Department of Computer Sciences
University of Salzburg

PS Natural Computation
SS 15/16

Evolution of a Poker Player Using Genetic Programming

July 14, 2016

Project Members:

Thomas Samy Dafir, 1331483, tdafir@cosy.sbg.ac.at
Laurentiu Vlad, 1423336, lvlad@cosy.sbg.ac.at
Dominik Baumgartner, 0920177, dbaumgart@cosy.sbg.ac.at
Sebastian Strumegger, 1420277, sstrumegg@cosy.sbg.ac.at

Academic Supervisor:

Helmut MAYER
helmut@cosy.sbg.ac.at

Correspondence to:

Universität Salzburg
Fachbereich Computerwissenschaften
Jakob–Haringer–Straße 2
A–5020 Salzburg
Austria

Contents

Abstract

The goal of this project is to evolve a GP poker player by means of genetic programming techniques. The GP Players evolve over generations through selection, mutation and cross mutation of their chromosome tree.
A good computer player has to emerge after a sufficient amount of generations have been played in a "heads up" texas-hold-em no limit poker game. Good players are determined by their fitness value which is determined through assessing a player's success after each poker round.
The focus of our work lies in the chromosome tree. Although the current API provides some basics functions for that tree, it is likely that they are not sufficient for the evolution of a good player. The second focus lies on the poker game itself. We have to determine what kind and how much information a player needs in order to evolve properly without interfering with its evolution. This means, that we don't want to "tell" the player what the next best action is, but rather let evolution decide. Additionally we will do some fine-tuning of evolutionary-parameters like population size or number of generations to achieve the best possible result.

1 Frameworks

This project requires the use of and interaction between two frameworks namely GPoker and JEvolution. In the following both frameworks and their interactions with one another will be briefly explained.

1.1 jEvolutin

JEvolution is the most essential part of this project. JEvolution provides functionality for Evolutionary Computation including Evolutionary Algorithms and Genetic Programming. We use the latter. JEvolution completely implements the evolution of syntax trees including selection, mutation and crossover. Syntax trees are made up from objects of type *ProgramNode*. Each node has an array containing all child nodes. In order for the evolution to work a set of functions and terminals has to be provided. These nodes are implemented as subclasses of the *ProgramNode* class. Selection is conducted based on the computed fitness of every syntax tree. For our application of the framework nodes are provided by the GPoker framework and by us. In order for JEvolution to work correctly a function calculating a tree's fitness has to be supplied (fitness function). In our case this function has to somehow compute how good a syntax tree (in our case representing a poker player) is able to complete the given task (here: play poker and win chips).

1.2 GPoker

GPoker is a poker-framework by Helmut Mayer. It provides a set of classes implementing a working poker setup including several game modes, different players and the possibility for a human player to play against computer players. The most essential functionality for our application is the possibility to include evolved players and hence use the framework to do genetic programming. GPoker provides nodes which are subclasses of the JEvolution *ProgramNode* class representing decisions, actions and queries a poker player might perform. GPoker can be parameterized through the gpoker.xml file.

1.3 Interaction

How do the two frameworks interact?
JEvolution evolves syntax trees made up from *ProgramNode* objects. It does not care about what the functionality of a node is. As long as it's specified as a subclass of *ProgramNode* it can be used in a syntax tree. Once GPoker detects that a GPlayer (evolved player) is specified in the gpoker.xml file communicates with JEvolution to invoke the evolutionary process. GPoker provides nodes for constructing a syntax tree. JEvolution then evolves syntax trees made up from these nodes. A vital component of the evolutionary process is the calculation of the fitness value. In the case of the evolution of a poker player the fitness value should of course reflect the player's skill. The most intuitive way of defining a player's skill is measuring the amount of won/lost chips. This value can be determined by letting the player play games of poker against other players and is then recorded as the player's fitness. JEvolution can then execute the selection part of the evolutionary process using the value provided by GPoker. Interactions can be summarized as follows:

- GPoker detects an evolved player in the xml file and invokes JEvolution providing poker specific nodes.
- JEvolution builds syntax trees and uses GPoker to calculate fitness values of syntax trees.

2 Genetic Programming

As mentioned, a GP Player is represented by a so-called chromosome tree. Or more exactly, the poker player's strategy is represented by this tree.
This means, that each player's tree has a root node and more or less child nodes, where nodes with children are called functions and leave nodes are called terminals. From the very beginning there is a fixed set of functions and terminals available. At first, a population of GP Players with random nodes are created which play a predetermined number of games. In each generation there is a chance for every player that it's tree is changed by mutation or crossover.

2.1 Chromosome Tree

2.1.1 Functions

Functions play the most important role in a GP Player's strategy, since functions are the decision-makers.
They can have one or more child nodes and always are of a specific return type. Further they need to have a parent node (except the root function). Child nodes can either be terminals or functions, where it is also very important that the child nodes return the very type, that the function expects.
As an example we could define a lower-than function that expects floating point values from both children and returns a boolean value to it's parent node.

2.1.2 Terminals

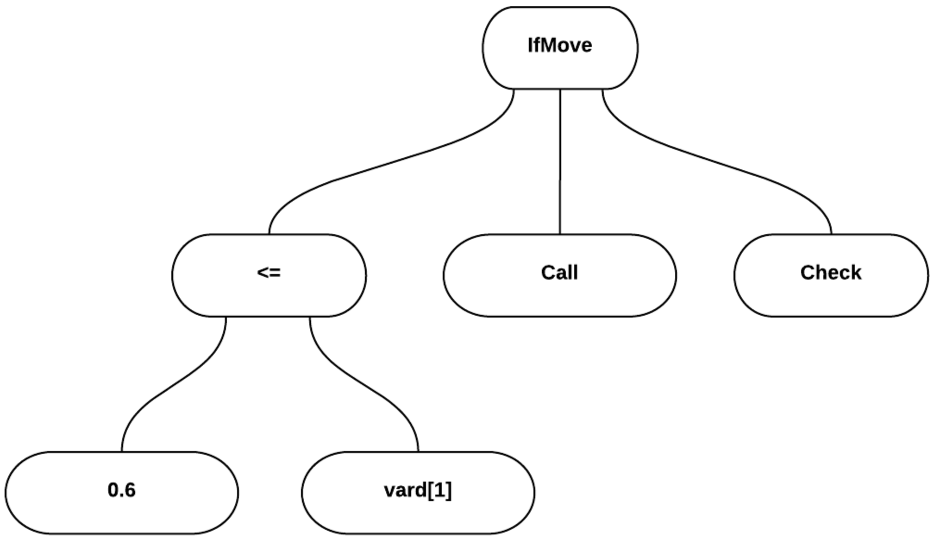
Terminals are the leave nodes in a player's tree. They cannot have child nodes but return a certain type themselves.
This does not necessarily mean that a terminal's value has to be static. They can have dynamic values like "pot size" or "higher hand card".

2.1.3 Root

The root node it a special function, since it does not have any parent nodes.
For a proper evaluation of the player's tree it needs to be rooted, so we can start making further decisions at one point. For GPoker we need to have a root node that allways returns a poker move which is returned to the game.

A GPlayer, for example, uses the root node "IfMove" which has three children:
The very left child is the if-branch, that needs to return a boolean type.
The middle child is the then-branch and returns a move.
And at last, the right child represents the else-branch, which also returns a move.

2.1.4 Example



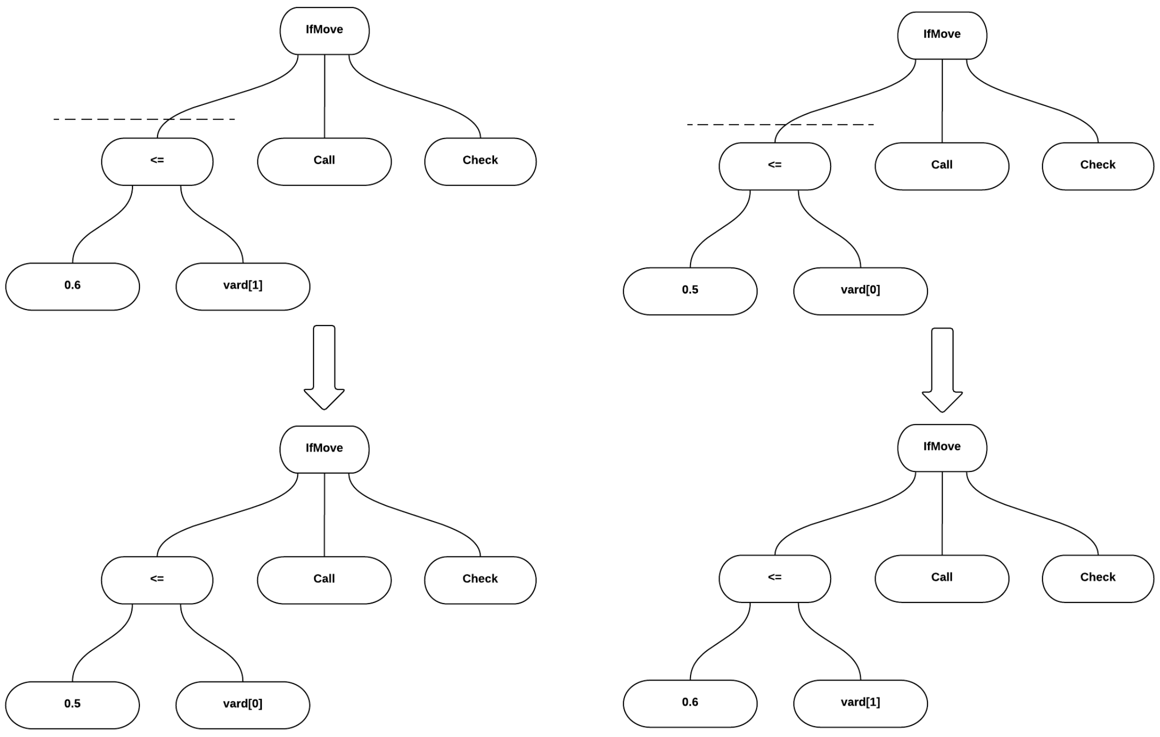
This very simple tree represents the following strategy:
If 0.6 (Two to Ace normalized in 0 to 1) is less than or equal to the player's lower hand card, make a call.
Else, check the game. Note that checking can also mean folding, in case there has been a raise.

2.2 Evolution

There are two fundamentally different ways for evolution to change a tree, namely crossover and mutation.

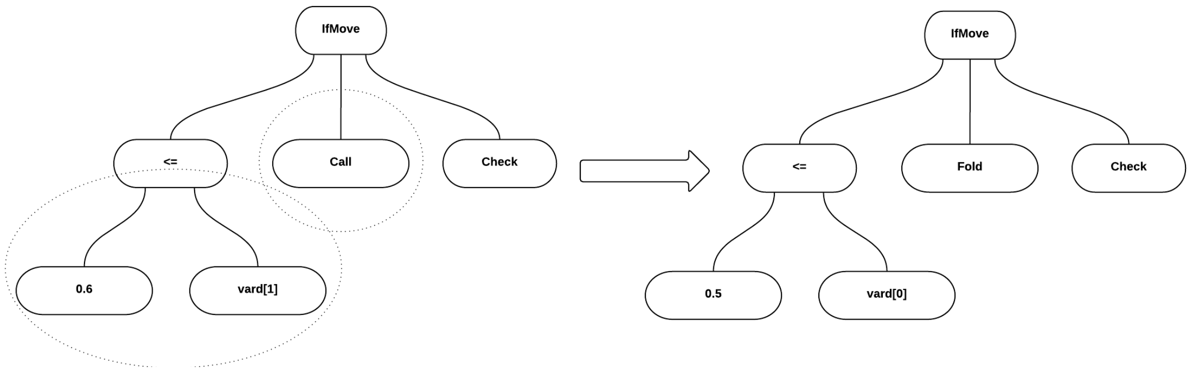
2.2.1 Crossover

Crossover takes two individual players and swaps subtrees in a random point. Of course both subtrees need to have the same return type. This method is more common, since good players that were not sorted out by evolution usually have good subtrees. So the the well-playing trees are not completely destroyed by the process of evolution, but there is much room for new combinations.



2.2.2 Mutation

In contrast to Crossover, mutation does not even need two individual trees for change. Either a single node is chosen randomly and changed to some other random node that returns the same type. Or even a randomly chosen subtree can be changed. Mutation has to be used carefully, since random changes can change a very good player to be useless.



3 Implementation

3.1 Initial changes

As mentioned above, our focus in this work was to change existing nodes and/or add new nodes to the chromosome tree. When we first evaluated the existing code, especially the implemented functions and terminals, we observed that the raise value was randomly calculated [0.0 to 2.0 * pot]. In this case, the value was neither decided nor influenced by evolution.
Now raising plays a vital part in a Texas Hold'em No Limit poker game and this was also noticed by the evolution which often evolved players who's tree was only a raise node. Therefore we decided to concentrate on making the raise node accessible to the evolution so its value can be part of the evolution.

The first attempt was to allow the raise leaf-node to have a subtree which would potentially get evolved. For this purpose we introduced a new function node called "IfDouble". It evaluates a boolean expression and returns a double value wich is the factor of the raise [i.e 1,9397 * pot size]. In this case, a ConstMove terminal could either be a terminal or a function node.

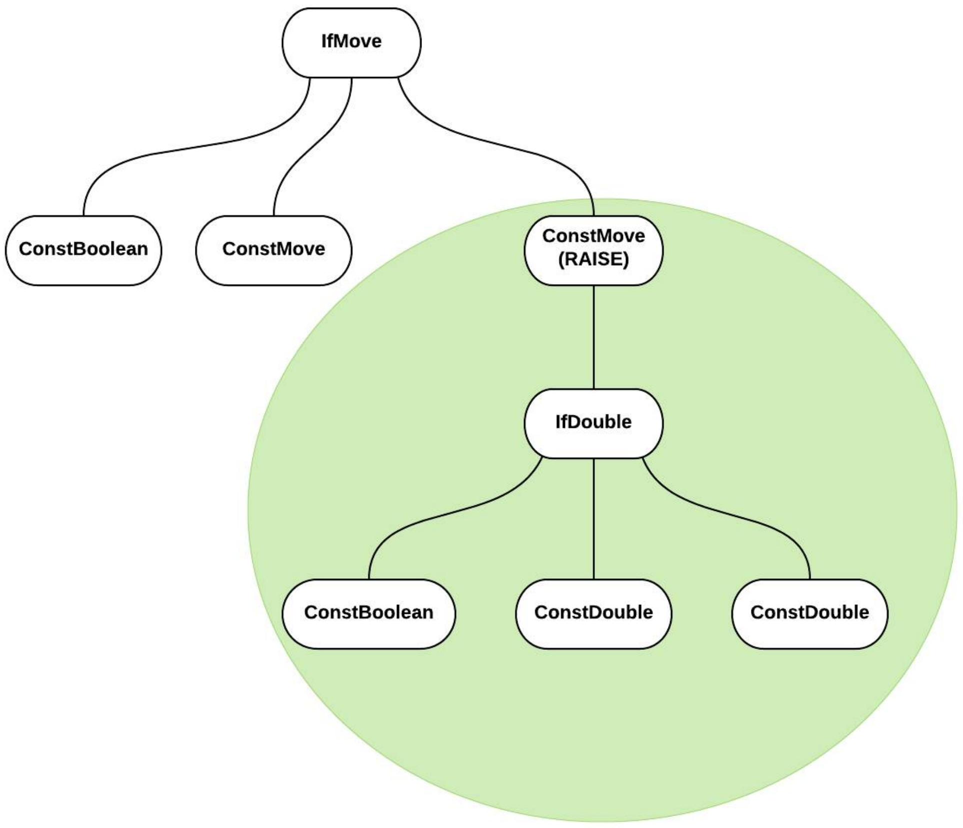


Figure 1: *Example tree of an evolved player with a IfDouble-subtree appended to raise*

This however lead to undesirable mutations like the following:

- GPoker produced IfDouble-Nodes that only have one child (ConstBoolean).
- GPoker produced ConstMove(RAISE) Nodes that have a ConstDouble child which do not get evaluated.

3.2 Final implementation

3.2.1 Raise

As a final solution, the raise move was moved out from ConstMove and redesigned as a independent function node. It still has only one child but now the lines between a function and a terminal are not blurred. With this change, the evoluion can now remove raise entirely if it finds that this is not a good functions. In addition to that, it is now easier to mutate the factor of a raise and which is more dynamic and granular.
This is because a raise subtree can now grow or shrink and build up different calculations for the raise factor like multiplying the high card with another factor. Also any node returning double can be attached to the new raise.

3.2.2 Fold

After some testing we noticed that sometimes a player would emerge that would fold the entire time. A fold-only player does not make any sense since it would lead gradually to the loss of all chips of that player. Although this is a crucial mechanism and a valid move, we deletet fold from the ConstMove class to avoid this kind of players and it now can only return a "Call" or "Check".
This is possible because the *checkMove(Player player, Move move)* method of the "Dealer" class handles invalid player moves and automatically folds [i.e. when a player checks in response to a raise].

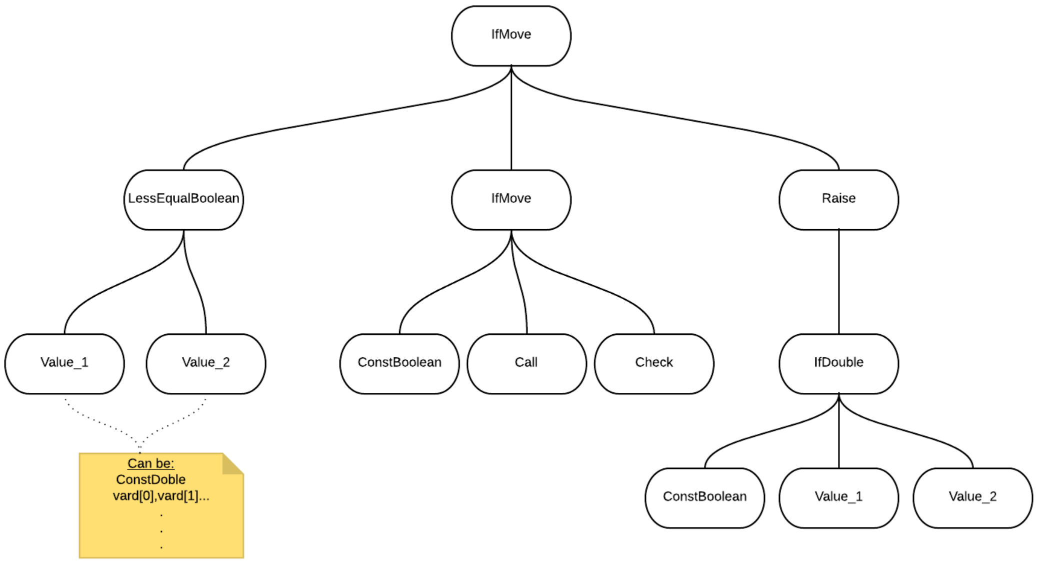


Figure 2: *Example tree of an evolved player after the final implementation*

4 Experiments

For the experiments we evolved two GPlayers both against a "PatternPlayer" and a "TestPlayer" in multiple runs. The first player was the one without any changes to his funcitons and terminals and had no other changes made to it by our group. The second player had the aforementioned changes applied. After all test runs were completet the fitness of the old and new player were confronted.

4.1 Setup

4.1.1 Evolving against the PatternPlayer

BIN MIR NICHT SICHER OB HIER WAS REIN SOLLTE. DER CODE VOM PATTERNPLAYER IS EWIG LANG UND DEN IN WORTEN ZU BESCHREIBEN IS AUCH A SCHAS!!!!!

EURE MEINUNG!!!

4.1.2 Evolving against the TestPlayer

One TestPlayer was choosen from previously evolved players that made sane moves and played as follows.

```
if (getCards().get(1).getRank() < Card.JACK) {
  if (dealer.getCallBet(this) < 9)
        move.setType(Move.CALL);
  else
        move.setType(Move.CHECK);
}else
  move.setBet((int)(0.9849 * dealer.getPot()));

return move;
```

In short, this player was playing more defensively when his highest card was lower than a JACK and only made a "Call" or "Check". If his highest card was greater he played aggressive and raised 0.9849 * pot-size.

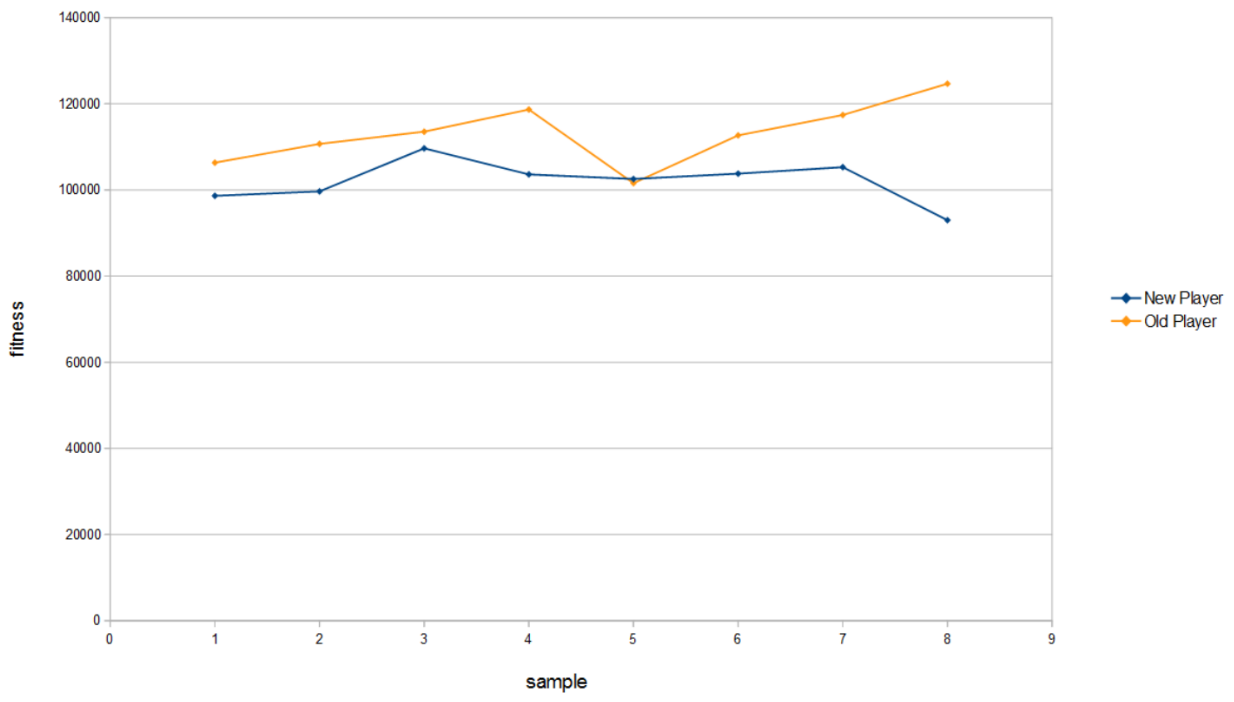
4.1.3 Experiment parameter

For both experiments the parameter were choosen as follows:

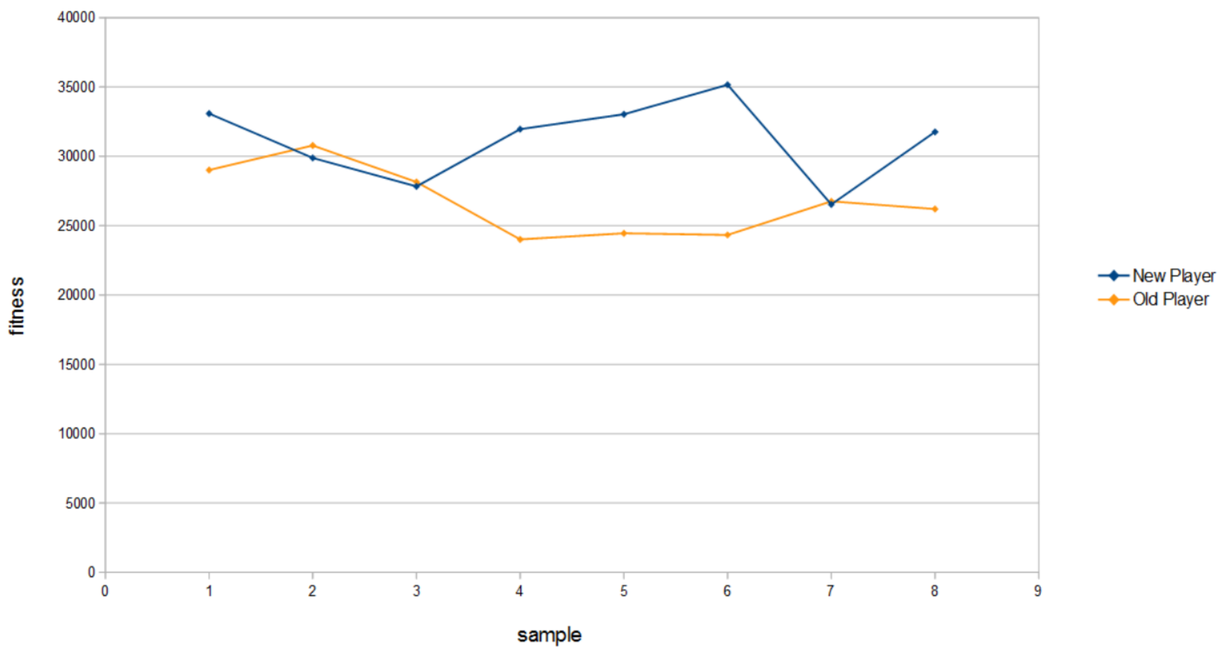
- Game Mode: Doyle
- Test-Runs: 8 for each player
- Generations: 10.000
- Population-Size: 100
- Rounds: 1000
- Mutation-Rate: 0.0369

4.2 Results

4.2.1 GPlayer VS. PatternPlayer



4.2.2 GPlayer VS. TestPlayer



5 Links

- Project Page: `https://student.cosy.sbg.ac.at/~tdafir/nc/`
- PS Page: `http://www.cosy.sbg.ac.at/~helmut/Teaching/NaturalComputation/proseminar.html`