Adam Brown

Allison Holt

**Programming Assignment – Deliverable 1**

1. **Heuristic Functions**
   1. H1 – Heurisitic One will be the total number of pebbles the player has in that state, calculated by **The Player’s Pebbles**

//Utiliity Function for H1

function UTILITY-H1(state) returns h:heuristic value

return state.playerPebbles

* 1. H2 – Heurisitic Two will examine how much farther the player is over his opponent calculated by **The Player’s Pebbles – The Opponent’s Pebbles**

//Utiliity Function for H2

function UTILITY-H2(state) returns h:heuristic value

return (state.playerPebbles – state.opponentPebbles)

1. **Modified Alpha-Beta Minimax Game Playing algorithm**
   1. **Look-ahead a fixed number of moves**

function ALPHA-BETA-SEARCH(state, depth) returns a:action

v = -∞; a = NIL; α = -∞; β = ∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MIN(state’, α, β, depth)

if v’ > v then v = v’; a=action end if

if v >= β then return a

else if v > α then α = v

return a

function MAX-VALUE (state, α, β, depth) returns v:utility-value

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

v = -∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MIN(state’, α, β, depth + 1)

if v’ > v then v = v’ end if

if v >= β then return v

else if v > α then α = v

return v

function MIN-VALUE (state, α, β, depth) returns v:utility-value

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

v = ∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MAX(state’, α, β, depth + 1)

if v’ < v then v = v’ end if

if v <= α then return v

else if v > β then β = v

return v

* 1. **Abandon a path if it reaches a game board or state that has already appeared in that path (i.e., no loops – note that this is different from redundant paths leading to transpositions)**

function ALPHA-BETA-SEARCH(state, depth) returns a:action

v = -∞; a = NIL; α = -∞; β = ∞

pathStates = NIL

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MIN(state’, α, β, depth,pathStates)

if v’ > v then v = v’; a=action end if

if v >= β then return a

else if v > α then α = v

return a

function MAX-VALUE (state, α, β, depth, pathStates) returns v:utility-value

if state in pathStates then return -∞

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

INSERT(pathStates, state) //inserts current state into pathStates

v = -∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MIN(state’, α, β, depth + 1, pathStates)

if v’ > v then v = v’ end if

if v >= β then return v

else if v > α then α = v

return v

function MIN-VALUE (state, α, β, depth, pathStates) returns v:utility-value

if state in pathStates then return ∞

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

INSERT(pathStates, state) //inserts current state into pathStates

v = ∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MAX(state’, α, β, depth + 1)

if v’ < v then v = v’ end if

if v <= α then return v

else if v > β then β = v

return v

* 1. **To store any new game board with its minimax value in a hash table (Transposition Table) and each time to first look up this table and execute the MAX-VALUE or MIN-VALUE algorithm for that state only if the game board is not in the Transposition Table (if it is, directly return the corresponding minimax value)**

function ALPHA-BETA-SEARCH(state, depth) returns a:action

v = -∞; a = NIL; α = -∞; β = ∞

pathStates = NIL

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

v’ = MIN(state’, α, β, depth,pathStates)

if v’ > v then v = v’; a=action end if

if v >= β then return a

else if v > α then α = v

return a

function MAX-VALUE (state, α, β, depth, pathStates) returns v:utility-value

if state in pathStates then return -∞

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

INSERT(pathStates, state) //inserts current state into pathStates

v = -∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

if state’ in TRANS-TABLE then

v’ = TRANS-TABLE(state’)

else

v’ = MIN(state’, α, β, depth + 1, pathStates)

INSERT(TRANS-TABLE, state’, v’)

if v’ > v then v = v’ end if

if v >= β then return v

else if v > α then α = v

return v

function MIN-VALUE (state, α, β, depth, pathStates) returns v:utility-value

if state in pathStates then return ∞

if CUTOFF-TEST(state, depth) == true then return UTLITY(state) //Either for H1 or H2

INSERT(pathStates, state) //inserts current state into pathStates

v = ∞

for each action in ACTIONS(state) do

state’ = RESULT(state, action)

if state’ in TRANS-TABLE then

v’ = TRANS-TABLE(state’)

else

v’ = MAX(state’, α, β, depth + 1)

INSERT(TRAN-TABLE, state’, v’)

if v’ < v then v = v’ end if

if v <= α then return v

else if v > β then β = v

return v

1. **Come up with a way to order the moves to improve the alpha-beta pruning performance; explain it in English and provide a corresponding algorithm that takes a game state/board and the player to move next as parameters and returns a set of ordered moves**
   1. The moves should be ordered based on the number of pebbles in each of the players squares minus the distance of that particular square to the that players last square. This analysis takes in all given information to come up with an ordered list of the best moves, starting with the best, and ending with the worst.

function ORDER-MOVES(state, player) return orderedMoves as set

for each square in player’s region

value 🡨 square’s number of pebbles – moves to last

square region

INSERT(orderedMoves, value)

SORT(orderedMoves)

return orderedMoves