Cylindrical Connect 4

Hunter Nieminski and Benjamin Maymir

May 2023

Abstract

The game connect four is a two player game traditionally played on a 7x6 grid. Players alternate placing a stone of their color in the first unoccupied row of any column. A player is declared the winner when they place a stone so that four of their stones are adjacent vertically, horizontally, or orthogonally. We can represent this this more precisely as

$$\exists R \in [0, 7], \exists C \in [0, 6]$$

such that

$$G_{R,C} = G_{R+1,C} = G_{R+2,C} = G_{R+3,C}$$

or

$$G_{R,C} = G_{R+1,C+1} = G_{R+2,C+2} = G_{R+3,C+3}$$

or

$$G_{R,C} = G_{R,C+1} = G_{R,C+2} = G_{R,C+3}$$

Connect four has been solved for nearly three decades, and it has been shown that with optimal play, the starting player will always win. We will examine a variation of connect four in which the normally flat game board is projected onto the surface of a cylinder. This introduces a new win condition in which the chain of four stones can wrap around the sides of the board. This is mathematically equivalent to taking the value modulo 7 when verifying if a turn resulted in victory for the player.

Problem description

Connect 4 is already solvable so we had to find a new variation that hasn't been done before to make for an interesting project. Wrapping the connect 4 game around a cylinder allowing for possible wins across edges creates a more complex game that is still simple to understand. For a real life implementation there would need to be some kind of win detector so players could know when they won as it would be slightly confusing to try and figure out where a win is when playing on a cylinder. This is a simpler implementation using a computer as

the computer can see everything without having to look around a cylinder.
There are multiple other variations on the classic board game such as "Connect 4 Spin" and "Connect 4 Blast" where there are small variations on the game such as spinning columns or shooting nerf guns with discs into the board. Connect 4 Spin does make the game more complex with possible changes to the board mid-game, and our implementation will also make the original game more complex.
This is a complex problem which adds many more possibilities to win and throughout this paper we will explore the problem in depth.

Background Review (Literature)

In the game of infinite cylindrical connect four, it is possible for for a game tree search to never terminate in positions that are drawn, however due to the undecidable nature of the halting problem, an optimal algorithm must be capable of interpreting patterns that lead to drawn positions. In their paper on solving cylinder-infinite connect four[1], the authors describe cannot-lose strategies for both the first and second players on infinite cylinders, except for cylinders with width 2, 6, and 11.

Another approach to training an agent to play connect four is to build a model through reinforcement learning. Rather than searching through the entire state space, a reinforcement algorithm rewards agents that make decisions that lead to winning positions, and punish decisions that lead to losing ones. In the paper *Reinforcement Learning with N-tuples on the Game Connect-4*, it is shown that this approach is sufficient to train an agent to play optimally through repeated games against an adversary running the same algorithm[2]. Since it is not always immediately obvious whether a move leads to a winning or losing scenario, the paper highlights a new system utilizing n tuples to track the relative strength of a move. This method was shown to be successful as it was demonstrated beating minimax algorithms in situations where wining was possible.

An additional approach taken towards solving connect four is to train existing algorithms on it. The AlphaZero AI has been used to create agents capable of exploring enormous state spaces. It was a pivotal part of one of the first go engines capable of beating the top humans at go, and is the basis for one of the best chess engines ever created. In the paper, *Improvements to Increase the Efficiency of the AlphaZero Algorithm: A Case Study in the Game 'Connect 4* several improvements are proposed for the AlphaZero using connect four as a benchmark[3]. It is very likely that a similar approach would be successful in training an agent to play cylindrical connect four.

The game of connect four, even without modifications, is a very complex game built upon simple rules. In order to train an agent to be able to solve a game such as connect four, the agent must be familiarized with the rules and win conditions. Traditionally this is hard coded by the creator of the agent, however *Learning to Win Games in a Few Examples: Using Game-Theory and Demonstrations to Learn the Win Conditions of a Connect Four Game* demonstrates a novel approach where the rules and representations are demonstrated by a human[4]. The paper demonstrated the ability of an agent to learn the win conditions

of connect four after a single demonstration and several questions initiated by the agent. This was shown to be irrespective of color or starting player, and the agent was also capable of adapting to a change in the win conditions after it was initially taught.

A possible extension of the project would be to expand the board to have infinite width instead of having the edges wrap. This would present additional challenges as not only would there be an infinite state space, making game tree search difficult, but there are an infinite number of possible moves for any given state, making most versions of mini max unviable. This particular variation is explored in *Infinite Connect-Four Is Solved: Draw* which demonstrates a strategy that is guarenteed to not lose[5] However we seek to create an agent or algorithm that is capable of beating opponents that do not engage in perfect play as well as drawing against opponents that play perfectly.

As seen so far, mini-max is our decided upon method of solving connect 4 due to its ability to predict the opponents move and then beating it. The main fallback to this method as mentioned in [6] is the fact that it takes up lots of memory and run-time to work. This is where alpha-beta pruning comes into play. Alpha-beta pruning cuts down the run-time significantly because it cuts off branches of moves so that our algorithm doesn't have to search every single possible move, reducing the total search time and allowing mini-max to both make good decisions and save memory. A good heuristic is also required for mini-max to optimally choose which moves are better than others, and [6] gives two potential heuristics: heuristic 1 uses the current state of the board to decide what the best possible move is, this is a very effective heuristic as it is tailor made to the game but it has high memory costs and will take more time to run. Heuristic 2 places values on each circle on the board and plays based off of this along with a simplified version of 1. For our project we will be implementing heuristic 2 because it is not as complex and allows for a deeper search due to less memory usage.

There are approximately $1.6 * 10^{13}$ possible board combinations possible, making this game feasibly impossible to completely calculate out completely in any reasonable amount of time. [7] computes a heuristic value for their alpha-beta pruning algorithm using the minimum number of moves to a win with any given move (or the maximum to lose) resulting in a min/max algorithm. All this algorithm is required to calculate are these numbers for each of the 7 possible moves, resulting in a much more cost effective solution. This paper was trying to find the optimal solution so some other optimizations were also tried including: An earlier determination of the winning state, and going for any winning path. For the earlier determination of a winning state, a win is identifiable in connect 4 one move before it happens and thus the search tree can be cut short by one move resulting in a faster search which would be very helpful in our project. Going for any winning path results in less computing time but sub-optimal solutions so we will not be using this.

In order to create a working connect 4 algorithm, we need to know how exactly the game works in order to format all of our code correctly. [8] has multiple control flow diagrams as shown at the bottom of this paper(Images 1 and 2). These completely break down the exact steps to a game of connect 4 and will serve as a blueprint to our code for this project. Knowing exactly how the steps go and in what order is very important if we are to have a

sensible product at the end of this project. These diagrams also implement a timer which we could use in our project to put another constraint on our algorithm and ensure a more consistent run-time. This gives us the bear bones of the project which will allow us to implement everything else we have discussed in this paper in an organized matter that will be easily readable. A difficulty is also implemented which would be an interesting idea, having an algorithm that runs at a lower depth to give players a chance against it would be a fun and practical addition to the project.

There are 4 different ai algorithms explored in [9] including random, defensive, aggressive, and mini-max. Random speaks for its-self in that it chooses one of seven possible moves at random. This is the worst of the four and usually results in a loss unless it gets lucky, so we will not be exploring it in this project. Defensive uses a heuristic that assigns point values to each move depending on how well they block the opponents pieces, no-matter what possible wins might be available. Aggressive is the opposite of defensive and only focuses on trying to get its-self the win. Aggressive does do some slight defense if the opponent is about to win but mainly focuses on getting to 4 in a row and doesn't look very far ahead in the search tree. Mini-max is the combination of aggressive and defensive giving an equal number of heuristic value to both offensive and defensive plays, while also looking far ahead in the search tree to obtain its heuristic. Random loses to all of the above, aggressive defeats defensive due to actually going for wins, and mini-max beats aggressive for taking the whole picture into account. We will stick with mini-max for our main project but may add in simplified aggressive and defensive versions for comparison.

For a possible example of a project like this, [10] is a project assigned to Cornell students where they had to create a connect 4 solver. They illustrate the steps of the project as follows: Construct the game state tree, associate values with each state (the heuristics discussed earlier), a winning board with have a value of infinity and a losing board has a value of negative infinity, evaluate the leaves, then the nodes of the parent tree, then finally implement this to a certain depth. For our uses in this project we will implement a choice in depth to the user for testing purposes. The amount of calculations is 7^n where n is the number of moves made, so we will test with multiple different depths to find an algorithm that consistently wins but is also efficient in its usage of time and memory.

Conclusion to Research

As discussed above there are many ways in which we can go about this project to try and gain the most information and have the most rock solid results possible. We will be using a wrap around board and creating our own heuristic function to make the optimal mini-max algorithm to play connect 4. Other algorithms will also be tested along with multiple different depths and heuristics, along with statistical proofs showing that the algorithm is optimal. This paper goes through much of the research done regarding this problem and gave us many ideas to implement into this project and with enough fine tuning we will be able to show all of this work throughout the rest of this project.

Approach

The majority of our research suggested that a game tree search was likely to yield the best results given the scope and time frame of the project. Initially we attempted to do an infinite depth search for exclusively game states that resulted in a win for either player, however with minimal optimization and 2^126 possible positions to evaluate, the search was too slow to compute even a single position. Instead we pivoted to use a depth limited search paired with a more powerful evaluation formula. Replacing the 3 state win/loss/draw evaluator, the new evaluation function finds the longest chain of stones for both players and subtracts the second players from the first. In the case where a player has a chain of four or more stones in a row, the evaluation function multiplies the length by 10 to ensure that it is the best possible move. Additionally, a change was added to the minimax algorithm to directly check for wins as otherwise it would evaluate positions where it could play a winning move the turn after it lost as even. With this approach, the algorithm could be run at depth 5 in a reasonable time. However in order to meet the 99% goal we set for ourselves, we needed to either improve our evaluation function or optimize our game tree search. We opted to add alpha-beta pruning to drastically reduce the number of branches explored. This time save resulted in roughly a 4x increase in speed, allowing depth 9 to be run efficiently.

Experiments

The focus of our experiments were games played against a random opponent. In order to reduce variance tests were done as three trials of 1000 games. Due to the time required to test at depth 9, only 1 trial was conducted. The results are as follows.

Depth	Wins	Losses
1	977	23
1	980	20
1	984	16
3	987	13
3	982	18
3	979	21
5	980	20
5	987	13
5	984	16
7	996	4
7	993	7
7	994	6
9	999	1

Analysis

From the experiments, it can be reasoned that our implementation was capable of making an intelligent move the majority of the time. However the relative improvement with additional depth was fairly lackluster. One possible cause is the simple evaluation function. A common trait demonstrated by the algorithm was to play the lowest possible row for many turns in a row. This is a result of the evaluation function treating all possible moves that do not win equally once a chain of 3 stones has been accomplished. In the event that 3 stones are lined up but blocked on both sides, the algorithm performs no better than random until a forced win sequence is discovered within its maximum depth. Additionally the evaluation function was by far the slowest part of the algorithm. It is possible that with a faster but weaker evaluation function, a higher depth could be explored for better results. There were several other improvements that could be made in future iterations, such as storing the result of every position evaluated in order to prune transposition positions, storing positions as two binary numbers rather than a string in order to evaluate positions using faster binary arithmetic, or a precomputed table of openings and endings.
In order to show that it is possible to create an agent that is able to effectively win games of cylindrical connect 4 using a limited depth search, we decided to determine the p value of the data presented in our experiments. Taking the average of the win percentage over each of the depths results in a mean of .9863, and the expected mean for a fair game of connect 4 is .5. Using the standard deviation of the results which is .055 and a sample size of 13, we calculated the z-score to be -115. This is a ridiculously high z-score and shows that for any reasonable alpha level our null hypothesis does not stand. This disproves the null hypothesis that the agent did not improve, and thus leads us to the result that this was a worthwhile experiment.

Conclusion

We have effectively shown that an agent operating with a limited depth search using alpha-beta pruning is able to consistently outplay a randomized bot, which implies that our new version of cylindrical connect 4 is able to be beaten using search algorithms. This was an interesting project in the idea that even with this new game, AI search algorithms are able to solve it effectively and work in a reasonable time frame especially at the lower depths. AI is a powerful tool and will be especially effective at creating solutions to tangible issues such as the one we presented here. We are finally left with the question, how far will AI be able to integrate itself into our culture?

Statements of Collaboration

Benjamin Maymir: abstract, half of background, approach, expiriments, half of analysis, code

Hunter Nieminski: description, half of background, conclusion to research, half of analysis, conclusion

References

[1] Y. Yamaguchi, T. Tanaka, and K. Yamaguchi, "Cylinder-infinite-connect-four except for widths 2, 6, and 11 is solved: Draw," in *Computers and Games* (H. J. van den Herik, H. Iida, and A. Plaat, eds.), (Cham), pp. 163–174, Springer International Publishing, 2014.

[2] M. Thill, P. Koch, and W. Konen, "Reinforcement learning with n-tuples on the game connect-4," in *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I 12*, pp. 184–194, Springer, 2012.

[3] C. Clausen, S. Reichhuber, I. Thomsen, and S. Tomforde, "Improvements to increase the efficiency of the alphazero algorithm: A case study in the game'connect 4'.," in *ICAART (2)*, pp. 803–811, 2021.

[4] A. Ayub and A. R. Wagner, "Learning to win games in a few examples: Using game-theory and demonstrations to learn the win conditions of a connect four game," in *Social Robotics* (S. S. Ge, J.-J. Cabibihan, M. A. Salichs, E. Broadbent, H. He, A. R. Wagner, and Á. Castro-González, eds.), (Cham), pp. 349–358, Springer International Publishing, 2018.

[5] Y. Yamaguchi, K. Yamaguchi, T. Tanaka, and T. Kaneko, "Infinite connect-four is solved: draw," in *Advances in Computer Games: 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers 13*, pp. 208–219, Springer, 2012.

[6] Y. H. Xiyu Kang, Yiqi Wang, *Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game*. Scientific Research Publishing Inc., 2019.

[7] G. Vandewiele, *Creating the (nearly) perfect connect-four bot with limited move time and file size*. Towards Data Science, 2017.

[8] A. S. Ahmad M. Sarhan and M. Shock, *Real-Time Connect 4 Game Using Artificial Intelligence*. Science Publications, 2009.

[9] C. Z. Erik Ibsen, *A Connect Four Playing AI Agent: Algorithm and Creation Process*. Khoury College of Computer Sciences Northeastern University Boston, MA, 2003.

[10] C. University, *Assignment A4: Connect Four AI*. Cornell University, 2014.

Pictures

Image 1:

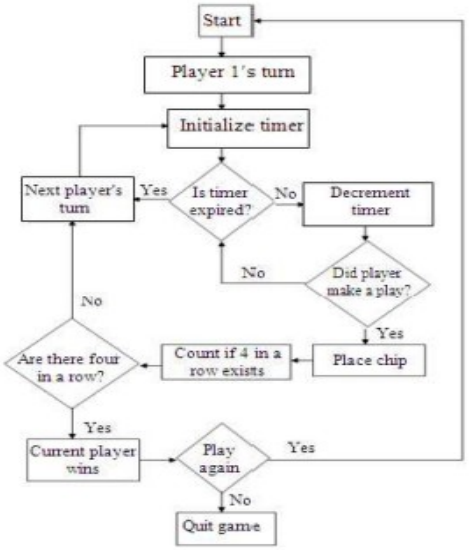

Fig. 2: Original software design flow chart

Image 2:

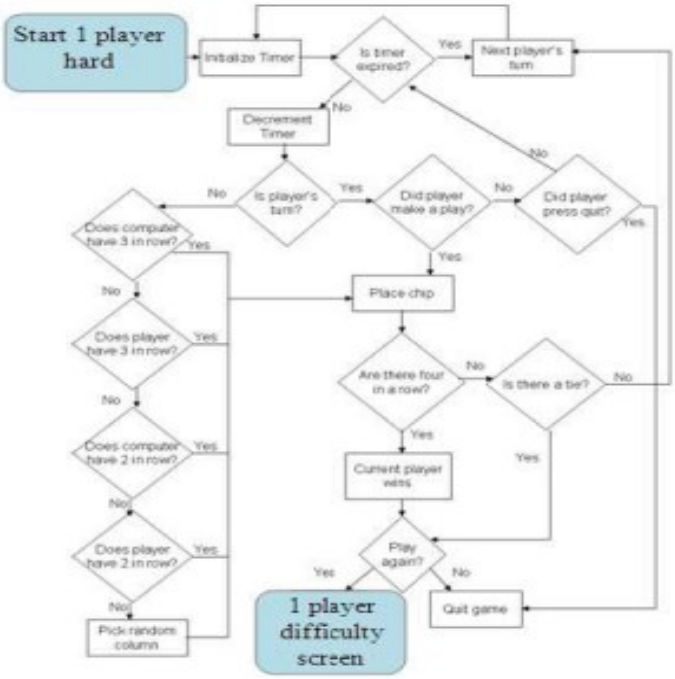

Fig. 10: Software flow for one player hard mode

Code Used:

```
    language
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

void displayBoard(char *board);
int takeTurn(char *board, int player, const char*);
int checkWin(char *board);
int checkFour(char *board, int, int, int, int);
int horizontalCheck(char *board);
int verticalCheck(char *board);
int diagonalCheck(char *board);
int evalBoard(char *board, const char *);
void maximize(int depth, char *board, const char*, int *ret, int alpha);
void minimize(int depth, char *board, const char*, int *ret, int beta);

int main(int argc, char *argv[]){
   const char *PIECES = "XO";
   char board[6 * 7];
   memset(board, ' ', 6 * 7);

   int turn, done = 0;

   for(turn = 0; turn < 6 * 7 && !done; turn++){
      displayBoard(board);
      int result[2];
      if(turn%2 == 0) {
        maximize(7, board, PIECES, result, INT_MAX);
      } else {
        minimize(7, board, PIECES, result, INT_MIN);
      }
      printf("Eval: %d\nBest Move: %d\n",result[0], result[1]+1);
      while(!takeTurn(board, turn % 2, PIECES)){
         displayBoard(board);
         printf("**Column full!**\n");
      }
      done = checkWin(board);
   }
   displayBoard(board);

   if(turn == 6 * 7 && !done){
      printf("It's a tie!\n");
   } else {
      turn--;
      printf("Player %d (%c) wins!\n", turn % 2 + 1, PIECES[turn % 2]);
   }

   return 0;

}
void displayBoard(char *board){
   int row, col;
   for(row = 0; row < 6; row++){
      for(col = 0; col < 7; col++){
         printf("%c", board[7 * row + col]);
      }
      printf("\n");
   }
   printf("1234567\n");

}
int takeTurn(char *board, int player, const char *PIECES){
   int row, col = 0;
   printf("Player %d (%c):\nEnter number coordinate: ", player + 1, PIECES[player]);

   while(1){
      if(1 != scanf("%d", &col) || col < 1 || col > 7 ){
         while(getchar() != '\n');
         printf("Number out of bounds! Try again.\n");
      } else {
         break;
      }
   }
   col--;

   for(row = 5; row >= 0; row--){
      if(board[7 * row + col] == ' '){
        board[7 * row + col] = PIECES[player];
        return 1;
      }
   }
   return 0;
```

```
}
int checkWin(char *board){
    return (horizontalCheck(board) || verticalCheck(board) || diagonalCheck(board));

}
int checkFour(char *board, int a, int b, int c, int d){
    return (board[a] == board[b] && board[b] == board[c] && board[c] == board[d] && board[a] != ' ');

}
int horizontalCheck(char *board){
    int row, col;

    for(row = 0; row < 6; row++){
       for(col = 0; col < 7; col++){
          if(checkFour(board, 7 * row + col, 7 * row + ((col + 1) % 7) ,
          7 * row + ((col + 2) % 7), 7 * row + ((col + 3) % 7))){
            return 1;
          }
       }
    }
    return 0;

}
int verticalCheck(char *board){
    int row, col;

    for(row = 0; row < 3; row++){
       for(col = 0; col < 7; col++){
          if(checkFour(board, 7 * row + col, 7 * row + col + 7, 7 * row + col + 14, 7 * row + col + 21)){
            return 1;
          }
       }
    }
    return 0;

}
int diagonalCheck(char *board){
   int row, col;

   for(row = 0; row < 3; row++){
      for(col = 0; col < 7; col++){
        if(checkFour(board, 7 * row + col, 7 * (row + 1) + ((col + 6) % 7),
        7 * (row + 2) + ((col + 5) % 7), 7 * (row + 3) + ((col + 4) % 7))
        || checkFour(board, 7 * row + col, 7 * (row + 1) + ((col + 1) % 7),
        7 * (row + 2) + ((col + 2) % 7), 7 * (row + 3) + ((col + 3) % 7))){
          return 1;
        }
      }
   }
   return 0;

}

int evalBoard(char *board, const char *PIECES){
  int row, col, max1 = 0, max2 = 0;
  for(row = 0; row < 6; row++){
    for(col = 0; col < 7; col ++){
      if(board[7 * row + col] != ' ') {
        int player = board[7 * row + col] == PIECES[0] ? 0 : 1;
        int length = 1, max = 0;
        while(board[7 * row + ((col + length) % 7)] == PIECES[player] && length < 7) {
          length ++;
        }
        max = length > max ? length : max;

        length = 1;
        while(row + length < 6 && board[7 * (row + length) + ((col + length) % 7)] == PIECES[player]) {
          length ++;
        }
        max = length > max ? length : max;

        length = 1;
        while(row + length < 6 && board[7 * (row + length) + ((col - length + 7) % 7)] == PIECES[player]) {
          length ++;
        }
        max = length > max ? length : max;

        length = 1;
        while(row + length < 6 && board[7 * (row + length) + col] == PIECES[player]) {
          length ++;
        }
        max = length > max ? length : max;

        if(max > 3) {
          max *= 10;
        }
        if(player == 0) {
```

```
            max1 = max > max1 ? max : max1;
          } else {
            max2 = max > max2 ? max : max2;
          }
        }
      }
    }
    return max1-max2;
}

void maximize(int depth, char *board, const char* PIECES, int *ret, int alpha) {
    int max = INT_MIN;
    int move = -1;
    if(depth == 0) {
      ret[0] = evalBoard(board, PIECES);
      ret[1] = move;
    } else if(checkWin(board)){
      ret[0] = max;
      ret[1] = move;
    } else {
      for(int col = 0; col < 7; col++) {
        char* newBoard = strdup(board);
        for(int row = 5; row >= 0; row--){
          if(newBoard[7 * row + col] == ' '){
            newBoard[7 * row + col] = PIECES[0];
            // displayBoard(newBoard);
            // printf("Depth: %d\nCol: %d\n",depth, col);
            int result[2];
            minimize(depth - 1, newBoard, PIECES, result, max);
            // printf("Eval: %d\nMax: %d\n\n", result[0], max);
            move = result[0] > max ? col : move;
            max = result[0] > max ? result[0] : max;
            if(max >= alpha) {
              ret[0] = max;
              ret[1] = move;
              return;
            }
            break;
          }
        }
      }
      if(move == -1) {
        max = evalBoard(board, PIECES);
      }
      ret[0] = max;
      ret[1] = move;
    }
}

void minimize(int depth, char*board, const char* PIECES, int *ret, int beta) {
    int min = INT_MAX;
    int move = -1;
    if(depth == 0) {
      ret[0] = evalBoard(board, PIECES);
      ret[1] = move;
    } else if(checkWin(board)){
      ret[0] = min;
      ret[1] = move;
    } else {
      for(int col = 0; col < 7; col++) {
        char* newBoard = strdup(board);
        for(int row = 5; row >= 0; row--){
          if(newBoard[7 * row + col] == ' '){
            newBoard[7 * row + col] = PIECES[1];
            // displayBoard(newBoard);
            // printf("Depth: %d\nCol: %d\n",depth, col);
            int result[2];
            maximize(depth - 1, newBoard, PIECES, result, min);
            // printf("Eval: %d\nMin: %d\n\n", result[0], min);
            move = result[0] < min ? col : move;
            min = result[0] < min ? result[0] : min;
            if(min <= beta) {
              ret[0] = min;
              ret[1] = move;
              return;
            }
            break;
          }
        }
      }
      if(move == -1) {
        min = evalBoard(board, PIECES);
      }
      ret[0] = min;
      ret[1] = move;
    }
}
```