Report of assignment 1

Joel Bäcker (jo4383ba-s) of EDAP01

February 9, 2022

1 Description of the solution

My solution is an implementation of the minimax-algorithm with the addition of alpha-beta pruning that tries to limit the number of visited nodes. The minimax-algorithm is a recursive algorithm that evaluates what the best move is in a n-player game by maximizing the player's score and minimizing the opponents' score. When alpha-beta pruning is added, nodes that never will yield a better move than does already found will not be explored further. The decision of when pruning is done with the variables α and β, where α is the minimum score of the player we are maximizing for and β is the maximum score of the opponents.

My implementation of the algorithm start of in the method `student_move` where all the available moves are found using the function `available_moves` found in the connect four environment package. For each available move a new state is created were the new move is applied. The next step calls the method `alpha_beta(curr_env, depth, alpha, beta, maximizing_player)` with the new state (`curr_env`) together with the starting values of the other which are `depth = MAX_DEPTH, alpha = -INF, beta = INF` and `maximizing_player = False`. The variable `MAX_DEPTH` is a global variable equal to 5. In the method `alpha_beta` the minimax and pruning is performed. The method can be broken down into three parts: when depth equals zero or the current state is a terminal one, maximizing a for play and minimizing for opponent.

1) If the bottom of the recursion is reached (5 moves ahead where depth is zero), no moves are available (the board is filled) or the victory condition is full filled by some player the board is evaluated and returned.

```
if depth == 0 or len(moves) == 0 or curr_env.is_win_state():
    return evaluate_board(curr_env.board)
```

2) When maximizing for the player all the available moves are explored and if any value is larger or equal than `beta` then the evaluation is stopped.

```
if maximizing_player:
    value = -INF
    for m in moves:
        new_env = copy.deepcopy(curr_env)
        new_env.change_player()
        new_env.step(m)

        value = max(value, alpha_beta(new_env, depth - 1,
    alpha, beta, False))
```

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

3) When instead the opposite is performed, the evaluation is stopped if any value is smaller or equal to `alpha`.

```
else:
    value = INF
    for m in moves:
        new_env = copy.deepcopy(curr_env)
        new_env.change_player()
        new_env.step(m)

        value = min(value, alpha_beta(new_env, depth - 1,
    alpha, beta, True))
        if value <= alpha:
            break
        beta = min(beta, value)
    return value
```

Evaluation function

The heuristic evaluation function will summarize the score of all lines (in every direction) of length four, the score of a given line is determined by how many discs of each player is present in a line. If two different player's discs are present in the same line, the score will be 0. The evaluation function will return a non zero value if the markers are of the same player (not counting zeros) and more than one is present. If there are four of the player's marker the maximum score will be given, and if the line is filled with the opponent's, the minimum score will be returned. The score for the opponent having 2 or 3 in a row is slightly larger (in absolute terms) than the player having it, the idea behind this is that is should be more favourable to block than to play aggressively.

2 How to launch and use the solution

I have implemented my solution in the skeleton code provided by the course administration, thus the solution is also dependant on the package `gym_connect_four`. Other packages that the program is dependant on is `logging`, `gym`, `random`, `requests`, `numpy`, `argparse`, `copy` and `json`.

As mentioned the program is written in the skeleton code file. To run the program one would simply type `python skeleton.py` together with an argument (note that python 3 is required). The argument description can be found in the skeleton code and is:

```
optional arguments:
  -h, --help    show this help message and exit
  -l, --local   Play locally
  -o, --online  Play online vs server
  -s, --stats   Show your current online stats
```

Example, if we want to play against the server we would use
`python skeleton.py -o`.

3 Peer-review

I peer-revied Fredrik Voigt (fr2405vo-s) and we discussed the points suggested on the Canvas page.

1) Because our solutions look a lot alike, we said that the implementations were correct.

2) We can test this by commenting out the part were pruning is performed, i.e. the `break`-lines above, and noticing that a move takes far longer than it otherwise does.

3) The requirements for the assignment states that each move should not take more than 5 seconds and win (or draw) 20 games in a row against the server, which both our solutions does.

4) We can easily measure the time it takes for us to make a move by clocking it and we can create a script that plays against the server 20 times and see how it performed.

5) Yes it does, it were removed and the performance dropped which suggests that the pruning works.

6) Yes, it evaluates when the depth has reached its limit or a wining condition is found.

7) It works by evaluating each "section" of a board and then summarizing all the evaluations. A section is defined as a possible lines (in every direction) that contains only one player, is of length four or longer and contains not more than one 0 in a row (a section can contain more than one 0 but cannot be next to each other).

8) The method performs well against the server and playing locally, I would say that it is good enough.

9) When playing against each other my implementation seemed to be better, and after some discussions we concluded that it were probably because of Fredrik's evaluation function not performing as well as mine that contributed to the differences. So if something should be improved it would probable be that.

3.1 Peer's Solution and Technical Differences of the Solutions

My and Fredrik's solutions were quite similar, the structure that I have in my code, described above, also appears in my peer's code. The difference between our solutions is the way we evaluate the board. Fredrik's approach is different from that I have done, his way of calculating a heuristic value of the board is described above in point 7.

However, one difference in our solutions is the way we handle the current board. I make a copy of the current environment (from the given code) and then make a step, this is done for all available moves and all recursive layers. This allows me to use the functions that already implemented in the environment.

Instead of this, Fredrik stores the board in a `Numpy` array, and thus he needed to implement his own methods to determine if the board is in a winning state or how to make a move. It should also be noted that Fredrik's code ran faster than mine, which is probably because he used his own methods.

3.2 Opinion and Performance

- Which differences are most important? As state above the most influential difference between mine and Fredrik's solutions is the way we keep track of the current state of the game as well as the evaluation method. Other than that the two solutions do not differ in a significant way. The implications of different functions are also discussed above.

- How does one assess the performance? We came up with one idea to asses the performance by setting up a friendly fight between our bots to see which of our solutions is the best. After completing 6 games it were found that my bot won 5 times and Fredrik's only 1, there seems to be some balancing between time and accuracy, my implementation is the slower one but seems to be the better of the two.

- Which solution would perform better? The scores hints that my solution is the more optimal one, but I believe that such a conclusion can not be drawn from such a small sample size. So with that in mind I can not definitely say which one is the best one. The only real conclusion is that mine in slower and Fredrik's is faster.

  One could perform a longer running test were the server (or simply performing more games against each other) is battled and then comparing number of wins and loses to then perhaps form a more rigid conclusion. The bot that would win, I cannot say, except that I hope mine is better.

4 Paper summary AlphaGo

The paper describes a new approach for artificial intelligence to play the board game Go, so well so that for the first time a computer program beat a professional player reliably. The new strategy does not necessarily explore more moves than the alternative Go programs, but it can better identify important moves and filtered the less significant ones. The success is made possible by a combination of supervised learning, reinforcement learning and Monte Carlo Tree Search (MCTS). The supervised part trained a so called "value network", used for evaluating a board, with data collected from professional Go matches and the reinforcement trained on data generated by playing against itself where a "policy network" were trained to select an optimal move.

Also worth saying is the board state is stored as 19×19 images, and with the use of convolutional layers to create a representation of the board.

4.1 Difference to the own solution

The differences between the two program is quite vast. What similarities there exists between the two solutions is they are both based on tree search algorithms, i.e. MCTS and minimax algorithm. But that is about as alike they are. The AlphaGo program combines the MCTS with machine learning to select moves, which is not implemented in my solution due to being far more complex than a simple minimax method.

For my evaluation function I had to manually program the weights (described above), however in the article they instead opted for a more general solution based on machine learning. The way they implemented an evaluation function where to approximate the prediction of the outcome of a certain move, these weights are found by training the model.

Another element in AlphaGo I found interesting is how the current board is stored as an image, and to determine and further investigate it convolutions are used. I suppose this is done to minimize memory usage and perhaps being able to store it some sort of a database where if a state that have already been examined appears again it do not need to do extensive evaluation again. Neither this approach is present in my.

4.2 Performance

Given sufficient time to train AlphaGo sibling on connect four instead of Go, I believe that there would be an increase in performance. Why this occur would mainly be because of the more sophisticated structure of AlphaGo, the selection of which move is optimal allows it to more explore fewer, but more important moves, deeper to determine what is the optimal one. However, given that a standard connect four board is quite small, the performances might be equal if the depth of search in my is higher. This is of course possible if we ignore the assignment's (or the article's) time requirement that a move should be calculated in 5 or fewer seconds.

If we would imagine that we would expand the game of connect four to a game on an arbitrary large board and length of winning length, then my solution would not be as good as the AlphaGo. When the size of the playing field increases the number of possible boards my solution goes through would

increase drastically. However, AlphaGo would need to train on this new board in order to be able to perform well but when that stage is done, the time for move selection would probably be shorter.