumber=59|Generation=798|Funds=85864.0|Wins=64410|BlackjackWins=7018|Losses=76835|Ties=10910|DoubleDowns=19022|Splits=2155|Fitness=6439.800000000003
JobNumber=59|Generation=798|Funds=86021.0|Wins=64407|BlackjackWins=7115|Losses=76845|Ties=10826|DoubleDowns=18167|Splits=2078|Fitness=6451.574999999997
JobNumber=59|Generation=799|Funds=84064.0|Wins=64514|BlackjackWins=7120|Losses=76609|Ties=10960|DoubleDowns=21466|Splits=2083|Fitness=6304.799999999996
JobNumber=59|Generation=799|Funds=85463.0|Wins=64365|BlackjackWins=7129|Losses=76884|Ties=10887|DoubleDowns=18029|Splits=2136|Fitness=6409.724999999991
JobNumber=59|Generation=799|Funds=85633.0|Wins=64228|BlackjackWins=7147|Losses=76821|Ties=11058|DoubleDowns=19371|Splits=2107|Fitness=6422.474999999991
JobNumber=59|Generation=799|Funds=85266.0|Wins=64181|BlackjackWins=7076|Losses=76761|Ties=11128|DoubleDowns=18451|Splits=2070|Fitness=6394.949999999997
JobNumber=59|Generation=799|Funds=79260.0|Wins=63378|BlackjackWins=7020|Losses=78648|Ties=10702|DoubleDowns=16455|Splits=2728|Fitness=5944.5
JobNumber=59|Generation=799|Funds=86647.0|Wins=64680|BlackjackWins=7069|Losses=76565|Ties=10947|DoubleDowns=19356|Splits=2192|Fitness=6498.524999999994
JobNumber=59|Generation=799|Funds=87333.0|Wins=64661|BlackjackWins=6953|Losses=76313|Ties=11055|DoubleDowns=18363|Splits=2029|Fitness=6549.974999999991
JobNumber=59|Generation=799|Funds=86453.0|Wins=64678|BlackjackWins=7103|Losses=76693|Ties=10682|DoubleDowns=18319|Splits=2053|Fitness=6267.842499999984
JobNumber=59|Generation=799|Funds=86910.0|Wins=64720|BlackjackWins=6978|Losses=76315|Ties=10981|DoubleDowns=19534|Splits=2016|Fitness=6518.25
JobNumber=59|Generation=799|Funds=85547.0|Wins=64242|BlackjackWins=7111|Losses=76679|Ties=11190|DoubleDowns=18480|Splits=2111|Fitness=6416.024999999994
JobNumber=59|Generation=799|Funds=86667.0|Wins=64538|BlackjackWins=7055|Losses=76521|Ties=11115|DoubleDowns=18865|Splits=2174|Fitness=6500.024999999994
JobNumber=59|Generation=799|Funds=82028.0|Wins=63492|BlackjackWins=6914|Losses=77612|Ties=10799|DoubleDowns=16868|Splits=1903|Fitness=6152.099999999991
JobNumber=59|Generation=799|Funds=84725.0|Wins=64336|BlackjackWins=6989|Losses=76889|Ties=10891|DoubleDowns=18397|Splits=2116|Fitness=6354.375
JobNumber=59|Generation=799|Funds=84241.0|Wins=64091|BlackjackWins=7073|Losses=76978|Ties=11038|DoubleDowns=19711|Splits=2107|Fitness=6318.075000000004
JobNumber=59|Generation=799|Funds=85394.0|Wins=64261|BlackjackWins=7114|Losses=76941|Ties=11021|DoubleDowns=18549|Splits=2223|Fitness=6404.550000000003
JobNumber=59|Generation=799|Funds=85844.0|Wins=64227|BlackjackWins=7088|Losses=76842|Ties=10991|DoubleDowns=18022|Splits=2060|Fitness=6438.300000000003
JobNumber=59|Generation=799|Funds=85089.0|Wins=64246|BlackjackWins=6979|Losses=76552|Ties=11273|DoubleDowns=18668|Splits=2071|Fitness=6381.675000000003
JobNumber=59|Generation=799|Funds=87988.0|Wins=64716|BlackjackWins=7032|Losses=76572|Ties=11037|DoubleDowns=18778|Splits=2325|Fitness=6599.099999999991
JobNumber=59|Generation=799|Funds=86005.0|Wins=64359|BlackjackWins=7121|Losses=76837|Ties=10899|DoubleDowns=18142|Splits=2095|Fitness=6450.375
JobNumber=59|Generation=799|Funds=85946.0|Wins=64461|BlackjackWins=7048|Losses=76761|Ties=10859|DoubleDowns=18111|Splits=2081|Fitness=6445.949999999997
JobNumber=59|Generation=799|Funds=84032.0|Wins=63928|BlackjackWins=6950|Losses=76910|Ties=10993|DoubleDowns=19732|Splits=1831|Fitness=6302.399999999994
JobNumber=59|Generation=799|Funds=86021.0|Wins=64259|BlackjackWins=7011|Losses=76801|Ties=11012|DoubleDowns=18144|Splits=2072|Fitness=6451.574999999997
JobNumber=59|Generation=799|Funds=85891.0|Wins=64537|BlackjackWins=7059|Losses=76747|Ties=11044|DoubleDowns=19118|Splits=2328|Fitness=6441.824999999997
JobNumber=59|Generation=799|Funds=86852.0|Wins=64700|BlackjackWins=7052|Losses=76796|Ties=11175|DoubleDowns=18798|Splits=2671|Fitness=6513.899999999994
JobNumber=59|Generation=799|Funds=86510.0|Wins=64689|BlackjackWins=7070|Losses=76649|Ties=10879|DoubleDowns=19153|Splits=2217|Fitness=6488.25
JobNumber=59|Generation=799|Funds=85880.0|Wins=64455|BlackjackWins=7280|Losses=76963|Ties=10883|DoubleDowns=18686|Splits=2301|Fitness=6441.0
JobNumber=59|Generation=799|Funds=84510.0|Wins=64064|BlackjackWins=7096|Losses=76971|Ties=11009|DoubleDowns=19615|Splits=2044|Fitness=6338.25
JobNumber=59|Generation=799|Funds=86029.0|Wins=64426|BlackjackWins=7061|Losses=76802|Ties=10926|DoubleDowns=18107|Splits=2154|Fitness=6237.102499999994
JobNumber=59|Generation=799|Funds=76153.0|Wins=66754|BlackjackWins=7697|Losses=83989|Ties=10308|DoubleDowns=20036|Splits=11051|Fitness=5711.474999999998
JobNumber=59|Generation=799|Funds=83055.0|Wins=64707|BlackjackWins=7133|Losses=78758|Ties=11181|DoubleDowns=18310|Splits=4646|Fitness=6229.125
JobNumber=59|Generation=799|Funds=86791.0|Wins=64472|BlackjackWins=6975|Losses=76634|Ties=11096|DoubleDowns=18623|Splits=2202|Fitness=6509.324999999997
JobNumber=59|Generation=799|Funds=84583.0|Wins=64215|BlackjackWins=7101|Losses=77007|Ties=10949|DoubleDowns=17806|Splits=2171|Fitness=6132.267499999987
JobNumber=59|Generation=799|Funds=83293.0|Wins=64924|BlackjackWins=7273|Losses=78671|Ties=11057|DoubleDowns=18381|Splits=4652|Fitness=6038.742499999993
JobNumber=59|Generation=799|Funds=85162.0|Wins=64219|BlackjackWins=6964|Losses=76933|Ties=10965|DoubleDowns=18733|Splits=2117|Fitness=6387.149999999994
JobNumber=59|Generation=799|Funds=82734.0|Wins=64002|BlackjackWins=7014|Losses=77258|Ties=11123|DoubleDowns=20591|Splits=2383|Fitness=6205.049999999996
JobNumber=59|Generation=799|Funds=84680.0|Wins=64477|BlackjackWins=6980|Losses=77313|Ties=11017|DoubleDowns=18532|Splits=2807|Fitness=6351.0
JobNumber=59|Generation=799|Funds=83191.0|Wins=63963|BlackjackWins=6993|Losses=77564|Ties=10536|DoubleDowns=17518|Splits=2063|Fitness=6239.325000000004
JobNumber=59|Generation=799|Funds=83739.0|Wins=64076|BlackjackWins=6967|Losses=77199|Ties=10786|DoubleDowns=17415|Splits=2061|Fitness=6071.077499999992
JobNumber=59|Generation=799|Funds=84558.0|Wins=64703|BlackjackWins=7002|Losses=77419|Ties=10695|DoubleDowns=19808|Splits=2817|Fitness=6341.849999999991
JobNumber=59|Generation=799|Funds=86090.0|Wins=64396|BlackjackWins=7106|Losses=76893|Ties=10819|DoubleDowns=18412|Splits=2108|Fitness=6456.75
JobNumber=59|Generation=799|Funds=83831.0|Wins=63905|BlackjackWins=6883|Losses=77272|Ties=10947|DoubleDowns=18726|Splits=2124|Fitness=6287.325000000004
JobNumber=59|Generation=799|Funds=87417.0|Wins=64623|BlackjackWins=7069|Losses=76281|Ties=11191|DoubleDowns=18649|Splits=2095|Fitness=6556.274999999994
```

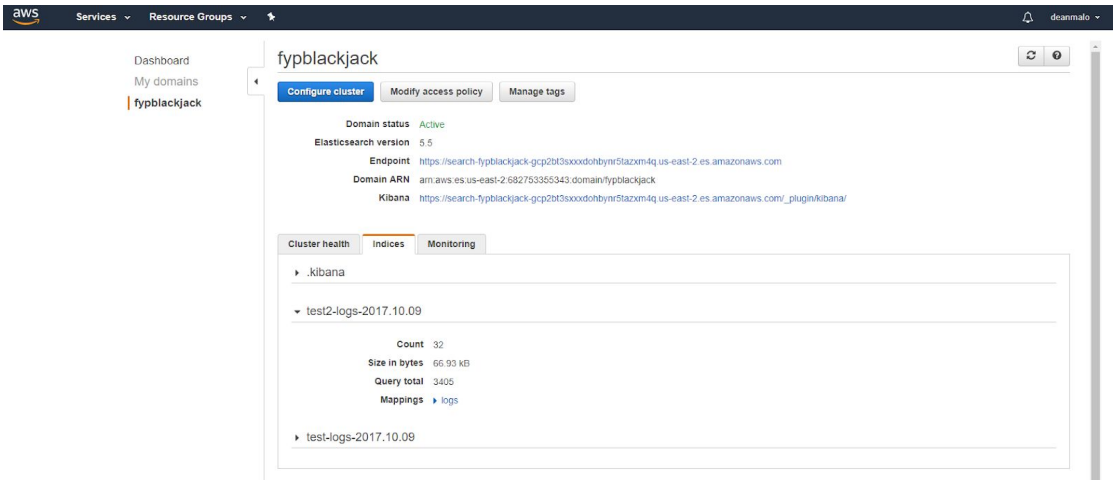
The “logs.txt” log file which Logstash parses. This contains data on each player, which job and generation they belonged in and how many wins/losses/ties etc. they had.

logstash.conf - Notepad

File Edit Format View Help

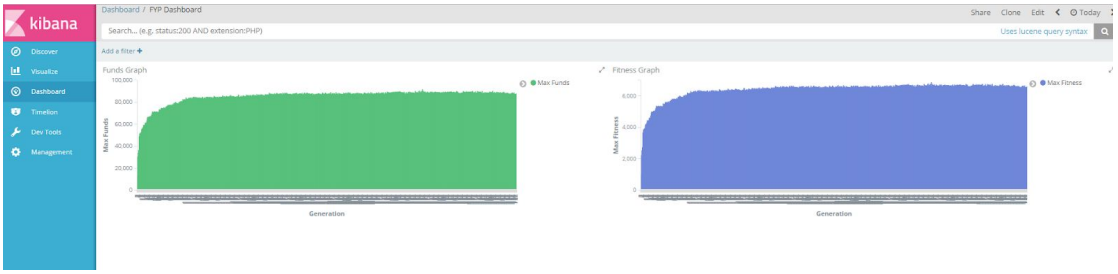
```
input {
    file {
        path => [ "C:/Users/Dean/Documents/logstash-5.6.2/logs.txt" ]
      }
    }

filter {
    kv {
        field_split => "|"
       }
    mutate {
        convert => {
            "JobNumber" => "integer"
            "Generation" => "integer"
            "Funds"  => "integer"
            "Wins" => "integer"
            "Losses" => "integer"
            "Ties" => "integer"
            "DoubleDowns" => "integer"
            "Splits" => "integer"
            "Fitness" => "float"
            }
       }
    }

output {
    amazon_es {
        hosts => ["search-fypblackjack-gcp2bt3sxxxdohbynr5tazxm4q.us-east-2.es.amazonaws.com"]
        aws_access_key_id =>
        aws_secret_access_key =>
        region => "us-east-2"
        index => "test2-logs-%{+YYYY.MM.dd}"
        }
    }
```

The Logstash configuration file. The Input plugin feeds the log to Logstash, the Filter plugin splits the data into headers and values of specific types, and the Output plugin sends the data to the Amazon Web Services Elasticsearch cluster.



The Elasticsearch cluster configuration interface in the Amazon Web Services console. The Kibana dashboard can be accessed from here.

A dashboard was created in Kibana that represents each generation by its average fitness or funds. This shows the improvement over time as the fittest players from the previous generation were used to create the next.



A Kibana dashboard which represents the data in the logs in a graph to show the increase in fitness/funds between generations.

Results & Thoughts

I feel that the product has highlighted the hidden complexity of the game of blackjack and shown that, when given options such as doubling down and splitting, it is difficult to determine an optimal strategy based on trial and

error and survival of the fittest. Implementing splitting was definitely the biggest challenge of the project, as it challenged how I had structured the product up to that point and also required me to optimize how players were being generated and mutated. While it's fair to say that genetic evolution isn't the most optimal way of devising an optimal blackjack strategy based on some of my results, I still feel that it was an interesting experiment, and it helped represent the individuals in a manner I would be familiar with ie. players in a game making simple decisions. A more realistic solution might have been repeating the same situation over and over but with different player decisions in order to rely less on luck.

The initial plan was to have an iteration of the product that implemented some form of grammatical evolution, however this unfortunately never came into fruition. When it came to representing the different decisions for a given number of scenarios, it seemed like using integers was a much cleaner solution than using an approach of function trees. Discussions were had regarding how grammatical evolution could be implemented into the system, such as counting cards or changing the player's bet depending on how much funds the player has left, but due to the amount of time spent optimizing splitting, there wasn't enough time to implement these ideas.

The idea of counting cards is of particular interest since it does increase a player's odds of overcoming the house edge, to the point that most casinos observe and remove anyone caught to be doing it despite it being completely legal. If the project had more time, I would have liked to focus on this and how the player can consciously take into account what cards have been revealed up to now when making a decision.

I feel that I have learned a lot about the process of researching genetic evolution as a result of this project and after building the product. I had very little experience with the topic before initiating the project, and I feel that I've come a long way in terms of understanding basic concepts. I understood

the purpose of optimization and how a more focused fitness function or parameters file can really benefit the performance of the system. I also had the opportunity to do some analytical research by looking at graphs generated from the system's log files and observing how the players improved throughout generations.

Despite the product not being able to implement grammatical evolution, I still feel like I've learned a lot about the subject as a result of being supervised by one of its innovators. The topic is still of interest to me, and I may research it in my own time to see how it can be expanded to other solutions. The aforementioned use of grammatical evolution to create an optimal player of Mrs. Pac Man was of particular interest, and I'll be sure to keep an eye on advancements in the future.

References

Conor Ryan, Michael O'Neill, 2003 'Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language'

Brian Benchoff, 2013 'Evolving a Better Blackjack Strategy' (http://graphsandwords.com/projects/evolving-a-better-blackjack-strategy)

George Mason University Department of Computer Science, 'Tutorial 1: Build a Genetic Algorithm for the MaxOnes Problem' (https://cs.gmu.edu/~eclab/projects/ecj/docs/tutorials/tutorial1)

Robert Hackett, June 12th 2014, 'Meet The Father Of Digital Life' (http://nautil.us/issue/14/mutation/meet-the-father-of-digital-life)

Wikipedia contributors, April 15th 2018, 'Genetic programming' (https://en.wikipedia.org/w/index.php?title=Genetic_programming&oldid=836637809)

Ken Smith, January 30th 2018, 'Why Does The Number Of Decks Matter In Blackjack?' (https://www.blackjackinfo.com/why-does-the-number-of-decks-matter-in-blackjack/)

Peter Adam, July 4th 2017, ‘Introduction to PonyGE2 for Grammatical Evolution’ (https://towardsdatascience.com/introduction-to-ponyge2-for-grammatical-evolution-d51c29f2315a)

Edgar Galvan-Lopez, John Mark Swafford, Michael O’Neill, Anthony Brabazon, 2010, ‘Evolving a Ms. PacMan Controller using Grammatical Evolution’ (http://ncra.ucd.ie/papers/evolvingMsPacmanControllerEvoGames2010.pdf)

Sean Luke, September 1st 2015, ‘The ECJ Owner’s Manual’ (https://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf)

Arun Prasath, June 25th 2016, ‘Openstack log analysis using ELK’ (http://bingoarun.github.io/openstack-log-analysis-elk.html)

David Labowsky, September 29th 2016, ‘How to play Blackjack: a quick basic tutorial’ (https://casinogrounds.com/play-blackjack-quick-basic-tutorial/)