SOS game

Narine Marutyan, Sona Hakobyan, Elen Petrosyan

November 12, 2022

Introduction

What is SOS?

SOS is a game that can be played with a pencil and paper. It is quite similar to tic-tac-toe but can have a larger grid and players in each turn can choose between “S” and“O”. Thus, the game has a much larger complexity than tic-tac-toe. While tic-tac-toe can become too predictable, SOS is guaranteed to keep players thinking about their future actions. The goal of this game is to cover the board with as many “S-O-S” formations as possible, and each formed SOS will increase the score of that particular player by 1. In SOS, there are little to no limitations for the board design. The uniqueness is in its connection with timing: the bigger the board, the longer the game will last.

Overview of rules

- ➢ Board size is n by n
- ➢ Number of players is two
- ➢ Number of turns is equal to n^2.
- ➢ During each turn the players choose between “S” and “O”
- ➢ A player gets a point if he created an S-O-S combination during his turn (either vertically, horizontally or diagonally).
- ➢ You receive a turn after each successful goal! (The likelihood of your opponent winning decreases as you accumulate more turns. More SOS sequences will be connected as a result by the agent, leaving the opponent with fewer streaks to deal with.)
- ➢ The game ends after the whole grid is filled.
- ➢ The winning player is the one who got the most points.

Nature of the problem

SOS is a multiplayer game, but when it is played by precisely two players it is considered a combinatorial problem. One side benefits and the other suffers an equivalent loss as a result that’s why it is called a zero-sum game. In other words, the net increase in the game's benefit is zero because when player one has advantage the player two at the same time carries the same amount of loss. Also, it is considered a sequential game with perfect information. In order to be clear, a sequential game entails a situation in which one player picks an action before the others do. To represent these kinds of games controlled by the time axis, they usually apply decision trees. Every player has complete knowledge of all prior occurrences, including the "initialization event" of the game, when making any decision. This is called “Perfect information”, which in its turn is a synonym for “fully observable”.

Game Implementation

The purpose of our project is to create a game where different agents will compete against each other playing a game called SOS. We have three agents in mind: an agent who`s algorithm is going to be MINIMAX, a greedy agent with some modifications and a human agent. Our agent will determine what action to take by considering all viable answers in a certain number of tree levels and selecting the best one. Three algorithms will be applied to evaluate the performance of our players. We hope to create a program which never loses to a random agent and a pure greedy agent. Also, when playing with a human player, the greedy agent will end up with a draw if the grid is 3x3, and is going to outplay the human player if the grid is larger. The empty 9x9 square grid is the starting state, which describes how the game is laid in the start. Putting a "S" or a "O" into an empty cell is an action. The transition model specifies the state that proceeds from a player's activity. When the grid is filled with letters, the terminal test is true. The utility function decides who won and who lost by defining the final scores of both players.

Literature Review

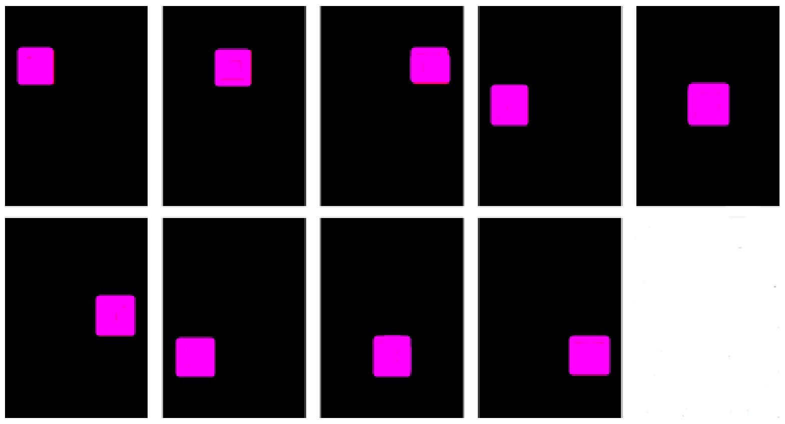
As the literature for the “SOS” game is not wide, we have decided to go through the papers and algorithms related to “Tic-tac-toe” as the logic of these two games is similar. The authors of one of the publications determined to address the tic-tac-toe problem using single-pixel imaging (SPI), which is essentially changing a two-dimensional pixelated sensor with a single-pixel detector and pattern illuminations into a single-pixel detector and pattern illuminations. (Jiao et al, 2022) To better explain how that works they introduced the instance of a simple camera’s pixelated sensor. The pixelated sensor array of a typical camera records a single image of a two-dimensional object, while in SPI,the object scene is lit in succession by several projected structured light patterns, and the total light intensity is recorded each time by a single-pixel detector. The recorded light intensity is determined by the inner product of an encoded illumination pattern and the object picture, according to mathematics. Afterwards, the object image may be computationally recreated using the series of single-pixel intensity data and the lighting pattern data. SPI is a physical application of optically based compressive sensing that has recently been widely employed with cutting-edge deep learning algorithms, with a primary focus on imaging and vision.

In the following manner, an SPI system may be customized as a Tic-Tac-Toe player.
To begin, the system must identify the game condition at any time using nine pattern illuminations. There is a printed game board with nine gray color squares in a grid. The human opponent player and the SPI player are represented by black and white square paper cards, respectively. If a player occupies an empty square with his or her sign, the initial gray hue will be

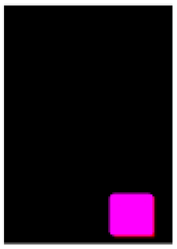
overlaid by black or white by connecting the paper card to the board. The projector will display nine successive light patterns during the detecting step. Only the region corresponding to one of the nine squares on the board is lighted in each design. The single-pixel detector can record nine related single-pixel intensity values. The intensity value can indicate whether the color of each square is black (highest light absorption and minimum light reflection), gray (mid light absorption and reflection), or white (lowest light absorption and greatest light reflection). As a result, the game state, represented by a nine-element vector, may be derived. Each square in the grid will have a vector element with a value of 1 (filled by human player), 2 (blank), or 3 (filled by SPI player).
Moreover, with minimum digital computing, the system can produce the ideal action for the next move depending on the current game situation. The vector has a total of 39=19683 possibilities in theory. In reality, the amount of white symbols vs black symbols is 0 or 1, because two players take turns placing one sign at a time. As a result, only part of the 19683 vector value combinations are permitted under this limitation, which may occur in real games. For all valid game states, the ideal action is made through the Minimax algorithm. It can be used to optimize the next move in advance. Furthermore, it will be determined if the human player wins in the current game state and whether the SPI player wins after the current decided move. The already computed outcomes for all conceivable game states are then saved in a lookup table. When the game is played online, the sole digital processing step is the extraction of results from the lookup table depending on the discovered game state.
Lastly, pattern lighting is used to represent the SPI player's activity as well as the human player's winning or losing status. In all, 19 different designs will be employed for the given circumstance. The square on which the SPI player plans to place a white symbol in the following move will be lit up. If required, the top and bottom squares on the board representing the human player's winning or losing status will also be lighted.

The authors employed 19 patterns to indicate the SPI player's activity as well as the human player's winning/losing status.

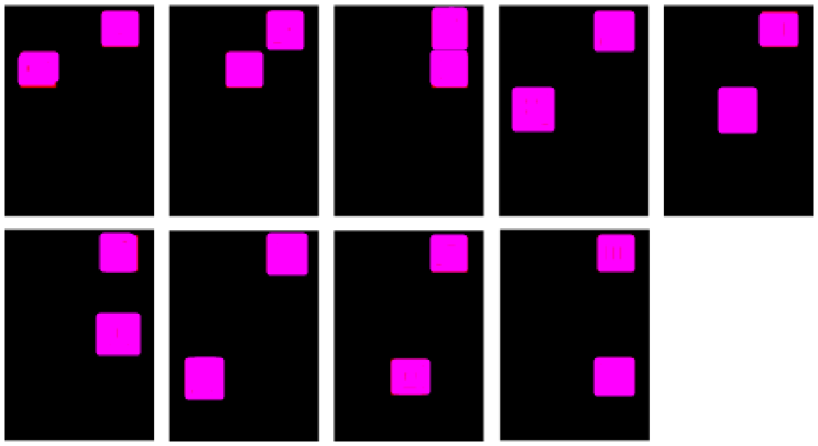
- SPI player's consecutive actions are depicted in the following patterns;



- The following pattern shows the human player's victory, so actions stop here



- The rest of the actions displayed below show SPI player's moves and the SPI agent will become the winner side



Multi-agent reinforcement learning in Markov games, with the objective of learning Nash equilibria or coarse correlated equilibria (CCE) sample-optimally, was another research study that provided an intriguing approach for the selection of algorithms of the SOS game (Li et al, 2022). Regardless of the sampling protocol, all the outcomes suffer from at least one of the two barriers: the curse of numerous agents and the obstacle of a lengthy horizon. The authors took a step toward resolution by assuming access to a flexible sampling mechanism: the generative model. A fast-learning algorithm was presented, named Q-FTRL and an adaptive sampling technique that exploit the optimism principle in online adversarial learning (especially the Follow-the-Regularized-Leader (FTRL) approach) for non-stationary finite-horizon Markov games. Now let's see what algorithms were used for this problem.

The technique is based on dynamic programming (Bertsekas, 2017). The procedure applies backward recursion from step h = H to h = 1; in reality, the sampling will be completed. The learning procedures for step h before returning to step h-1. The i-th player calls the generative model for K rounds, with each round drawing SAi as a consequence of independent sampling, the total sample size is given by KSH∑Ai from i to m(Li et al, 8). The above algorithms requires 3 implementation steps, which are:

1. Sampling and model estimation
2. Q-function estimation
3. Policy updates((Li et al, 9)

As a result, the suggested algorithm turned out to be manageable and sample efficient.
So these papers gave useful insights for the construction of our “SOS”, however, they would be hard to implement at this stage, so finally we decided to end up with a minimax algorithm with alpha-beta pruning working on it and introduced a new agent working “Smart-Random” algorithm.

Method

The Design

Our implementation of the SOS game is that different agents play against each other (from a human playing against an artificial agent to two artificial agents playing against each other). We concentrated on creating a professional design of the code including all standards of object oriented programming, design patterns and focusing on the speed of functions used in code infrastructures. Each agent is represented by a class. In total we have 3 + 1(not explicitly defined) agents.

The first and the easiest agent implemented is the user agent. The latter is a human who inserts values for rows and columns he/she wants to place his/her “S” or “O” on.

The second agent is the random one. It just seeks for an empty cell and arbitrarily puts an “S” or an “O” in there.

The next and, of course, the most important agent uses the alpha-beta pruning algorithm. The player is going to try to maximize its performance. The minimax value is going to be the highest one that a player can receive without knowing what move its opponent is going to make. The evaluation function is chosen so that the better move has the highest utility value. The most important idea in this specific implementation of the “SOS” game is that the game is not presented as a win/lose situation; the game has scores, which are utilized in its implementation. Since the algorithm's outcome is represented by scores, the decision was made to utilize the scores as minimax`s evaluation function. Thus, this evaluation function is based mainly on the scores. A depth is chosen and the alpha-beta algorithm finds the utility values for them and choses the one with the highest score. If there are a few high scores, then one of them is chosen randomly. GIven that all the utility values are equal, which means there is no “best move”, our algorithm uses a smart random function to make a better move than just a random one. If the player`s opponent has an opportunity to make an “S-O-S” after the player`s move, the utility is one divided by the score. The more “S-O-S”s the opponent will be able to make after a certain move, the less the utility will be. Thus, the less the utility the less the chances that that move is going to be chosen. If that function again returns equal values, then finally, our agent will act totally randomly. To finish up, it is worthy to mention that when we define alpha-beta agent`s depth to be zero, our agent becomes purely smart-random. We are going to consider it an agent, too.

Evaluation

To make our implementation`s advantages more vivid we decided to compare it to the Minimax algorithm. As we know Minimax is quite time and space consuming. Hence, the alpha-beta pruning algorithm was invented to make Minimax more efficient. It prunes the nodes which are unnecessary. Our game, after each iteration, writes the percentage of nodes that were pruned by the alpha-beta agent. We did these calculations, by finding the number of nodes that minimax would have expanded((Free Space)!/(Free Space - Depth)!) and then we divide the pruned nodes by the number of nodes we calculated for minimax and we multiplied by 100.

First, we tried to test our agent`s capabilities against an online SOS game. Our grid was accustomed to the online game`s grid. Our game was played in the following way: on an 8x8 grid, alpha-beta with depth 1 was playing against a "human player". The first step was done by our game, that step was inserted to the online SOS game from the human player's side, and when the online game made a step we inserted it in our game as a human agent. Surprisingly, our alpha-beta algorithm won the game with a final score of 35:3.

The second series of trials was with the human player against alpha-beta. The human agent was our teammate who had a pretty good understanding of the game; however, all the games were won by our alpha-beta agent except for one.

The next trial, the result of which is quite worthy of mentioning, is that our blocking/smart random agent mostly acts like a human. It does not look into depths. It just tries to block future "SOS" formations for its opponent. Thus, when playing against each other, those two agents most of the time end up in a draw.

Since the minimax and alpha-beta algorithms are famous for being very slow and their maximum depths are uncomputable, we tried only playing with the depths 0, 1, 2, 3 and 4. They compose 25% of maximum depths, if not even less; thus, our conclusions on evaluations are mainly based on agents playing with those depths. We tried to use 50% of our depths, but it took hours to finish the game. If we try writing the program in C programming language, we will be able to reach its full potential, but python is not suitable for such computations.

The next trial was between the smart random agent and the alpha-beta agent. Surprisingly, during these trials, the alpha-beta agent plays well with relatively small depths. We interpret this phenomenon this way. The alpha-beta does so because it is just trying to ensure its "Sos" and high score, whereas with higher depths, it tries to minimize its opponent`s future good moves while the latter just plays by blocking its opponent and does not try to maximize its performance. In the first 25% of depth, the higher the depth, the worse the result.

For the purpose of getting some real insights about our alpha-beta agent we decided to finally make them play against each other having each of them different depths. Somehow in the depth levels in which we were capable of testing our agent, which means 1,2,3 and 4, our alpha-beta with the smaller depths played better. We think that concluding about our agents` goodness based only on 25% or less percent depth is quite biased. When we tested for hours for depth 5 we actually saw that it beats the agent with depth 1. It changes the overall picture quite much, since the depth 3 lost to depth 1.

Conclusion

In our report we presented some algorithms for constructing a program in which an intelligent agent plays against a human player or against another intelligent agent. They are playing a game called SOS, in which they are choosing optimal or approximately optimal strategies. In our game, the alpha-beta agent is either going to win or is going to end up in a draw when playing with the random agent. In the case of two alpha-beta agents playing against each other, the outcome is not definite. For solving the mentioned problems, we defined a random agent; a user-player, which inserts the position and the value it wants to put to a cell on its turn; additionally, we defined an agent which uses alpha-beta pruning algorithm, with an evaluation function connected with its scores. In the latter algorithm we defined a "smart random" function, which would help the agent to make a decision in the case when the utilities for alpha-beta are equal. Since, the function was quite helpful, we also added the smart random as an agent, it was just the alpha-beta, but with depth 0. The games we conducted gave us pretty impressive insights, we believe that the methods we applied on this game can have further applications in other search games.

References

[1] E. (n.d.) GitHub - realKfiros/SOS. Retrieved November 6, 2022, from
https://github.com/realKfiros/SOS
[2] Hellerstein, L., Lidbetter, T. and Pirutinsky, D. (2019). Solving Zero-Sum Games Using Best-Response Oracles with Applications to Search Games. *Operations Research*, 67(3), pp.731–743. doi:10.1287/opre.2019.1853.
https://arxiv.org/pdf/1704.02657.pdf
[3] Abas Setiawan (2020) Playing the SOS Game Using Feasible Greedy Strategy
https://www.researchgate.net/publication/340883423_Playing_the_SOS_Game_Using_Feasible_Greedy_Strategy
[4] Gabriella Bee Balista (2016) Color as a Game Design Tool
https://medium.com/@interama/color-as-a-game-design-tool-2b9b38ad228e
[5] Playing Tic-Tac-Toe Games with Intelligent Single-pixel Imaging
2205.03663.pdf (arxiv.org)

[6] Minimax-Optimal Multi-Agent RL in Zero-Sum Markov Games With a Generative Model
2208.10458.pdf (arxiv.org)
[7] Bertsekas, D. P. (2017). Dynamic programming and optimal control (4th edition). Athena Scientific.