AI plays Ludo with a Genetic Algorithm

August Ravn Mader

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
aumad17@student.sdu.dk

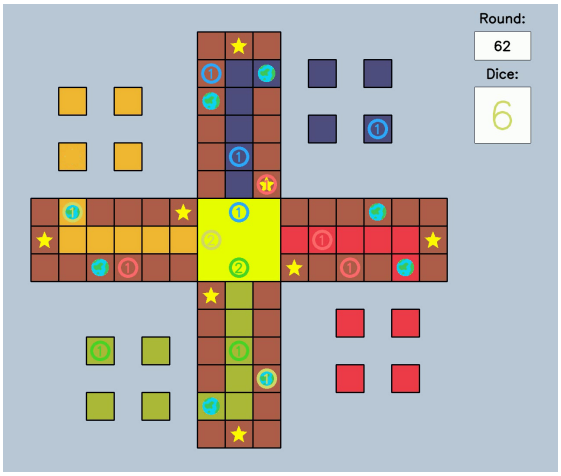

Fig. 1. The Ludo board with an example of configuration of the players' pieces.

Abstract. In this paper a Genetic Algorithm is presented and implemented to play the board game Ludo. The algorithm uses a refined state space representation and a utility function encoded into the individual's chromosome. The Genetic Algorithm performs satisfyingly and achieves a best win-rate of 79%. This result is compared to a SARSA approach with a best win-rate of 46.3%.

1 Introduction

This paper is written as part of the concluding evaluation of the 5 ECTS course Tools of Artificial intelligence at the master level for Robot Technology at SDU. The paper will describe how to implement an Artificial Intelligence (AI) to play and maximize the chance of wining in the board game Ludo. The Ludo board can be seen in figure 1. Multiple AI approaches are described during the course. One of approaches learned is a Genetic Algorithm (GA). Personally I have no experience working with evolutionary computation and based on previous students experience GA should provide satisfying results. On this background the it is chosen to apply a Genetic Algorithm (GA) to play Ludo. This paper will describe how a GA can be implemented with the corresponding results.

2 Methods

This section will describe how the GA is implemented and trained to win the Ludo game.

GA is a evolutionary method to solve optimization problems. It based on a fixed size population with individuals. The population size applied for this implementation of the GA consist of 100 individuals. Each individual contains of a chromosome. The best individuals need to be determined on a fitness score, which will be used in reproduction for the next generation. The reproduction process consists of the selection, crossover and mutation steps. These steps will be described further in section 2.4 The GA is explained in pseudo code in algorithm 1.
Over the each generations the algorithm will yield in theory better chromosomes with better fitness until the algorithm has converged. The GA is applied to determine which brick is best to move. This depends on the state representation and an utility function. This will be described in section 2.1 and 2.2.

Algorithm 1: GA Pseudo code

```
   Result: Best individual with corresponding chromosome
 1 generations = 0
 2 initPopulation()
 3 score = computeFitness()
 4 while not succesReached(score) and generations <= 25 do
 5 |   elite = selection(score)
 6 |   crossover(elite)
 7 |   mutation()
 8 |   score = computeFitness()
 9 |   generations += 1
10 end
11 bestFitness = max(score)
12 bestIndividual = find(bestFitness)
13 return bestIndividual
```

2.1 State space representation

The implementation of GA is inspired by the paper *"Man - Machine Interface"*[1]. The Ludo game consists 16 bricks in total divided among 4 players. Each brick can have 52 different configurations, resulting in a full state space representation of 1652. Therefore, it is necessary to simplify the state representation for the GA learn to play the Ludo game in a realistic time. The state space is crucial for the success of the GA no matter the training time and the parameters how to evolve the population. The state is derived from the special board positions as well as good moves to performed to increase your chance of winning the game. Each state is a consequence of moving a brick with a given dice roll. The following 11 states for are applied.

1. Beat opponent
2. Enter goal straight
3. Hit star
4. Hit globe
5. Go out from home
6. Get into goal
7. Escape possible beat
8. Hit yourself home
9. Dangerous position
10. Standing on globe
11. None of the above - Move piece

Each brick needs to be evaluated to figure out which states are relevant. A brick can be in one or multiple states depending on the brick's position, opponents' positions and dice roll. However, state 11 is only set if none of the other states do not apply.

2.2 Utility function

When each brick's states is determined, the GA should estimate which brick is the best to move. This is done through an utility function which is a set of numeric values for each state. These values are encoded en the individual's chromosome. Thereby, using value encoding for the chromosome where a gene corresponds to a state. Depending on which states apply a utility for each brick can be calculated. This is done through combining the value states/genes into a sum. This sum will be referred as the utility value. The brick with the highest utility value is determined to be the best to move.
State 8, 9 and 10 has a negative affect on the utility value. State 10 is introduced to have the bricks standing on globe less likely to be moved. Thereby, reducing the chance of them to get beaten home.

The utility function is thereby, used to internally evaluate the states. As an example if brick 1 is in state 1 and 3, and brick 2 is in state 7 and 8 the utilities values will be calculated as shown in equation (1)

$$\begin{aligned} u_1 &= g_1 + g_3 \\ u_2 &= g_7 - g_8 \end{aligned} \tag{1}$$

The brick to move will then be the brick with the highest utility value. In this instance it is most likely brick 1 which is determined to be the best move.

Each individual in the population has a different utility function for the chromosome encoding. The GA needs to estimate the best chromosome by evolving the individuals by a fitness and the reproduction step.
The population is initialized with 100 individuals with random utility functions from values from 0 to 100.

2.3 Fitness function

Each individuals needs to evaluated depending on a fitness function, which described how well it wins in Ludo. The applied fitness function is a score which counts how many wins each individual achieved out of a 100 Ludo games against three opponents which do random moves. The score is thereby applied in the selection process to determined the best chromosomes.

2.4 Reproduction - Selection, Crossover & Mutation

As described previously the reproduction process consists of the selection, crossover and mutation steps. The selection process is based on the elitism approach. Where the top 10% of individuals with the highest score is selected for the crossover step. Thereby having a generational replacement of 90%. In the crossover step 90 children are created, thus the population remains with 100 individuals. Where the parents are chosen from an uniform distribution, so each individual was the same chance of being chosen, regardless of the amount of wins. The same individual can not be chosen to be both parents. The children are made with the cross breeding strategy where the even genes comes from the first parent and the odd genes from the second parent. The last step in the reproduction is the mutation where a gene from each individual is randomly selected from a uniform distribution. Then a is mutation value add to the gene form a Gaussian distribution with mean at 0 and a standard deviation of 0.25.

A new generation is now made and the fitness score is computed again to calculate if it can improve upon the previous generation. This process iterates until a success criteria is satisfied or more than 25 generations is made. The success criteria being a individual can achieve more or equal to 85 wins. The process is shown in algorithm 1.

The algorithm is developed in Python with the provided ludopy library. The code can be found at the github repository [3].

2.5 Comparison AI method - SARSA

Another AI method presented in the course is SARSA. This approach tries to estimate action-values depending on the current state S. This is done through given the agent a reward R when a desirable action A is taken with the following next A' in the subsequently state. This results in a Q-table with action-values or Q-values which will describe which actions yields the highest rewards in the possible states. SARSA is a variation of Q-learning but uses a on-policy temporal difference control, compared to Q-learning which is a off-policy method. The SARSA approach is described in pseudo code in algorithm 2.[2] Both SARSA and GA needs to have a state space representation. But SARSA differentiates GA by also importantly needs define the rewards beforehand. Where GA learns these rewards with the utility function encoded in the chromosome.

In the results section the performance of GA and SARSA is compared. The result of the SARSA is provided by a fellow student Troels Andreasen (tandr17@student.sdu.dk).

Algorithm 2: SARSA Pseudo code

Result: Q-table with action-values
Initialize $Q(s, a)$
while *minimum change in* Q **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    while *not* S *is not terminal* **do**
        Take A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\left(R + \gamma Q(S', A') - Q(S, A)\right)$
        $S \leftarrow S'$
        $A \leftarrow A'$
    end
end
return Q

3 Results

In this section the results of the implemented GA will be presented. The algorithm is tested against three opponents which do random moves. The fitness score is as explained previously wins out of 100 games.

The results of the GA is shown in figure 2 where the fitness score for each individual per generation. The bigger the dot the more recurrent the fitness score. In figure 3 can the statistical mean and standard deviation be seen. The GA is tried to be improved be introducing a extra state: Possible best opponent next round. This tries to check if a brick has the possibility to beat an opponent next round. Thereby, increasing the chance to hit your opponents home. The results of the this can be seen in figure 4 and 5.

In table 1 the best chromosome can be seen of the best performing individual throughout the whole training. Furthermore, the mean best score, mean score and the standard deviation is compared between the original GA and the GA with the extra state.

The result from the SARSA method is provided by a fellow student Troels Andreasen these can be seen in table 2. The SARSA is also tested against three opponents which do random moves.

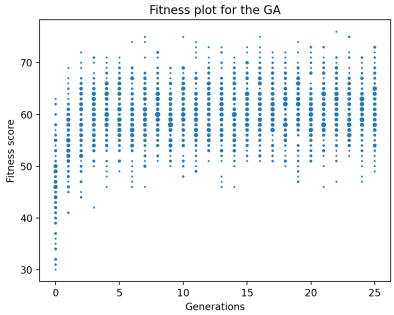

Fig. 2. Fitness plot for each individual per generation.

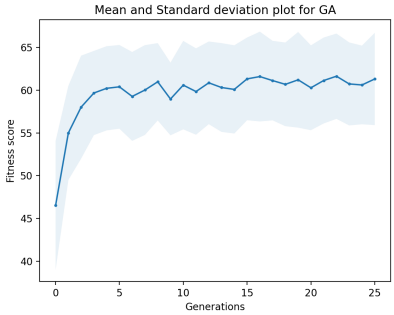

Fig. 3. The statistical plot with the mean and the surrounding standard deviation.

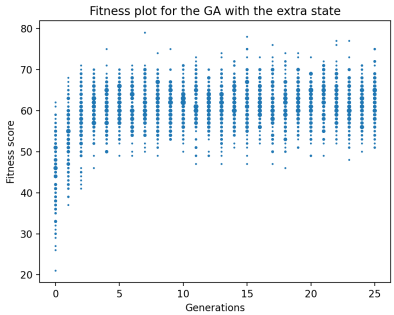

Fig. 4. Fitness plot for each individual per generation of the GA with extra state.

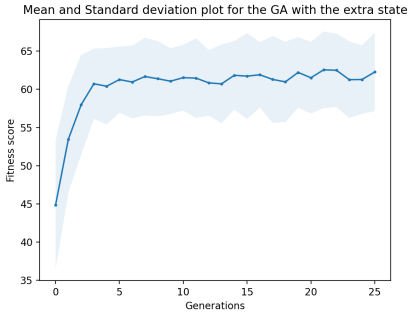

Fig. 5. The statistical plot with the mean and the surrounding standard deviation of the GA with extra state.

Beat opponent	93	48
Hit itself home	20	13
Enter goal straight	88	95
Hit star	74	95
Dangerous field	18	70
Hit globe	10	20
Go of from home	66	87
Default - move piece	19	25
Get into goal	91	35
Escape possible beat	23	26
Standing on globe	64	51
Possible best opponent next round (Extra state)		20
Best score	**75**	**79**

Mean best score	**72**	**74**
Mean score	**61**	**63**
Standard deviation	**5.4**	**5.1**

Table 1. The best chromosome with the best score as well as the mean best score, overall mean and standard deviation for the GAs.

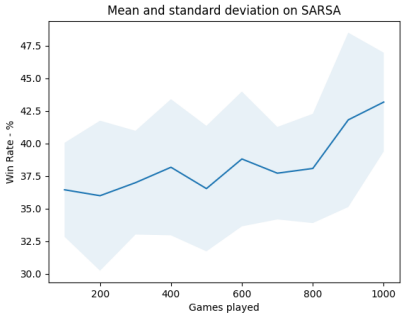

Fig. 6. The statistical plot with the mean and the surrounding standard deviation of the SARSA.

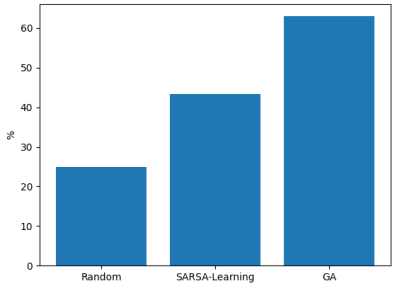

Fig. 7. Mean win-rate for the different AI approaches and a random player.

States / Actions	Move out	Normal	Goal	Star	Globe	Protect	Kill	Die	Goal Zone	Nothing
Home	7	0	3	3	5	9	3	1	6	4
Goal zone	9	0	18	4	3	10	17	7	2	9
Safe	7	6	3	1	-1	-1	0	0	28	2
Danger	7	5	3	6	13	9	3	0	18	4
Goal	9	6	2	4	8	4	3	2	1	4

Best win-rate	**46.3%**
Mean win-rate	**43.5%**
Standard deviation	**1.42**

Table 2. Q-table with the result of the SARSA provided by the fellow Troels Andreasen.

4 Analysis and Discussion

From the results it can be seen the first GA on the first generation outperforms the statically average win-rate of a random player with 25% due to it can be seen in figure 2 the lowest fitness score is 30% and the average over the first generation is 47%. This indicates the states implemented in the chromosome with the utility function is providing valuable information to the agent even then not trained.

The performance of the first and second iteration with the extra state is compared. The second iteration performs better. This can be seen in table 1 where it outperform the first iteration in every category. Therefore, it must be interpreted the extra state improves the state representation and increase the success of the GA. Thereby proving how crucial the state space representation is for the performance of the GA. However, the GA do not meet the success criteria of 85 wins, and terminates after 25 generations. The maximum amount of wins achieved is 79. If the performance of the GA should be further improved the state space representation should be the most critical part to refine. However, Ludo is a luck based game and the results is thereby influenced be chance. Therefore, it can be argued what is the maximum percentage of wins achievable.

The convergence time of the two GAs is around 4-5 generations as seen in figure 3 and 5. Afterwards the results only improve slightly. Thereby, it can be concluded the GA has learn to play Ludo to its best ability with the derived state space. Compared with a the first and second iteration with extra state. The second iteration achieves a more steady development of the mean generation score. Where opposite the first iteration the score is not as constant development. Furthermore, the second iteration also achieves a lower overall standard deviation as seen in table 1. Thereby, the second iteration achieves a more consistent success as well as higher due to a mean best score of 74.

Also seen in table 1 the weight of the best chromosome is presented. This chromosome achieves a win-rate of 79%. Both GAs have similar weighting of the

genes. Hitting stars, beating a opponent, entering goal straight and get into goal are important actions to win the game. The low value for hitting yourself home for both GAs can be explained by if this state is relevant no other states would apply and the utility value is the negative weight of this gene. Thereby given the brick a very low chance to be move regardless of the low gene value.

When the GA approach is compared to the SARSA. The GA outperform the SARSA with a best-win of 79% compared to SARSA 46.3% as seen in table 2. This may be due the reward for desirable states should be refined. As expressed in section 2.5 the rewards for these states are defined beforehand. Where the GA adaptively learns which states are more preferable per evolution. Furthermore, as proven previous the important of the state space representation can not be understated. The implemented state and actions in the SARSA may not be sufficient and more information is needed to improve the agent success. Compared to the GA 12 states are applied where SARSA only implement 5. These reasons may course the GA to perform better than the SARSA. Furthermore, the increased success while learning is lower for the SARSA it starts at around 36% and growth to 43.5% compared to the GA whit starts with 45% and growth to 63%. However, the performance of the SARSA is more consistent due to the lower standard deviation of 1.42 as seen in table 2.
Finally is the mean win-rate compared the two AI approaches and a random player as seen in figure 7. It shows both the AI agents are more successful compared to a random player. Thereby, have both AIs fulfilled their purpose and is a improvement of a random player. Furthermore, of the two AIs GA is a more successful implementation and is a better method to win in Ludo compared to the SARSA implementation.

5 Conclusion

It can be concluded the genetic algorithm provide a satisfying results with a best win-rate of 79% and an average best win-rate per generation of 74%. Furthermore, it can be concluded the learns to play Ludo with a convergence time of 4-5 generations. However, the GA do not meet the success criteria of 85% wins. If the the performance was to be improved the state space representation should be refined. Compared with a SARSA approach the GA perform better due to best win-rate for SARSA is 46.3%. Thereby, it can be concluded the GA is implemented successfully and is a better method to win in Ludo compared to the SARSA implementation.

Bibliography

[1] S Bhuvaneswari. "Man - Machine Interface". In: *International Journal of Artificial Intelligence & Applications* 3 (Jan. 2012), pp. 139–147. DOI: 10.5121/ijaia.2012.3111.

[2] Beakcheol Jang et al. "Q-Learning Algorithms: A Comprehensive Classification and Applications". In: *IEEE Access* PP (Sept. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2941229.

[3] August Mader. *AugustMader/Genetic-algorithm-Ludo-game-in-Python*. URL: https://github.com/AugustMader/Genetic-algorithm-Ludo-game-in-Python.git.