Assignment 1 : Adversarial Search

Christoffer Lindell Bolin

February 14, 2022

1 Description of the solution

I based my minmax Alpha-Beta pruning algorithm on the Wikipedia[1] pseudocode. The hardest part for me was not the implementation of the minmax Alpha-Beta algorithm, since I understood it pretty well. It was getting the evaluation function right due to my sheer lack of knowledge with python. One of the tasks of this assignment was to complete a single move within five seconds. Even though I have a high end computer, it grows exponential for each depth you traverse. Therefore I had to cut the search as demonstrated below in Python to reach the goal, which in turn required me to implement an evaluation function, more on that later. I found that a depth of six was the limit to meet the stated goal, although to win against the server a depth of five was plenty sufficient and much faster. Further optimizations could be done via implementing a transposition table, which keeps tracks of previously seen positions. Therefore the algorithm would not have to re-evaluate, saving precious time. I also cut the search if the move is considered to lead to a game over position.

```
def student_move(board, depth, alpha, beta, maximizing_player) -> (int,
    int):
    terminal = is_terminal(board)
    if depth == 0 or terminal != 0:
        match terminal:
            case 1: # Winning move player (Student)
                return math.inf, None
            case 2: # Winning move AI (Server/local)
                return -math.inf, None
            case 3: # Draw
                return 0, None

        # depth == 0, evaluate the position on the board
        return evaluate(board), None
```

As seen below, is the way I implemented the rest of the Minmax Alpha-Beta algorithm from the maximizing players perspective. The minimizing perspective is quite similar, but with a few differences. The full source code is available on GitHub.[2] The major difference from Wikipedia[1] is how I assign the value. This

[1] *Alpha–beta pruning*. visited on 2022-01-24. URL: https://en.wikipedia.org/wiki/Alphabeta_pruning

[2] *Github code*. URL: https://github.com/chrisse22222/AI_Course.

is due to that I also wanna track what move leads to a favorably position for the maximizing player.

```
    moves = available_moves(board) # get legal moves on the board
    candidate = random.choice(moves) # init candidate, in this case a
        random valid move
    if maximizing_player:
        value = -math.inf
        for move in moves:
            b_copy = board.copy() # copy boards current state
            place_piece(b_copy, move, PLAYER_PIECE)
            new_value = student_move(b_copy, depth - 1, alpha, beta,
                False)[0]

            if new_value > value: # max:(value, new_value)
                value = new_value
                candidate = move

            alpha = max(alpha, value)
            if value >= beta:
                break

        return value, candidate
```

The general idea I had in mind with the evaluation function was to score based on the number of two and three pieces in a row on the board from the maximizing and minimizing players perspective. Then I took the difference between them and got a value, which reflects the overall position on the board. The higher the value is, the better it is for the maximizing player and the opposite for the minimizing. Since I considered three in a row to be better than two in a row, the latter gave a higher score. The final addition to the evaluation function was to change so it favored the middle of the board, since connect-four is a solved game,[3] placing the pieces in the middle of the board gives the highest chance of winning.

```
    # Evaluates players (students) and "AI" position and compares them.
def evaluate(board) -> int:
    player_score = score_count(board, PLAYER_PIECE)
    ai_score = score_count(board, AI_PIECE)
    return player_score - ai_score
```

As stated before, I am a beginner with python with limited knowledge, I had to lookup in ways of implementing how to count the number of pieces in a row on the board. I would like to acknowledge Keith Galli[4] regarding this matter.

[3] Victor Allis. "A Knowledge-based Approach of Connect-Four". In: *Department of Mathematics and Computer Science Vrije Universiteit* (1988). URL: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf.

[4] Keith Galli. *How to Program a Connect 4 AI (implementing the minimax algorithm)*. visited on 2022-01-28. URL: https://www.youtube.com/watch?v=MMLtza3CZFM.

2 How to launch and use the solution

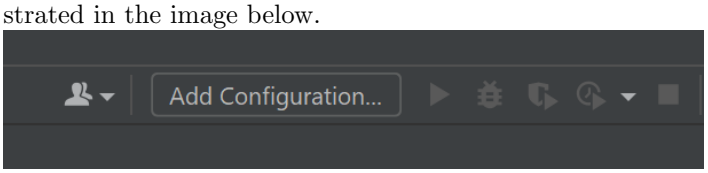
For my implementation I have utilized the template that was handed out from the course, written in Python. Follow the steps below in order to run it. I will base this guide on Pycharm IDE.

Prerequisites: Python 3.10, Pycharm

1. Start by downloading the source code from GitHub, preferably import to an IDE/editor of preference. In Pycharm select get from VCS.

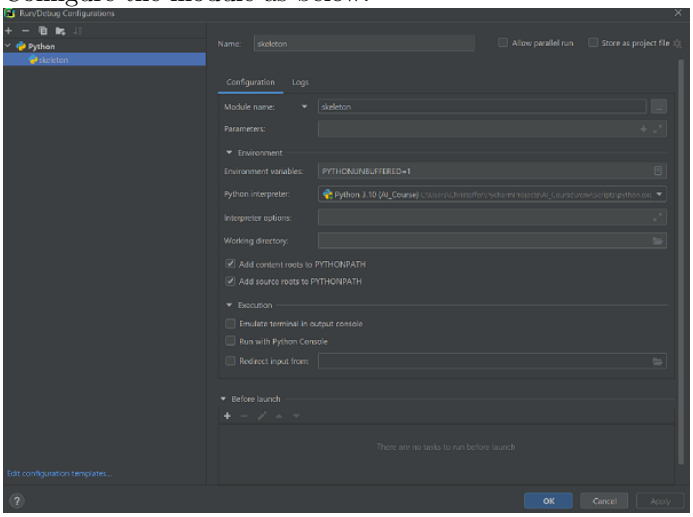
2. Select skeleton.py

3. In the right hand corner of the IDE, select add configuration as demonstrated in the image below.



4. A new window pops up which prompt you to add a new run configuration. Select add new run configuration and select Python.

5. Configure the module as below.



6. In the Parameters, depending on sought after action, type -l for local, -o for online, -s for stats and hit apply.

7. When playing against local as default it will ask you for moves, if you do not want this behavior follow the current step, otherwise skip this. Go to function opponents_move. If you rather want local to play random moves, comment out first action and uncomment last.

```
# TODO: Optional? change this to select actions with your policy too
# that way you get way more interesting games, and you can see if starting
# is enough to guarrantee a win
action = int(input("Select move between 0 - 6: "))# Play against AI (student)
#action = student_move(state, 3, -math.inf, math.inf, False)[1] # AI Against AI
#action = random.choice(list(avmoves)) # Random choice
```

8. Run the application by pressing the play button or alternatively shift + f10.

3 Peer-review

I have peer-reviewed with Alexander Ryde. We have during our peer-review session discussed points related to our solutions such as if our implementation is correct, if it is efficient enough, ways of improving it and so fourth.

3.1 Peer's Solution

Since we both have based our algorithms on the Wikipedia[1] Min-max alpha-beta pruning pseudocode, ours is quite similar with a few differences, mainly how our heuristic evaluation function works.

Regarding the evaluation function, he calculates the score from the players (student AI) perspective, and gives a score based on the number of two and three pieces in a row the player have. In this matter, ours is similar. However, he also penalizes if the opponent have three in a row since it is considered to be dangerous. Although, since since I evaluate from both perspectives (players and opponents) and calculate the $\Delta score$, this is not needed for me to get the intended behavior. Another noteworthy difference to point out is that he does not favor placing the pieces in the middle of the board, hence if ours would to play against each other, mine would most likely win.

3.2 Opinion and Performance

Alexander Ryde's soulution looks to be implemented correctly, it has a won consecutively more than 20 times in a row against the server AI.

To measure the performance of the solution, in python timers could be implemented that measure the time before t1, and after the move t2 has been made. Then you can calculate the Δt = t2 - t1. This gives a rough estimate of the time it takes to make a move. Although I want to point out that our hardware differs significantly which has a major affect on the performance. In order to get a fair comparison, both solutions were run on Alexander's hardware. During our peer-review we measured our solutions average time it took to make a move and it turned out that Alexander's solution is faster. Since he does not evaluate the board from two perspectives. Although the difference is considerably small. With a depth of five, he reaches the goal of completing a single move within five seconds.

In conclusion, Alexander's solution is faster but mine is considered to generally perform better.

4 Paper summary AlphaGo

A new algorithm has been developed my Google Deepmind[5] to tackle the problem artificial intelligence had with the widely more complex classical board game Go. Previous Go AI has been based on Monte Carlo tree search (MCTS) with enhanced policies to narrow the search. Despite this, evaluating all possible moves in the game of Go using this approach can not handle the sheer amount of possible moves 250^{150}. Using only MCTS have not been able to beet a human professional without handicap in a full game of Go and only been able to compete on a strong amateur play.

With the recent progression of deep convolutional neural networks, especially in the field of visual domains, Deepmind have been able to utilize the leanings and create a new algorithm AlphaGo[5] that combines from both worlds, the traditional approach with MCTS and neural networks. In AlphaGo, the current board position is passed in as an image and convolutional layers are used to construct a representation of the position. This effectively reduces the depth and breadth of the search tree. A supervised learning policy network ρ_σ and a value network v_θ was trained with the help of expert human moves. The result is an algorithm that is vastly superior that won 99.8% against other Go programs and even won a championship in 2015 against a professional player 5-0 without any handicap.

4.1 Difference to own solution

The main difference between my algoritm and AlphaGo[5] is that AlphaGo is a hybrid algorithm which combines a tree-based search (MCTS) and a learned model that incorporates neural networks while mine is purely a planning algorithm based on Minmax Alpha-Beta pruning. Regarding the evaluation function, mine is purely a hard coded Heuristic evaluation function based on the rules of connect-four where I determined what moves should be favored. Meanwhile, AlphaGo utilizes neural networks to determine what move is considered to be good or bad. Throughout the game, AlphaGo tracks a probability for winning and works on maximizing that probability by traversing states that have historically led to many wins. Monte Carlo Tree Search, which AlphaGo incorporates does not require a heuristic evaluation function since it reaches conclusions when terminal nodes are reached where it estimates a probability to win or loose. AlphaGo with a value policy network goes a step further where it estimates a value even before terminal node is reached. Generally mine and AlphaGo approach are on the opposite side of the spectrum, they are completely different.

[5] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* (2016). DOI: 10.1038/nature16961

4.2 Performance

Since connect-four is an uncomplicated, solved game that has in comparison to games such as Go few possible alternations, I doubt AlphaGo[5] approach would significantly alter the performance of my solution to be worth the substantial increase in computational power. It is worth to point out though that AlphaGo in the match against the professional Go Player Fan Hui, the algorithm evaluated thousands of times fewer positions than Deep Blue did in its chess match against Kasparov despite Go being a far more complex game. So there is obvious potential, although I still no not belive it to have a major impact in the case of Connect-four since the number of possible moves are limited. If the board would expand however, It might come in handy.

References

Allis, Victor. "A Knowledge-based Approach of Connect-Four". In: *Department of Mathematics and Computer Science Vrije Universiteit* (1988). URL: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf.

Alpha–beta pruning. visited on 2022-01-24. URL: https://en.wikipedia.org/wiki/Alphabeta_pruning.

Galli, Keith. *How to Program a Connect 4 AI (implementing the minimax algorithm)*. visited on 2022-01-28. URL: https://www.youtube.com/watch?v=MMLtza3CZFM.

Github code. URL: https://github.com/chrisse22222/AI_Course.

Silver, David et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* (2016). DOI: 10.1038/nature16961.