This article is written by students. It may have omissions and imperfections, reported as far as possible by our reviewers in the editing notes.

The machine to play the sticks game

Students:

Petru Săveanu, 11^{th} grade
Matei Coldea, 11^{th} grade
George Iorga, 11^{th} grade
Petra Hedeșiu, 10^{th} grade
Ștefan Oprean, 10^{th} grade
Matei Bojan, 10^{th}grade

Coordinator Teachers:

Ariana Văcărețu

Researcher:

George Țurcaș, Lecturer at Babeș-Bolyai University

School:

Colegiul Național "Emil Racoviță" Cluj-Napoca, Romania

This article is written by students. It may have omissions and imperfections, reported as far as possible by our reviewers in the editing notes.

Contents

1 Research Topic **3**

2 Introduction **3**

3 Simulating the game **4**
3.1 The simulation of the game 4
3.2 Obtained results 4

4 Computer Science approach **5**
4.1 Determination of the winning strategy 5
4.1.1 The Minimax algorithm 6
4.1.2 Alpha-Beta pruning and Minimax algorithm 6
4.1.3 Complexity 8
4.1.4 Conclusions 8
4.2 The Monte Carlo Simulation Algorithm 8
4.2.1 Complexity 9

5 Probabilistic approach **9**
5.1 Poisson Distribution 9
5.2 Recurrences 10

6 Conclusion **11**

1 Research Topic

The game of sticks is simple: two players play with 8 sticks, each in turn removing one or two sticks, the one who takes the last one wins.
We have a machine with 8 cups numbered from 1 to 8. We initialize the machine with two balls of each color (yellow/red) in each cup except in cup 1 with only two yellow balls.

We suggest you play several games against a machine as follows:
- The machine starts.
- When it is the machine's turn to play, it draws a ball at random from the cup corresponding to the number of sticks left in the game. If the cup is empty, two yellow and two red balls are placed in the cup (except in cup n°1, where only two yellow balls are placed).
- If the drawn ball is yellow it removes one stick, otherwise two.
- The drawn balls are placed in front of the corresponding cup.

At the end of a game, if the machine has won, it is "rewarded" by putting the played balls back into the corresponding cups, and for each ball that is put back, a ball of the same color is added to the cup. If the machine loses, it is "punished" by removing the balls it has played.

Our research focuses on determining how many games it takes for the machine to always win. After completing the task, we programmed a new machine for altered game parameters, such as the number of initial sticks, while also analyzing the outcomes.

2 Introduction

The problem of game simulation, particularly where a machine learns to play against a human, is of significant importance in the field of artificial intelligence and machine learning. This type of research is crucial as it explores the capabilities of machines to learn from past decisions, prioritize successful strategies, and penalize unsuccessful ones, ultimately leading to the development of an unbeatable strategy. These simulations aim to capture the complexities and dynamics of real-world games, providing a way for machines to learn, adapt, and improve their performance through repeated interactions.

This learning process, often referred to as reinforcement learning [1], is a key aspect of artificial intelligence. It enables machines to interact with their environment and learn to make optimal decisions over time. This approach involves training machines by providing them with feedback in the form of rewards or penalties based on their actions and outcomes. Over time, through trial and error, the machine can learn to associate certain actions with favorable outcomes and adjust its strategy accordingly. The development of a machine that can consistently win a game against a human player demonstrates the potential of artificial intelligence to master complex tasks and adapt to new situations.

Research in this area has broad implications beyond games. For instance, the same principles can be applied to develop self-driving cars that learn to nav-

igate roads safely, financial systems that adapt to market changes, or healthcare algorithms that improve patient outcomes based on past medical data.

3 Simulating the game

3.1 The simulation of the game

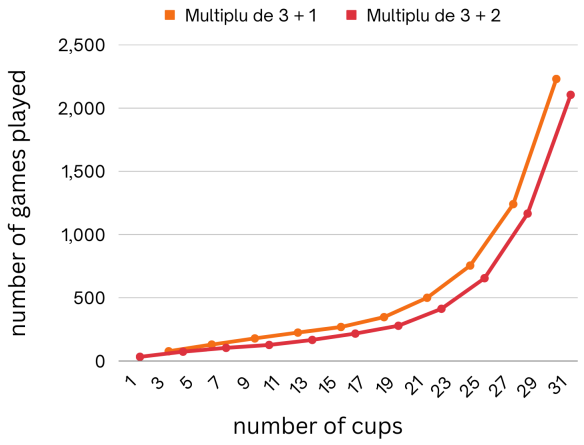
Our algorithm answers the problem for any number of cups in the game. We found the optimal strategy for the player, and in this way, the machine maximizes its chances of winning, "learning" from the player by adding the played balls back into the cups, in case of a win, and removing the played balls, in case of a loss. We consider that the machine will always win when there is only one type of ball left in all the cups it extracts from. This program runs 50000 times, calculating the average number of games after which the machine always wins.

The player's strategy: j[] = {1, 2, 1, 1, 2, 0, 1, 2, 0,. . . }
1 - takes a yellow ball (one stick)
2 - takes a red ball (two sticks)
0 - takes a random ball

Observation. If n is a multiple of 3, the player has a winning strategy. No matter what number of balls the machine takes out, the player will always extract 1 or 2 balls such that the remaining number of balls is a multiple of three. This means that for any ball extracted, the player will go to one of the cups with an exact strategy, causing the machine to fall, once again, in a multiple of 3 cup.

3.2 Obtained results

As explained before, when the initial number of sticks is a multiple of 3, the player has a winning strategy. Therefore, we obtain two graphs: corresponding to the two possible values $n \equiv 1$ or $n \equiv 2 \pmod 3$.



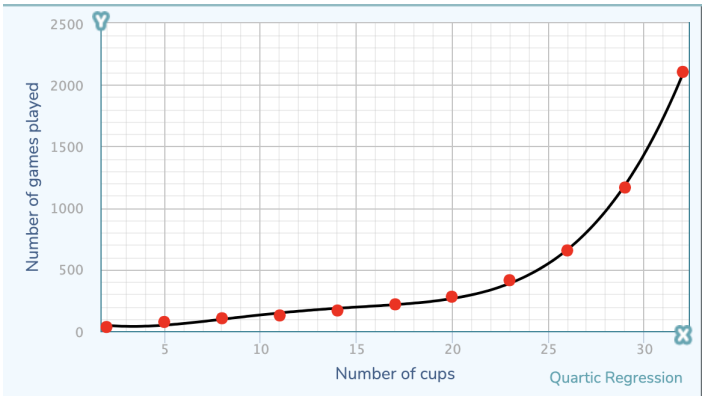
Using a curve fitting algorithm we noticed the polynomial quartic regression fits best our values. The two obtained functions are:

- Multiple of 3 + 1:

$$y = 232.14 - 79.56 * x + 13.59 * x^2 - 0.77 * x^3 + 0.01 * x^4$$

- Multiple of 3 + 2:

$$y = 101.20 - 43.63 * x + 9.17 * x^2 - 0.57 * x^3 + 0.01 * x^4$$



In determining the results, the precision with which it is calculated is essential.

4 Computer Science approach

4.1 Determination of the winning strategy

We are constructing a binary tree in which the left branch represents taking one stick and the right branch represents taking two sticks. This tree is commonly referred to as a game tree.[2] Beginning with a starting point of n unclaimed sticks, the machine takes the first move. By considering all possible choices, we expand the tree.

As the machine and the human take turns, the even levels will describe all possible moves for the human while the odd levels will describe all possible moves for the machine at a certain point. The tree is fully constructed until all moves on every branch have been exhausted. Terminal nodes are marked with either 1 or -1, based on the outcome they represent.

A win for the machine is marked with 1 and a loss is marked with -1. Therefore, the terminal node is on an even level it will be marked with -1, otherwise, it will be marked with 1.

By applying the Minimax algorithm [3] on the constructed game tree, we can decide whether or not the machine has a winning strategy and, we can compute, the number of move the machine has to do in order to win.

4.1.1 The Minimax algorithm

In this section, we give a brief presentation of the Minimax algorithm, an algorithm that can be used to determine the best move for the player in decision-making and game-theory [4]. It is commonly used in two-player, turn-based games such as Tic-Tac-Toe, Chess, and more.

The algorithm considers two players, known as the maximizer and minimizer, where the maximizer aims to achieve the highest possible score, while the minimizer aims to achieve the lowest possible score.

Every state of the game, or board state has a value associated with it. If the maximizer has the advantage in a given state, the score of the board will tend to be positive, while if the minimizer has the advantage, it will tend to be negative. These values are determined by heuristics specific to each type of game.

Code:

```
// scores[] stores leaves of Game tree.
// h is maximum height of Game tree

int minimax(int depth, int nodeIndex, bool isMax)
{
// terminating condition: leaf node is reached
if (depth == h)
    return scores[nodeIndex];
// If current move is maximizer, find the maximum attainable value
if (isMax){
    return max(minimax(depth + 1, nodeIndex * 2, false),
    minimax(depth + 1, nodeIndex * 2 + 1, false));
}
// If current move is minimizer, find the minimum attainable value
else{
    return min(minimax(depth + 1, nodeIndex * 2, true),
    minimax(depth + 1, nodeIndex * 2 + 1, true));
}
}
```

If the minmax function returns 1, then the machine can win, otherwise the human always wins.

The use of this function in the Driver code looks like this:
minimax(0, 0, true, scores, h) - starting from depth 0 and node Index 0 and with the maximizer to move

4.1.2 Alpha-Beta pruning and Minimax algorithm

Alpha-Beta pruning is a method for optimizing the minimax algorithm by greatly reducing the computation time. This optimization allows for faster search and the ability to delve deeper into the game tree. [5]

It eliminates branches that do not need to be searched because a better move has already been identified. The technique is named for the two parameters it

adds to the minimax function: alpha and beta.

Alpha represents the highest value that the maximizer can currently ensure at that level or higher, while beta represents the lowest value that the minimizer can currently ensure at that level or lower. [6]

Code:

```
int minimax(int depth, int nodeIndex,bool maximizingPlayer, int alpha,int beta)
{

// Terminating condition leaf node is reached
if (depth == h)
    return scores[nodeIndex];

if (maximizingPlayer)
{
int best = MIN;

// Recur for left and right children
for (int i = 0; i < 2; i++)
{
int val = minimax(depth + 1, nodeIndex * 2 + i, false, alpha, beta);
best = max(best, val);
alpha = max(alpha, best);

// Alpha Beta Pruning
if (beta <= alpha)
    break;
}
return best;
}
else
{
int best = MAX;

// Recur for left and right children
for (int i = 0; i < 2; i++)
{
    int val = minimax(depth + 1, nodeIndex * 2 + i, true, alpha, beta);

    best = min(best, val);
    beta = min(beta, best);

// Alpha Beta Pruning
if (beta <= alpha)
    break;

}

return best;
}
```

```
}
```

4.1.3 Complexity

The minimax algorithm with alpha-beta pruning has a time complexity of $O(\frac{b^d}{2})$, where b is the branching factor and d is the maximum depth of the search tree. By pruning irrelevant branches, alpha-beta pruning significantly reduces the number of nodes evaluated, enhancing the algorithm's efficiency. The degree of improvement varies based on factors such as game tree structure, move ordering, and game-specific aspects. In optimal scenarios, alpha-beta pruning can eliminate around 50% of nodes at each level. This technique empowers the algorithm to delve deeper into the game tree, evaluate fewer nodes, and achieve faster and more effective decision-making. In the realm of game theory and artificial intelligence, alpha-beta pruning is considered a vital optimization technique [7].

4.1.4 Conclusions

Based on the game tree we can determine the best move any player can make in a given position. Knowing this, we can determine all the optimal moves that either the man or the machine can make when given a certain position. We will make use of the lists of all the optimal moves the two entities can make, later.

4.2 The Monte Carlo Simulation Algorithm

The Monte Carlo Simulation algorithm [8] is a method for solving problems using simulations. The basic idea behind the algorithm is to simulate many possible scenarios, track the outcomes, and then use the collected data to make predictions or decisions.

In the case of the game of sticks, the Monte Carlo algorithm can be used to determine the average number of games it takes for the machine to always win. [9] The algorithm would involve the following steps:

1. Initialize the machine with the given parameters (8 sticks, 8 cups with two yellow and two red balls in each cup except cup 1 which has only two yellow balls).

2.Run a large number of simulations of the game, where the machine takes a random ball from the cup corresponding to the number of sticks left in the game and removes either one or two sticks based on the color of the ball.

3.Track the machine's performance in each simulation and keep track of the number of wins and losses.

4.Analyze the results of the simulations to determine the average number of games it takes for the machine to always win. The data collected from the simulations can be used to calculate the average number of games the machine needs to play to win, the percentage of wins, the percentage of losses

If we want to program a new machine with different parameters, such as a different number of sticks or a different number of sticks that can be removed in each turn, the process would be similar. The algorithm would need to be adapted to the new parameters and the simulations would need to be run again to track the machine's performance and determine the average number of games it takes for the machine to always win.

It's important to note that the Monte Carlo algorithm is an approximate method and the results will get closer to the real value as the number of simulations increases. More simulations will give more accurate results but it will also be computationally expensive.

4.2.1 Complexity

The time complexity of the algorithm is dependent on the number of iterations performed in the Monte Carlo simulation, denoted as num_iterations. Considering a constant number of sticks and cups, the simulate _game() function, representing a single game simulation, can be considered a constant time operation. The monte_carlo_simulation() function calls simulate_game() num_iterations times, resulting in an overall time complexity of approximately O(num_iterations). The space complexity is constant, as the algorithm primarily utilizes fixed-size arrays. The analysis highlights the scalability and efficiency of the algorithm in estimating outcomes and informs decision-making in game scenarios [10].

5 Probabilistic approach

5.1 Poisson Distribution

A Poisson distribution [11] is a tool that helps to predict the probability of certain events happening when you know how often the event has occurred. It gives us the probability of a given number of events happening in a fixed interval of time. [12]

Bad ball = balls that according to the array of optimal moves calculated with the minimax algorithm, do not represent the optimal move.

Example: From position 7, if the machine draws a yellow ball, it would place itself in a winning position, therefore the yellow ball is a good ball for position 7 (if a type of ball is not good according to the calculations of this type, it is generally considered a bad ball)

Based on the values calculated through the game simulation program(described above), we will calculate the average number of bad balls we have in the system after x games. Having this in mind, we can calculate the probability of having z bad balls after x games, using the formula:

$$P(x; z) = \frac{(e^{-y(x)} \cdot y(x)^z)}{z!}$$

y(x) - means of bad balls (calculated using the Monte Carlo algorithm), the average number of balls remaining in the system after x games

We know that once the system is out of bad balls, it can not get any new bad balls, therefore if the system gets rid of all the bad balls in game k, then it will be out of bad balls for the remaining games. To calculate the possibility of having got rid of all bad balls by game x, we can make an average of P(k,0), where k is an integer from 0 to x.

5.2 Recurrences

First, we will calculate the expected number of rounds when the game has only 2 cups. We will consider 3 arrays of numbers which represent the probability that after n rounds there will be 0,1 or 2 wrong balls. We can calculate these arrays using previous members. We will notate N0, N1, and N2(there are still 2 bad balls). Using probability theory we can obtain:
N0[n]= N1[n-1]/(n+1)
N1[n]= N1[n-1]*(n)/(n+1)+N2[n-1]*2/(n+3)
N2[n]= N2[n-1]*(n+1)/(n+3)

The expected value is equal to x where

$$x = \sum_{i=1}^{\infty} N0[i] \cdot i$$

$$\sum_{i=1}^{\infty} N0[i] = 1$$

N0[i] converges to 0 faster than i to infinity, so

$$N0[i] \cdot i$$

converges to 0 so x is rational.

Using a computer we obtain the average number of games for only 2 cups which is about 33. Now, we will consider that in every game the second cup is cleared after 35 rounds to solve the case with 4 cups. Because we still have to make 2 correct choices, the cases with 4 or 5 cups have the same result. The idea of recurrences is the same but we have to change the number of balls left in the cup. From the first 35 balls we will double 33 and lose the other 2, then we will double every good ball. So, we can say that we get 33/35 balls per turn. After calculating the average precisely as in the previous case, multiply by 5, add 39, and divide by 6 to obtain the real average because in 1/6 cases we have to refill the cup because we lost all the balls after 4 rounds, but this means that we cleared the second cup so we have 35 more games. We must take into account that no rounds end in less than 37 games(35 games for the second cup, and 2 for the red balls from the fourth cup), so we will multiply by 37 instead of n for every probability from N0 up to 37.

Until now, I obtained 3 results. It needs 33 rounds to finish the game if the machine starts from the second cup, and about 75 if the machine starts from the fourth or fifth cup and 104 for 8 cups.

6 Conclusion

After putting together all the solutions it is observed that:

1. The expected value of laps increases exponentially with the increase in the number of cups
2. In practice, the person may win almost one hundred percent every round but there are still bad balls
3. A large number of simulations is required to obtain a good precision of the result

References

[1] P. Winder, *Reinforcement Learning: Industrial Applications of Intelligent Agents*. 2020.

[2] T. h. Cormen, C. E. Leiserson, R. L. Rivest, and C. stein, *"Introduction to Algorithms"*. 2009.

[3] M. Eppes, "Game theory — the minimax algorithm explained."

[4] R. E. Korf and D. M. Chickering, *Best-first minimax search*, ch. 1.2. 1996.

[5] G. L. Team, "Alpha beta pruning in ai," *Great Learning Blog*.

[6] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning,," *Great Learning Blog*.

[7] J. Allis, "Searching for solutions in games and artificial intelligence.," 1994.

[8] A. B. Owen, *Handbook of Computational Statistics*. 2008.

[9] I. research, *What is Monte Carlo simulation?* 2017.

[10] R. Y. Rubinstein and D. P. Kroese, *Probability and Statistics*. 2016.

[11] G. Mihoc and N. Micu, *Elements of probabilistic and statistics*. 1979.

[12] J. Tsitsiklis, "Part iii: Random processes - definition of the poisson process," *MITOpenCourseWare*.