Ludo AI Agent - Generic Algorithm

Jacob Falgren Christensen

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
jacoc17@student.sdu.dk

Keywords: Artificial Intelligence, Evolutionary Computation, Genetic Algorithm, Reinforcement Learning, Q-learning.

Abstract. This paper will present methods for applying evolutionary and reinforcement learning to the board game Ludo. The evolutionary method will be using the Generic Algorithm and the reinforcement learning method will be using Q-Learning. Furthermore, this paper will present the solutions to the problem of the enormous state space that comes with a complex game such as Ludo. The results of these two methods will then be compared to see which methods yield the best win rates when tested across several games of Ludo.

1 Introduction

This paper will present different methods that can be used to teach an AI agent how to play the board game Ludo. The game of Ludo is quite simple for a human to grasp but to a computer, it can be quite hard, this is due to the enormous state space that would be required to represent the state of the board and all the players at any given time. As such two methods were chosen, both are of the category of unsupervised learning, the first of these methods involves evolutionary computation and the second involves reinforcement learning.
The first method is the Generic Algorithm [1] which is an evolutionary learning method in which the agents evolve across generations where the best traits are carried over from a set of parents to their children and as such the success rate of the agents evolve over multiple generations. This method uses an initial population of agents each containing its own chromosome that determines how they respond to different situations. Through the evolutionary process, these chromosomes mix and create new agents that have the potential to outperform their parents.
The second method is a reinforcement learning method where the agent is learning by trail and error, here the action of the agent is either punished or rewarded based on how good the move is. The specific reinforcement learning method used is Q-learning [4], where the agent transitions through a set of states with a given set of action. These actions are either exploitative or exploratory, when exploring random actions are taken, when exploiting a Q-table is consulted as to what action will return the highest reward.

2 Methods - The Generic Algorithm

In this section the method Generic Algorithm will be explored in depth along with the different tests that are done to test the fitness of the agents that are produced by the algorithm. This method will require a state space representation of the current state of the game, and the challenge that arises is that the complete state space for a game of ludo becomes quite large as each brick a player has would require its own representation of the states. This is shown in [2] where the state space representation of ludo can achieve a magnitude of between $5 \cdot 10^{21}$ and $1.6 \cdot 10^{22}$ states, and thus it would not be feasible to attempts to make a complete representation. The methods that are tested in this paper therefor do not rely on a complete representation.
The GA method excels in the ludo board game do to its ability to work well on problems where a complete state space representation cannot be easily provided. As such this method can learn something from an incomplete set of information. With this in mind a set of states can be developed that are then used to train the GA agents, these agents all contain genetic material in the form of a chromosome, this chromosome is a vector containing a set of weights that are used to determine ho the agent reacts in different situation, thus enabling the agent to chose the best brick for each situation. These weights are initialised randomly with a value from $[-100; 100]$ thus allowing for negative weights such that bad states can be punished and good states are rewarded. An example can be seen in equation 1.

$$\text{Chromosome} \longrightarrow \begin{pmatrix} 86 \\ -59 \\ -5 \\ -52 \\ 14 \\ -74 \\ 39 \\ -84 \end{pmatrix} \tag{1}$$

For this project 8 states are designed, these states are designed using the dice roll that the agent makes at the start of each round. The states then describe the resulting position of a brick after it has been moved according to the dice roll, the states are listed bellow:

- Brick can be moved out of start field
- Brick can be moved into end field
- Brick can be moved into striking range of an opponent
- Brick can hit an opponent home
- Brick can be moved in front of another players home position
- Brick can be moved into danger
- Brick can be moved onto a star
- Brick can be moved onto a globe

All of the states above are in designed with the rules of the ludo board game in mind such that they do not violate these. The states that a applicable and thus active for a single brick is evaluated using a utility function which produces an 8 bit string representing the states of a brick.

$$\text{Brick} \longrightarrow 11011010$$

Using this representation the score of each brick can be computed by finding the dot product between the activation string and the chromosome, both represented as a vector. This can be seen in equation 2.

$$Score = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 86 \\ -59 \\ -5 \\ -52 \\ 14 \\ -74 \\ 39 \\ -84 \end{pmatrix} \tag{2}$$

This is repeated for all bricks and the brick with the highest score is chosen. To evaluate the fitness of a single agent it plays 100 ludo games against random-walk opponents and the amount of wins the agents achieves is recorded and used as its fitness.

2.1 Selection Step

This has been divided into to parts, the first part culls the weak and the second part is used to find pairs of parents that can then produce offspring. The first step involves removing the half of the population with the lowest fitness, this ensures that the next step of the process will not use the weak when generating offspring. To find a pair of parents tournament selection is used, here three random agents are chosen from the remaining population and and the best agent wins the tournament and becomes parent one, this is repeated once more and the winner is chosen to be parent two. This process of selecting pairs of parents is repeated until a desired amount of pairs have been selected.

2.2 Crossover and Mutation Step

Once the pairs of parents have been selected they can now be used to produce offspring, the method chosen to do crossover is single point crossover where the point at which the crossover occurs is chosen at random, therefore a large amount of children is also wanted as this would ensure that a good mix of the chromosomes are made. Once crossover has occurred the different chromosomes are mutated, here a mutation factor and a mutation strength are chosen and these determine how often mutation occurs and the strength if the mutation. The mutation chance works on each weight of the chromosome such that all weights have a set percentage chance of mutating. The strength of this mutation is then determined by the mutation strength and this is a random number generate in a interval of +/- mutation strength which is the added to the weight. The mutation chance is chosen to be 20% as this ensures that the chromosomes will mutate more frequently thus allowing for more exploration. The mutation strength is set to 10 which allows for sizeable mutations to occur which also enhances exploration.

2.3 Steady State Step

In this step the members of the new generation are selected, to do this the method steady state is used where the parents along with their children are all polled together along with a batch of new blood which consists of a set of new randomly generated agents. This ensures that the GA is able to escape from local maximums. From the pool the best members are then selected and become part of the new generation, this will ensure that good members are carried over through generations such that they have a chance of creating offspring with a potential to outperform them.

2.4 Test Design

To test the generic algorithm there are multiple different parameter that can be tuned to increase performance, however an increase in most of these parameters will result in a large increase in computation time. The following parameters are considered for tuning and 2 parameters are chosen for the tests:

– Amount of Generations
– Population size
– Mutation rate and Strength
– Amount of Children
– Number of games played

The two parameters that are chosen is the amount of generations and the population size, this is due to preliminary tests that have showed that a change in the number of games played did not have a significant improvement in the quality of the fitness of the chromosomes. It should however be noted that the higher the amount of games played the more accurate the fitness is. This test can be seen in table 1.

	Win Rate [%]
50 Games	57.2
100 Games	56.9
200 Games	56.1

Table 1: Preliminary tests on the fitness over multiple games.

The amount of children is kept constant to ensure a good mix of the genes, for the tests this is chosen to be 16 children per pair of parents, this value was chosen to be double the size of a chromosome. Furthermore, the mutation rate and the strength were kept constant as well due to time constraints.

Test	Population Size	# Generation
Control Test	25	25
# Generations 1	25	50
# Generations 2	25	75
Population Size 1	50	25
Population Size 2	75	25

Table 2: Table of the tests and the values that change.

A control test is first run with the base parameters which can be seen in table 2, this creates a baseline where the lowest win rate, the average win rate and the highest win rate is saved for each generation. Following this control test 2 additional tests are carried out per variable for a total of 4 tests, resulting in the tests seen in table 2. All the tests are carried out multiple times to eliminate outliers. The Code used to run these tests along with the Images in full size can be found the GitHub Repository [5].

3 Methods - Q-Learning

The method used for comparison is Q-learning which is based on [4], in this paper a state space representation similar to the GA representation is used, where the only states that are tracked are those relating to the local surroundings of the bricks. However the number of states used in this method is only six instead of eight. These states are in a similar fashion represented with a six bit long string that indicates which states that are applicable to a given brick.

Home	Safe	Vulnerable	Attacking	Finish Line	Finish

Table 3: States used for Q-learning

These states are then used for the Q-table that contains the Q-values. The function used to compute the rewards a given action will achieve is based on the change in state that the action will cause, here leaving positive states or entering negative states are punished and entering good states are rewarded. This is represented with a 6 element vector which is then dotted with a set of weights to achieve the reward, seen in equation 3.

$$reward = \begin{pmatrix} 0 \\ -1 \\ -1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} w_{home} \\ w_{safe} \\ w_{vulnerable} \\ w_{attacking} \\ w_{finishline} \\ w_{finished} \end{pmatrix} \tag{3}$$

Here the left vector is based on the states found in table 3 and the right vector is a vector of weights. The Q-learning agent will make its moves based on the highest values in the Q-table based in the state that it is currently in, this is the exploitation part which occurs most often. Exploration is done 15% of the time where a random action is taken, this ensures that the agent explores new possible best moves during training. To update the Q-values in the Q-table the following equation is used:

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \cdot maxQ(s_{t+1}, a) - Q(s_t, a_t)) \tag{4}$$

where Q is the Q-table; α is the learning rate; r_t is the reward at time step t; γ is the discount factor; s_t is the state at time step t and a is the action taken.

4 Results

In this section the results of the tests described in section 2.4 will be shown and explained. First the test results of the Control test can be seen in figure 1. Then the results of the tests for varying amounts of generations can be seen in figure 2. Lastly the test results for the varying population sizes can be seen in figure 3.

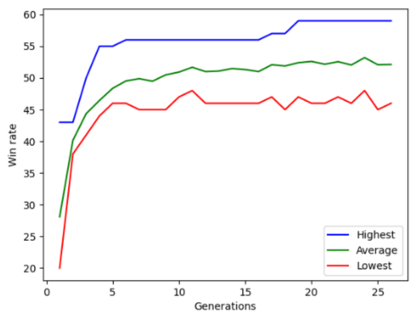
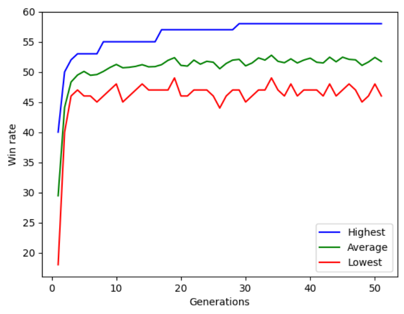
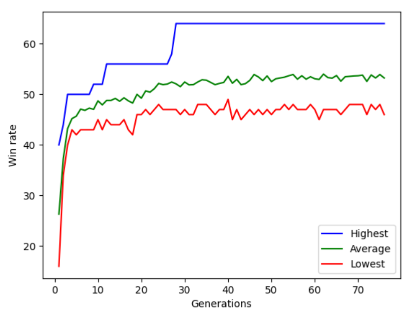


Fig. 1: Control Test.

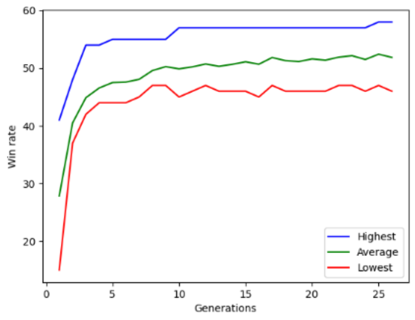


(a) 50 Generations.

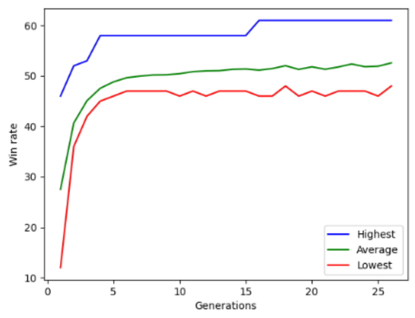


(b) 75 Generations.

Fig. 2: Tests with a varying amount of Generations.



(a) Population size 50.



(b) Population size 75.

Fig. 3: Tests with a varying Population size.

For the Q-learning method two sets of weights are used; one with even weights and one with a set of weights created by the author of the paper. These can be seen for the training of the agent in table 4, and for the evaluation of the agent in table 5.

Agent Training	Equal Weights	Human Weights
Win Rate [%]	40.20	37.11

Table 4: Final win rate over 30,000 training games.

Agent Evaluation	Equal Weights	Human Weights
Win Rate [%]	47.2	42.4

Table 5: Final win rate for agent evaluated over 20,000 games.

5 Analysis and Discussion

As it can be seen in the test results the GA method is able to produce chromosomes that are able to achieve a win rate over 60% in most cases, here the highest win rate that was achieved over 100 games was 64% this can be seen in figure 2b. It should however be noted that this win rate is achieved after about 30 generations and therefore it is not a result of running the full 75 generations that the test were about. And thus this high win rate is most likely a moment in the evolution where the best agent is found, and as it can also be seen this agent does not improve or get overtaken by another agent during the remaining generations.

One of the most important metrics that are tracked is the average win rate, this gives a great indication of what is happening in the population, here it can be seen that it looks like most of the tests plateaus which is unwanted as this means the learning has stopped or significantly slowed. This could be combated by increasing the mutation rates when consecutive generation do not make significant changes in win rate, here even negative changes could be good as it is required to move away from a local maximum. When looking at the generation sizes it can be seen that the graphs here are much smoother compared to the generation graphs which indicates that a larger population size results in less fluctuation in the win rats.

This can then be compared to the Q-Learning method which as seen in the tables perform considerably worse than the GA method this could be caused by the weights that were chosen for the training and the evaluation, it could however also be caused by the states that were chosen. As with the GA the states have a very large effect on the quality of the agents that are produced, thus more and better states would improve the win rate of both methods as the agents would receive more information to base its decision on.

6 Conclusion

In conclusion the Generic algorithm and the Q-learning method both are able to learn the board game ludo to a certain degree and when compared to each other GA outperforms Q-learning by a significant margin when it comes to the best achievable win rates. The GA method works very well based on the information that it is given with the limited amount of states that it takes into consideration.

7 Acknowledgements

I wish to show my gratitude to Associate Professor Poramate Manoonpong for his support in writing this paper. With his contribution of teaching the course Tools of Artificial intelligence on the second semester of Advanced Robotics Technology at the University of Southern Denmark. As well as my colleague Jens Otto Hee Iversen for allowing me to use his paper as a comparison and my colleague Søren Gejl Ilstrup Madsen for sparing with me throughout this project.

References

1. S.N. Sivanandam S.N. Deepa. Introduction to Genetic Algorithms. Springer-Verlag Berlin Heidelberg. 2008.
2. Faisal Alvi and Moataz Ahmed. Complexity Analysis and Playing Strategies for Ludo and its Variant Race Games . 2011 IEEE Conference on Computational Intelligence and Games (CIG'11). 31 Aug.-3 Sept. 2011.
3. SimonLBS. SimonLBSoerensen/LUDOpy. url: `https://github.com/SimonLBSoerensen/LUDOpy`
4. Jens Otto Hee Iversen. Q-Learning - Ludo.
5. Jacob Falgren Christernsen. DenJacob/LudoProject. url: `https://github.com/DenJacob/LudoProject`