Miłosz Staniszewski 220629

Artificial Intelligence and Knowledge Engineering Laboratory

1. Game Playing Algorithm
2. **Introduction**

The goal of this task was to choose a board game without any random elements, implement its rules, create a simple playing interface and implement two algorithms that will try to find the best possible move. Used algorithms are min-max and min-max with alpha-beta pruning. The algorithms search through all possible moves in a game for few steps ahead and based on the value of the state of the game, they try to find the best possible moves regarding both players and opponents reactions to them.

1. **Chosen game – Gomoku**

The game chosen for this assignment was Gomoku – game similar to tic-tac-toe. Players place their stones in the cells of a 15x15 diagram. The first player to put 5 stones of their color in a row either horizontally, vertically or diagonally, wins the game. During the game players are trying to block their opponent’s combos while building theirs at the same moment.

1. **Algorithms – min-max and alpha-beta**

Min-max algorithm scans all possible moves for the game for the given amount of turns (the “depth of search”). By using recursion, algorithm finds out all possible states of the game after specified amount of turns and evaluates them. After receiving an evaluation for a given state, algorithm goes further, trying to figure out, which combination of moves will provide the best result and, potentially, a win. After checking all outcomes, the best move is chosen and performed. Algorithm takes into consideration possible reactions of the opponent and tries to minimize loss that will be a result of the opponent’s moves while maximizing the gain that will be a result of the player’s moves.

Pseudocode:

function minimax(move, depth, maximizingPlayer)

if depth = 0 or move is a final move

return the heuristic value of the game state

if maximizingPlayer

bestValue := -infinity

for each child of move

performMove(child)

v := minimax(child, depth - 1, FALSE)

if v > bestValue

bestValue := max(bestValue, v)

save move parameters

return bestValue

else (\* minimizing player \*)

bestValue := +infinity

for each child of move

v := minimax(child, depth - 1, TRUE)

if v > bestValue

bestValue := max(bestValue, v)

save move parameters

return bestValue

Alpha-beta pruning extents the min-max algorithm by adding two extra variables, alpha and beta. Those variables store values of the worst values for maximizing player and the best values for the minimizing player. When values for the moves of the minimizing player are worse that the values for the maximizing player, that means that all further moves won’t be taken into notice for the final consideration and can be omitted. Algorithm breaks the recursion at given level in order to avoid unnecessary complexity and to save computational time.

Pseudocode:

function alphabeta(move, depth, alpha, beta, maximizingPlayer) is

if depth = 0 or move is a final move

return the heuristic value of the game state

if maximizingPlayer then

v := -infinity

for each child of move do

performMove(child)

v := max(v, alphabeta(child, depth – 1, alpha, beta, FALSE))

if v > alpha

alpha := max(alpha, v)

save move parameters

if beta <= alpha then

break (\* beta cut-off \*)

return v

else

v := +inf

for each child of node do

v := min(v, alphabeta(child, depth – 1, alpha, beta, TRUE))

if v < beta

beta := min(beta, v)

save move parameters

if beta <= alpha then

break (\* alpha cut-off \*)

return v

1. **Implementation**

Evaluation function used for calculating the value of the move searches for the combos on the board – how many stones belonging to the same player appear in a sequence of five places. Each stone belonging to the opponent zeroes the score obtained from considered 5 slots, because it makes it impossible to finish the game within this area, so it should not be considered. The total score is calculated by the formula: , where n is the number of stones belonging to the player in a 5-place area. Extreme cases, such as win or loss are graded by the positive and negative values of infinity, respectively.

1. **Evaluation**

Min-max:

1st move – 554887 ms – 11089344 nodes

2nd move – 554655 ms – 10793640 nodes

3rd move – 555181 ms – 10503240 nodes

4th move – 550445 ms – 10218096 nodes

Alpha-beta pruning:

1st move – 4902 ms – 103741 nodes

2nd move – 5978 ms – 135242 nodes

3rd move – 7654 ms – 173391 nodes

4th move – 8849 ms – 188195 nodes

As seen above, alpha-beta pruning is way more efficient than pure min-max algorithm. Min-max is checking every possible move, which for a board of 15x15 places is a lot of work that takes significantly more time than alpha-beta. Another significant difference between those two algorithms is that in time, min-max considers less moves, as there are less free spaces on the board, where alpha-beta is considering more and more possible moves – this is the result of the game status being more complicated and more moves are starting to be more significant.