Introduction to Artificial Intelligence (2022)

Group Project 1
Othello

Student 1: Bjørnar Haugstad Jåtten (bjja@itu.dk)

Student 2: Jakob Henriksen (jarh@itu.dk)

Student 3: Danyal Yorulmaz (dayo@itu.dk)

Student 4: Jonas Borella Jakobsen (jboj@itu.dk)

Group: 4

Date: 17.03.2022

Introduction to Artificial Intelligence 2022

IT UNIVERSITY OF COPENHAGEN

0.1 Search algorithm explanation

For our search algorithm we used minimax search with pruning to optimize it.

This algorithm's purpose is to find the best move for MAX by trying all actions, comparing the resulting states and then choosing the state with the highest minimax value.

The search algorithm is recursive and goes through to the bottom node(s) of the tree and then goes back up again while noting all the minimax values through the tree. In order to note all the minimax values, our algorithm uses the *findScore* function, which is explained *evaluation explanation* part. Let's say the max depth of our tree is d and the number of legal moves is m then we have that:
The time complexity: $O(m^d)$
The space complexity: $O(md)$ [1]

The way our search algorithm is then optimized with pruning is that it runs through the tree is by performing a depth-first exploration of the tree. Along the way it also updates the values of *alpha* and *beta* in order to prune the remaining branches at a node. In an ideal case, this helps us reduce to the need for examined nodes to only $O(b^{m/2})$, instead of ($O(b^m)$ for minimax. [2] The alpha-beta pruning addition, as well as depth cut-off is explained further in section *0.2*

The code is recursively calling the minValue and maxValue functions, until if either reaches the given depth (as describes in the *cut-off function explanation* part), or if the game is finished. The way we check if the game is finished is by calling the *isFinished*-function, which checks if there is more legal moves left.

Both *minValue* and *maxValue* both evaluates all of the legal moves at the current game state. This evaluation is then compared to the already given value. For *minValue*, if the evaluated value is less than the already given one, the value is updated with the evaluated value. If the new value is smaller than the already given beta value, the beta value is updated as well.
The same goes for *maxValue*, which is updated when the new value is greater than the existing one. Instead of the beta value, it is the alpha value that is updated in *maxValue*.

0.2 Evaluation explanation

As Othello leaves no doubt if the outcome of an action is either beneficial or an disadvantage for a given player, it is ideal to implement an evaluation function.

In our implementation, the evaluation replaces the utility in the pseudo-code. The evaluation function we have implemented is quite simple - it takes a game state as a parameter, and retrieves the scores from this (using the countTokens-function). Depending on the current player, which is a global variable, it subtracts the score of the opponent from the one of the current player, and returns the result.

The result is used to give the value (utility) for the reached end-point of the search. The score which the value holds, is based on the tokens played.

0.3 Cut-off function explanation

As the number of game states in our mini-max tree is exponential to the depth of the tree [2], we need to implement some sort of mechanism to avoid going trough all states, as this would take to much time. By adding pruning, we can avoid the game states that makes no difference to the outcome, but still be able to compute the right answer.

As suggested by the assignment, we have implemented alpha-beta pruning as our pruning technique, based on the pseudo-code on page 200. The main difference from our original implementation is the addition of the two integers, alpha and beta. These are passed as parameters in the max- and min-value functions. Alpha represents the value of the best choice found so far for max, and beta represents the same just for min. [2]

When looping through our tree, we are constantly using these alpha and beta values to see if we know about a higher value further up in the tree. This done by passing along the alpha and beta as parameters, so that we are always able to cut and update them when traversing trough the tree. Alpha and beta needs to be known by both max and min, as it will be used both in the function itself, as well as the in the call for the other function. Max needs to know alpha, as it needs to know if it should be improved and thus updated, and we know that the min will call according to it. Max also needs to know beta, as it might cut according to it, and Min might improve it.

We have also added a depth cut-off function, which cuts off the search when reached a given depth. This is done by adding one to a step counter for each call to minValue and maxValue, and then checking each time if this step counter goes beyond the given depth.

When testing, we found that our implementation can return a move in under 10 seconds when the depth is less than 8. When exceeding 7, the algorithm used beyond the maximum time to deliver a move.

Character count: $4771 \approx 2.6 pages$

Bibliography

[1] Russel Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition, 4ed.* Pearson Education Limited, 2020. pp. 195-197.

[2] Russel Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition, 4ed.* Pearson Education Limited, 2020. pp. 198-201.