Ludo AI

Rasmus Haugaard

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
rahau15@student.sdu.dk

1 Introduction

As long as there have been games, there has been an interest in autonomous players (AIs). One way to avoid handcrafting AIs could be through Evolutionary Algorithms (EA) where a population of agents compete and undergo some combination of exploitation by natural selection and exploration by introducing variation. In this work, multiple Ludo players will be evolved through EA and compared to both random- and handcrafted players.

The provided Ludo game has been ported to python by Haukur Kristinsson `https://github.com/haukri/python-ludo`. I have continued the work to make it align with the C++ version and packaged it as a python package for modularity, `https://github.com/RasmusHaugaard/pyludo`. The motivation behind the re-implementation in Python was faster development.

The agent in this work is thus also implemented in python. No EA package is used. All methods are implemented using the python standard libraries and numpy. The implementation will be available at `https://github.com/RasmusHaugaard/pyludo-ai`.

2 Methods

The methods developed in this work are based on scoring the possible actions by a relative value function with some parameters, where the function is defined by the state representation, and the parameters are found with EA. More specifically, an agent policy, π, is defined by a value function, V, depending on the agent chromosome, θ, and the agent action representation, A, which can be dependent on the current state and the next state.

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, V(\theta, A(s, s'(s, a))) \tag{1}$$

, where s is the current state, a is the action, and s' is the next state defined by a known model function of s and a. Note that the next state is the immediate next state, before the next player makes it's move. Also, note that the state representation is then fully defined by the action representations for the four actions. Since only Equation 1 is examined, the full state representation will not be referred to, and for simplicity, the action representation $A(s, s'(s, a))$ will simply be denoted A from now on.

2.1 Action representations and Value functions

Three action representations and their corresponding value functions are explored in this work.

Simple The simple action representation, A_{sim} has four binary components. 1: Whether a player token was moved from home onto the board, 2: whether a player token entered the safe end zone, 3: whether a player token entered goal, and 4: whether an opponent token was sent home.

The corresponding value function is a chromosome-weighted sum of the action representation components:

$$V_{sim}(\theta, A_{sim}) = \sum_{i=1}^{4} \theta_i A_{sim,i} \tag{2}$$

Note that no bias parameters are needed, since

$$\underset{a}{\operatorname{argmax}} \sum_{i=1}^{4} \theta_i A_{sim,i} = \underset{a}{\operatorname{argmax}}\, b_5 + \sum_{i=1}^{4} \theta_i (A_{sim,i} + b_i)$$

Note also, that

$$\underset{a}{\operatorname{argmax}} \sum_{i=1}^{4} \theta_i A_{sim,i} = \underset{a}{\operatorname{argmax}} \sum_{i=1}^{4} c \cdot \theta_i A_{sim,i}, \quad c > 0$$

, and thus the parameters are normalized after each change, so that $\sum_i |\theta_i| = 4$. The normalization should make the parameter space unambiguous and the resulting chromosomes more comprehensible.

Reinforcement learning analogy In V-value iteration, the policy, π is ti choose the action that results in the highest expected value is chosen

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s,a) V'(s') \tag{3}$$

, where a is the action, s is the current state, s' is the next state, and P denotes the environment model, the probability of ending up in state s' given the current state and action, and V' denotes the expected return given a certain state.

In the case of Ludo, since the immediate state transition is deterministic, Equation 3 boils down to

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, V'(s'(s,a)) \tag{4}$$

, where s' is the known model function. Note that s' is not the next observable state like it would be in Reinforcement learning, but rather the immediate state after an action is taken, before the next player makes it's move. Also note that V' doesn't have to actually estimate the expected reward but should just be able to relatively score states.

The following two players only tries to relatively score the next immediate states, and does thus not look at the current state.

Advanced The advanced player assigns a value to each token for both the player and the opponents. Each token is given a score based on 4 properties. 1: Weather it is on the board (not home), 2: a normalized (from 0 to 1) distance it has traveled along the common path, 3: weather it is in the end zone, and 4: weather it is in the goal position. The value assigned to each property of the tokens is decided by the EA parameters, $\theta_1..\theta_4$, and are shared among the player and opponents. All token values are then weighted according to the probability of no opponent being able to hit the token home before the player's next turn. p is approximated by looking at how many, N, opponent tokens could possibly hit the token, and then setting $\hat{p} = (5/6)^N$. The three opponents summed token values are weighted per opponent by θ_5, θ_6, and θ_7 and subtracted from the summed player token values, which yields the final action score.

Full Both the *simple* and especially the *advanced* player requires a lot domain knowledge. The assumptions that the embedded domain knowledge make might actually inhibit the player performance. In many other problems than Ludo, obtaining this domain knowledge might be difficult. It is interesting to examine how well an action representation with no domain knowledge will be able to perform.

Both the *simple* and the *advanced* player only has between 4 and 7 genes which could very likely be optimized manually with no algorithm.

The *full* player has no embedded domain knowledge, only looks at the immediate next states s', and since it looks at a full state representation, there are many parameters.

A_{full} is a 4×59 array, describing for each of the four players, how many of it's tokens are in each position, where the token positions have been mapped from -1, 0 .. 56, 99 to 1, 2 .. 59. An action score is then determined by feeding A_{full} through a neural network with parameters found by EA. The network has one hidden layer of 100 neurons and one output neuron. Tanh is used as the hidden layer activation to enable non-linearity in the learned value estimate function. With a bias neuron added to the input, there are a total of $(4 * 59 + 1) * 100 + 100 * 1 = 23.800$ parameters or genes.

With back-propagation in machine learning it is desirable to have the neurons be in the active region of the activation function to avoid diminishing gradients. In the case of EA there are no gradients, but it is still desirable to have the neurons near the active region for changes in the parameters to affect the player policy. To be in the active region of tanh, the weights in a neural network layer before the activation would often be initialized with a std deviation of $\sqrt{1/N}$ [2016], where N is the amount of input neurons to the layer. As described later in subsection 2.2, for some of the methods in this work, constant mutation strengths are used, and thus, the genes should preferably be initialized in the same range for all representations. Instead of initializing the genes differently, the hidden layer outputs are multiplied by $\sqrt{1/N}$ before activation, which is equivalent but allows the genes to be in the same dynamic range as for the other representations.

2.2 Evolutionary Algorithms

All methods in this work are based on [2015].

The agents in this work are evolved playing against each other instead of fixed opponents, which means that instead of designing an AI to be good against a certain opponent or a certain group of opponents, the resulting AIs will have tried to explore and exploit it's own shortcomings, ideally making it more robust to opponents in general.

A few considerations has been done while choosing the EA methods. Ludo has a decent amount of randomness to it, which means that to be able to evaluate a player with low uncertainty, a lot of game runs are required. Searching the problem space through EA takes time, and thus ways to increase efficiency are of great interest.

Letting agents play against each other instead of against fixed opponents, required game runs are conveniently reduced by a factor of four.

The selection is done through different versions of Tournament Selection. Four agents from the population are chosen to play N games against each other. The two agents with the most wins are bred, and the offspring replaces the two lowest ranked agents. The choice of Tournament Selection is discussed in section 4. Most experiments were done with ten game runs per tournament. The population size is chosen to be a multiple of four, and a generation is defined to be when all agents have played in a tournament.

A problem of EA is the concept of modality. Depending on the problem, many modes in the search area might be good candidates for solutions, but in practice, the mode that shows best initial results often takes over the whole population. Different methods try to alleviate this. In this work, Cellular Tournament Selection and Island Tournament Selection are examined briefly.

Since many breeding- and mutation methods based on real-valued genes are discussed in [2015], real-valued genes are used in this work.

When two parent agents are selected in a tournament, the genes from the parents need to be recombined in some way during breeding. In this work Real Uniform Recombination and Real Whole Arithmetic Recombination are examined. Real Uniform Recombination only allows a new combination of the same genes, Real Whole Arithmetic Recombination allows to interpolate between the genes from the two parents, and Real Blend Recombination allows to sample genes that are out of the range between parent genes. Increasing new information also increases instability, but Blend Recombination allows to tune the exploration with an α, where $\alpha = 0$ corresponds to Real Whole Arithmetic Recombination.

According to [2015], the recommended way to mutate the offspring with real-valued genes is by sampling from a normal distribution around the current gene value. The mutation rate is set to 100 %, and variation is controlled by adjusting the standard deviation, σ, of the normal distribution from where the mutations are drawn. In this work, a Real Normal Mutation, where a user specified σ is chosen is examined. A question arises of how to choose sigma. Also, it might make sense to change sigma over time, e.g. lowering it while approaching a

local maximum. For this reason, two other Mutation methods are examined, namely the Real Adaptive One Step Mutation, where σ is stored inside the agent chromosome during EA, so that the EA itself finds an appropriate σ. Realizing that different genes might have different appropriate σ's, the Real Adaptive N Step Mutation is also examined, where a σ per gene is stored within the chromosome. The adaptive methods only have one or two learning rates that are set by the user and define how quickly the sigma can change. The learning rates are all set to 0.1 in this work.

The populations need to be initialized. All genes are initially sampled from a standard normal distribution. All adaptive sigmas are initially sampled from a standard lognormal distribution.

2.3 Live population evaluation

It is desirable to be able to plot the performance of populations as they are training to guide method combinations and hyper-parameters. The agents in this work are evolved with self-play, so the win rates during EA can only be used as relative performance. To obtain live absolute performances for the populations, each agent in a population is played against three random players for a certain number of games. Limited by compute power, the number of games is chosen to be 100. The worst case std deviation for each observed win rate is $\sigma_{agent} = 0.05$. (See **??** for the statistics). Because of this rather large std deviation, if the individual agent win rates are plotted, they should be taken as an indication rather than the real win rate. The most sample efficient way to get reliable population means would be to play games uniformly distributed between the population agents. This would give a worst case std deviation of $\sigma_{pop} \approx 0.011$, but then it would not be possible to say anything about the distribution of win rates in the population. Instead, independently of the population size, win rates for 20 randomly chosen agents in a population is sampled and the population win rate is found by the mean of the sampled agent win rates. Though this mean estimate has a higher variation, it allows an indication of the population std deviation.

2.4 Player evaluation

The live plots discussed in subsection 2.3 are used to guide the evolution of the *simple-*, *advanced-* and *full* player discussed in subsection 2.1.

Based on the plots for different hyper parameters and methods, the best population is chosen for a given player. A recursive population reduction is used to choose the best agent from the population. For each reduction level, elimination tournaments are played with 500 games per tournament where only the best agent in a tournament survives and pass on to the next reduction level. If the population size is not a multiple of four, the remaining agents are automatically passed on to the next reduction level in the start of the population to make sure, they will enter a tournament next. In the end, if there are two or three agents left, the remaining spots in the tournament are filled up by random players, and

the best non-random player is chosen. The final agents represent the players and are used for player comparisons in section 3.

2.5 Fellow student method

This reports findings will be compared to Christian Quist Nielsen's. Christian also scores each action and then takes the action with the highest score. His action representation consists of 6 components. Normalized player game completeness after the action, weather the token was moved onto the board from home, weather an opponent was knocked home, weather the token moves into a position where it could potentially hit an opponent the next turn if the opponent doesn't move, weather the token moves from an unsafe position to a position where an opponent can not knock it home during the next turn, and weather the token moves to an unsafe position. The action components are multiplied element-wise with the 6 genes and summed for each action.

The agents found in this report will be compared to two agents from Christian. One was evolved playing against three random players, and one was evolved playing against three 'smart' players, which choose to move a token onto the board if possible, or else takes the action where the summed player tokens minus the summed opponent tokens is highest.

Christian's initial population is sampled from a standard normal distribution, he uses tournament selection with a tournament size of four, and plays 100 games against random players per player per tournament, so in total 400 games per tournament. The two best players of a tournament is recombined by uniform crossover, and the offspring is mutated by adding samples from a standard normal distribution.

3 Results

The players are compared in tournaments of 2500 games, which result in a win rate worst case std deviation of $\sigma = \sqrt{0.5(1-0.5)/2500} = 0.01$. When comparing which of two players are best in Table 1, they go into a tournament with two instances of each player. When obtaining a score for an agent in Table 2, it enters a tournament with three opponent agents. Like for all tournaments in this work, the player order is randomized for every game. The statistical significance of the results presented in the tables will be discussed in section 4.

4 Analysis and Discussion

First, there will be a statistic perspective of the results, second the players, the evolutionary algorithm methods and the results will be discussed. This work covers many players and methods why all combinations and findings cannot be covered.

player\opp.	random	smart	chr-random	chr-smart	simple	advanced	full
random	50.4*	15.2	17.1	15.6	20.1	7.8	7.1
smart	84.8	49.7*	50.3*	44.0	54.3	30.8	33.4
chr-random	82.9	49.7*	49.6*	45.7	47.0	26.7	28.0
chr-smart	84.4	56.0	54.3	51.0*	56.2	33.4	34.1
simple	79.9	45.7	53.0	43.8	50.8*	33.2	26.8
advanced	92.2	**69.2**	**73.3**	**66.6**	66.8	50.4*	**45.8**
full	**92.9**	66.6	72.0	65.9	**73.2**	**54.2**	50.3*

Table 1: Win rates in percent for players (rows) against opponents (columns). The win rates are determined by 2500 game tournaments with two instances of both the player and the opponent. The best player for a given opponent is marked in bold. *Results that are statistically insignificant at a significance level of 5 %. The diagonal is kept as validation of the evaluation method.

player\opp.	random	smart
random	24.8*	6.3
smart	62.9	25.2*
chr-random	61.1	23.8*
chr-smart	66.3	27.0
simple	56.6	21.4
advanced	**78.2**	**44.0**
full	77.4	41.4

Table 2: Win rates in percent for player (rows) against opponents (columns). The win rates are determined by having one instance of the player and three of the opponent in a tournament of 2500 games. *Results that are statistically insignificant at a significance level of 5 %.

Statistical significance Considering the sampled win rate for a player, each game outcome, weather the player wins or not, can be thought of as being drawn from a binary random variable with $\mu = p$, where p is the true player win rate. According to the Central Limit Theorem, the average of a sequence of independent random variables drawn from the same distribution is normally distributed when the sequence size, n, is large. The estimated win rate is thus approximately normally distributed $\hat{p} \sim N(p, \sigma^2),\ \sigma^2 = p(1-p)/n$. The estimated variance is $s^2 = \hat{p}(1-\hat{p})/n$, and two single sided tests can be done to test weather the player is significantly better or worse than the opponent. The null hypothesis is that $p = 0.5$ or $p = 0.25$ for Table 1 and Table 2, respectively. The normalized observed difference from the null hypothesis, z, is:

$$z = \frac{\hat{p} - 0.5}{s} \quad \vee \quad z = \frac{\hat{p} - 0.25}{s}$$

With a critical region of 95%, the z-thresholds are ± 1.645. By solving for $\hat{p}$, the results in Table 1 are statistically significant, if $\hat{p} \leq 48.3\% \ \vee \ 51.7\% \leq \hat{p}$, and the results in Table 2 are statistically if $\hat{p} \leq 23.6\% \ \vee \ 26.5\% \leq \hat{p}$. The results that are statistically insignificant has been marked with a star for convenience.

Chosen player populations The agent chosen to represent the *simple* player is from generation 100 in Figure 1 with $sigma = 0.1$. Generation 20 might look slightly better. A z-test reveals the difference is not significant with 5% significance level. The agent chosen to represent the *advanced* player is from generation 55 in Figure 2. The agent chosen to represent the *full* player is from the last generation of the population with size 160 in Figure 3a.

Mutation experiments In Figure 1 three tournament evolution processes are plotted for the simple player with no recombination but normal mutation with varying σ. It is apparent how the mean win rate of the evolution with $\sigma = 1$ is lower. This could suggest that adaptive mutation as described in subsection 2.2 would perform more stable.

A test was done with full player tournament evolution processes, but neither one-step nor n-step adaptive mutation with a learning rate of 0.1 showed any improvement over normal mutation with $\sigma = 0.1$. All the processes were done with a population size of 40, 10 games per tournament, whole arithmetic recombination and were run for 1000 generations.

Recombination experiment Figure 4 shows genes of the *simple* player during evolution process, and since they are normalized as discussed in subsection 2.1, the genes are comprehensible. All four end-populations perform similarly. Note that since the *simple* representation is binary, more gene combinations might perform equally, which could explain the difference in the end agents. Notice how the column with no recombination doesn't explore the space very uniformly. Uniform does better exploration than no recombination. Whole recombination

seems to sample the space evenly, but the end population has converged tightly, which could hint that it would be more prone to settling on a local maximum for harder problems. Blend recombination seems to accommodate the issues with both diversity and even search.

Figure 5 shows a recombination test on the full player. No apparently significant difference was seen between the recombination methods. Letting them run for longer and on bigger populations could potentially have showed differences.

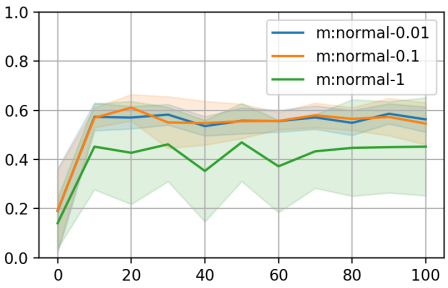

Fig. 1: Evolving the simple player with tournament selection, population size of 20, 10 games per tournament, no recombination and normal mutation with σ equal to 0.01, 0.1 and 1 respectively. Generation number along the first axis and win rate along the second axis. Population mean and std deviation is plotted.

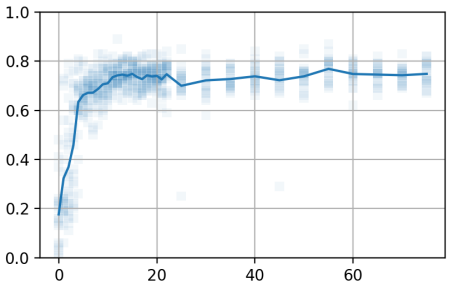

Fig. 2: Evolution of the advanced player with tournament selection, population size of 80, 10 games per tournament, whole arithmetic recombination, and normal mutation with $\sigma = 0.1$. Generation number along the first axis and win rate along the second axis. Agent win rates are scatter plotted. Population mean is plotted.

The Tournament Selection The scalability of tournaments what was inspired the choice of tournaments, but since this was run on a single laptop and only few tests were done with parallel, Fitness Selection or Rank Selection would probably have worked at least as well.

5 Conclusion

Three players have been evolved using different evolutionary methods. One of them has no embedded domain knowledge but is actually better than the player with the most domain knowledge winning 54.2% of 2500 Ludo Games with two instances of each player. The player with most domain knowledge wins 78.2% against random players, while the player with no domain knowledge wins 77.4% against random.

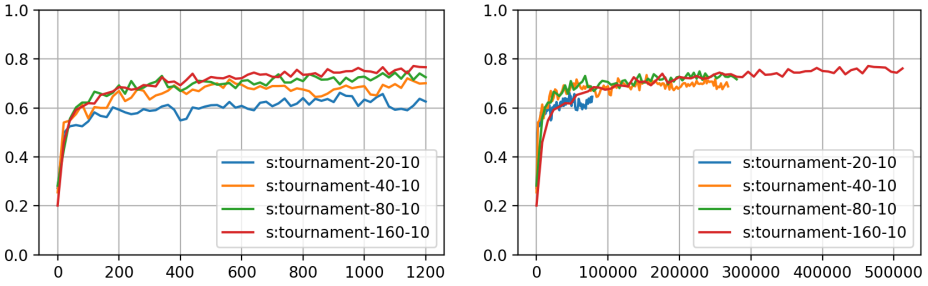

Fig. 3: Population mean win rates while evolving the *full* player with tournament selection, 10 games per tournament, whole recombination, normal mutation with $\sigma = 0.1$, and varying population sizes from 20 to 160. a) Generation number along the first axis. b) Total number of games played along the first axis.

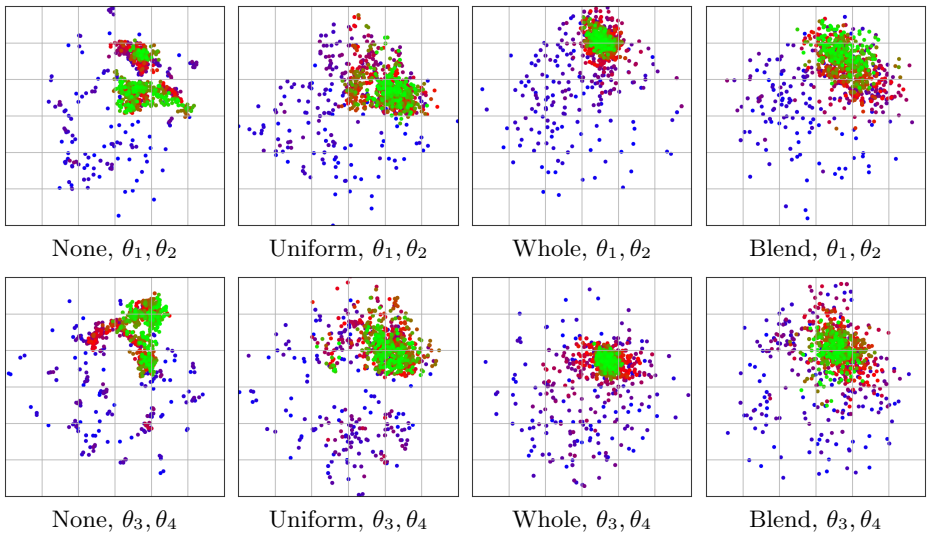

Fig. 4: Simple player population genes during 20 generation evolution processes with normal mutation, $\sigma = 0.1$ for no recombination, uniform recombination, whole recombination and blend recombination. Genes are plotted with a color range, representing the progress through the evolution process, going from blue, through red to green. Grid size is one. Origos are in the center of the plots.

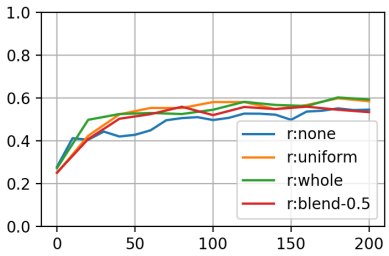

Fig. 5: Evolving the full player with tournament selection, population size of 20, 10 games per tournament, normal mutation with σ equal to 0.01, and different recombination methods. Generation number along the first axis and win rate along the second axis. Population mean is plotted.

6 Acknowledgements

Thanks to Haukur Kristinsson for the work on the python version of ludo and to Christian Quist Nielsen for letting me use his evolved players in this report.

References

Eiben, A. E. and Smith, James E.: Introduction to Evolutionary Computing. Springer Publishing Company, Incorporated. 978-3-662-44874-8 (2015)

Ian Goodfellow and Yoshua Bengio and Aaron Courville: Deep Learning. MIT Press. `http://www.deeplearningbook.org` (2016)