The 22nd Annual Undergraduate Research Conference

Wittenberg University
Department of Computer Science
Brian Morrow (s10.bmorrow@wittenberg.edu)
Thomas DeBell (s10.tdebell@wittenberg.edu)
Dr. Steven A. Bogaerts (sbogaerts@wittenberg.edu)

A Genetic Algorithm to Optimize a Connect Four Minimax Player

Connect Four is a classic game played by many people across the world. We have implemented a fully functional Connect Four game with a computer player that uses the minimax algorithm to choose its next move. This algorithm first creates every possible state that can come from the current state within a definable number of moves, and then scores each state using the concept of an n-sequence. An n-sequence is a window of four slots with n pieces and (4 – n) blanks. Intuitively, a higher n-sequence should have a greater effect on the score of a state than a lower n-sequence, but the exact relationship is unclear. Therefore, we used a genetic algorithm to determine the scoring importance of each n-sequence. The genetic algorithm sets up a pool of computer players, all with different weights for each n-sequence evaluation, and plays a round robin tournament, recording wins and losses. At the end of each round robin, a select number of players with the most wins move on to the next round and are then randomly mutated and crossed-over to create a new pool of players. Once a select few weights continue to win over several iterations of the genetic algorithm, we will use these weights to create a highly sophisticated and difficult-to-beat Connect Four computer player.

Introduction

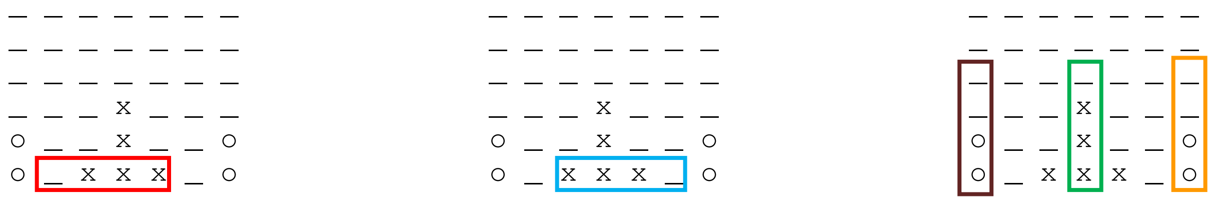
One question almost every computer user has asked while playing a computer game like solitaire, Connect Four, or chess is, "How did I just lose to a box of circuits? How did it decide to make that move with the pawn that led to checkmate?" For our purposes, we decided to attack this question in the context of the classic Hasbro game Connect Four. In this game, a board has seven columns and six rows. The goal is for one of the two players to attain a sequence of four of their own pieces in a horizontal, vertical, or diagonal fashion. The tricky part is that both players are trying to achieve this goal while simultaneously trying to prevent the other player from doing so. Our goal became creating an efficient computer player for Connect Four that provides a significant challenge to a human player. A key concept in designing a computer-simulated player for this game, or any game for that matter, is that it will not win effectively if it makes each of its moves based on the current state of the game only. It must look "down the road," taking into account the fact that the opponent will make the best decision on his/her next move based on what decision the computer player makes about its own move. This way the computer player can decide the best move that not only maximizes its chances of winning, but also minimizes the opponent's chances. Thus we decided to implement the minimax algorithm.

Algorithms

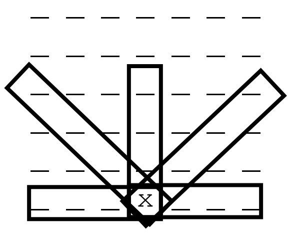
For the minimax algorithm to be effective, we first needed to design a way for the computer player to evaluate a given state of the game board so when it looks "down the road," it knows whether a certain move will be advantageous or not. To accomplish this evaluation, we created a function called *evalState* that returns a score of a game state based on some particular characteristics. This score would tell the computer player whether the given state has a positive or negative effect on its chances of winning. This algorithm is also zero-sum – if the evaluation of a game state by player A results in 100 points, the evaluation of the same state by player B would result in (-100) points.

evalState

Our scoring system first relied only on the concept of a terminal state. This is simply a game state where the board is full (there are no more moves possible) or where one of the players has completed a sequence of four consecutive pieces either horizontally, vertically, or diagonally. This required the minimax algorithm to continue to expand each state down the tree until a terminal state was reached. However, this is extremely time-inefficient because of the time it would take for every state in the tree to be expanded until that "game" ended. Therefore, we decided to more comprehensively score a state based on the pieces already placed rather than based on pieces that can be placed later. Our new scoring system is based on what we call *n*-sequences. This is a window of four consecutive spots on the board where *n* spots contain the same piece and the remaining spots are open. For example, in the following game state, there are three different 3-sequences for 'x', highlighted in red, green, and blue boxes. The 'o' piece also has two 2-sequences, highlighted in brown and gold in the right example only.



Similar to the 4-piece sequences found in terminal states, these *n*-sequences can go horizontally, vertically, or diagonally. Thus, the following board actually has five 1-sequences for the 'x' piece, highlighted in black boxes:



The *evalState* algorithm first checks if the state it's evaluating is a terminal state. If it is, this state is said to be the best (or worst, depending on which player has the winning sequence and for which player *evalState* is evaluating) in terms of scoring and the algorithm stops. If a winning sequence does not exist, *evalState* scores a state by counting the number of 1-sequences, 2-sequences, and 3-sequences for each player. To determine these counts, the algorithm loops through each spot on the game board, starting at the top-left, and checks what *n*-sequence exists, if any. This process is shown in the following pseudo-code:

```
for each row in the game board:
    for each column in the current row:
        for each four-space window in the right, down,
        up-right, and down-right directions:
            check which n-sequence exists here, if any
```

Once these counts are determined, *evalState* scores a state by subtracting the opponent's count from the count of the player of which *evalState* is evaluating and adding up those values (listed below).

(difference of the number of 1-sequences for player 1 and player 2)

(difference of the number of 2-sequences for player 1 and player 2)

(difference of the number of 3-sequences for player 1 and player 2)

However, this scoring was ineffective because the weight each *n*-sequence had on the scoring was equal. Intuitively, the scoring importance of a 3-sequence should be greater than a 1-sequence because a 3-sequence only needs one more piece to become a winning 4-sequence. Thus we added a weight to each of the differences so they can have different effects on the scoring of a state. This was accomplished by changing the formula to add these three values instead:

(weight of a 1-sequence) * (difference of the number of 1-sequences for player 1 and player 2)

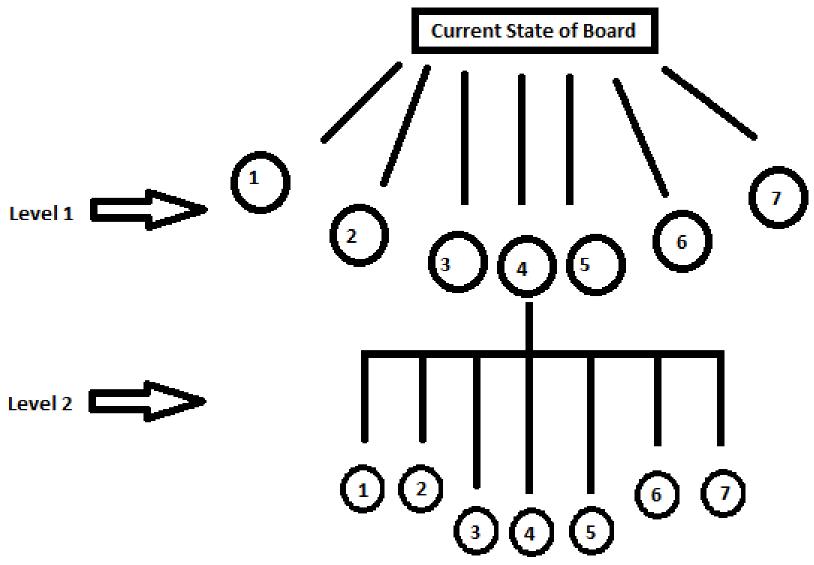
(weight of a 2-sequence) * (difference of the number of 2-sequences for player 1 and player 2)

(weight of a 3-sequence) * (difference of the number of 3-sequences for player 1 and player 2)

With our scoring algorithm in place, it was time to implement the minimax algorithm.

Minimax Algorithm

At its core, the minimax algorithm can be implemented in any one-on-one strategic game situation. Integral to the algorithm is the ability to maximize your chances for success while minimizing the opponent's opportunities to succeed. The diagram below illustrates the first step in the algorithm. We are assuming it is a game between two players, Player X and Player Y, and that it is Player X's turn.



Assume it is Player X's turn, and his opponent is Player Y.

Level 1 of the minimax tree represents the seven boards which are possible in one move, given the current state of the board. Each number corresponds to a player making a move into that column.

At Level 2 of the tree, there will be seven sets of future boards (only one is shown here), and each of these seven future board sets will contain seven nodes each representing Player Y's 7 choices given Player x's previous move.

The algorithm starts by taking in the current state of the Connect Four board. From there, it expands out the next seven possible states, each child of the current state corresponding to Player X making a move into that column of the Connect Four board. Then, each node in Level 1is traversed, and each one has seven children generated corresponding to Player Y's seven possible moves. Thus Level 2 will contain 49 nodes, because from the initial current state, Player X has seven possible moves, and then Player Y has 49 possible moves. This continues so that at any level of the tree the maximum number of nodes will be $7^{(\text{current-depth-1})}$. The algorithm runs in this manner with the alternating levels representing the alternating players and will end at either a specified search depth or if a board becomes a terminal state.

Once this terminal state or set depth is reached, the algorithm will then begin to traverse upwards through the tree, running our *evalState* function on each node in order to obtain a score. Assuming we are looking at the game from Player X's perspective, at each level which represents Player Y's choices of moves such as Level 2 above, the algorithm will pick the choice with the lowest score. At each level which corresponds to Player X, we want to pick the best move and maximize our score, so the process picks the move which *evalState* rates as the best, meaning it has the maximum score of the group of seven choices. Thus, every time our computer player needs to decide where to place a piece, this algorithm simulates many future moves and picks the move that, based on our scoring gives Player X the highest chance of winning and Player Y the lowest chance of winning.

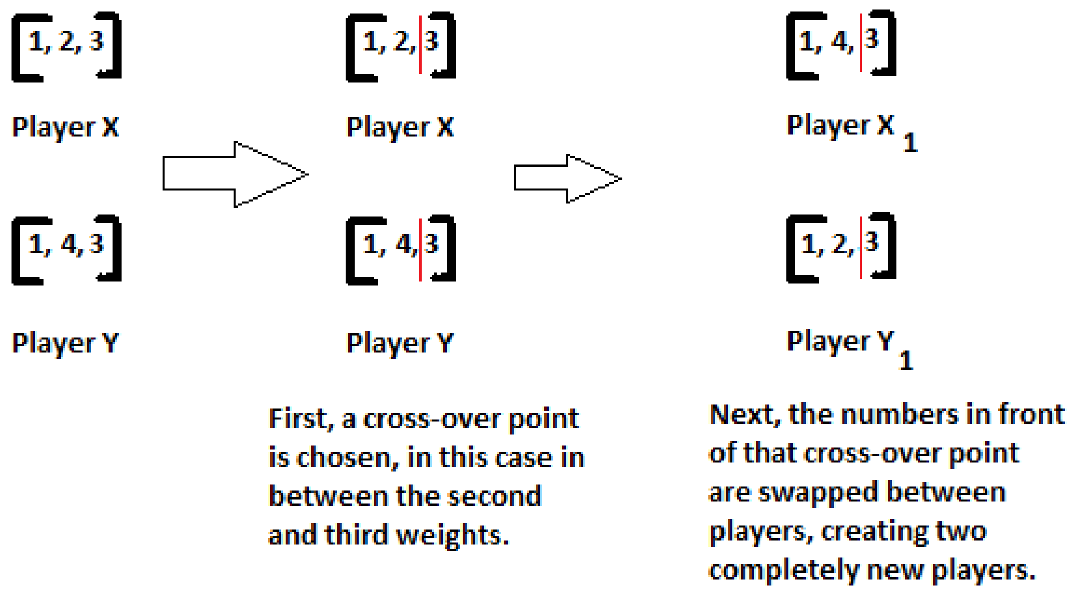
Genetic Algorithm

Recall that *evalState* scores a state by counting and weighing different *n*-sequences for each piece. We needed weights for each *n*-sequence, so instead of using our intuition or simply guessing, we decided to implement a genetic algorithm to determine an ideal set of weights. In order to do this, we first had to come up with a way to represent an individual player and also a way to give that player different characteristics than another computerized player. Then, we had to devise a method which would be able to pit these different players against each other, and pick winners based on a scoring system.

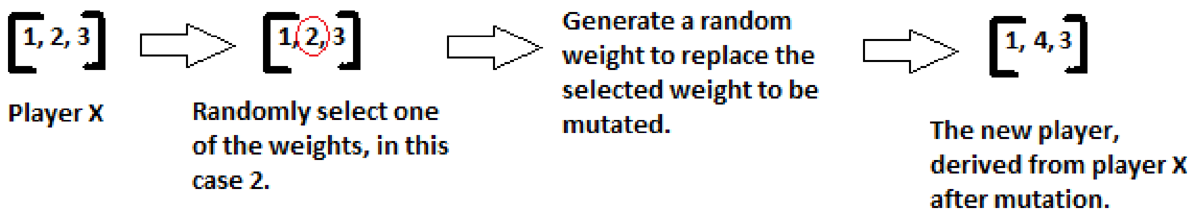
A 'Player'

In order to accurately score a board, we decided that a 'player' would consist of a list of three values or weights, such as [1,2,3]. This constituted an individual player with specific characteristics. As mentioned before in the discussion of *evalState*, we put importance on 1-, 2-, and 3-sequence windows that occurred on the playing board. In order to make different competitors, we decide to create a list of weights to represent a player as stated above, with each weight corresponding to a 1-, 2-, or 3-sequence for the current player. So for the above player, our *evalState* will now use the values [1,2,3] to score a board. This list represents the weights *evalState* will use for each *n*-sequence, so in this example, the weight for a 1-sequence is 1, the weight for a 2-sequence is 2, and the weight for a 3-sequence is 3. These values weigh the scores of a sequence and give more power to different sequences depending on the player.

Once the basic player definition was put into place, we had to create ways to manipulate those players, in order to diversify our search for the best set of weights we could find. To do so, we used crossovers and mutations on these objects. A cross over is a method which takes two of the players and switches some of their values. A cross-over would then occur as follows:


1, 2, 3
Player X
1, 4, 3
Player Y
1, 2, 3
Player X
1, 4, 3
Player Y
First, a cross-over point is chosen, in this case in between the second and third weights.
1, 4, 3
Player X 1
1, 2, 3
Player Y 1
Next, the numbers in front of that cross-over point are swapped between players, creating two completely new players.

A mutation is a method which will pick one of the three weights for the player and swap it out with a random number. This would look as follows:


1, 2, 3
Player X
1, 2, 3
Randomly select one of the weights, in this case 2.
Generate a random weight to replace the selected weight to be mutated.
1, 4, 3
The new player, derived from player X after mutation.

After this coding was completed, we had our players defined as well as our procedures to ensure the diversity of our weights. It was time to create the framework to play against one another.

Implementation

In its definition, a genetic algorithm is a search technique used to find solutions to an optimization or search problem. For us, it was all about optimizing our weights to get the most competitive computer player because the player with the best weights would then make the best moves and be the most challenging for other players. Our genetic algorithm generates several random players as discussed above, where the weights for each player are random numbers between 0 and 99. Once the players are generated, every player plays every other player. We then take the top winners and run the algorithm again with additional players; however, these players are not randomly generated. Instead, a number of new players are created using mutations and crossovers of the winning players. Thus, we have a new list of players of the same number as the previous round, with the hope that after many rounds the best players will begin to rise to the top and continuously win. The pseudo code is as follows:

```
GeneticAlgorithm(numRounds):

Input: The number of rounds you wish to simulate

Output:  After each round, the top four players are saved and passed
to the next round.  This continues until the round limit is reached,
and the final list of four winners from all the rounds is returned.

playersList = [] (empty list to be filled with "players")
for I in range Number_of_PLAYERS:
     playerXWeightList = [] (an instance of an individual player)
     on first run only :
          generate three random numbers between 0 and 99
          append to playerXWeightList
          append playerXWeightList to playerList
     any other run:
          mutate or crossover some of the previous rounds' winning
          players to create a new list based on and including previous
          winners
          for each player in playersList:
               play against every other player, recording a win as 1
               and a loss as -1 (round robin tournament)
          pick top n scores to move on, pass them to the next round
```

Once a select few players continue to win round after round, a threshold test is performed to determine if the genetic algorithm can stop. This test looks at each of the sequence weights of each of the winners separately and determines if they are within a threshold of each other. In other words, if the spreads of the weights of each sequence is less than the threshold, the algorithm stops. The spread is calculated by taking the difference of the maximum and the minimum of the weights for each sequence. The genetic algorithm can also stop after a specified number of rounds. The following is an example run of the genetic algorithm through one round with only two players:

```
Create two players with random weights:
     Player 1: [3,52,84]        Player 2: [52,34,47]
Play a round robin tournament:
     Player 1 vs. Player 2 (Player 1 goes first)
          Results in a win for Player 1
     Player 2 vs. Player 1 (Player 2 goes first)
          Results in a win for Player 1
Now Player 1 has a score of 2 (2 wins)
And Player 2 has a score of -2 (2 losses)
Player 1 will move on to the next round
Player 1 is mutated in the 2-seq slot to
     create a new Player 2: [3,83,84]
The algorithm continues from here with these two players
```

Experimentation

Once we had put into place all the code and had tested it to validate its correctness, it was now time to begin running tests in order to create the best Connect Four player possible given our constraints. When we decided to use a weighting system to create a more powerful computer player, our basic assumption was that a longer sequence would be more important, and thus should receive a higher weight. In the end, our hope was to identify a set of weights that, based upon our testing constraints and our *evalState*, gives us the most powerful computer player. To see if this intuition is true, we began running a series of different tests.

Setup

The first step in verifying our guess at a weighting system was to setup a series of experiments that would show us the differing behaviors of our computer player based on different factors. One distinction between tests was the number of rounds for which we would allow a simulation to run. Since our weights could range anywhere from 0-99, we assumed that the longer we allowed the simulation to run, the closer to the actual “best weights” we could get. The second factor we took into account was how far down the tree the minimax function was looking. For all experiments listed in this paper, the results are from a minimax function that searched to a depth of three; this means it looked three moves into the future in order to determine its best move. Once we began to test a depth of four, our hardware constraints became evident as the tests took far too long for us to run and obtain proper data. Once we had all this in place, we began running two identical sets of tests: two sets of 50-round simulations as well as two sets of 100-round simulations. At this point we had multiple valid data sets and the results were beginning to look like what we were expecting, but it led us to wonder if increasing the amount of rounds would help decrease our deviations. Thus, we began running two more sets of tests: one set to attempt to run for 200 rounds, and one set to run for 300 rounds.

To Summarize, we ran the following tests:

A) Search Depth of 3, 50 rounds
B) Search Depth of 3, 100 rounds
C) Search depth of 3, 200 rounds
D) Search depth of 3, 300 rounds

Tests A and B were run four times, with tests C and D run twice each.

Results

First, we calculated the averages and standard deviations based on the number of rounds.

Number of Rounds	1-seq Weight Average	1-seq Weight Standard Deviation	2-seq Weight	2-seq Weight Standard Deviation	3-seq Weight	3-seq Weight Standard Deviation
50	6.7	4.53	30.5	22.5	78.125	22.9
100	5.125	5.7	24.25	11.7	78.875	27.45
200	6	1	18	1.5	55.5	2.8
300	7	1.54	19.5	1.7	43.5	1.73

Then we calculated the ratios of 1-seq weights to 2-seq weights, and 1-seq weights to 3-seq weights. Here are the results:

Number of Rounds	(2-seq weight)/(1-seq weight)	(3-seq weight)/(1-seq weight)
50	4.5	11.6
100	4.7	15
200	3	9.25
300	2.8	6.21

Analysis

Now that we had collected all our data, we began to see some interesting trends appearing. In the beginning of our testing, we felt that if we increased the number of rounds in the simulation, the deviation would shrink considerably as the number of rounds in the trials was increased. This is due to the fact that to start we are using random number generation between 0 and 99 for the player's weights. The high values of these standard deviations in the 50- and 100-round trials led us to believe that we had not run our tests long enough to get an accurate set of weights for the "best player." Also, in looking at our ratio's in comparison, there was still a large difference in the (3-seq weight)/(1-seq weight) values for 50 and 100 rounds.

This led us into our next round of testing, which would involve increasing the number of rounds in the simulation. We felt that this would allow more weights to be tested and allow the best and most accurate weights to rise to the top, hopefully with a much lower deviation. As you can see from the results above, our expectations were correct. For both of the latter tests, all of our deviations were under 3, with only one being over 1.75. This showed that we had eliminated the outliers and were very close to the best set of possible weights. Without running this test exhaustively however, we will never reach the exact number of that best set. However, based on the ratios we received and the averages we calculated, the best relationship between weights is approximately *x*, 3**x* for a 2-seq weight, and 6**x* for a 3-seq weight where *x* is any value chosen for a 1-seq weight.

Thus we have found that in Connect Four the number of pieces in a sequence does determine its importance in scoring a state and it is an increasing relationship. As the number of pieces in the sequence increases, so does the importance of that sequence to a player's chance of winning, and thus our computer player's decision on where to move next. Our data gave us our best possible player based on our tests and using our evaluation function, along with following our time and hardware constraints. This player has a set of weights that correspond to a ratio of 1-3-6 for a one, two, and three sequence weight, respectively.

Future Possibilities

There are many ways one could go with this algorithm. For example, one with a lot of experience or knowledge in the game of Connect Four can implement some strategies that are used by Connect 4 players. Instead of just choosing places where the player can win, it would deliberately choose certain places according to some strategy in order to give itself a greater chance of winning. For example, it could intentionally set up a situation where the other player, no matter where they choose to place a piece, will lose in the following turn. An example of this is shown below:

```
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ _ o o _ _
_ _ x x x _ _
```

The ‘x’ player has two 3-sequences in this case. If ‘o’ blocks the win on the left, ‘x’ can simply place a piece on the other side. If ‘o’ blocks the win on the right, ‘x’ can simply place a piece on the left.

```
_ _ _ _ _ _ _              _ _ _ _ _ _ _
_ _ _ _ _ _ _              _ _ _ _ _ _ _
_ _ _ _ _ _ _              _ _ _ _ _ _ _
_ _ _ _ _ _ _              _ _ _ _ _ _ _
_ _ _ o o _ _              _ _ _ o o _ _
_ o x x x x _              _ x x x x o _
```

The minimax algorithm occasionally accomplishes this through its evaluation of future states, but not intentionally (other than in its desire to win). Other ways to evaluate the score of a state could also be developed. In order for the minimax algorithm to evaluate states further down the tree of expanded states, efficiency would have to be greatly increased. This can be accomplished a few ways - for example, our implementation could be translated into another language that supports multi-threading in order to speed up the minimax algorithm's expansion and evaluation of future game states. Different threads could simultaneously evaluate states in different branches. With seven threads, the minimax algorithm could evaluate each of the seven branches of expanded states nearly simultaneously. Our minimax player could also be used in real-world applications; if someone was developing a full Connect 4 computer game, they would most likely prefer to work on the cosmetics and functionality of the game rather than worry about what their computer player will do to win. Along those same lines, a difficulty modifier could also be implemented by using less successful weights, not looking “down the road” as far, or by scoring states with fewer tests.

Conclusion

After the hours of experimentation, our genetic algorithm was successful in finding an optimal set of weights for our computer-simulated Connect Four player. With these weights, the computer can use a concrete scoring system to determine how well it is doing given any state of the game. By “looking down the road” and evaluating possible future states with its implementation of the minimax algorithm, our computer player can make intelligent and challenging decisions every time it places a piece on the board. This makes our minimax computer player a very challenging and formidable opponent for any player of our Connect Four game.

Acknowledgments

Thanks to our project advisor, Dr. Steven Bogaerts, Assistant Professor of Computer Science at Wittenberg University.