APPLIED REINFORCEMENT LEARNING WITH USE OF GENETIC ALGORITHMS

How an agent learns to play the Ludo game

Daryosh Derakhshan
The Mærsk Mc-Kinney Moller Institute (MMMI), University of Southern Denmark (SDU), Odense, Denmark
dader@student.sdu.dk

Keywords: Artificial Intelligence, Reinforcement Learning, Q-learning, Genetic Algorithms, Ludo playing Agent.

Abstract: **This paper will present a method to apply reinforcement learning on Ludo playing agents. The reinforcement learning method will be the Q-learning algorithm. Moreover the paper will present a key concept in this implementation called the parenting recipe. The parenting recipe is a complete recipe for which kind of rewards and punishments an agent should get, when making a particular move. Experiments has shown that different recipes creates different agents with different skills. This paper will therefore also present a genetic algorithm to optimize the parenting recipe, and the paper will also present experiments which conclude that the method works as intended.**

1 INTRODUCTION

This paper will present a strategy for teaching clueless Ludo players the game of Ludo by applying the techniques of the learning paradigms in artificial intelligence. Many different learning paradigms are present, such as supervised learning, e.g Artificial Neural Networks (ANN's) and unsupervised learning in terms of self organizing networks, e.g. Kohonen Maps or Growing Neural Gas. Another learning paradigm is the reinforcement learning paradigm, which is quite different from the supervised learning paradigm. In reinforcement learning, unlike in supervised learning, the agents are not told which outcomes are desired. Instead they take action and get feedback from the environment, which makes them learn from their experience. This is much like how children are raised. Parents cannot make their children do a certain action in a given situation, the only thing they can do, is to provide them with proper feedback for the actions they have made. If the actions are not well behaved, the parents will most likely punish them, and if the actions are well behaved, the children will be rewarded. This kind of parenting is used to form a well behaved child according to the parents' goals. Whether the child is well behaved depends if it takes advantage of the experience it has, or it wants to explore unvisited territory. This is known as exploitation vs. exploration. As the child grows up, the feedback from its parents will tend to weight less as it did when the child was younger, and the child will not learn as fast as it once did. These social aspects of parenting, can be directly mapped to the reinforcement learning of artificial agents. The reinforcement learning technique that will be presented in this paper is the Q-learning algorithm with delayed rewards (Watkins, 1989). The agent cannot always get an immediate reward for the action it has made, since the potential consequences will not be revealed until a number of rounds has passed. Because of this social aspect of raising ludo players as they were children, parents will be able to inspect the different children they have raised to compare their social success in reference to how the were brought up. In this way, parents can learn when and how much to punish and when and how much to reward the children in different situations. If their child does not succeed in society, the parenting has most likely not been appropriate. The parents can then try to raise a new child by different parenting methods, and see whether the child succeeds more in life. This introduces how to combine genetic algorithms with reinforcement learning (Cline, 2004) for optimizing what I call the parenting recipe - how to raise a child to be successful.

These techniques will be used in the vision of raising an intelligent Ludo player, that will learn the game as it plays and ultimately beats a Semi-Smart Player (SSP), which is the goal for the project.

2 THE REINFORCEMENT LEARNING METOD

The reinforcement learning method that will be used in this paper is the Q-learning (Watkins, 1989). The agent creates a Q-map of all the states S it can be situated in, and all the actions a it is able to take in that specific state. For each action in that specific state, a Q-value is attached. The Q-value $Q(s,a)$ is an estimate of the value of future rewards, if the agent takes that particular action a. For now, consider a state as being the position of all the bricks. Even though there are many different combinations of how the bricks can be placed on the board, it is still a finite state problem. The agent will then use its Q-map to determine which action is optimal in a given state, and unless the agent is exploring, it will choose the action which produces the highest Q-value. According to (Leslie Pack Keabling, 1996) the reinforcement learning model can be said to consist of:

- a discrete set of environment states, S;
- a discrete set of agent actions, A; and
- a set of reinforcement signals.

How this has been applied to the Ludo game will be described in more detail in later sections. First some aspects of reinforcement learning must be fully understood.

2.1 The Markov Property

Although both transitions and rewards may be probabilistic, they depend only on the current state and the current action. That means that there is no dependence on previous states, actions, or rewards. This is the *Markov property*. This property is very important: it means that the current state of the game is all the information that is needed to choose which action is the best. Because of the knowledge of the current state, it is unnecessary to know about the system's past. Any process can be modeled as a Markov process if the state-space is made detailed enough to ensure that the description of the current state captures those aspects of the world that are relevant to predict state-transitions and rewards. However there are two types of Markov processes which considers deterministic and nondeterministic environments. Deterministic environments is that the same action in the same state on two different occasions results in the same state-transition and the same reinforcement signal. A nondeterministic environment is then that the same action in the same state on two different occasions may result in different state-transitions and/or reinforcement signals. Whether this environment is deterministic or nondeterministic will be determined in section 2.6.

2.2 Policies

A *policy* is a mapping from state to actions. In other words, a policy defines a rule for deciding what action to take when knowledge is given about the current state. A policy should be defined over the entire state-space: the policy should specify what to do in any situation. A policy that specifies the same action each time a state is visited is termed a *stationary policy*. A policy that specifies that an action be independently chosen from the same probability distribution over the possible actions every time a state is visited is termed a *stochastic policy*. However during learning, the learner's behavior will change. The behavior will neither be stationary nor stochastic, but the optimal policies the learner seek to construct will be stationary.

2.3 Return

The aim of the agent is to maximize the rewards it receives. The agent does not wish to maximize the immediate reward in the current state, but wishes to maximize the rewards it will receive over a period of time. That is, it wishes to make a sequence of good moves rather than making one good move. There are three main methods of assessing future rewards: total reward, average reward, and total discounted reward. The total discounted rewards has been chosen, since it is the simplest case. The total discounted reward from t is defined to be:

$$r_1 + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n} + \cdots \qquad (1)$$

where r_k is the rewards received at time k. γ is termed the *discount factor* and is a number between 0 and 1, usually slightly less than 1. The total discounted reward is called the return. The purpose of γ is to determine the current value of future rewards. If γ is set to 0, a reward at time $t+1$ is worth nothing at time t, and the return is the same as the immediate reward, that is no future rewards is taken into account. If γ is set to be slightly less than one, the weight of future rewards will become less the longer in the future it is represented. In this way, is it possible to weigh next turn's rewards higher than the future reward that is

given after 10 turns. Nevertheless, for any value of γ strictly less than one, the value of future rewards will eventually become negligible.

2.4 The Credit-Assignment Problem

It is not obvious how to compute the optimal policy. The problem is that some clever moves now, may enable high rewards to be achieved later; each of a sequence of actions may be essential to achieving a reward. In the Ludo game an action may be to move a brick in an area where many enemies are gathered, and the chances of being hit home is great. Maybe the brick will not get hit home right away, but after maybe three turns, it would be wrong to blame the decision taken immediately before the brick was hit home, for these decisions may have been the best that could be taken in the circumstances. The actual mistake may have been made when the brick entered the enemy-zone, if other bricks could be moved without taking any chances. Because of this difficulty of determining which decisions were right and which were wrong, it may be difficult to decide which changes should be made in the policy. This problem of assigning credit or blame to one of a set of interaction decisions is known as the 'credit assignment problem'. This problem can to some extend be avoided in the Ludo Game. If a brick gets hit home, it must either be because the brick moved near an enemy, or an enemy moved near the brick afterwards. To avoid potential danger, the agent could be punished if it moves a brick near the enemy, if there are other bricks that can be moved risk free. However this means that all potential dangerous situations, which could lead to a brick is hit home, must be determined.

2.5 The State-Space Representation

Recall that the reinforcement learning model consists of a discrete set of environment states S. This state-space representation must be determined. Many different state representations can be made, e.g. a state can include the position of every brick on the board. This would however create a huge and complex state-space representation, since the positions must be represented somehow. To keep the representation as simple as possible, a brick's state could simply be represented by a binary format, where each index said something true/false about the brick's possibilities. Depending on how many indexes that are wanted, it is able to keep the state-space representation quite simple. Just like in (Sæderup, 2010), there are 9 different pieces of information for each brick that is interesting to take into account when the die has been rolled. These informations are:

- the brick can get to goal
- the brick is able to hit an enemy home
- the brick is able to hit itself home
- the brick can move to a star
- the brick can move to a globe
- the brick can get out of home
- the brick can move near an enemy
- the brick is currently in the safe zone
- the brick can get into the safe zone

These 9 elements will be each brick's state. Notice that all the elements can either be true or false, which keeps the representation simple. The state is represented by a binary format, e.g. 010000100. In this case the brick is able to hit an enemy home, due to 1in the second index, but the brick will at the same time move near an enemy, due to the 7th. index, meaning that the brick will risk being hit home the next round. However an agent has 4 bricks, and which brick to move can be a hard choice. A good agent must take all the bricks into account, before choosing the best brick to move. Thus the state representation must be extended to a 9 times 4 binary bit combination, that is a 36 binary bit combination, where the bits 1-9 represents the first brick's internal state, 10-18 represents the second brick's internal state etc. In this way, one environmental state represents the four brick's internal state, which allows the agent to take all four bricks into account in a given state. Notice that the current state of the environment, does not represent how the bricks are placed at the current moment, but what possibilities each brick has when the die has been rolled.

2.6 The Q-map

Every agent must build a Q-map where every state combination that is possible is saved. The Q-map is simply constructed by playing millions of games, so every new state is saved to the map. For each state in the map, there is a set of actions A. These actions are simply represented by the four different bricks. When the agent is situated in some state, the agent will always have a maximum of four moveable bricks. Sometimes even less, if e.g. three out of four bricks are at the goal, the only option is the last brick, since the others are not movable. Figure 1 illustrates an example of how the Q-map is represented. For each state in the environment, there is attached a Q-value for every brick. The agent's job is to choose the best action in a given state. However whether the action that was taken, was a good or a bad move, will not

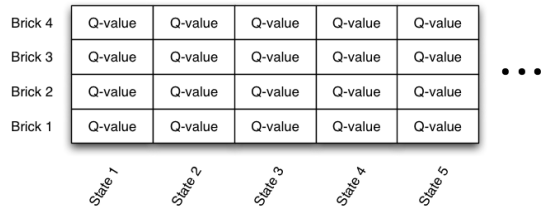

Figure 1: The Q-map of the Ludo player

always be visible until later. The agent can choose to move a certain brick, that get hit home the round after, which clearly indicates that the agent made a bad move. The agent will therefore get what is called *delayed rewards*: only when the agent gets another turn, it can evaluate upon the recent move, and the "parents" will reward it or punish it for the recent move. The agent will then update its belief system (Q-map), for that specific state it was situated in and the bricked that it chose to move. One might think that the Q-map's state-space would get incredibly huge, since a state is a 36 binary bit combination, which would result in 2^{36} different combinations. However many of the states are combinations are not possible, e.g. the brick cannot both move to a star, goal and globe at the same time. This is one of many states that are not valid, so the amount of states will not even be close to 2^{36}. Calculations and experiments has shown that the approximate state-space size will be near 100.000 different states. This means that there will approximately be near 400.000 Q-values in the entire map. Recall that the Q-values is an estimate of the value of future rewards, if the agent takes a particular action *a*. With this Q-map representation, the environment will be nondeterministic, that is taking the same action in the same state on two different occasions may result in different state-transitions and/or different reinforcement values in terms of rewards. An example would be if the agent chooses to move a brick near an enemy, risking it to be hit home the following round. The enemy did not have any luck hitting the brick home the first round and the move the agent made, perhaps gets a reward since it is closer to the goal than before. However the enemy could easily rolled the die to its benefit, and the brick would be hit home, resulting the agent to get a punishment. Figure 2 shows an illustration of the described example. In state 1 the best action is to move brick 4, so the agent takes that action and moves near the enemy. The enemy is currently 2 squares behind the brick, and if the enemy rolls 2, it is able to hit the brick home. However the enemy hits 3 and moves 1 square ahead of the brick. The agent then receives a delayed reward

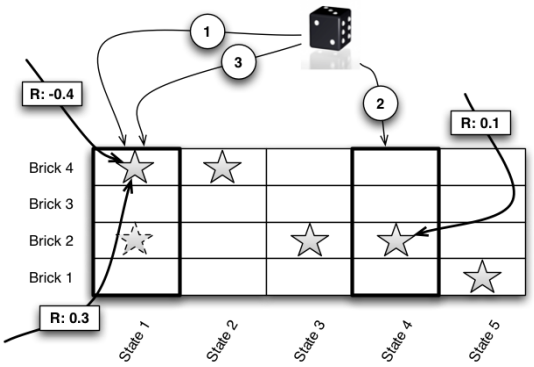

Figure 2: Example showing a deterministic state-transition but a nondeterministic reward

(R: 0.3) since it was not hit home last round. When the die has been rolled again, the agent ends up in state 5. In state 5 the best brick to move is brick 2, and the agent takes that action. After the die has been rolled again the agent ends up in state 1 once again. Still the best brick to move is brick 4, even though the brick gets near an enemy again and risking being hit home. The agent moves brick 4 again and takes the risk of being hit home. Unfortunately the brick gets hit home by an enemy, and the agent this time receives a punishment (R: -0.4) for the same action it received a reward from. Notice that it is not the action made by the agent, that determines which state is the next, but the next state transition depends both on the die, the action and the enemies' moves. Recall that each state only represents the different possibilities each of the bricks have, and what possibilities one brick has, depends on the die, the current state, and what move the enemy made last turn. Maybe due to the punishment the agent got for moving brick 4 in state 1, it can result in a new best action e.g. to move brick 2 in state 1. Some states will therefore have stationary policies and others will have stochastic. That the state-transition function includes the enemies' moves indirectly, the method has potential to adapt to different enemies' strategies, however this has not been studied further.

2.7 The Parenting Recipe

The agent that is playing the Ludo game for the first time can be considered as a new born child. The agent can not distinguish right from wrong nor danger from safe. It does not know whether it is a good thing to be hit home, or if it is a bad thing to get to goal. An agent that is as clueless as this, will have all zeroes in its Q-map. The agent is born with the Q-map where

all the valid states are present, that means that it already has a map of all the situations it can get itself into. However, as mentioned, all the Q-values will be zero, hence the agent will not know what to do when situated in a particular state. Recall that the main difference between supervised learning and reinforcement learning is, that in supervised learning there is a presentation of input/output paris, and the agent then learns to produce that output when that specific input is given. In reinforcement learning the agent learns from the rewards it gets from the environment - in this case that is the parent. The agent is not told which action would have been in its best long-term interest. After all the parent does not know which action is the best, it can simply just give the child a reward or a punishment depending on which action it takes. The parent's job is now to discipline the child, to learn it right from wrong. To learn by punishments and rewards, seem to be a good idea, however how does the child separate from two bad moves which it got punished for? It simply gets two different levels of punishments, so it can distinguish bad from worse. In the same way an agent should maybe get a higher reward from move which hits an enemy home than from a move that got a brick to a globe. However the parent does not know which move should get the highest reward, and after all which of those moves are best, maybe depends on the type of player the agent is playing against. A good parent with good parenting skills, knows exactly how much to punish its child when doing something wrong, and how much to reward them when doing something right. As mentioned the level of punishment and reward depends on the specific state the agent is situated in. This aspect of leveling the punishments and rewards I call the *parenting recipe*. A parenting recipe exactly explains how much reward a child should get if e.g. it hits an enemy home, and how big of a punishment it should get, if the child makes a move, that causes a brick in the following round to be hit home. Obviously different parenting recipes will create different Q-maps, which is the belief system of the agent. Agents with different Q-maps will obviously play the game differently, because they might chose different actions if situated in the same state. Therefore is the parenting recipe crucial for making a child that knows how to play the game, since it is a recipe for parents to follow when raising a child.

The parenting recipe is represented as a 10-bit array with float values. The values in the parenting recipe includes:

- the learning rate
- the discount factor
- a reward when the action did not result in a brick got hit home the following round
- a punishment when the action did result in a brick got hit home the following round
- a reward when the action hit an enemy home
- a punishment when the action moved a brick near an enemy, and risking being hit home in the near future
- a reward when the action moved a brick to a safe zone
- a reward when the action moved a brick to the goal
- a reward when the action moved a brick to a star
- a reward when the action moved a brick to a globe

These values define how a parent raises its child. Notice that two of the values are defined as a punishment, where six are defined as being a reward. These definitions are statically defined, since it is never wanted to give a reward if the agent takes an action which results in a brick being hit home. Similarly it is never wanted to give a punishment if the agent moves a brick to the goal. The only dilemma is how big the reward in that case should be - it should however never result in a punishment. Therefore every punishment is defined as being between $[-1;0]$ and every reward is defined as being between $[0;1]$. The two first elements, are the discount factor γ and the learning rate α. One might wonder why this is part of the parenting recipe. The rewards and punishments in the parenting recipes, is the immediate reward, however the parent want to return the total discounted reward. Recall that the discount factor γ determines the current value of future rewards. That means that when the discount factor changes, so will the total discounted reward that is given to the agent when making a move. Two different agents with an equal set of rewards and punishments, but a different discount factor, might take different actions, since they take future rewards into account differently. An agent with a zero discount factor will get a reward that is equal to the immediate reward, where an agent with a high discount factor might take a different move because it can see that it will lead to a dead end. The learning rate in the parenting recipe, can be seen as the ability for the parent to teach its child. If the learning rate is too low, the child will learn very slowly, and if the learning rate is too high, then the child will never be sure what action might be the best one, sine it's Q-values wont stabilize. Therefore all these parameters are crucial for which kind of child the parent will end up raising.

2.8 The Q-learning Algorithm

The child that is playing the Ludo game for the first time, must build its Q-map up with help from the rewards it receives from the environment (parent). The Q-learning algorithm has three parameters:

- the learning rate α
- the discount factor γ
- the greedy parameter ε

The learning rate α and the discount factor γ has already been described. The greedy parameter ε is a probability of taking a non-greedy (exploratory) action in greedy action selection methods. A non-zero value of ε insures that all state/action pairs will be explored as the number of trials goes to infinity. In a greedy method, $\varepsilon = 0$ the algorithm might miss optimal solutions, however, in a near greedy method, a value of ε that is too high, will cause agents to waste time exploring suboptimal actions that have already been visited. The Q-learning rule is

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max Q\left(s',a'\right) - Q(s,a)\right) \quad (2)$$

Where $maxQ(s',a')$ is the maximum Q-value of the state that the agent got into due to the action a. This maximum Q-value is weighted by the discount factor γ and the reward r is added. Hereafter the Q-value of the action that was taken is subtracted to that value to and the entire expression is multiplied by the learning rate α. This value is then added to the Q-value of the action that was taken. If each action is executed in each state an infinite number of times on an infinite run and α is decayed appropriately, the Q-values will converge to the optimal Q-values according to (Leslie Pack Keabling, 1996). When the Q-values are nearly converged to their optimal values, it is appropriate for the agent to act greedily, taking in each state the action with the highest Q-value. However, during learning, there is a difficult exploitation vs. exploration trade-off to be made. When the agent is learning by getting rewards by the parent recipe, the greedy parameter ε is set to 0, forcing the agent to pick an action by random. This is done, since it does not really matter if the agent win the games while it learns. The important thing is for the agent to explore every possible actions in every state an infinite number of times, and not favor any actions in the process. Once the states have been visited enough times, and the Q-values have converged, ε can be set to 1, forcing the agent to choose the best action it has learned in the specific state. Alternatively some randomized strategies could have been used, if needed, such as the *Boltzmann exploration*. The Q-learning rule given in equation 2 with the described algorithm has been proven to converge for deterministic Markov processes, and indicated by (Cline, 2004), it sometimes converges in environments that is not deterministic. There is a nondeterministic version of the Q-learning which is guaranteed to converge in systems where the reward and state transitions are given in a nondeterministic fashion, as long as they are based on reasonable probability distributions. The equation for the nondeterministic algorithm is presented in (Cline, 2004). The deterministic Q-learning rule has been applied, and this might result in that some nondeterministic state-transitions and Q-values will never converge, as is sometimes will generate punishments, and other times rewards. This is obviously one of the big problems of the Q-learning algorithm, that it sometimes have nondeterministic state-transitions. Another problem with the Q-learning algorithm is the state representation. The state-space often tend to get huge, and the quality of the representation if hard to keep high.

3 THE GENETIC ALGORITHM

Even though the Q-learning algorithm alone will be able to develop an agent that is able to beat a SSP, it will, as discussed earlier, depend on which parenting recipe the parent raises the child with. As we will see in section 4, it clearly makes a difference which parenting recipe is applied to the child that has to learn the game. This section will present a genetic evolving algorithm, which will seek to optimize the way the parent raises a child - that is optimizing the parenting recipe.

As mentioned, different parenting recipes will create different agents, but will the same parenting recipe create the same agent? For now, we assume it will. An experiment will be conducted in section 4 to illustrate this. If all it is needed to reproduce the same agent, is the parenting recipe, then the recipe will be considered as the agent's *chromosome* (Ross, 1999). A chromosome is composed of *genes*. Each gene has a *locus*, that is the position within the chromosome, and these can take any set of values. The locus' possible values are called *alleles*. As described in section 2.7, the parenting recipe consists of 10 elements, which are the genes, e.g. the final gene in the chromosome is the gene that defines the reward an agent gets if a brick is moved to a globe. Where this gene is placed in the chromosome is defined by the index (locus), and the set of values each gene can take is also described in section 2.7, i.e. a punishments cannot be above zero and a reward cannot be beneath zero.

To fully understand the algorithm used, one must un-

derstand the operations such as:

- the mutation of a chromosome
- the breeding/mating of pairs of chromosomes

How these operations are implemented, is crucial for the success of the algorithm, as we will see in the following sections.

3.1 Mutating a Chromosome

When mutating a chromosome, it can be done in many different variations. First one must define how many genes of the chromosome a mutation has to mutate. Can it be all of the genes, only one of the genes, or perhaps a random amount. Hereafter one must define which genes to mutate - should it be the first half of the genes, or should it be picked by random? Finally one must define how much a gene should be mutated - how much can the genes value change? So there are three parameters that should be defined:

1. the amount of genes to mutate
2. which genes to mutate
3. how much to mutate each chosen gene

In the implementation the values has chosen to be:

The amount of genes	40% of the genes
Genes to mutate	Random gene
Mutation rate	$\pm[0.1;0.3]$

Table 1: The mutation specifications chosen

To give an example, consider table 2. The first chromosome is the parenting recipe that needs to be mutated. The second chromosome is the mutated chromosome. As illustrated, the fourth gene has decreased a bit, and so has the fifth gene. However the ninth and tenth gene has increased a bit. So four out of ten genes has been mutated within the acceptable mutation rate. Whether the values are increased or decreased is random.

0.41, 0.34, 0.0, -0.12, 0.73, -0.15, 0.67, 0.48, 0.6, 0.09
0.41, 0.34, 0.0, -0.23, 0.57, -0.15, 0.67, 0.48, 0.72, 0.25

Table 2: An example of a chromosome that has been mutated

The purpose of a mutation like this, is to create a slightly different parenting recipe from the original. Often mutation is used in combination with mating a pair of chromosomes. However it can be used independently, but this has not been done in this implementation.

3.2 Mating Pairs of Chromosomes

When two humans make a child, the child will get a set of properties from one parent, and get another set of properties from the other parent. Fortunately, that is not the only process happening when producing a child. In the same time, the child gets mutated a bit which results in that the child gets its own fingerprints, size etc. By mating pairs of parenting recipes, this aspect of mating is respected. Mating is done either by a process called *crossover* applied with probability p_c, or by copying of the parents with probability $(1-p_c)$ (Ross, 1999). Crossover comes in various flavors, but the simplest one is the *single point cross over*. This type of crossover simply take a random point between the start and the end of the chromosome, and everything left from that point in parent 1's chromosome is copied to the child, and every genes right from that point in parent 2's chromosome is copied to the child. To complete the mating of the two chromosome, the new chromosome is mutated as described in section 3.1. To give an example, consider table 3

0.91 0.55 0.13 -0.14 0.15 -0.16 0.17 0.18 0.19 0.11
0.51 0.82 0.23 -0.24 0.25 **-0.26 0.27 0.28 0.29 0.31**
0.91 0.55 0.13 -0.14 0.15 -0.26 0.27 0.28 0.29 0.31
1.0 0.71 0.13 -0.28 0.15 -0.26 0.09 0.28 0.29 0.31

Table 3: An example of a mating of a pair of chromosomes, to produce a chromosome of a child

The first row represents parent 1's chromosome, the second row represents parent 2's chromosome, the third row represents the child's chromosome after the crossover, and finally the fourth row represents the child's final chromosome after mutation. The single point has been chosen by random, and obviously that point is located in the middle. That means that the first half of parent 1's chromosome is copied to the child, the second half of parent 2's chromosome is copied to the child. This state is represented by the third row in table 3. To complete the mate, a mutation is done on the child's chromosome, and the result and the final chromosome produced by mating, is represented in the last row. A new parenting recipe has now been created by mating two other recipes. This operation is the key process of the genetic algorithm.

3.3 The Evolutionary Process

As described earlier, the purpose of the genetic algorithm is to optimize the parenting recipe, in hopes of creating an agent that is able to beat the SSP. Whether the parenting recipe is a good or a bad one, is determined by the *fitness function*. The fitness function

in this case, is simply the amount of points the agent gets by playing the game. If an agent out of four, wins 10.000 games by points, that agent must have the best parenting recipe. The implemented evolutionary process is illustrated in figure 3. The steps are the following:

1. One agent is created by a custom made parenting recipe. This recipe is then mutated three times to create a population of four agents. Every agent must be raised (trained) a according to their parenting recipe.
2. Let the population of 4 agents play against each other.
3. Let the two best agents of the population mate with each other to create a new parenting recipe.
4. A new child is created, and must be raised according to the new parenting recipe.
5. Let the child learn the game from scratch by playing an appropriate amount of games.
6. The new child, which has finished its learning, replaces the worse of the recent population of 4 agents, and the process continues from step 2.

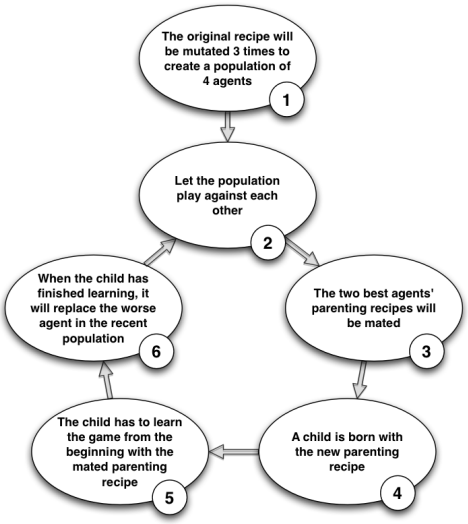

Figure 3: The steps of the evolutionary process

This process could have been designed in thousand other ways, however it has to be taken into account that is has to be a simple process, since the learning process of a new child is very time consuming. Alternatively, an island model could have been used, where different populations compete against each other internally, where at some point the best of each island, visits another island to compete with a foreign population. Other variations exists, where the total population is replaced by children, however this is not a solid solution, since it is not known whether the child is better than the parent. Therefore the chosen model is the *elitist* generation-based GA (Ross, 1999), where the best agents remain, to compete with their children. Once the children becomes better than their parents, they will start producing children of their own. If not, they will be replaced by a new set of children. This concept is taken directly out of evolutionary darwinism - the survival of the fittest.
Another important thing to notice is that this evolutionary process continues for all eternity, unless a stopping criteria has been defined. Since the goal is to beat the SSP, the stopping criteria will satisfied when the agent beats the SSP with a certain amount of points. This results in that every time the population has ranked its own agents, the best of them will compete against the SSP, and if it wins with an acceptable amount of points, the evolutionary process may be stopped.

4 EXPERIMENTS

This section will provide different experiments along with tests to analyze the implementation. This section will conduct four different experiments, and the results will be discussed. The four experiments will try to answer the following:

1. How many games does a child need to play in order for it to be considered trained?
2. Till now, we have assumed that the same parenting recipe is all that is needed to reproduce a child, but is that really the case?
3. Does the evolutionary process find a better parenting recipe, than the custom parenting recipe created?
4. Which agent, between the one that is referred to in this paper, and the one referred to in (Sæderup, 2010) is the best Ludo player?

Some of the experiments, has been made with 1 vs. 3 players with 10.000 rounds of Ludo, that is why the points get between 15.000-20.000 points in the different charts. Other experiments has been made with 2 vs. 2 players playing 10 matches with 1000 rounds each; points between (3200 - 3400) where each teams' points are added together.

4.1 The Number Of Training Games

This experiment will test how many games it takes for an agent to stabilize its points gained when competing against SSP's. The agent is born with an empty

map, only the states are present, but the Q-values are all zero. The agent must then compete against three SSP's. The result is illustrated in figure 4.

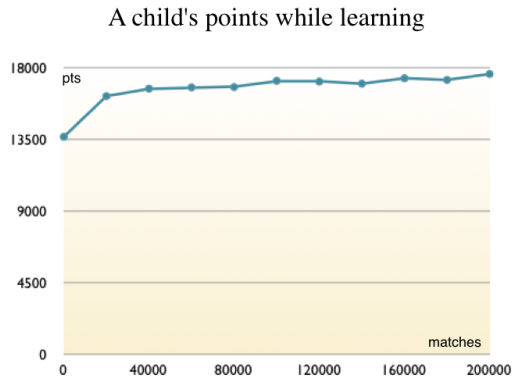

Figure 4: The learning of a child as it plays more games

A totally clueless agent will always try the move the same brick if it is possible. This is however not a bad strategy, but not enough to beat a SSP. As illustrated in figure 4 the first 20.000 games creates the fundament of the decisions the agent makes. Afterwards the curve's slope is decreased, and the agent's Q-map has converged around 160.000-200.000 games. Because of this result, the evolutionary process uses 200.000 games to raise a new child.

4.2 Identical Parenting Recipes

As described earlier it is obvious that different parenting recipes creates different agents. However that does not result in that the same recipe then creates the same agent. Two different agents are raised by the same parenting recipe, and after training in 200.000 games, they both compete with three SSP's in 10 matches with 1000 rounds each. Figure 5 illustrates the points that each of the children got. It seems to be quite similar. To clarify whether these two datasets come from the same population, an unpaired t-test must be conducted. The purpose of an unpaired t-test is to see whether two samples could have been drawn from populations with equal means (Cohen, 1995). The null and alternative hypothesis for the unpaired t-test are:
$H_0 : \mu_1 = \mu_2 \Rightarrow$ the agents are similar
$H_1 : \mu_1 \neq \mu_2 \Rightarrow$ the agents are not similar
We use a 95% confidence-interval, and the t-value has been calculated to be 1.7415, and since t is less than 2.262 (Cohen, 1995), the null hypothesis cannot be rejected, meaning that these two samples most likely come from a population with equal means. Therefore

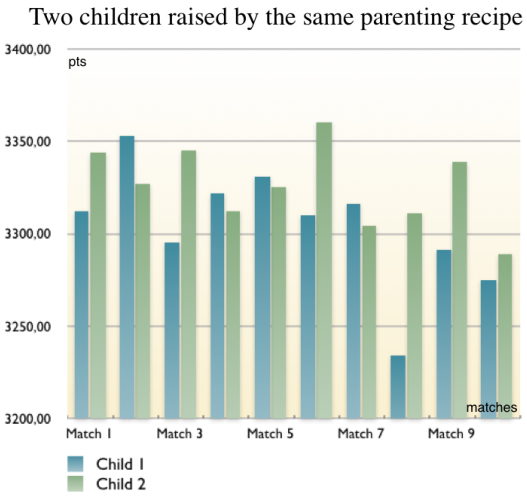

Figure 5: The results of two children of the same parenting recipe competing against 3 SSP's

it can with 95% certainty be concluded that a child can be reproduced by use of the parenting recipe.

4.3 The Winner Of The Evolutionary Process

When running the evolutionary process, one does not know whether a better parenting recipe will come along, and how long it takes for it to get to that state.

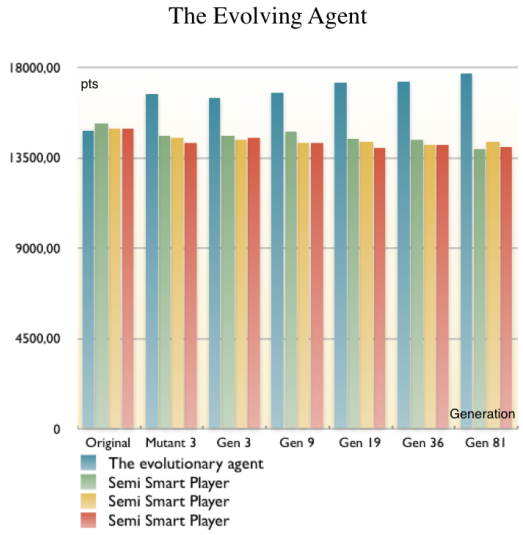

Figure 6: The results of two children of the same parenting recipe competing against 3 SSP's

The evolutionary process begun with the original cus-

tom made parenting recipe. The result of its competition against three other SSP's is illustrated in picture 6. For the first round, a unpaired t-test has yet again been conducted, since it looks like a close call and the results showed that it with 95% certainty cannot be confirmed that the original agent plays better than the 3 SSP's. As the generations evolve, the agent gets better and better, and ends in generation 81, which has taken over 20 hours to produce. Since this result is more than acceptable, that the SSP's are not even close at beating the agent, the evolutionary process has been stopped. As seen by the chart, it can confidently be concluded that the child after generation 81, beats the three SSP's when playing 10.000 rounds.

0.41, 0.34, 0.0, -0.12, 0.73, -0.15, 0.67, 0.48, 0.6, 0.09

Table 4: The parenting recipe of generation 81, which got the best results in the experiment

For convenience the resulting parenting recipe after more than 20 hours, and 81 generations is presented in the above table.

4.4 Reinforcement Learning Player Vs. Neural Network Player

As the last experiment, it was interesting to compete between two completely different agents. One made by reinforcement learning with GA optimization, and another agent made by ANN with GA optimization. As figure 7 illustrates, it is not even a close call. The agent that has been developed with ANN+GA beats the RL+GA agent by many points repeatedly. A reasonable explanation for this defeat, is that the evolutionary process was stopped, since its goal had been reached. Maybe if the process had been running for a couple of weeks, a new parenting recipe would have been generated that might beat the ANN+GA agent if trained against it. Again a unpaired t-test has been conducted, which concluded with a 99.5% confidence interval, that the ANN+GA agent is better, which does not come as a surprise.

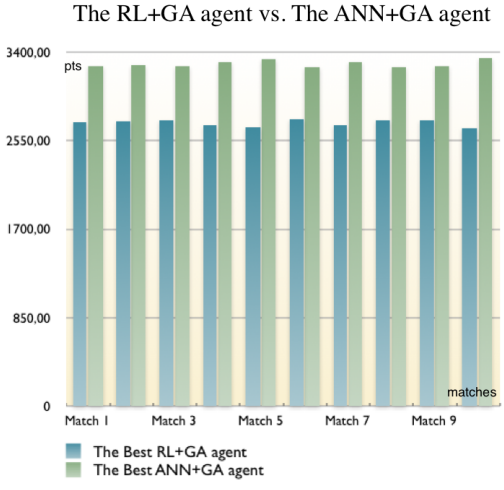

Figure 7: Competition result of two agents developed by two different methods

5 CONCLUSION

A method for developing an agent with the methods of reinforcement learning has been presented. How the Q-learning algorithm has been implemented has been described. Moreover an genetic algorithm has been developed on top of the reinforcement learning, to optimize the parenting recipes used when raising a new child. Experiments has been conducted, which clearly indicated that both the reinforcement learning model and the genetic algorithm worked quite well, even though the process has been time consuming. The goal was to create a reinforcement learning agent which could beat a Semi Smart Player, and as experiments show, this goal has with great satisfaction been accomplished.

ACKNOWLEDGEMENTS

I would like to thank Kasper Sæderup, that has, along with myself, been developing both the RL+GA and the ANN+GA agents. I am happy we got to develop two agents with two radically different methods. Moreover we are happy to have come at a 3^{rd} place in the tournament with the ANN+GA agent.

REFERENCES

Cline, B. E. (2004). Tuning q-learning parameters with a genetic algorithm.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence*. The MIT Press.

Leslie Pack Keabling, Michaeal L. Littman, A. W. M. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research 4*.

Ross, P. (1999). Neural networks: an introduction. Technical report, AI Aplications Institute, Univeristy of Edinburgh.

Sæderup, K. (2010). An evolving ludo player based on neural networks, backpropagation and ga.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College.