Connect-Four Agent Using Particle Swarm Optimization

Etienne Le
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01852, USA
etienne.h.le@gmail.com

Pratik Maharjan
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01852, USA
pratikmaharjan88@gmail.com

Lok Sum Leung
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01852, USA
loksumleung.lsl@gmail.com

***Abstract*—In this project for Artificial Intelligence class, we have developed a best-move agent for a Python-programmed version of the board game Connect-Four. The agent forms its style for best moves around a specifically-formatted evaluation function that inserts a set of fitness to a Particle Swarm Optimization algorithm, which uses graph movements and iterations to guide the agent to the most optimal solution.**

Keywords—PSO, Connect-Four, AI, agent, game

I. Introduction

Connect Four is a well-known strategy game originally published in 1974.[1] It is a two-player game which players take turns to drop a colored piece/disc into a seven-column and six-row grid. The pieces will fall straight down and occupy the lowest available space within the column. If the column contains 6 pieces, no other piece can be dropped into that column. The winner of the game is the first player who forms a connected horizontal, vertical or diagonal line of four of their own pieces. It does not matter which part of the four-string was placed last, whether it was one of the endpoints or "middlemen"; if the player has made a consecutive length of four, they have achieved victory. [5] (see fig. 1., fig. 2., fig. 3., fig. 4.)

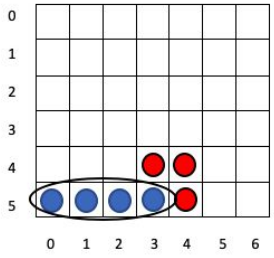


Fig. 1. Player Blue won with a horizontal line of four blue pieces.

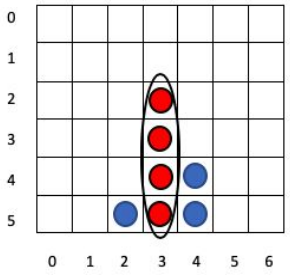


Fig. 2. Player Red won with a vertical line of four red pieces.

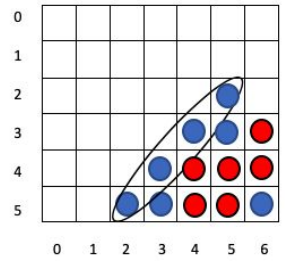


Fig. 3. Player Blue won with a diagonal line of four blue pieces.

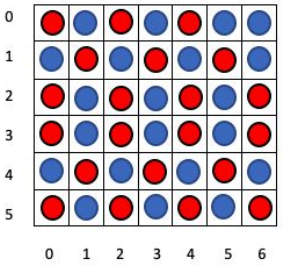


Fig. 4. Draw.

In this work, we have implemented an AI agent to play the current best move given a random assortment of pre-arranged game boards simulating various instances of competitive play using Particle Swarm Optimization. Particle Swarm Optimization (PSO) is an evolutionary algorithm that homes in on an optimal solution iteratively by utilizing a batch of candidate solutions that are initialized on random sections of a graph. Introduced in 1995 by J. Kennedy and R. Eberhart, it was originally proposed to simulate birds searching for food or the movement of fishes' shoals. In search of a best solution, the algorithm moves the particles with certain velocity which is calculated in each iteration. The movement of each particle is influenced by it own best known position and also the best known position in the search space. The final result is the particle swarm converging to the best solution.

II. Literature Review

For the purpose of this study, we will be reviewing available works on the Connect Four game playing Agent using Artificial Intelligence.

Thil and Konen [2] developed a perfect playing Connect Four agent. They used Minimax algorithm and optimized the algorithm using alpha-beta pruning, move-ordering mechanisms, symmetries, two-stage transposition tables in order to identify identical positions created by permutations

of a certain move sequence, Zobrist Hashing, Enhanced Transposition Cutoffs and more.

Saveski [3] used minimax search algorithm on the agent of Connect Four game. He then improves the speed by using alpha-beta pruning to minimize the search space. He also implemented iterative deepening search, which called the alpha beta pruning search with increased depth in each iteration and returned the value of the last iteration.

Marcus [4] applied minimax algorithm on his Connect Four AI. He improved the program using alpha-beta pruning algorithm. He also employed the heuristic function. The function can guide the AI towards a winnable position.

To prepare for implementing the Connect-Four agent using the Python programming language, there was a lot of research put into finding out how previous researchers implemented a proper, competent AI for the board game. Throughout our experience with this, we have found many scholarly articles and repositories in which teams have developed the game and had the AI run on a minimax or alpha-beta pruning algorithm. These algorithms were effective. Naturally however, none of these articles give any indication on how evolutionary algorithms might be used to serve a similar, if not identical, purpose.

Therefore, our research on such algorithms, most prominently genetic algorithms and particle swarm optimization, was without much reference to the board game itself, though there was still a lot to be learned from that. For instance, these evolutionary algorithms require several components, namely a population of solutions, an evaluation function to assess them, and a means of generating a new population with the intent that an even better solution was within it.

Genetic Algorithms and Particle Swarm Optimization differ in how they intend to optimize the next generation of population. GAs utilize genes, or particular, essential characteristics that factor into improvement. Finding the fittest among them based off of these, the algorithm then makes these into a set that cross over between each other, generating the new population with their genes blended. If all permutations of crossover are achieved and the previous best had almost all the ideal genes already, then it is likely the algorithm will have the max-fitness solution in the next generation. Other factors that help reach this eventual conclusion include mutation, which has a random chance of altering one random gene, in the hopes that it helps optimize the solutions further.

Particle Swarm Optimization views solutions as coordinates on a graph rather than as chromosomes in GAs, and modify the possibilities differently than them. This is reinforced by how it finds the best global solution through calculating velocities for individual particles. The velocity function in PSO is based off of a particle's current position, a local position that is the most fit (often referred to as pBest), and finally the global best (gBest). It is the velocity function's job to guide each and every particle towards the gBest by traveling across multiple iterations of movements, and the PSO will have a plethora of particles that point to the most optimal coordinate.

III. Methodology

A. Task Environment Properties

1) *Fully observable:* The complete board (that meaning all of the spaces, occupied by AI, opponent, or not at all) is observable. The board always remains at 7x6, and though it will eventually run out of spots, prior arrangements influence the next position of play a lot.
2) *Single-agent:* The task environment is single-agent in this case, since there is only one agent trying to find the best solution for the board.
3) *Deterministic:* The task environment is deterministic because every move is made by either player with certainty (meaning players won't make accidents and drop into columns it doesn't want to play at).
4) *Episodic:* The task environment is episodic because the agent's current action will not affect a future action. While this is mostly due to how the programs were developed, this could also be valid in full games, in which an opponent cannot stop a player's victory or did not see a four-string coming.
5) *Dynamic:* The environment is changing due to the game board filling up with both players' moves. This is shown in our pre-arranged testing boards, with the latter ones being theoretical extensions of the ones before; the algorithm must reconsider its best options to find the best move.
6) *Discrete:* The task environment is discrete because the grid will not increase in space. Players may be prone to not recognizing imminent victory and defeat in Connect-Four, but there is indeed a finite amount of win and lose conditions, due to the lack of modifications to the game board.

B. Procedures

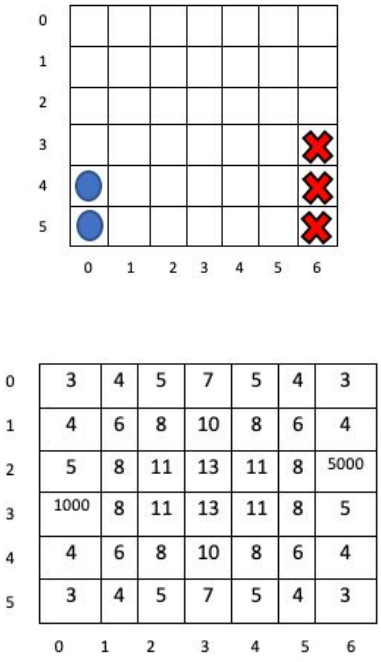
In order to develop an AI that works well in playing an instance of a game of Connect-Four, then the game itself must be implemented first, as it will give us full understanding of the rules of the game and how all of the components and functions view the board.

After the aspects of the board game have been internalized, we then moved on to making the evaluation, or fitness, function to determine all of the best moves available to a player. This function took precedence over all others because at this stage of development, we were not certain whether we wanted to develop either a GA or a PSO. We did know by this point that in either case, a fitness function must be implemented, so we worked on that first. This worked out very well for us in the long run, as how we evaluate fitness served well in realizing how best to look at the game board as an AI. With an ideal perspective found, one algorithm sounded more enticing than the other.

The evaluation function searches from the top-left of the game board, sliding across the row before moving

down to the next one and repeating, until it reaches the bottom-right corner.

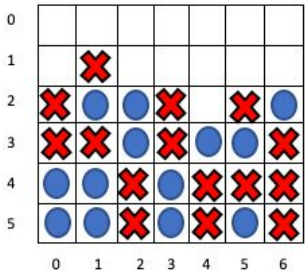
The function begins by assessing defensive options, first by seeing each of the enemy player's pieces on the game board (see fig. 5.). It disregards any sets of enemy pieces until it discovers a three-string, in which the function would locate the anticipated fourth and put a heavy weight on that specific coordinate, emphasizing that to not lose, it should play there in almost every scenario. Only one case bypasses this weight.



0	3	4	5	7	5	4	3
1	4	6	8	10	8	6	4
2	5	8	11	13	11	8	5000
3	1000	8	11	13	11	8	5
4	4	6	8	10	8	6	4
5	3	4	5	7	5	4	3
	0	1	2	3	4	5	6

Fig. 5. Fitness values of a random board (AI is circle)

Now that countermeasures have been placed, the agent now goes on the offense. It identifies every place on the game board in which the AI has its pieces, and attempts to find any two or three-strings of them that they have. If they find any two strings, the fitness function will allocate an intermediate weight to the place where a third-string could be formed. However, if it finds out that the third disc is already there, then it will identify the fourth spot in the string and allocate a supreme weight to that space (see fig. 6.). This makes it so that at the end of evaluation, the agent knows where it should play to end the game and achieve victory, with even more emphasis than preventing loss if it is so close.



0	3	4	5	7	5	4	3
1	6000	6	100	10	8	6	4
2	5	8	11	13	11	8	5
3	5	8	11	13	11	8	5
4	4	6	8	10	8	6	4
5	3	4	5	7	5	4	3
	0	1	2	3	4	5	6

Fig. 6. Fitness values of a random board (AI is circle)

The evaluation function would clear the modified fitness values of any spots on the grid that it could not reach, whether due to it being on top of empty spaces, or because they were already occupied. This made it so that the PSO algorithm would not deadlock the particles to these spaces that were impossible to play on.

If the AI was given an empty game board (which was one of our pre-arranged game boards), then the function would not update the fitness table and use the default values (see fig. 7.).

0	3	4	5	7	5	4	3
1	4	6	8	10	8	6	4
2	5	8	11	13	11	8	5
3	5	8	11	13	11	8	5
4	4	6	8	10	8	6	4
5	3	4	5	7	5	4	3
	0	1	2	3	4	5	6

Fig. 7. Default fitness

Said default values were a given integer between three and thirteen. They were properly allocated to each spot based on how many ways that particular space can be involved in making a four-string. For example, the corner spots of the grid have an initial evaluation of three, because there are only three ways a player can win with that space. The middle spaces have thirteen possible ways to win, and so are given an initial value of thirteen. With the bottom middle of the game board boasting a value of seven, the agent will play there first to maximize its winning options.

IV. Results

TABLE I.
Relationship Between Number Of Particles And Number Of Iterations

Particles	Iterations	Correct Best Move (Out of 10)
5	5	3
5	200	5
10	5	5
10	200	6
25	5	7

25	200	8
50	5	8
50	200	9
100	5	9
100	200	9
200	5	10

As Table I. indicates, the number of particles is more significant to finding the optimal solution than the number of iterations that the PSO algorithm runs. This is most likely because of the limited size of the game board when compared to a graph. In essence, the seven-by-six grid makes it so that there are only forty-two spots we must consider (decimals are not done even if the velocity function creates them because it is impossible to put a piece through a column in the board; instead the end result is converted to an integer). Subsequently, particle counts that increase past 42 are more likely to be randomly arranged across the entire board and will have a good chunk already on top and next to the global best position, assisting the PSO algorithm to find the gBests with ease.

V. Discussion

The agent proves effective at playing best moves during the consecutive cases that we presented in the Results and Methodology section. However, these are not the only cases in which the player or AI can win or lose. For instance, the player can win by placing one of their own pieces between a two-string and a lone piece, making four. The agent's evaluation function does not take "middlemen" into consideration, and so they wouldn't put significant weights behind those spots unless said spot was also involved for a traditional three-string case. The vice-versa is also true; in offense, it will only do middle-man four-strings if by chance there was one of its pieces on the other side of a two-string it is accounting for.

During initial testing, the evaluation function failed at times to deliver the best move due to a prior misinterpretation of gameplay. The agent originally believed in the notion that it should make sure it doesn't lose more than it should find out how to win. While defense is vital to the game, the AI needed to know that if it can win with its next move, it should really perform the move that does that. For this reason, the aforementioned supreme weight was attached to forming a four-string, while forming three-strings was given the lowest increased weight.

In addition, while we included a source code file for our PSO agent to play a full game against a human player, it often fails to play too effectively. Part of that comes from the shortcomings of the evaluation function listed above. However, what also made it not very effective is the agent's episodic nature; none of the PSO agent's functions save data for later use, rather recreating an entirely new fitness table as its turn comes along. It could theoretically take a while to build a two-string and start playing off of that.

VI. Conclusion

This project serves as the first step to creating a highly competent AI for Connect-Four that could rival the tried and true methods of implementing an agent player using Minimax and/or Alpha-Beta Pruning. If the shortcomings listed in the Discussion section can be remedied, the program using our algorithm and fitness function to play a full game may prove to be a very daunting opponent for any players of all levels of experience. With an improved evaluation function that can work with different sized boards, the PSO agent can display its ability more significantly on a larger board with particles that are spread out more.

Acknowledgment

We would like to express our sincere gratitude to our research mentor, Professor Jonathan Mwaura from the department of Computer Science at University of Massachusetts Lowell, for providing invaluable guidance throughout this project.

References

[1] Wikipedia, the free encyclopedia. Connect Four.https://en.wikipedia.org/wiki/Connect Four

[2] Markus Thill, Wolfgang Konen. Connect-Four Game Playing and Learning Framework.https://github.com/MarkusThill/Connect-Four

[3] Martin Saveski. AI Agent for Connect Four. http://web.media.mit.edu/ msaveski/projects/2009 connectfour.html

[4] Ryan Marcus. A JavaScript Connect Four AI. https://rmarcus.info/blog/2014/12/23/connect4.html

[5] Allen, James Dow. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Puzzle Wright Press, 2010.