INTELLIGENT GAME PLAY FOR PYLOS USING ML AND AI

Rahul Nalawade
Dept. of Computer Science,
University of Texas at Dallas
rsn170330@utdallas.edu

Anshul Dupare
Dept. of Computer Science,
University of Texas at Dallas
axd171630@utdallas.edu

Venkatesh Gotimukul
Dept. of Computer Science,
University of Texas at Dallas
vxg173330@utdallas.edu

***Abstract*- In this paper, Pylos playing agents are discussed. These agents are combined with optimal game winning strategies. This results in an artificial companion which plays the board game Pylos in a fully interactive manner and up to the highest possible level. In the long term this could lead to interesting competitions between various board agents playing several board games in an interactive way. We aim to develop an agent for board game Pylos as discussed above which is a zero-sum game. The agent will learn by playing against another agent which would have a hard-coded evaluation function. We plan to use decision trees for moves where each move has a score extracted from a game board. We aim to pick the optimal next move using alpha-beta pruning and minimax algorithms. We then intend to use SVM to solve SVR and also to optimize the score of potential moves. We will generate the data set by letting the learning agent play against another agent which would have a hardcoded evaluation function. We plan to select only those data points which would have a certain depth in the decision tree.**

***Key Words*—Support Vector Machine(SVM), Random agent, Mini-max algorithm, Decision tree, Support Vector Regressor(SVR).**

I. INTRODUCTION

Pylos is a two-player strategy game played on a square board with 16 indentations laid out on a 4x4 base, seen in Fig. 1. Each player starts with fifteen balls, one player with light balls and the other player with dark. Players take turns placing balls onto the game board. If four balls of any color are placed on the game board in a 2x2 square formation, a ball can be placed on top of them forming a second level of play with 3x3 available positions to place a ball. As the game continues, players may position balls on this second level and once a square is formed, a ball can be played on the third level with 2x2 positions. Once the third level has been filled, a final ball can be placed on the fourth level.

The player that places the final ball wins the game. If a move results in a 2x2 square of any color, the player can remove one of their balls from anywhere on the board that is not supporting another level and place it on top of the square, saving the balls in the player's reserve for future moves. This locks all the underlying balls in place. These mechanics introduce much of the complexity in the game. Otherwise, the second player to place a ball would win every game. This also introduces the possibility of loops that can result in the game going on indefinitely, making a general game AI difficult to derive.

Fig 1. Pylos is played on a 4x4 indented game board. Players may position balls on multiple levels until a final ball is placed on the top

II. GAME THEORY

Pylos is a perfect information two-player zero-sum board game. That is, there is no hidden information (like hidden cards) and no element of chance (like flipping a coin or rolling a dice). Moreover, the two players make their moves in turns, have full information when making their decisions, and improve their situation at the expense of their opponent.

In principle it is possible to compute perfect strategies for such games. But of course for many interesting games, like for example chess, time and space constraints disallow that task. The interested reader might look for two-person zero sum games in any text book on game theory to learn more about the basic theory behind this class of games.

There are several degrees of 'solving' a game. The weakest version, usually called ultra weakly solved, is to just know which player (first or second) will win a game, without knowing how this can be achieved. If a game is weakly solved, there exists a winning strategy, if the player uses it from the very beginning of the game. In a strongly solved game we can compute optimal moves for any reachable position. Here a reachable position is a situation of the game which can be reached from the starting position through any

sequence of valid moves - regardless whether they are optimal or not. Thus this last version is of special interest for interactive games, where the user might want to challenge the robot by giving it difficult situations.

We have been able to solve Pylos in the strong way. The resulting game database has about 30 GB and contains for each reachable position a rating reflecting the quality of possible moves. This allows to obtain optimal moves for any possible game situation, as well as to crosscheck the plausibility of follow-up positions during an interactive game.

Fig 2. A wooden version of the board game Pylos

III. RELATED WORK

We chose Pylos for this machine learning application because it is deterministic and previously unsolved. Autonomous play of Pylos has been explored by only one previous publication. However, they solved the game by generating a huge database with all possible positions from which a win can be forced. This obviously defeats any human player. We are planning to implement our solution with more understanding of the game as opposed to the brute force method of a full game tree analysis

IV. METHODOLOGY

I. AGENTS INVOLVED IN BOARD GAME

Random agent: Agent that chooses random moves which are legal which may or may be the moves in the winning path but are appropriate and completely legal.

Maximizing agent: A slightly more sophisticated strategy looks at every available move and evaluates them. Once we get a list of legal moves, we choose the move that results in the best board configuration based on the static evaluation function that computes the difference between the number of balls each player has on the board. The Maximizing Agent returns the move that results in the best immediate score. This agent plays similarly to a beginner who is still getting familiar with the rules.

II. BOARD LEVEL REPRESENTATION

There are four levels in pylos game board. Each level with n^2 number of possible chances. We represent the bottom most level in the board by a_{ij} where i=1 to 4 and j=1 to 4 and the next level by b_{ij} where i=1 to 3 and j=1 to 3 and the next level by c_{ij} where i=1 and 2 and j=1 and 2 and the top most level by d_{ij} where i=1 and j=1.

Level a

	1	2	3	4
1	a11	a12	a13	a14
2	a21	a22	a23	a24
3	a31	a32	a33	a34
4	a41	a42	a43	a44

Fig 3. Level a of the board

Level b

	1	2	3
1	b11	b12	b13
2	b21	b22	b23
3	b31	b32	b33

Fig 4. Level b of the board

Level c

	1	2
1	c11	c12
2	c21	c22

Fig 5. Level c of the board

Level d

	1
1	d11

Fig 6. Level d of the board

III. BOARD STATE REPRESENTATION

We represent the Board state as the state or ball present in every individual position in the board level as discussed above. We represent the position as i=0,1 or 2 where i=1 is the player 1 and i=2 is the player 2 and i=0 represents that the position is null or void which means it is unoccupied at that particular instance.

```
State:[
        [[_,_,_,_],[_,_,_,_],[_,_,_,_],[_,_,_
        [[_,_,_],[_,_,_],[_,_,_]],
        [[_,_],[_,_]],
        [[_]] ].
```

Fig 7. State of the board

IV. DECISION TREE

From any given state, there are a finite number of moves for a player to consider. We chose to represent these moves in a decision tree and use an algorithm to pick the best next move in a game. The tree contains all possible moves from each position; the nodes represent positions in a game and the edges represent moves.

Pylos is too complex a game to enumerate an entire tree efficiently. Since players can remove their own pieces from the board, the game may not terminate after a finite number of steps and begin looping, resulting in a draw. Therefore, the decision tree for this game can be essentially infinite given these loops. At a single node, the branching factor can be over 100, which means that the player has over 100 legal moves to choose from at each turn. To put this in perspective, the average branching factor is about 35 in a game of chess and 250 for the game Go [2]. In order to determine the best move out of all the potential moves available to a player, we need to consider an algorithm that assigns node scores based on the current state of the board.

For the AI to practically play the game, we must limit the depth of the search down the tree to only a few moves ahead in the game, since it is too computationally expensive to expand all the way to the end of each game to determine whether a move results in a win. We settled on a layer-based scoring approach, whereby the AI makes a heuristic estimate for the score of each possible approach for the next step and follows each one until the game ends.

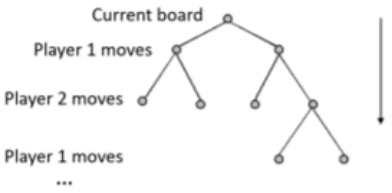

Fig 8. Decision tree contains all possible moves from any board configuration. Each layer alternates between players.

V. SCORE EVALUATION

We created four different, hard-coded player agents to simulate game play using various strategies, described below. The hard-coded player simulations all use a version of the decision tree search and a static evaluation function, which takes in a game state and computes a score correlating to the likelihood of a win. However, the agents differ in how they treat and combine these scores into more meaningful metrics. The score is extracted from the current state of the board. We experimented with evaluators that gave more points for balls placed at higher levels and balls placed in more strategic positions on the board. However, as is quite typical in other similar games such as chess or checkers, it was more useful to use the simple maximizing heuristic:

opponent's balls - # player's balls

and get better performance by a deeper search. The evaluator wants to maximize the number of opponent's balls on the board while minimizing the number of the player's own balls on the board.

VI. SUPPORT VECTOR MACHINE AND REGRESSOR

The AI's performance is based upon its ability to develop node scores that accurately represent the correlation of game states with a successful result. Once we acquired results from playing the four hard-coded agents against one another, we processed the data using a Support Vector Machine. The Gaussian kernel function we used transforms the data into a higher dimensional feature space, making it possible to perform the linear separation.

We let Maximizing agent play against the opponent with first move as Random so as to avoid similar game play every time limiting each move to 0.5 s, and kept a record of all relevant elements (board configuration, search depth, node score) encountered during game play. This generated many data points. However, we were only interested in evaluations that were generated using information greater than a certain depth, as those scores contain information about possible future evolutions correlated to the current board.

We used SVM to solve the Support Vector regression with a Gaussian kernel on the selected data points. We experimented with different kernels, but Gaussian worked the best due to its nonlinearity. The advantage of SVM is that it does not try to fit as closely as possible to the hyperplane and instead relaxes at a certain distance from the desired point. Since the score function is integer-valued, it does not need to have a more precise fit.

V. RESULTS

We quantified the performance of our AI by simulating games between various agents. We allowed player 1 to play against player 2 for 87, 199 and 1000 times, generating the data below. This data showed that both players played according to a heuristic which was the most effective, so we used it to generate the SVM model. For the table below, we show data corresponding to the wins and losses of a player from the header row column against a player in the left column.

TABLE I

NUMBER OF TRIALS- WINS , LOSSES

Number of trials	Player 1	Player 2
87	60	27
199	140	59
1000	422	578

As expected, the results were as usual when the trials were not much in size i.e for 87 and 199 trials we can see that player 1 won 60 out of 87 games and 140 out of 199 games. But, when we play considerably large number of games we can observe the probability that player 1 or player 2 win is almost equal in fact player 2 won more games than player 1 which is not surprising because the maximizing heuristic is applied for player given the player 1 starts with a random move.

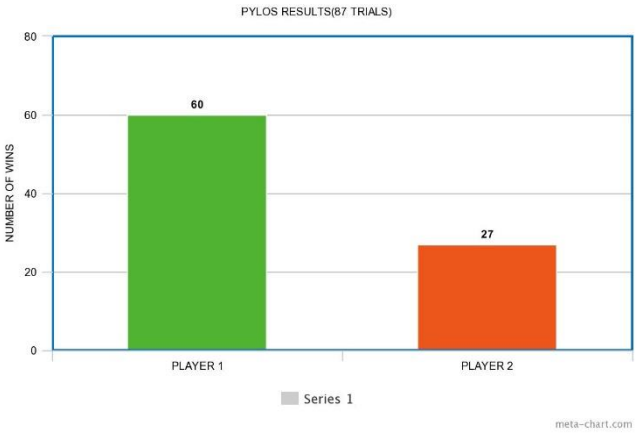

Fig 9. Bar diagram showing results for 87 trials

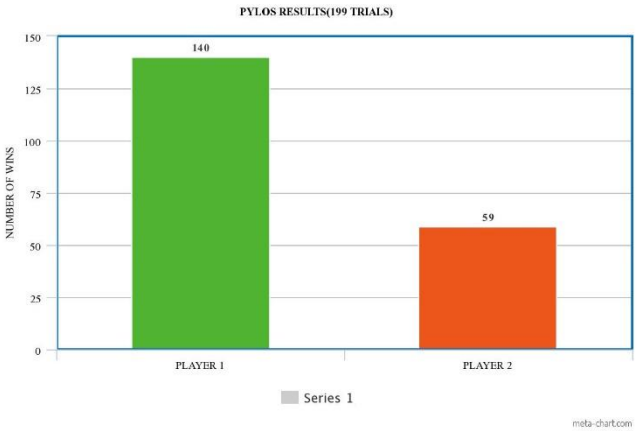

Fig 10. Bar diagram showing results for 199 trials

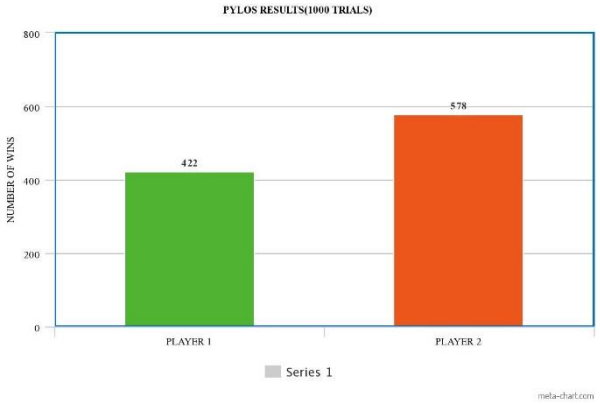

Fig 11. Bar diagram showing results for 1000 trials

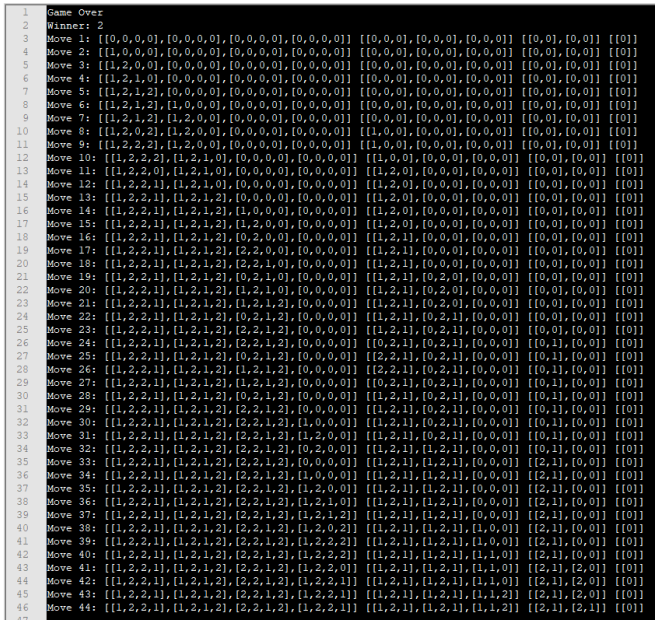

Fig 12. Output showing moves when winner is player 2

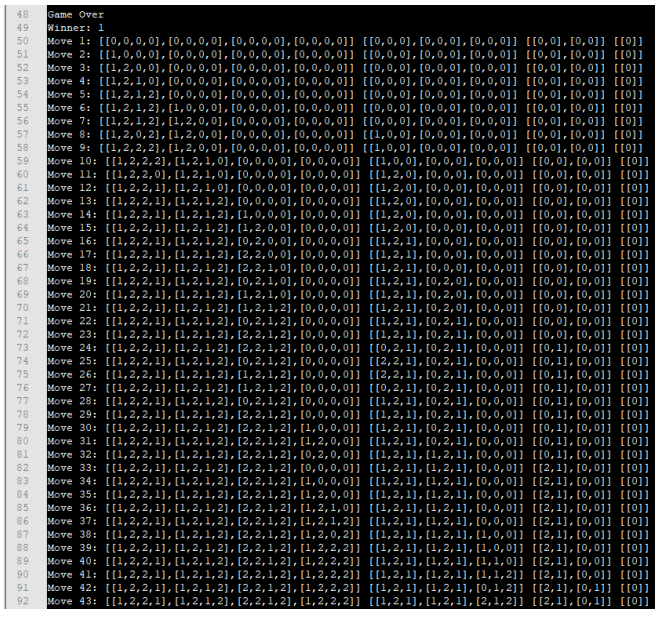

Fig 13. Output showing moves when winner is player 1

VI. FUTURE ENHANCEMENTS AND SCOPE

We have built a maximizing agent combined it with optimal game winning strategies. This results in an artificial companion which plays the board game Pylos in a fully interactive manner and up to the highest possible level.

Challenging tasks for the future are on the one hand to extend our approach to other popular board games, like connect four or checkers. On the other hand we would like to use standard hardware platforms, i.e., robots off the shelf. In the long term this could lead to interesting competitions between various autonomous robots playing several board games in an interactive way, possibly at exhibitions very similar to what happens for soccer in the RoboCup initiative.

The problem with the minimax and alpha-beta pruning algorithms is that they take an impractically long amount of time to search more than a few turns ahead. Additionally, we selected features that we thought could evaluate whether a move is likely to result in a win, based on prior knowledge we had from playing the game. Given that Pylos has a high number of available moves per turn, using these algorithms can result in somewhat slow gameplay.

In the future, we could look to a technique that is used for playing complex games (such as Go) known as Monte Carlo Tree Search (MCTS), which could improve the performance of the AI. MCTS is a method in which the algorithm randomly traverses a tree to find the end result of the game and decides which move is best based on which edges were chosen most often. We would like to compare the performance (in terms of speed and win rate) between our SVM agent and a MCTS player. Finally, we would like to implement a GUI for the game so that a human can click on buttons to make a move instead of inputting code manually through terminal.

VII. REFERENCES

[1]. Feng-Hsiung Hsu. Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press, 2004.

[2]. Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. RoboCup: A Challenge Problem for AI. AI Magazine, 18(1):73–85, 1997.

[3]. Frank Wallhoff, Alexander Bannat, Jurgen Gast, Tobias Rehrl, Moritz ¨ Dausinger, and Gerhard Rigoll. Statistics-based cognitive human-robot interfaces for board games — let's play! In Proceedings of the Symposium on Human Interface 2009 on Human Interface and the Management of Information. Information and Interaction. Part II, pages 708–715, Berlin, Heidelberg, 2009. Springer-Verlag.

[4]. G. Kronreif, B. Prazak, M. Kornfeld, A. Hochgatterer, and M. Furst. Robot assistant "playrob" - user trials and results. In Robot and Human interactive Communication, 2007. RO-MAN 2007. The 16th IEEE International Symposium on, pages 113–117, Aug. 2007.

[5]. O. Aichholzer, et al, "Playing Pylos with an autonomous robot," in IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Taipei, Taiwan, 2010, pp. 2507-8.

[6]. D. Silver, et al, "Mastering the game of Go with deep neural networks and tree search," Nature, vol. 529, no. 7587, pp. 484-489, Jan. 2016.