Wrocław University
of Science and Technology

Introduction to Artificial Intelligence

Laboratory 3: Game Playing Algorithm

By:

Sindy Kola 245903

Introduction

To conduct the required experiments and formulate my solution to this laboratory assignment, I have chosen the board game Checkers, otherwise known as Draughts, where the objective of the game is to remove all of the opponents pieces from the board, and thus, win. Following the report requirements described in the assignment given, I shall go over the following points(with the addition of some noted remarks):

1. Short description of the rules for the chosen game

Board:

1. Played on a 10x10 board
2. Only the darker fields are in the game

Player:

1. Players start on opposite sides of the board
2. One player is assigned the black piece set. The other, the white piece set.
3. Players alternate turns
4. Player may not move the pieces of the opponent
5. Each player starts with 10 uncrowned pieces

Pieces(uncrowned):

1. Can move one unit diagonally both forwards and backwards
2. Can capture the opponents piece by moving two subsequent units on the same line, jumping over the piece on the first unit.
3. Multiple pieces can be captured in a single turn, provided that it is done by successive jumps by a single piece (the jumps do not need to be on the same line, thus the diagonal direction can change)
4. Can become a King when it reaches the farthest in the board.

Kings:

1. Can move any distance along unblocked diagonals
2. Can capture an opponent’s uncrowned piece by jumping to any of the unoccupied squares right behind it.

2. Short description of the min-max algorithm with pseudocode

Min-max algorithm, or otherwise known as Minimax algorithm is a backtracking algorithm used in decision making and for our purposes, it will be used in this Game Playing Algorithm assignment. This algorithm provides an optimal move for the player under the general assumption is that the opponent is also playing optimally.
In this algorithm, two player play the game. One player is called the Maximizer(MAX) and the other is called Minimizer(MIN). Both "play" with the goal that the opponent gets the minimum benefit while they themselves get the maximum benefit.
For the exploration of the whole game tree a DFS is performed. The Minimax algorithm proceeds down to the terminal node of the tree, then it backtracks.

Pseudo-Code for Minimax

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)        //gives Maximum of the values
return maxEva

else
 minEva= +infinity
 for each child of node do
 eva= minimax(child, depth-1, true)
 minEva= min(minEva, eva)        //gives minimum of the values
 return minEva
```

3. Short description of alpha-beta algorithm with pseudocode

Alpha-beta pruning is a modified version of the minimax algorithm designed to improve performance. In other words, it is an optimisation technique. This technique involves a two threshold parameter Alpha and Beta for future expansion, and by which, without checking each node of the game tree, we can compute the correct minimax decision.

The main condition required is: $\alpha >= \beta$

This technique can be applied to any depth and sometimes, other than pruning the leaves of the tree, it prunes an entire sub-tree. As mentioned above, it is characterised by two parameters:

1. Alpha:
   - Initial value = -infinity
   - The highest value choice we have found so far at any point along the path of Maximizer
2. Beta:
   - Initial value = +infinity
   - The lowest value choice we have found so far at any point along the path of the Minimizer

Pseudo-Code for Alpha-Beta

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break
 return maxEva

else
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break
 return minEva
```

3. Short description of chosen evaluation functions

In my proposed solution I have two evaluation functions, namely:

1. *def simple_score(game, player)*
   This evaluation is quite simple, as the name suggest. It will calculate opposite side points. It assigns a set number of points to the two types of pieces(uncrowned = 100, kings = 175). Depending on what pieces are captured, the points are given.
2. *def piece_rank(game, player)*
   This evaluation method gives points depending on the "rank" of the piece. It checks their position on the board and it evaluates by that means. Additionally it subtracts points, in the case when a king becomes trapped on an edge.

4. Short description of chosen available move generation function

The function "*def avail_moves(board, player)*" will store the moves and jumps that are available to make on a list and then in the end return that list. At first it will traverse the 10x10 board and will check if pieces are placed on the board at the specific index of every iteration. Then at first, it will check if any jumps can be performed (this is done by a separate function). If a jump can be made, it will append that position to the list of moves.
If there are no jumps to be made, it will check next for any possible moves. It will do so very similarly as has been described for the jumps.

5. Analysis of performed tests

For the sake of simplicity, during the tests performed, a ply depth of 4 was used for both players of the game to conduct the experiments. Those experiments yielded the results as follows in the table.

Row: Player 1 Column:Player 2	**Minimax**	**Alpha-beta**	**Alpha-beta + ordering**
Minimax	210.34	135.86	93.26
Alpha-beta	131.85	58.78	*
Alpha-beta + ordering	106.81	*	43.17

***The additional heuristic for performance improvement was implemented as an extension to the Alpha-beta itself, therefore comparing against "normal" Alpha-beta would make no difference.*

As observed from the results collected from different experiments performed, the results are as one would expect from the study of the algorithms themselves.

At the set ply depth of 4, when both players utilise the Minimax game strategy, the execution time is the highest.
When both use Alpha-beta, the execution time dramatically decreases at a value of 58.78.
When experimenting with a combination of those strategies for the two players, intermediate values are yielded.

Finally, after adding the extra heuristic to improve performance, execution time decreased even more to the final and best value of 43.17.

Analysis of the tasks for extra 3 points:
Implementation of selected heuristics for improving the complexity of the game playing algorithm based on alpha-beta pruning scheme.

Since Alpha-beta pruning is very dependent on the order in which each node is examined, consequentially, the move order is an important aspect of this algorithm.
For the purpose of this task I chose to implement, the most natural and maybe straightforward heuristic, that is ordering. This heuristic performs ordering to search earlier parts of the tree that are likely to force alpha-beta cutoffs.

Even though most sources claim that ordering or sorting the nodes in such way that we try the most likely ones first(i.e those most likely to force an alpha/beta cutoff) will improve performance without changing results, it appears to not always be completely true.
Alpha-beta pruning will not change the value of the result. That means that if the regular minimax says that move *X* is worth 9, according to the heuristic evaluation function that is used, alpha-beta pruning will return that as-well. However, if two moves are of equal value, that is neither one is better than the other, it is normal for alpha-beta pruning to return a different move, as the change in move ordering can lead to one move being seen first.

In order to deal with this and evaluate if I had an error or not, I had to check if it was returning a different move of equal value and still evaluating all the moves to be of the same value as minimax, or if they were evaluated differently. After performing the check, I concluded that they were evaluated the same as minimax, which is what a normal behaviour should be.
Thus, the implementation of this heuristic was correct and indeed improved the overall performance.