Artificial intelligence

King Saud university/csc 361/second semester 2022

Connect-4 project

Instructor: Ms. EmanAlbilali

Student Name	Student ID
Aljawharh Alotaibi	441200846
Aljwhra Almakhdoub	441201225
Raghad Aljuhaimi	441201212

Contents

Introduction

Connect 4 game is a zero-sum deterministic perfect information game, which depends on the action of the agent also the other agent.

Using the random agent against the user is the fastest choice but the outcome of the implementation is not as required. The random agent chose any available spot without thinking of the outcome, so the chance of dropping the piece in the perfect spot in 6x7 board is 1/7. Additionally, if we want the chance of winning that would be approximate to 7^10 which is a huge probability to win. Thus, one solution for the problem is to use the minimax algorithm which gives the approximate best move to drop the piece in the board.

Description of minimax algorithm

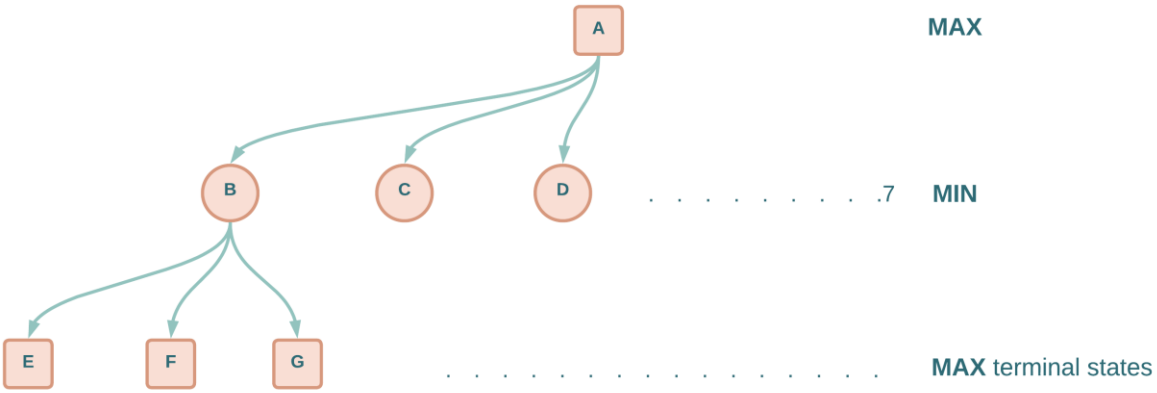

figure 1: minimax algorithm

The main idea behind the AI is adversarial search. It is to think of all the possible games of Connect Four represented as a tree. The root of the tree is the empty board. The second tier of the tree represents all the possible moves that can be made by the first player. The third tier of the tree represents all the possible moves that can be made by the second player.[1]

In figure 1 we have a board which will generate successors, the successors in minimizing states will predict the min player move in each column, the same goes for the maximizing player until we reach depth 3, if we reach depth 3 then we are in terminal state so we calculate the score for the move then return it to the max, the highest value shall be picked by the maximizing player. If we use alpha beta pruning the same prosses of picking the right move goes on, by pruning the unimportant scores it reduces the complexity to half of its time.

Minimax algorithm pseudocode

```
(* Initial call *)
minimax(origin, depth)

function  minimax(node, depth, maximizingPlayer) is
   if depth = 0 or node is a terminal node then
      return the heuristic value of node
function max
      value := −∞
      for each child of node do
         value := max(value, min(child, depth − 1))
      return value
function min
      value := +∞
      for each child of node do
         value := min(value, max(child, depth − 1))
      return value
```

Minimax alpha beta algorithm pseudocode

```
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)

function alphabeta(node, depth, α, β, maximizingPlayer) is
   if depth = 0 or node is a terminal node then
      return the heuristic value of node
function max
      value := −∞
      for each child of node do
         value := max(value, min(child, depth − 1, α, β))
         if value ≥ β then
            break (* β cutoff *)
         α := max(α, value)
```

```
    return value
function min
    value := +∞
    for each child of node do
        value := min(value, max(child, depth − 1, α, β))
        if value ≤ α then
            break (* α cutoff *)
        β := min(β, value)
    return value
```

The time complexity for alpha beta pruning

Worst case:

The worst case complexity of alpha-beta will be the same as the mini-max best case complexity i.e. O(b^d).

Best case:

The best case for a-b will be when all the moves are optimal, then the pruning can be done and a-b can prune almost half the node so the complexity is:

O(b^d/2).

It means that the a-b algorithm can search more than half the nodes compared to the min-max algorithm in the same time.

Average case:

Average case is a case in which the utility values for leave nodes were selected at random then the complexity will be: O(b^3d/4).

This case is the most practical situation. It also can be written as : O(b^0.75d).

Implemntation

State class

- Method- State dublicateBoard(State board): this method will make a duplicate of the received board to be used in the minimax algorithm so the main board will not be affected by the change.

- Method- LinkedList<State> getBoards(char agent): this method will generate the successor board from the received board by dropping a piece in each successor to generate the possibility and store it in linked list. The number of successors depends on the number of legal moves which has been known from the method getLegalAction() that checks if the column is free to drop the piece.

- Method-evaluationFunction(): this method evaluates the score of the received board if the dropped pieces are making a goal for the agent it will give it either positive high scores or negative high scores depending on whether it is a max or a min player, if we have different number of pieces then we use method calculateScore(char agent, double[] arr) then we multiply the number of 3 pieces with the high score and the number of 2 pieces multiplied by a less high score for the min agent 'X', and for the max agent 'O' has a less score than the min. then we subtract the min and max to return it.

- Method- calculateScore(char agent, double[] arr): this method will receive the agent and array of length 2 the first index is to calculate the number of 2 pieces and the second index the number of 3 pieces in the board row/column/left and right diagonal the method will initialize the value val which counts the number of the agent pieces. After each check we make it zero so we can check the other possibilities.

- Method- void scoring(int val, double[] arr): it increases the value of the array indexes depending on the value.

- Method- boolean checkForDraw(): this method has been used in the minimax algorithm to check if we reach the terminal state by checking if it is a draw board that means there is no empty spot to add a piece.

MinMax class

- The class contains a constructor to initialize the depth, the player(max) and enemy(min) agent. Also, the INF value is set to 99999.

- Method- Result minimax(State b): it has the initial call to the min method.

- Method- max(State b,int depth): since the method is a recursive method we first check if we reach the terminal state by checking if it the goal board or if we reach the last depth or the draw status. The method will return Move of type Result.

  The Move will contain the last move and the score from the evaluationFunction(). We initialize a Linked list of boards from the method getBoards in class state, we initialize it of type Result a maxMove then change it value to -INF, then for each board we send it to the min method after we reach terminal it will be stored in move of type Result. We check if the value of the maxMove equals the move, if they are equal we store them in an array then we choose a random index to set the move. Else we take the maximum value and set it to the maxMove. After the for loop ends we return the maxMove.

- Method- min(State b,int depth): the same prosses of the max method goes in the min but we choose the minimum value to return it.

AlphaBeta class

- The class contains a constructor to initialize the depth, the player(max) and enemy(min) agent. Also, the INF value is set to 99999.
- Method- Result minimax(State b): it has the initial call to the min method the same as minMax class.

- Method- max(State b,int depth,double alpha,double beta): since the method is a recursive method we first check if we reach the terminal state by checking if it the goal board or if we reach the last depth or the draw status. The method will return Move of type Result.

  The Move will contain the last move and the score from the evaluationFunction(). We initialize a Linked list of boards from the method getBoards in class state, we initialize it of type Result a maxMove then change it value to -INF, then for each board we send it to the min method after we reach terminal it will be stored in move of type Result. we check if the value of the maxMove equals the move, if they are equal we store them in an array then we choose a random index to set the move. Else we take the maximum value and set it to the maxMove, then we

compare the value with beta if it is bigger or equal then we return it and we don't explore the next boards. After the for loop ends we return the maxMove.

- Method- min(State b,int depth,double alpha,double beta): the same process of the max method goes in the min but we choose the minimum value and compare the value with alpha if the value is less than or equal to alpha then we don't have to explore the next boards. The method will return minMove.

Result class

- It contains constructor to initialize the move to -1 and the value to 0.

- Method- Result moveDone(int move): it will receive the move then make a new Result for that move.

- Method- possibleMove(int move,double value): it returns a new result that contains the move and value from the minimax algorithm.

- Method- Result changeValue(double value): it makes a new Result to store the new value and initialize the move.

Scores interface

- Contains the possible scores from average to very high.

Main class

- We create a new AlphaBeta class and new MinMax and the agent character is 'O'.

- Test the minimax and alpha beta against the random agent and human agent and catching the exception from the random and human agent which is an illegal move.

Results:

Human VS MiniMax agent Depth=5	choose your move 6 -------------- . . . . O O . . . . O O X . . . O O X X O X O X X X O X X -------------- O wins duration = 59624ms Duration in seconds = 59.624s
MiniMax agent Depth=5 VS Random agent	-------------- . . . . X O . . . . . O O . . X . O O X . . X O O X X X -------------- O wins duration = 613ms Duration in seconds = 0.613s
MiniMax agent Depth=7 VS Random agent	-------------- . . . . X . . X . . . O O O O . X X X O O X . -------------- O wins duration = 20179ms Duration in seconds = 20.179s

AlphaBeta agent Depth=5 VS Random agent	``` -------------- . . . . 0 . . . . . 0 . . . . . 0 X . . X . . 0 X . X X 0 . -------------- 0 wins duration = 315ms ``` Duration in seconds = 0.315s
AlphaBeta agent Depth=7 VS Random agent	``` -------------- . . . . 0 . . . . . . 0 . . . . . 0 0 X X . . . X 0 X X . . . -------------- 0 wins duration = 518ms ``` Duration in seconds = 0.518s

Discussion:

Based on the result we can see that the alpha beta agent has a better time performance, the difference of the time performance has appeared in depth 7, the minMax agent took 20 seconds to defeat the random agent, compared to alpha beta which took less than one second in the same depth. But minMax agent is more accurate to win since it explores all the possibilities. Also, if we increase the depth for both algorithms it will have a more accurate result. Alpha beta agent will have the same accuracy of the minimax algorithm if it has a higher depth.

The branching factor is for the minimax algortim approximately seven, and the depth of the tree could be as high as 42 so it will be around 7^42 which is a huge number of possiblites so the solution to the problem is to use a fast evalution function.

Albano and Orsini (1979) used a heuristic method that packs similar rectangles into strips for large-problem instances ranging between 400 to 4,000 rectangles[2]. Additionally, In (E. K. Burke …) they have demonstrated that, even with extremely large problems, the proposed best-fit heuristic is able to outperform currently published and established heuristic and metaheuristic methods to produce solutions that are very close to optimal, using very small execution times.[3]

One improvement could be made by implementing a broader range of policy networks and comparing their individual performances before attempting hyperparamter tuning, thus giving the best agent the best chance of developing a solved Connect 4 policy. In terms of developing a solved Connect 4 policy, a beneficial factor may be an increased set of rewards which coach the agent on a strategy that resembles the solved Connect 4 algorithm.[4]

There is many different approaches to solve the problem, and there can be many different algorithms to solve this problem, for example:

- Minimax.
- Minimax with alpha-beta pruning.
- Q Learning.
- Monte Carlo Tree Search.
- And many more.

Alpha beta pruning: time complexity O(b^(d/2))

Q Learning: However, in this paper, we depicted a reinforcement learning approach to the popular board game Connect Four in order to increase the probability of winning a game. Our Q-learning agent was able to achieve a peak win percentage of 73.4% when starting as player one versus the Sarsa agent. Alternatively when Sarsa started as player one versus Q-learning, our Sarsa agent was able to achieve a peak win percentage of 73.5%. Both algorithms never dipped below a win percentage of 50%, regardless of their opponent.
Sarsa agent maximizes over all of the possibleactions.
Time complexity O(en)[5]

Monte Carlo Tree Search: Main advantage of distributed MCTS is that its scale invariant to game complexity, i.e.to mk, where m is possible number of moves, and k is the variance of the outcome of a random game
Time complexity: O(mkI/C)[6]

Conclusion:

The complexity for minimax algorithm was the highset, but it can be so fast by using the proper evaluation function after researching for better algorithms, rather than using the minimax we found out that using the algorithm is the most accurate and fast solution for board game. From our implementation we found out that its suitable for the connect-4 game since it does not require high space memory and it’s really fast in depth of 5 and below.

Contribution:

Raghad: minimax algorithm/ evaluation function/ result class

Aljawharh Alotaibi: alpha beta algorithm/ result class/ testing

Aljwhra Almakhdoub: main testing/ report / contributed in evaluation function

Refrences:

[1] Rmarcus.info. n.d. *A JavaScript Connect Four AI · Ryan Marcus*. [online] Available at: <https://rmarcus.info/blog/2014/12/23/connect4.html>

[2] Albano, A., R. Orsini. 1979. A heuristic solution of the rectangular cutting stock problem. Comput. J. 23(4) 338–343.

[3] Burke, E., Kendall, G. and Whitwell, G., 2004. A New Placement Heuristic for the Orthogonal Stock-Cutting Problem. *Operations Research*, 52(4), pp.655-671.

[4] Thejacobthom.ca. n.d. [online] Available at: <https://thejacobthom.ca/wp-content/uploads/2021/11/SENG-474.pdf>

[5] *International Journal of Advanced Trends in Computer Science and Engineering*, 2021. Reinforcement learning in Connect 4 Game. 10(2), pp.578-582.

[6] *Stanford.edu. n.d. [online] Available at: <https://stanford.edu/~rezab/classes/cme323/S15/projects/montecarlo_search_tree_report.pdf>].*

https://github.com/raulgonzalezcz/Connect4-AI-Java/blob/master/src/Main.java

https://github.com/zakuraevs/connect4-ai/blob/a7e2e52745d68ae100d81293daabe422c710f01a/connect4_ai_no_comments.py#L33

https://github.com/kupshah/Connect-Four/blob/master/board.py

https://github.com/stratzilla/connect-four/blob/master/source.cpp

https://github.com/Gimu/connect-four-js