AI - final project

Thor Gunnlaugsson Jensen

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
thorj14@student.sdu.dk

1 Introduction

The objective of this paper is to describe the work gone into implementing an Artificial Intelligence (AI) Ludo player as part of the the *Tools of Artificial Intelligence* class in the SDU Robotics masters program.
To begin with a framework of a Ludo game with 2 types of players was given beforehand and the goal was then to implement and AI of ones own choosing. This paper will describe why I choose to solve the task with an Evolutionary Algorithm, how the my algorithm is structured and works and the results gathered from the algorithm.

2 Methods

To solve this problem a Genetic algorithm (GA) was chosen and implemented with a uniform crossover strategy[1] and a mutation rate of 70% but with a very small impact. The GA used was build fairly crude and with the philosophy of "less is more" therefore the GA has been build in one *.cpp* file with one class that mostly takes care of everything. The reason for making such a bloated class was to make the implementing of another students AI as painless as possible and to ease the transfer of my own AI to another student for comparing.

One could also have chosen another method for creating an AI Ludo player but after some discussing among fellow students it somehow seemed more approachable to build the player as a GA over other methods. The most important parts of the GA's implementation will be described in the upcoming subsections to better clarify how exactly the algorithm was build. All decision regarding those parts will be described there as well.

2.1 Representation

The whole Ludo game, besides the GUI, is being represented with integers in vectors, where each integer is the corresponding field of where the piece is and this is all taken care of in the provided code and does not need much alterations. In order to implement a AI player and take decisions the AI needs to know where

[1] http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

it itself has its pieces and where the "enemy" pieces are located. Also the dice roll for its own turn needs to be taken into account.
When the AI is aware of where it has its pieces and the eyes of the die, then it is time to implement a method for deciding which pieces should be moved, which is explained in subsection 2.3. The possible moves are encoding binary to indicate whether a piece can be make a certain move or not and a move is associated with a certain importance/"Weight" which will be described in further detail in section 2.2.

2.2 Initialization

In order to start the GA correct it needs to have its chromosomes initialized with some random values in order to have a starting point to work from. A separate *Struct* has been written in the GA class which only takes care of putting the weights for all possible moves/genes in the whole game into a vector full of floats. These values are used in the decision making process in order to choose the correct move. This based on a greedy approach where the highest value will always be the one chosen. When a chromosome full of initialized randomized chromosomes are initialized it is then pushed onto another vector and this process is repeated until a desired candidate size has been reached aka. the elements in the vector containing the genes. The size of my population hence forward called chromosome-pool are chosen to be 100 because of some general recommendations[2] but could be of any desired size. The values for each gene in the initialization process is randomly chosen to be between 0 and 1.

2.3 Explore board

In order to decide which piece to move all the possible moves for each piece has to be investigated. A matrix of moves was created were the rows were the pieces and the columns are the moves. Only nine moves was defined in this project, so nine Boolean were used, which also required nine if statements to check for that particular move. The moves defined can be seen in table 1. The location of the GA's own pieces were given in the game and also the location of enemies.

Each move is a gene in the chromosome and each gene/move has a weight associated in order to make possible for the algorithm to learn the correct values for each gene. In order to create a lot of chromosomes with different genes the chromosomes were build as a separate *struct* and the content of the *struct* can be seen in table 1.

2.4 Evaluation fitness

Each chromosome in the chromosome-pool is evaluated after its lifetime has expired. The lifetime is decided by the user. Tests with different life times can be seen in the results section. The fitness function is a aggregate fitness function[3]

[2] http://www.optiwater.com/optiga/ga.html
[3] http://www.nelsonrobotics.org/paper_archive_nelson/nelson-jras-2009.pdf

Table 1: *The constructor for all the possible moves hence forward referenced as genes. All the different chromosomes are then put in a vector to make it easier to cycle through them all. The weights for each gene is randomly initialized in the beginning of the program.*

enterBoard
move2SafeZone
sendEnemyHome
block
moveNormal
move2Star
move2Globe
moveInSafeZone
finishPiece
vector<float>AllMoves

and is decided on how many game the each chromosome won and then dived with the games it played without any concerns about how the task of winning was done.

2.5 Selection

The selection procces is based on Elitism because according to [1, Chapter 5, p.- 168] "Many researchers have found that elitism significantly improves the GAs performance". Only the chromosomes that scored the highest and second highest are used for repopulating a whole new generation which means the parental generation dies in order to make room for the new generation.

The genetics operator used for creating the new generator is based on a uniform crossover described in [1, Chapter 5, p.- 172] with a randomly chance to inherit genes from one or the other parent[4].

The way it has been implemented in this algorithm is to take the two parents and then generate a random number between 0 and 1 and if the random number is above 0.5 the child inherits from the first parent else it inherit from the other parent.

2.6 Mutation

The mutation done with this algorithm differs slighty from various websites[5] [6] [7] which points out that the mutation rate should be fairly low however they

[4] http://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php
[5] https://www.researchgate.net/post/Why_is_the_mutation_rate_in_genetic_algorithms_very_small
[6] https://www.researchgate.net/post/How_to_calculate_the_Crossover_Mutation_rate_and_population_size_for_Genetic_algorithm
[7] https://www.researchgate.net/post/How_to_implement_mutation_and_crossover_probability_rates_in_Genetic_algorithm

do not inform which crossover method is used. According to [1, Chapter 5, p.-174] the ratio between the two are more important and the mutation are often underrated hence this algorithm has a very high mutation of 70% however the mutated genes are not allowed to differ more than 0.2.
The method used for mutating the chromosome-pool is to take the whole pool and then shuffle them randomly. Then it is possible to simple take the first 70% in the chromosome-pool which is stored in a vector and mutate that particular chromosome genes. This is done with a Gaussian distribution where the mean is the genes value from its parent with a variance of 0.2. The new value created with the Gaussion distribution then replaces the old value and thus the gene has been mutated.
The reason for the variance being hard coded to 0.2 is because the initial weights of each gene is a random number between 0 and 1 and 0.2 is then a maximum of 20% randomization seen from the initial beginning of the GA.

2.7 Training

This GA was trained with a fixed size chromosome-pool of a 100 different chromosomes and played a million games against three Ludo players that would just randomly choose a piece. Then over four training session the only difference would be the amount of times each chromosome were allowed to play before its fitness was calculated.

2.8 Challenging another AI

To verify how smart this GA was it was put to the test by another algorithm to see who was the better Ludo player because it is not very impressive to beat a random player, however to beat a real player would be impressive. When nobody volunteered to play 1.000.000 games against the algorithm another AI was borrowed.

The borrowed AI was based on a reinforced learning technique called Q-learning. The technique is build around an agent who can be in different states and the agent can perform several actions. The objective in Q-learning is to estimate the Q-values based on the matrix made up of the states and actions[8]. So in the sense that the GA algorithm tries to find the appropriate weights for each gene the Q-learning tries to estimate the correct reward in order to make better decisions.

3 Results

Table 2 is created by taking the chromosome-pool with 100 different chromosome and having each chromosome play 500 games and reproduce based on the two

Games pr. Chromosome	Win-rate when training (1 million games)
500	39%
700	39%
1000	41%
1200	38%

Table 2: *The table is made up by the win-rate of a million games with different practise games for each chromosome.*

best scoring chromosomes. This procedure was repeated 700, 1000 and 1200 games before evaluation over a duration of a million games each.

Every time a chromosome won it was collected in order to estimate a mean win-rate in order to see how the chromosomes converged and to verify if the GA learned anything. This can be seen in figure 1. The different colors represents the different length each chromosome were allowed to play. Obviously the purple line is higher than the others because it could train 1200 times per chromosome. The interesting is to see how many generations it took for the chromosomes to converge.

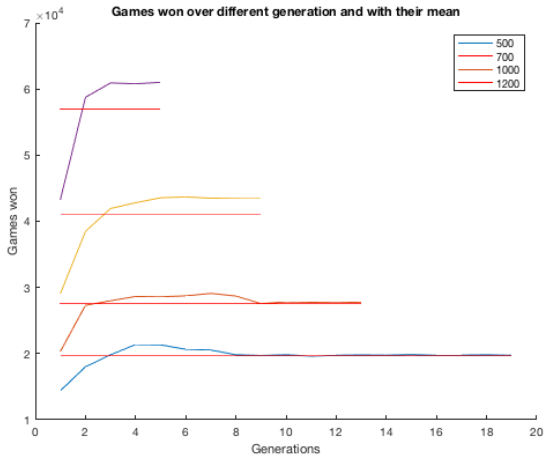

Fig. 1: *A plot showing how many chromosome wins each generation had and the win-rate mean of all of the chromosome with the same practise time.*

The plot seen in figure 2 is the different weights associated with each gene. There are 4 chromosomes being depicted there, each with their own color. The

[8] http://www.gatsby.ucl.ac.uk/ dayan/papers/cjch.pdf

color indicates how many games the chromosome were allowed to train. The plot is of the highest scoring chromosome of each of the four training sessions.

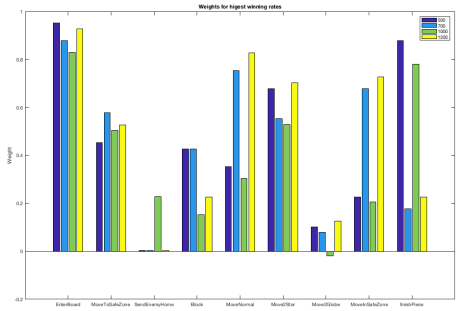

Fig. 2: *Plot showing the weights for each gene. The colors of the bar indicates how many games each chromosome where allowed to practise.*

When the most promising chromosome had been chosen the values are hard coded and 1 million games were played with these gene weights. In a attempt to introduce some randomness and fairplay the chromosome was tested as all four players (colors). The result can be seen in figure 3.

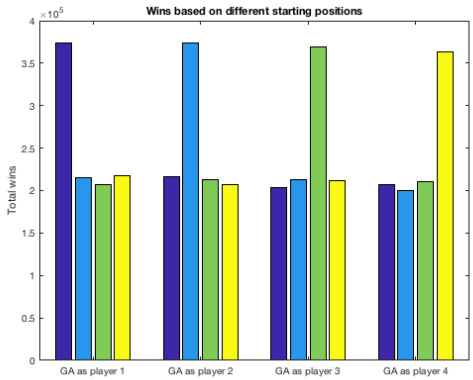

Fig. 3: *A plot showing how much the GA won with different starting position (colors)*

This GA was test in a game of Ludo aginst another AI trained with the Q-learning technique over a series of 10 trials where each trial was 10.000 games and the result can be seen figure 4.

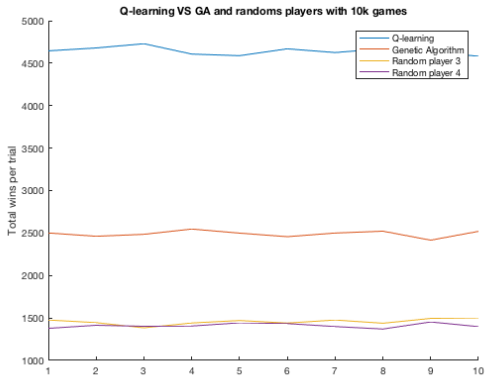

Fig. 4: *A combined plot of how many times the GA, Q-learning and randoms won in 10.000 games over 10 trials.*

4 Analysis and discussion

The weight in figure 2 seems to converge nicely together and by looking in table 2 the win-rate does not change much. However from plot 2 the chromosomes allowed to play 1.000 games are not following the weights as nicely as the rest, especially when it comes to sending an enemy home and moving to a globe. Here the 1.000 game chromosomes varies a lot from the others.
This apparent aggressive nature seems to increase its win-rate.

From figure 1 one gets an impression that the algorithm is converging however the GA should have been allowed to play more than 1 million games in order to go through more generations when it is supposed to train each chromosome 1.000 and 1.200 times.

The GA was trained as the same player for all of its training sessions which one could suspect would hinder its ability to play from other positions however from plot 3 there does not seem to be much difference if in where the GA starts from.

The game against another AI as seen from figure 4 showed that this GA definitely has room for improvements. Some of the possible improvements that could be implemented would be different parameters for mutation to see if that has any effect and avoid being caught in a local maximum. Also to expand the GA choices by defining other states the pieces can be in.
Another idea for improving the GA would be to expand the fitness function to not only evaluate win-loose but how much the chromosome lost and how much the chromosome won, by counting the distance from goal to the pieces that did not make it in case of a loose and count how far the enemy pieces were from

their goal in case of a win. This might give a better idea of what worked.

5 Conclusion

A genetic algorithm was successfully build and the algorithm was able to learn to play Ludo with a win-rate around 40% against three players who played randomly.

The win-rate being above 25% indicates some sort of intelligence has been implemented and according to the results in this paper the algorithm learns and improves itself.

Letting the GA play 1 million was not sufficient as soon as the chromosome-pool got big and each chromosome played enough to get at good idea of its capability.

The GA performed poorly and inadequate against another AI trained with Q-learning. Not to say Q-learning in general is superior to GAs but the difference in intelligence between these two algorithms was substantial.

6 Acknowledge

First of all I would like to thank Mads Tilgaard for borrowing me his AI to play against and rate my own. If you had not i would still have been going around thinking my algorithm was good enough.

The second person i would like to express my gratitude towards is Peter Gilsaa for taking the time and giving useful feedback while programming the AI. The short discussions and bouncing of ideas between us really helped me a lot to move forward with my algorithm and improve my coding in general.

The last person i will mention by name is Anne Groth. Thank you for listening to me explain what I've done to better help me debug and for given the advice to add 2 lines of code in the main source file and execute the project in the terminal making possible to run the program for longer. That really helped a lot generating data!

References

1. Melanie Mitchell, *An introduction to Genetic Algorithms*, The MIT Press, Massachusetts, 1996.