# Artificial Intelligence - Assignment 1

Szymon Stypa - sz6102st-s

February 9, 2021

## 1 Solution

In this first assignment, the task was to build a program playing and consistently not losing Connect Four games. This was to be done using the minimax algorithm for turn-based games in conjunction with alpha-beta pruning. A Python template with a complete environment for playing games locally and against the course server was provided. I chose to use this template as a starting point, since Python is a language I feel quite comfortable with.

### 1.1 Static Board Evaluation

I started this assignment by designing a static evaluation function for leaf nodes in the minimax algorithm. After a lengthy drawing session, I decided to treat the board as a matrix and use Numpy to extract each row, column and diagonal with length greater than three. This way, I get every sequence which might contain a four in a row as an array. To evaluate one such sequence, I step through it in quartets (four elements at a time) and count the number of ones and minus ones present. These values get stored in the variables `pos` and `neg` respectively. I also count the maximum number of consecutive duplicate elements (non-zero) and store that in `_max`. The actual scoring has the following logic.

```
if pos and not neg:
    score += pos * _max + 1000 * int(pos == 4)
if neg and not pos:
    score -= neg * _max + 1000 * int(neg == 4)
```

This works because a quartet does not give any points if it contains both coin types or none. However, if the quartet contains one and only one type of coin, we want to score it. To do so, we use the total number of coins in the quartet and multiply this with the maximum number of consecutive duplicates in the quartet. This logic means that the term is increasing linearly when putting coins in a valid quartet and exponentially when this insertion is also directly beside another allied coin. Keep in mind that these calculations are done for every quartet on all four axes. This means that positions which are members of many quartets (e.g center columns) will initially be more attractive to the algorithm than others. The exponential growth component along with the extra 1000 points makes the agent lean towards actually finishing the game (or blocking the opponent from doing so), as opposed to just placing coins in positions of high potential value.

## 1.2 Minimax

When researching practical examples of the algorithm, I stumbled upon a video by Sebastian Lague which brilliantly explained the minimax concept and implemented it in the form of a recursive function [1]. The video also extended to alpha-beta pruning, but I chose to put that part aside at first to get an intuitive feel for the algorithm and test my static evaluation. Naturally, I took a lot of inspiration from the code in the video and wrote a slightly modified version of this function which also keeps track of player moves.

```
def minimax(state, depth, _max=True, move=-1):
  children = next_states(state, 1 if _max else -1)

  if depth == 0 or len(children) == 0:
    return static_eval(state), move

  bestEval = -inf if _max else inf

  for col, child in children.items():
    currEval = minimax(child, depth-1, not _max)[0]
    if _max:
      if currEval > bestEval:
        bestEval = currEval
        move = col
    else:
      if currEval < bestEval:
        bestEval = currEval
        move = col

  return bestEval, move
```

As can be seen, the arguments consist of the current state, depth and goal of the call. The function starts by calculating all possible next states with a helper function which returns a dictionary of moves and the board states which that move results in. We then check if we should return a static evaluation, which is the case if we've reached a leaf node or if there are no more valid moves. Lastly, we loop through all children and evaluate them by calling the minimax function recursively with a decreased depth and opposite goal. Finally, we return either the best (maximum or minimum depending on goal) evaluation along with the move that produced it.

## 1.3 Alpha-Beta Pruning

To add alpha-beta pruning to the minimax algorithm, the arguments alpha and beta were added to the function. These represent the best scores either coin type can get assuming perfect play, and will accordingly be initiated as the lowest and highest possible value.

```
def minimax(state, depth, alpha=-inf, beta=inf, _max=True, move=-1)
```

These will of course be passed on in the recursive calls when traversing the tree. When evaluating child states, these values will be updated depending on the goal of the function call (max or min).

```
if _max:
  alpha = max(alpha, currEval)
else:
  beta = min(beta, currEval)
```

Finally, we will break the evaluation of children if alpha exceeds beta.

```
if beta <= alpha:
  break
```

Intuitively, this means that a parent further up the tree had a better option available already and will definitely not take the route through the direct parent of this child. An interesting note here is that I observed a noticeable performance improvement when ordering the children from the middle column moves and outward. This increases the rate at which pruning happens, as the inner columns are often times the ones containing the best moves.

# 2 Launch Solution

The solution mostly uses standard libraries, the only dependency being Numpy. Additionally, it requires all libraries used in the skeleton code. To play locally or versus server, simply set your preference in the main function of skeleton.py and run it. To play versus the agent, run playagent.py. Note that the code which actually plays the game lives in agent.py.

# 3 Peer-review

My peer review was done with Cem Alptürk, with whom I discussed and compared solutions in detail. Most importantly, we conversed about our minimax implementation, evaluation functions, performance improvements and the difficulties of debugging and evaluating ones code.

## 3.1 Peer's Solution

The solution presented by Cem was object oriented and slightly less compact than mine. He decided to represent his nodes as objects and call methods of these objects to traverse the minimax tree. The logic of the algorithm itself was, as expected, very similar to mine. Alpha-beta pruning was achieved in pretty much the exact same way. The evaluation function on the other hand, uses pre-calculated point values for every position on the board. These point values represented all the possible four-in-rows a certain position could be a member of.

```
scores = [[3, 4,  5,  7,  5, 4, 3],
          [4, 6,  8, 10,  8, 6, 4],
          [5, 8, 11, 13, 11, 8, 5],
          [5, 8, 11, 13, 11, 8, 5],
          [4, 6,  8, 10,  8, 6, 4],
          [3, 4,  5,  7,  5, 4, 3]]
```

To statically score a board, he simply looks up the corresponding point values for every coin position and adds it (or subtracts, depending on coin type) to the total board score. He also divides the score with the total number of points to normalize the values.

### 3.2 Technical Differences of the Solutions

Cem's solution was very different from mine in many aspects, although very similar in others. As has been mentioned, his solution was object oriented while mine consisted of a total of five functions. One might think that the evaluation was very different, but the pre-calculated scores actually correspond to the first term in my scoring function. If I were to evaluate a board with zeros and only one coin, that coin (depending on position) would yield exactly the amount of points shown in Cem's pre-calculated scoring matrix. However as we discussed, there are two problems with this evaluation. First off, it does not take into account that the opponent might have blocked the path for some of the potential four-in-rows which a position is part of. Secondly, pre-calculated scores for positions mean that the scoring only takes into account the potential value of certain positions and does not care if there are coins in a row or not - making it vulnerable to unorthodox tactics.

### 3.3 Opinion and Performance

During the peer review, we talked about the difficulties of evaluating ones solution. Because of all the recursive calls and the sheer amount of computations, it's really hard to keep track of what your program is doing. Mine, for example, worked best at depth 4 while Cem observed good performance at depth 6. We tried to reason our way to why that is, but the truth is that we're not sure at this point. We both agreed that it would be way easier to assess the quality of a solution if beating the course server was more of a challenge. Given the differences in evaluation, I'm tempted to assume that my solution would beat Cem's. When it comes to computation times, I suspect that his solution would win as there are substantially less computations being done for every evaluation.

An interesting thought which Cem brought up was that one could in theory increase the depth further in to the game without sacrificing performance. At the beginning of a game, every one of the seven moves should be expanded and explored by the algorithm (if not pruned) every turn. This means that $7^4 = 2401$ nodes are explored assuming a depth of four. However, if we fill for example four of the columns, the remaining three moves could be explored at a depth of seven ($3^7 = 2187$) without exceeding the original number of explored nodes per turn. It is however not clear if that would actually improve the algorithm that much. Another idea we discussed was treating boards and sequences as strings and using bitwise operations to calculate scores. This could in theory speed up

the evaluation by a lot and in turn enable higher depths, so it might be worth exploring if we ever try to improve our bots.

# 4 Paper summary AlphaGo

The paper on AlphaGo starts by discussing the computational difficulties with games of high depth and breadth, and different approaches for tackling these problems. First off, the depth can be reduced by truncating the tree and using an approximate value function which predicts the outcome from a given state. Next, the breadth can be reduced by using a policy of probability distributions to narrow down the number of branches. Before AlphaGo, the strongest go programs used such policies derived from data of professional human go player moves. This project, on the other hand, uses neural networks for both breadth and depth reduction. The training pipeline consists of a supervised learning (SL) policy network which starts off by imitating human moves. This is done with randomly sampled action-state pairs from expert games and stochastic gradient ascent. Another policy, the rollout policy, is also trained for faster predictions. The weights of the SL policy are then copied to a reinforcement learning (RL) policy which learns through self play versus random previous iterations of itself. At this point, the RL policy wins 80 percent of the games versus the SL policy. The RL policy is then used to fit an estimate value function with regression on state-outcome pairs.

The search itself combines the policy networks and estimated value function with the Monte Carlo Tree Search algorithm. Every node of the search tree stores the action value, visit count and prior probability. The tree is traversed by simulation, where each action is selected by maximizing the action value plus a bonus term which consists of a relation between prior probability and visit count. When the search reaches a leaf node, the SL policy is used to assign prior probabilities for each action. The leaf node itself is evaluated by the estimated value function (derived from RL policy) in combination with a rollout done by the rollout policy. At the end of a simulation, the action values and visit counts of all traversed nodes are updated accordingly.

## 4.1 Difference to the own solution

The main difference of my solution is that the breadth reduction is non-existing and the depth reduction is way simpler. Every time an action is to be selected, the breadth of the search is equal to the number of possible moves. This is more reasonable for connect four, since the breadth will never exceed seven. The depth on the other hand, is set to be a static value which never changes. At a maximum of this depth value, an evaluation is always performed. This evaluation is also very simple compared to that of AlphaGo, which uses a function regressed with the help of a RL policy together with a fast rollout. My solution simply uses a basic board evaluation function for this task.

## 4.2 Performance

My solution might not be perfect, but I don't think that any of the mechanisms involved in reducing depth and breath for AlphaGo would make much sense in

this scenario. Statically setting a depth and using a simple evaluation function with clever rules is probably enough to make an unbeatable connect-four bot. The reason I believe that is the case is simply the fact that it is quite easy to determine who is winning and who is losing in connect-four. In go, however, that's one of the more difficult things to do.

# References

[1] S. Lague, "Algorithms explained – minimax and alpha-beta pruning." https://www.youtube.com/watch?v=l-hh51ncgDI.