Assignment 1

By Olivia Mattsson (ol3270ma-s)

February 9, 2021

1 Description of the solution

The chosen approach for the implementation of a Connect Four solver was the minimax search algorithm with alpha beta pruning on a depth of 5.

Throughout this report, *the player* will refer to the Connect Four solver that was implemented whereas *the opponent* will be the server that it played against.

Whenever it is the player's turn, the current state of the game will be sent to the player's function where it will decide its move. The state is a numpy array in 2D, where a player move is represented as 1, the opponents move as -1 and an empty slot as 0.

The player's turn starts by making a copy of the current state, and then passing it on to the minimax algorithm. The algorithm returns a list of the chosen move together with the score of the move. The chosen move is an integer between 0-6, and represents in which column to play its next move. We will return to what the score represents when talking about the algorithm.

Minimax algorithm

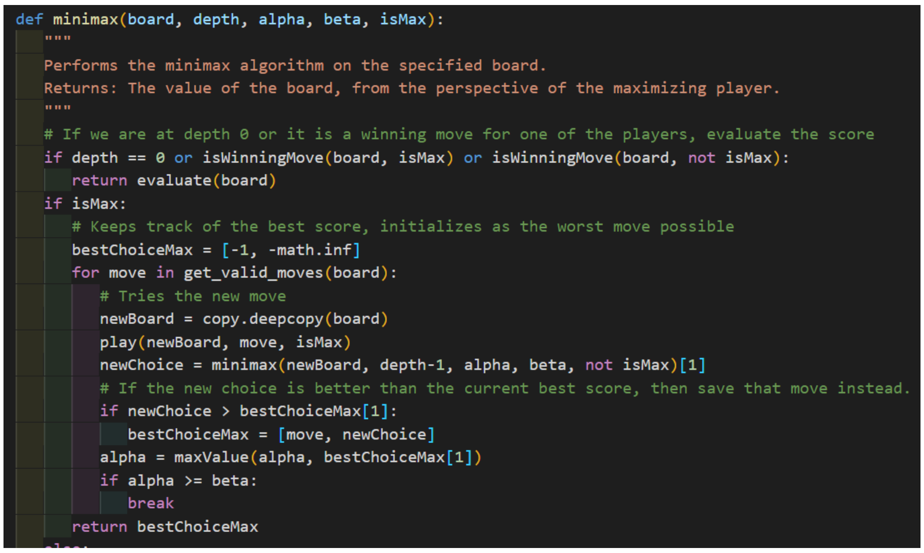
The recursive algorithm Minimax is often used different types of turn based games, trying to predict moves of their opponent and maximixing their gain for each move.

When deciding which move for the player to take, each of the possible moves are tested. And for each move that the player makes, the algorithm checks what the possible moves that the opponent could answer with and so forth. It predicts the different outcomes and evaluates them towards one another, trying to find the move where the player gets the highest value without knowing what move the opponent would take.

Since the algorithm is recursive, it would continue until all the possible moves are searched. This would be performance wise very slow and would not be able to decide for a move within 5 seconds. A maximum depth is introduced to speed up the deciding process. This implementation uses a depth of 5, where it still predicts a good amount of possible outcomes. It gives different outcomes and evaluates if the opponent is in a leading position as negative scores, and if the player is leading the score will be positive.

An excerpt from the minimax function is presented in Figure 1. The algorithm starts by checking if the maximum depth is reached or that there has been a winning move on the current board. If so, the evaluation function will be returned. If not, it checks whether it is the player or the opponent that called the function. The algorithm will go through the available moves, plays them as the player or the opponent and calls the minimax algorithm as the opposite player, simulating their response to the current move.

Eventually, each move would have been simulated and called to the depth of 0, where a score has been returned from the evaluation function. The current move would then be compared with the current best move. If it is the player that has called the function, the algorithm will look for the highest score between the current and the best move. If it is the opponent, then the lowest score would be saved. Eventually, it will go up back to the first call of the function, returning the best possible move with its score and that move would be played.



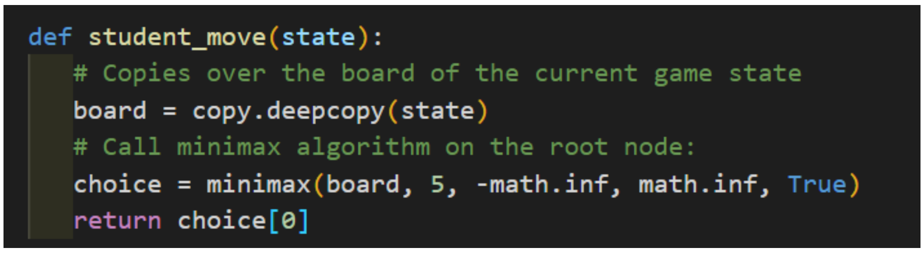
```
def minimax(board, depth, alpha, beta, isMax):
    """
    Performs the minimax algorithm on the specified board.
    Returns: The value of the board, from the perspective of the maximizing player.
    """
    # If we are at depth 0 or it is a winning move for one of the players, evaluate the score
    if depth == 0 or isWinningMove(board, isMax) or isWinningMove(board, not isMax):
        return evaluate(board)
    if isMax:
        # Keeps track of the best score, initializes as the worst move possible
        bestChoiceMax = [-1, -math.inf]
        for move in get_valid_moves(board):
            # Tries the new move
            newBoard = copy.deepcopy(board)
            play(newBoard, move, isMax)
            newChoice = minimax(newBoard, depth-1, alpha, beta, not isMax)[1]
            # If the new choice is better than the current best score, then save that move instead.
            if newChoice > bestChoiceMax[1]:
                bestChoiceMax = [move, newChoice]
            alpha = maxValue(alpha, bestChoiceMax[1])
            if alpha >= beta:
                break
        return bestChoiceMax
```

Figure 1: The minimax function

Aplha-Beta pruning

When deciding between all possible moves, Alpha-Beta pruning was used to limit the amount of searches of possible nodes. The values Alpha and Beta are used to stop the evaluation if the possible outcome would be worse than the ones that has already been examined.

In the initial call to the minimax algorithm, see Figure 2, alpha is set to the lowest and beta to the highest possible value. Throughout the minimax function, the values of alpha and beta will be propagated to the next calls. When the caller is the maximizing player, the value of alpha will be compared to the current best move (See Figure 1). If the best move has a higher score than alpha, alpha is set to the same score, whereas in the turn for the minimizing opponent beta is set to the lowest score between beta and the current best move for the opponent. Whenever alpha is greater than or equal to beta, we will break the search of moves. It means that the outcome will not be better than the ones that have already been searched.



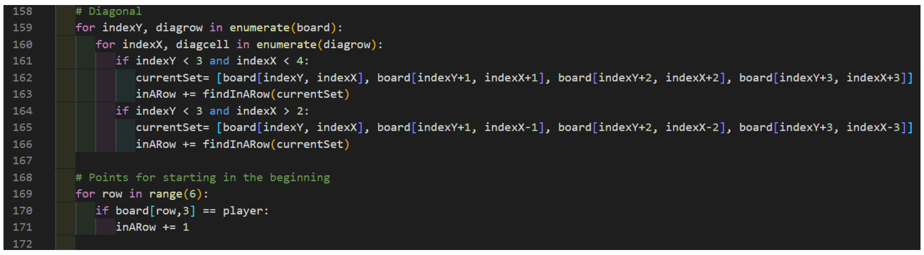
```python
def student_move(state):
    # Copies over the board of the current game state
    board = copy.deepcopy(state)
    # Call minimax algorithm on the root node:
    choice = minimax(board, 5, -math.inf, math.inf, True)
    return choice[0]
```

Figure 2: The initial call of the minimax function

Evaluation and heuristic functions

In the evaluation function, a set of four cells in a row is taken and passed to a heuristic function, determining the score of these cells. The evaluation function looks on horizontal, vertical, and diagonal sets. These scores are added to a total score of the entire board, and later on returned with a move of -1. This move will never be passed straight back to the initial call of the algorithm, but is simply there to follow the same return pattern as the function.

As seen in Figure 3, the possible digonal rows where four in a row is checked and sent to the function `inARow()`. The score of that row is then added to the total score. To make sure that the player constantly steers towards the middle columns in the beginning, a simple count of the player's moves in the middle is added. Otherwise, the player would always begin in the columns on the sides.



```python
    # Diagonal
    for indexY, diagrow in enumerate(board):
        for indexX, diagcell in enumerate(diagrow):
            if indexY < 3 and indexX < 4:
                currentSet= [board[indexY, indexX], board[indexY+1, indexX+1], board[indexY+2, indexX+2], board[indexY+3, indexX+3]]
                inARow += findInARow(currentSet)
            if indexY < 3 and indexX > 2:
                currentSet= [board[indexY, indexX], board[indexY+1, indexX-1], board[indexY+2, indexX-2], board[indexY+3, indexX-3]]
                inARow += findInARow(currentSet)

    # Points for starting in the beginning
    for row in range(6):
        if board[row,3] == player:
            inARow += 1

```

Figure 3: An excerpt of the evaluation function, checking diagonal rows to find their score.

The function `inARow()` is the heuristic function, where the different compositions of player and opponent moves are compared. When a row is passed into the function, it counts how many of the cells that belongs to the player, the opponent, and are empty. As seen in 4, the possible constellations will return different scores. If there is four in a row for the player, it should always play this move. If the opponent has four in a row, it will return the lowest score. To makes sure that the player will always try to block instead of playing two and three in a row, those values will return a bit more than playing these types of moves themselves.

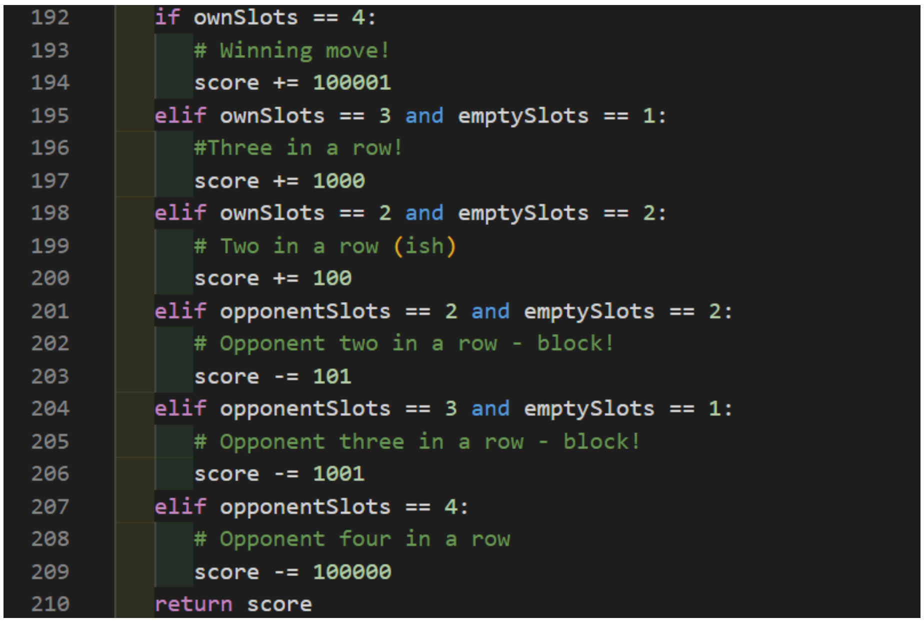

```
    if ownSlots == 4:
        # Winning move!
        score += 100001
    elif ownSlots == 3 and emptySlots == 1:
        #Three in a row!
        score += 1000
    elif ownSlots == 2 and emptySlots == 2:
        # Two in a row (ish)
        score += 100
    elif opponentSlots == 2 and emptySlots == 2:
        # Opponent two in a row - block!
        score -= 101
    elif opponentSlots == 3 and emptySlots == 1:
        # Opponent three in a row - block!
        score -= 1001
    elif opponentSlots == 4:
        # Opponent four in a row
        score -= 100000
    return score
```

Figure 4: An excerpt of the heuristic function `inARow()`, where the score is set.

2 How to launch and use the solution

Make sure that all of the prerequisites that is needed for the shell code are installed by running:

```
pip install requests
pip install numpy
pip install gym
pip install pygame
```

The rest of the packages that are used are included in Python 3. Launch the program as you would normally launch a Python program. The current version plays against the server.

3 Peer-review (1-2 pages)

The peer review was done on the solution of Sofia Cristiansson (so2877ch-s).

3.1 Peer's Solution

In my peer's implementation, the minimax algorithm happens in the function `student_move()`. The values of alpha, beta, depth, maxPlayer, and state are passed down from the `play_game()` function. The return values are the value of the selected move as well as the move, where the value looks like it has been used in debug purposes.

To retrieve the available moves, the global variable `env` is used with the built in function `available_moves()`. When playing a certain column, env is used as well and then resets back to the current state after the way has been tested. Other than that, the minimax algorithm follows the flow of the minimax / alpha-beta pruning algorithm.

When depth 0 is reached or the `step()` function returned a result that says that the game is finished, the `eval_func()` is returned. If the game isn't over or the depth is larger than 0, it will continuoulsy call `student_move()` as the maximizing or minimizing player and returns the best move for the players.

The `eval_func()` counts the current score of the board in the `env`, where it loops over the diagonal, vertical, and horizonal rows of four to add to the total score. The row is passed to `count_score()`, where the heuristic function is implemented.
The amount of zeros, minus ones and ones are counted, and depending on the constellation different scores are added. Similar to my solution, whenever the maximizing players have four, three, or two in a row they reward a positive score. When it is the minimizing player that has the same, the score will be negative.

3.2 Technical Differences of the Solutions

The algorithm implementation looks and behaves quite similar. While my solution uses the state and does not interact with the `env` variable, her solution uses the `env` and the built in functions for retrieving the available moves, playing a column and overall visualizes the upcoming moves by the `env`. As my first implementation used the `env` as well, I found that using `env` took a lot more time than just using the state variable and manipulating the array. My implementation uses a depth of 5, whereas my peer's implementation currently runs on depth 2. As I noticed by making the depth higher when using `env`, the execution time of the minimax algorithm scaled a lot quicker than with the use of the state itself, allowing me to have a higher depth at a shorter execution time.

Another important difference is the return values of our heuristic functions. Whereas my implementation gives scores that will enforce blocking by setting the score for the opponent's three in a row to -1001 and the player's score for the same result to 1000, my peer's solution gives the same score to both outcomes. I would argue that my implementation would put a bit more effort in making sure that the opponent would not end up in a position of winning than my peer's, given our implementations.

3.3 Opinion and Performance

Given the technical differences presented in the section above, there is some impacts on the performance of the two solutions. With a higher depth that is achieved in my solution, there are more possible outcomes that is examined in each move. I would argue, without having the peer's solution tested, that mine would try to avoid ending up in a bad position at a higher rate than my peer's. But if the depth was increased to the same as mine, there would be lees of a difference but would lead to the peer's solution taking longer time to determine each move.

The general heuristic and eval functions work the same, with the minor difference that mine interprets blocking three and two is a row as more rewarding than playing these moves myself. I am also rewarding the different constellations with a factor of 10 between each, as seen in Figure 4, whereas my peer's solution uses the values of ±10, ±5, and ±2. While I'm not sure how this would affect the outcome, there is less that differentiates the different moves in terms of score than my solution. I would argue that since my three in a rows reward a lot more than my two in a rows, the move selected would prefer a move which creates one three in a row instead of a pair of two in a rows. I'm assuming that the opposite could happen in my peer's solution, since the given scores are very close to each other.

These two differences argues for that my implementation would probably perform a bit better when facing a more advanced opponent.

4 Paper summary of AlphaGo

AlphaGo was the first computer program to beat a professional player in the full sized game of Go with no handicaps. It uses Monte Carlo Tree Search (MCTS) simulation combined with networks of values and policies in its search algorithm, leading to an impressive win-rate against other programs for Go.

AlphaGo uses a combination of supervised and reinforcement learning. Supervised learning takes data, often represented input-output pairs, to match new incoming inputs to the expected output. This is done in the first stage of the training, where the algorithm loads millions of positions from the KGS Go Server. The supervised learning network together with a rollout policy, e.g. how it should select the optional move, then learns to predict the upcoming move that would be done by a human expert.

In the next stage, the reinforcement learning (RL) is initiated on the supervised learning policy network (SL). The RL network will compete against earlier iterations of the policy network to make sure that the results would vary, and that the network did not *overfit* to its current version.

4.1 Difference to the Connect Four solution

First of all, AlphaGo uses a different algorithm than the one used in the solution presented in this report. While the Connect Four uses minimax search, AlphaGo uses MCTS. Monte Carlo combines the minimax search with an element of randomness, as described in the last section, instead of purely the evaluation function that has been implemented in this case.

In its search algorithm, AlphaGo stores several values in each node. While the current implementation only contains the scores of the different nodes, AlphaGo saves information about the probability of choosing the specific node in earlier games. Monte Carlo does estimates about its action value and more. It takes the most common plays into account and values them differently.

One could argue that the own solution only has a rollout policy, e.g. the evaluation function, while the AlphaGo solution have a lot of more factors taken into consideration.

4.2 Performance

This search algorithm needs a lot of training material. Since it is hardware heavy and uses a combination of supervised and reinforcement learning, it would take a while for the AI to become better. It would perhaps not even be possible to run on the current setup that the alpha-beta pruning was used on.

AlphaGo is also trained to predict human expert moves. When it is put against another machine, they might not do the moves that the AI expects. Since this is the case when playing against the given server, I would not see that this implementation would perform better. However, when facing better opponents and human players, this solution would probably excel. Since Go is a much more complex game, there are a lot more factors to take into consideration than just the score of each possible move.