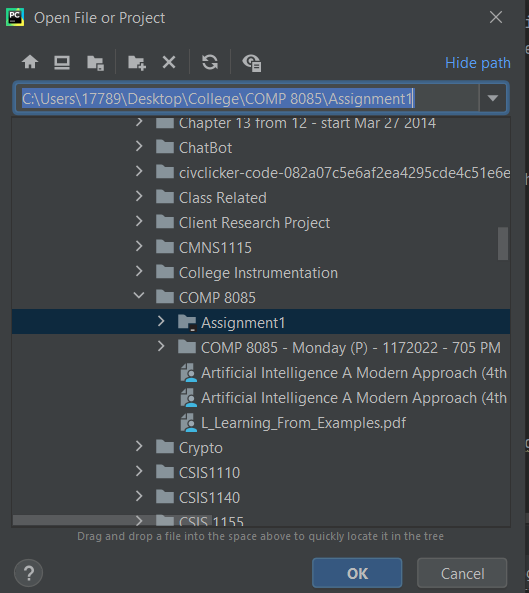
# Assignment 1:

How to Set up:

Simply Extract the Folder’s contents into Folder

You might need to do a pip install to run the n

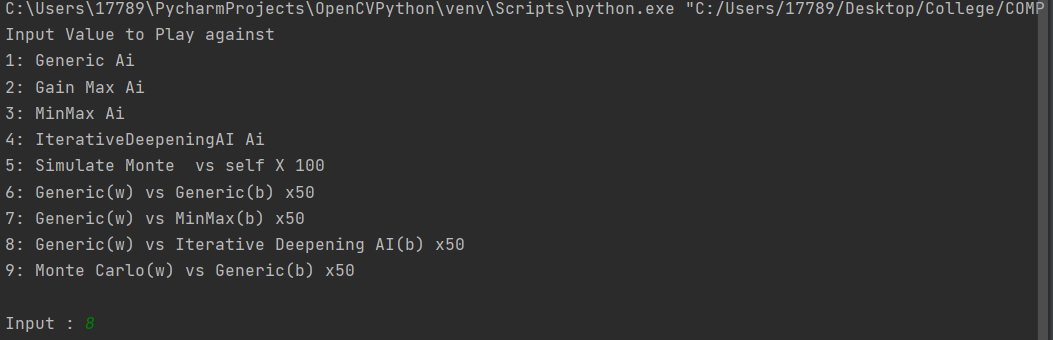
Open folder in your preferred IDE. ( PyCharm is the one used in this project)



Run engine.py



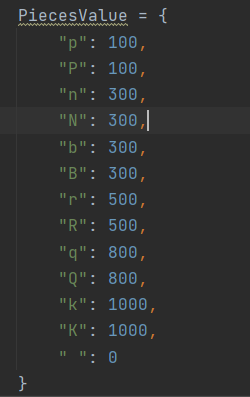
Follow the instructions:



## Generic Max Gain AI

### Pseudo-Code:

**#This function evaluates the current board/game state and outputs an value that is a representation of current #position of the game. This evaluation uses only Number of Pieces and Pieces value. The values of the Chess pieces are #abritory but as a general rule, Kings and QUeens are highest while pawns are the lowest**



**function evaluation(boardstate) return a evaluation\_value**

player1evaluation = weightofPlayer1Piece1 \* AmountofPlayer1Piece1 + weightofPlayer1Piece2 \* NumberofPlayer1Piece2..

player2evaluation = weightofPlayer2Piece1 \* AmountofPlayer2Piece1 + weightofPlayer2Piece2 \* NumberofPlayer2Piece2..

return player1Piecesvalue - playerPieces2evaluation

**#Now that we can evaluate the board. we simply look for the High Board State that give us the most Gain**

**FUNCTION MAX\_GAIN(game,State) returns best Evaluated Move**

V 🡨 -inf

for Each ACtion in GAME.ACTIONS do

ACTION\_Evaluation 🡨 Evaluation(A)

if (ACTION\_EVALUATION >= V) then

BEST\_ACTION 🡨ACTION

V 🡸 ACTION\_EVALUATION

Return BEST\_ACTION, V

#On the other Hand, if we are the opposing player, then we look for the SMallest. Because the evaluation is based on #White – Black. So if you’re the black player then you are looking for the Smallest White evaluation

FUNCTION MIN\_GAIN(game,State) returns best Evaluated Move

V 🡨 -inf

for Each ACtion in GAME.ACTIONS do

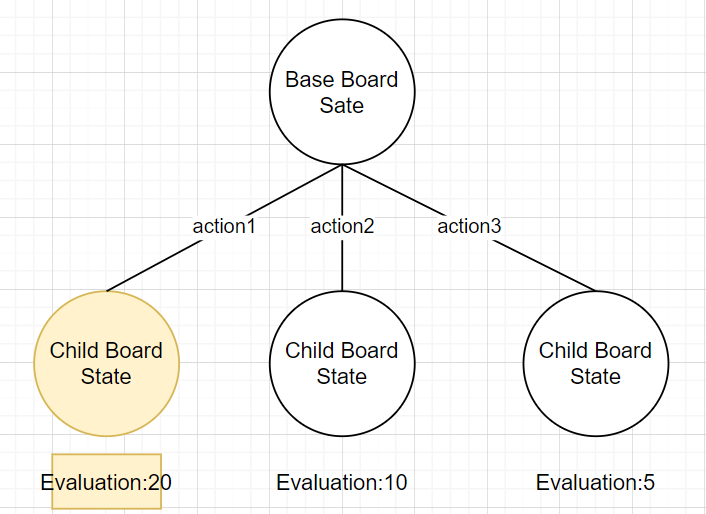
ACTION\_Evaluation 🡨 Evaluation(A)

if (ACTION\_EVALUATION<= V) then

BEST\_ACTION 🡨ACTION

V 🡸 ACTION\_EVALUATION

Return BEST\_ACTION, V



### Problems:

* AI simply looks for the highest gain for next Action and doesn’t account for captured pieces, overall game state and moving piece values. So, it is prone to fall into traps where it will always capture low value pieces with any of pieces, including its high valued pieces
* AI doesn’t have any prediction of possible responses of the opposing player, so it can suicide pieces without knowing
* It has no ability to discover new moves so the moves are always fixed if left on its own. The Game’s result will always be the Same

### Runs Against Generic AI(RANDOM AI) - 20 times Win% 0:

|  |  |  |  |
| --- | --- | --- | --- |
| DRAWBY50M | WHITE WIN | BLACK WIN | STALE MATE |
| 20 | 0 | 0 | 0 |

### Generic Random AI(Itself) - 20 times Win % 6/20:

|  |  |  |  |
| --- | --- | --- | --- |
| DRAWBY50M | WHITE WIN | BLACK WIN | STALE MATE |
| 10 | 2 | 4 | 4 |

When we run it against Generic AI, it will always have more captured pieces than its opponent. However, it lacks the ability to use check to checkmate the opponent so it will always end in a DRAW

## Minimax AI(with Alpha-Beta Pruning)

### Pseudo-Code:

function evaluation(boardstate)..Same prevous AI

FUNCTION MINIMAX-SEARCH(game,state,maxDepth,alpha = -inf,beta = inf) returns Best\_action

if Player == FIRST

value,move = MAXVALUE(GAME,STATE,0 , alpha,beta, MAXDEPTH)

IF Player == Second

value,move = MINVALUE(GAME,STATE,0, alpha,beta, MAXDEPTH)

Return move

**#MAXValUE FUNCTION Doesn’t Just look Highest Gain but it also Considers OPPONENT’s RESPONSES**

**#WE START AT DEPTH OF 0. To Find OUR CURRENT BEST POSSIBLE MOVE, We start evaluating at the Last Depth and work our Way #Backwards**

FUNCTION MAXVALUE(GAME,STATE,CURRENT\_DEPTH = 0,ALPHA,BETA, MAXDEPTH)

IF GAME.IS-TERMINAL(state) OR CURRENT\_DEPTH = MAXDEPTH Then

RETURN evaluation(STATE), NULL #ONCE we Reach the Maximum Depth of the search

V 🡨 -inf

**#For POSSIBLE ACTION WE CAN THEN CHECK FOR THE RESULT OF THE ACTION TO FIND WHAT IS BEST OPPONENT RESPONSE TO THE MOVE.**

FOR EACH ACTION in GAME.ACTIONS(STATE)

MovingPieceVALUE 🡨 ACTION.GETPIECE(START)

TARGETPIECEVALUE 🡨 ACTION.GETPIECE(END)

CLONERESULT 🡨 GAME(state)

CLONERESULT.TO-MOVE(ACTION)

v2, a2 🡨MIN\_VALUE CLONERESULT,current\_DEPTH+1,ALPHA,BETA,MAXDEPTH) **#WE PASS THE RESULT TO FIND OPPONENT RESPONSE**

IF (V2 > V) THEN **#IF WE FOUND A BEST MOVE THAT DOESN”T RESULT IN A STRONGER OPPONENT RESPONSE THEN we APPLY the MOVE**

V,BESTACTION 🡨 V2,ACTION

ALPHA🡨MAX(ALPHA,V)

IF (v >= BETA) THEN RETURN v, BESTACTION

IF (TARGETPIECEVALUE >= MovingPieceVALUE) THEN # **always take smaller value piece to take a bigger value piece.**

BESTACTION🡨ACTION

RETURN V,BESTACTION

RETURN V, BESTACTION

**#TO AVOID CHECKING Unnecessary Actions we Implement ALPHA BETA Cutoff .**

**# ALPHA IS set to the largest of the scores of its successors explored up to now. At MIN node, The alpha value is of its predecessor. IF Evevaluation Of a MIN Node is Lower than its PREDECESSOR Then we Do not Look at further Nodes**

**# BETA IS set to the largest of the scores of its successors explored up to now. At MAX node, The BETA value is of its predecessor. IF Evaluation Of a MAX Node is Higher than its PREDECESSOR Then we Do not Look at further Nodes**

**#Another Cut-OFF implemented is to use a smaller value piece to take a bigger value piece.**

**#We use the same logic as Max except at we are evaluation our OPPONENTS’ responses so it will be a MIN Instead. We can of course search another depth deeper and look at our follow up response to our opponent’s responses…and so on**

FUNCTION MINVALUE(GAME,STATE,CURRENT\_DEPTH = 0,ALPHA,BETA, MAXDEPTH)

IF GAME.IS-TERMINAL(state) OR CURRENT\_DEPTH = MAXDEPTH Then

RETURN evaluation(STATE), NULL

V 🡨 inf

FOR EACH ACTION in GAME.ACTIONS(STATE)

MovingPieceVALUE 🡨 ACTION.GETPIECE(START)

TARGETPIECEVALUE 🡨 ACTION.GETPIECE(END)

CLONERESULT 🡨 GAME(state)

CLONERESULT.TO-MOVE(ACTION)

v2, a2 🡨MIN\_VALUE CLONERESULT,current\_DEPTH+1,ALPHA,BETA,MAXDEPTH)

IF (V2 < V) THEN

V,BESTACTION 🡨 V2,ACTION

BETA🡨MIN(BETA,V)

IF (v <= ALPHA) THEN RETURN v, BESTACTION

IF (TARGETPIECEVALUE >= MovingPieceVALUE) THEN

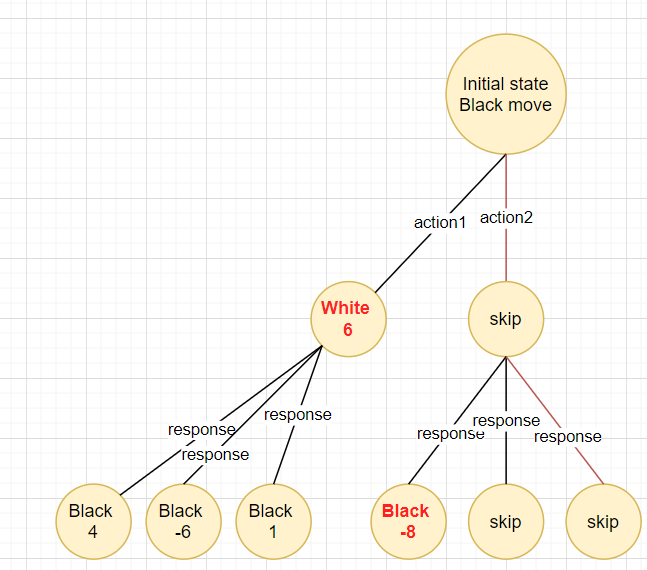
BESTACTION🡨ACTION

RETURN V,BESTACTION

RETURN V, BESTACTION

### Explanation:

### Here we see Red is the path that Black should take



Alpha-Beta CutOff( Black -8 branch are skipped because it is further results will only be worse than our current selection so far

### Problems:

* Search Intensive, the deeper the allowed depth, the more positions it must search for an optimal step. It takes a long time to make each move. Reducing the number of depths also runs into the possibly of not finding any optimal paths
* It lacks the ability to check mate the oppose King properly. When played against a random opponent. It doesn’t know how to corner and check mate the king thus wasting moves.

### Runs Against GENERIC RANDOM AI(RANDOM AI) - 20 times Win % 1/20:

|  |  |  |  |
| --- | --- | --- | --- |
| DRAWBY50M | WHITE WIN(AI) | BLACK WIN | STALE MATE |
| 16 | 1 | 1 | 2 |

The Biggest Issue with this AI, is that it is still lacking a Check mate implementation. Thus, it will always most likely to DRAWBY50TURNs

## Iterative Deepening AI

### Pseudo-Code:

CUTOFF 🡨 inf

# Iterative Deepening AI is exactly like Min Max AI Except it has an abritory cut off point, such as time

**FUNCTION Itrminimax-SEARCH(game,state,alpha = -inf,beta = inf) returns Best\_action**

for MAXDepth = 1 to inf do

value move = MinMAX-SEARCH(game,state,MAXDEPTH.-inf,inf)

if Value IS not EQUAL to CUTOFF then RETURN RESULT

FUNCTION MINMAX-SEARCH(game,state,maxDepth,alpha = -inf,beta = inf) returns Best\_action

if Player == FIRST

value,move = MAXVALUE(GAME,STATE,0 , alpha,beta, MAXDEPTH)

IF Player == Second

value,move = MINVALUE(GAME,STATE,0, alpha,beta, MAXDEPTH)

Return Value, move

StartTime = TIME()

TimeLIMIT = 5

**FUNCTION MAX\_VALUE(GAME,STATE,CURRENT\_DEPTH = 0,ALPHA,BETA, MAXDEPTH)**

IF GAME.IS-TERMINAL(state) OR CURRENT\_DEPTH = MAXDEPTH Then

RETURN evaluation(STATE), NULL

V 🡨 -inf

FOR EACH ACTION in GAME.ACTIONS(STATE)

MovingPieceVALUE 🡨 ACTION.GETPIECE(START)

TARGETPIECEVALUE 🡨 ACTION.GETPIECE(END)

CLONERESULT 🡨 GAME(state)

CLONERESULT.TO-MOVE(ACTION)

v2, a2 🡨MIN\_VALUE CLONERESULT,current\_DEPTH+1,ALPHA,BETA,MAXDEPTH)

IF (V2 > V) THEN

V,BESTACTION 🡨 V2,ACTION

ALPHA🡨MAX(ALPHA,V)

IF (v >= BETA) THEN RETURN v, BESTACTION

IF (TARGETPIECEVALUE >= MovingPieceVALUE) THEN

BESTACTION🡨ACTION

RETURN V,BESTACTION

IF(CUrrent.Time() – StartTime()) > TIMELIMT THEN RETURN v<--CUTOFF, BESTACTION **#CUTOFF BY TIME**

RETURN V, BESTACTION

**FUNCTION MIN\_VALUE(GAME,STATE,CURRENT\_DEPTH = 0,ALPHA,BETA, MAXDEPTH)**

IF GAME.IS-TERMINAL(state) OR CURRENT\_DEPTH = MAXDEPTH Then

RETURN evaluation(STATE), NULL

V 🡨 inf

FOR EACH ACTION in GAME.ACTIONS(STATE)

MovingPieceVALUE 🡨 ACTION.GETPIECE(START)

TARGETPIECEVALUE 🡨 ACTION.GETPIECE(END)

CLONERESULT 🡨 GAME(state)

CLONERESULT.TO-MOVE(ACTION)

v2, a2 🡨MIN\_VALUE CLONERESULT,current\_DEPTH+1,ALPHA,BETA,MAXDEPTH)

IF (V2 < V) THEN

V,BESTACTION 🡨 V2,ACTION

BETA🡨MIN(BETA,V)

IF (v <= ALPHA) THEN RETURN v, BESTACTION

IF (TARGETPIECEVALUE >= MovingPieceVALUE) THEN

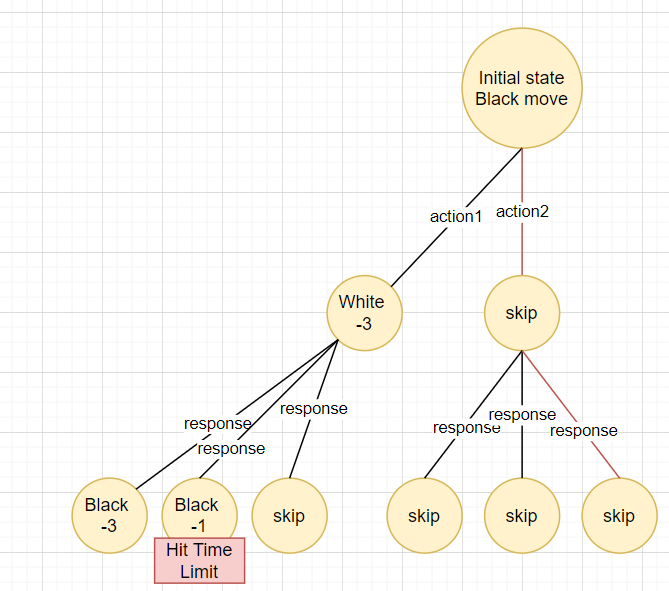
BESTACTION🡨ACTION

RETURN V,BESTACTION

IF(CUrrent.Time() – StartTime()) > TIMELIMT THEN RETURN v<--CUTOFF, BESTACTION

RETURN V, BESTACTION

### Explanation:



We only look at our current node so far if we hit a processing time limit

### Problems:

• Although we have implemented a cutoff, it is still search Intensive, the deeper the allowed depth, the more positions it must search for an optimal step. The higher the cutoff, the more optimal each step are required.

### Runs Against Generic AI:

|  |  |  |  |
| --- | --- | --- | --- |
| DRAWBY50M | WHITE WIN | BLACK WIN | STALE MATE |
| 20 | 0 | 0 | 0 |

Time cutoff: 5 s

Like MinmaxAI, biggest Issue with this AI, is that it is still lacking a Check mate implementation. It doesn’t know the value of a Check and Checkmate since it is only evaluating by piece value and # of pieces. Additional modification is required to improve this AI.

## Monte Carlo

### Pseudo-Code:

Simulate:

#we create 2 MCTSAI to try and find this an optimal Win path

**FUNCTION Self\_Simluate(GAME,STATE,TREE)**

**MCTSAI\_1(Game,STATE)**

**MCTSAI\_2(Game,STATE)**

**GameClone(GAME,STATE)**

**MOVES = []**

**Current\_node = TREE.ROOTNOTE**

**#In our .GetMOVE Function, We are actually expanding our current NODE based on the current GAME state**

**While GAMECLONE.STATE IS NOT TERMINATE:**

**move1 = MCTSAI\_1.GETMOVE(GAMECLONE.GAME.STATE,CurrentNODE)**

**#After we expanded the node, used our selection policy**

**GAMECLONE.GAME.APPLY(MOVE1)**

**MOVES.APPEND [MOVE1]**

**MOVE2= MCTSAI\_2.GETMOVE(GAMECLONE.GAME)**

**GAMECLONE.GAME.APPLY(MOVE2)**

**MOVES.APPEND[MOVE2]**

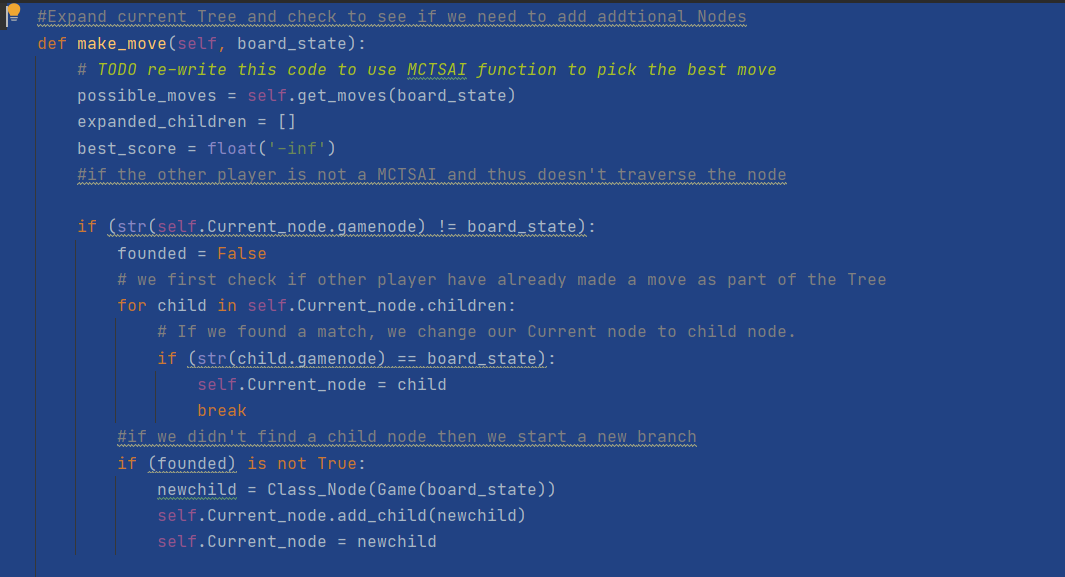
**#Game Reaches a terminal point, a leaf node**

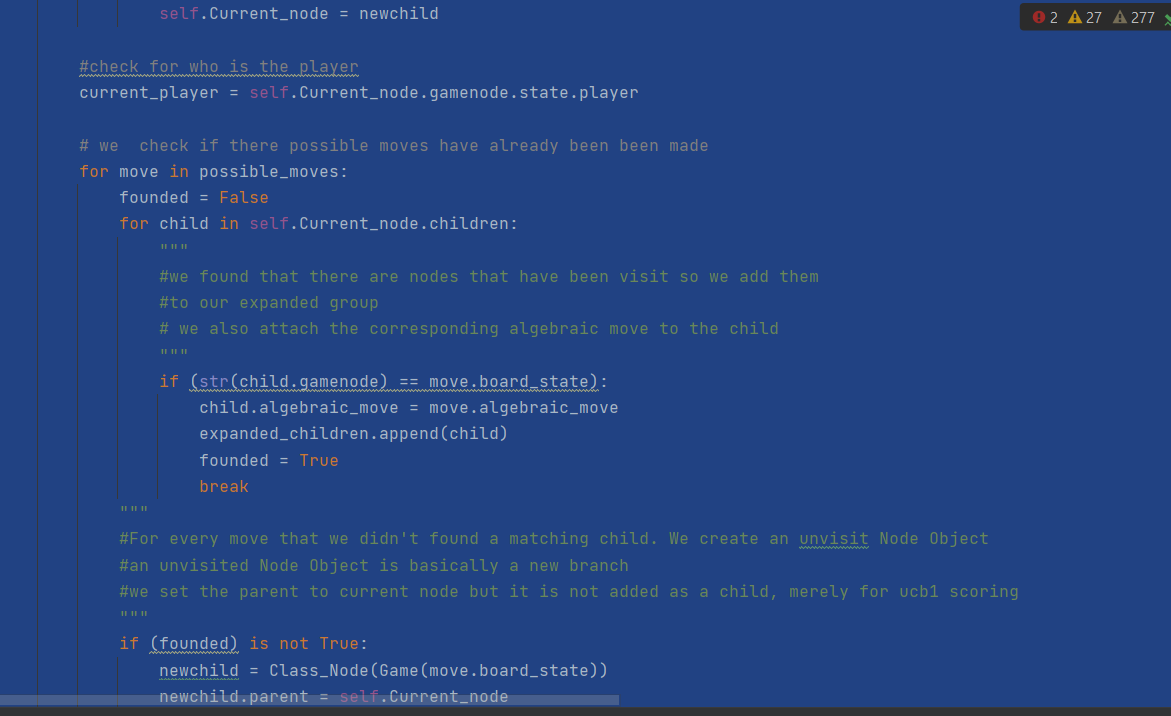
**Open file(“training.model” as model**

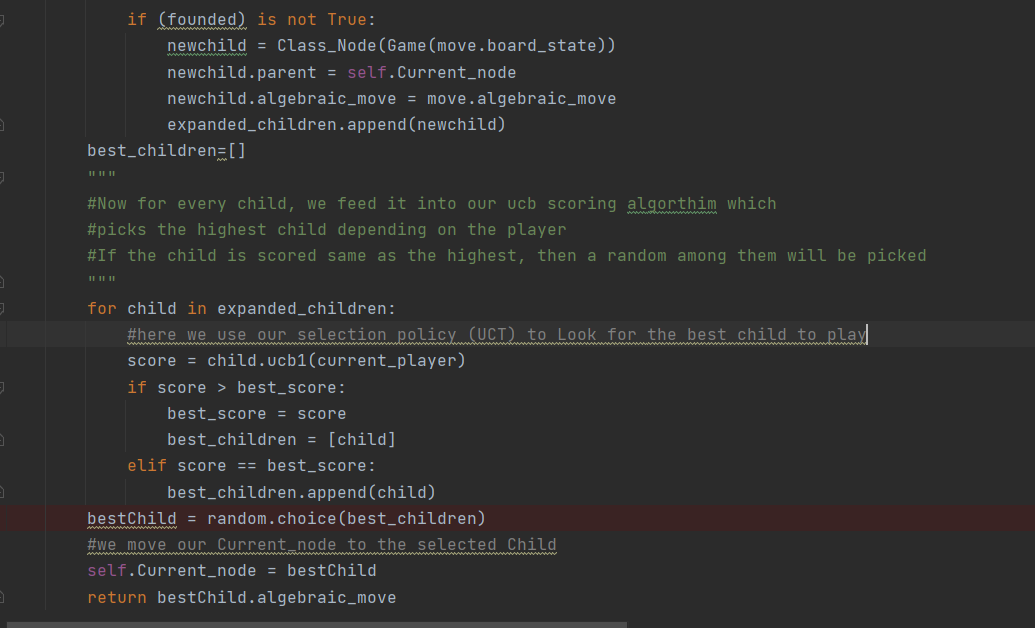
**model.WriteToFile(Moves)**

**model.WriteToFile(GameClone.Result)**

**Expansion Function(**Pseudo-Code was cut due to time restraints)



1



**#straightForward function, we simply backprogogate from our current node and increase black and white values**

**#The black and white win values determines how likely each player will take the path again**

**FunCTION BACKPROGOGATE(CURRENTNODE,WhiteWINVALUE, BLACKWINVALUE)**

**CURRENTNODE.visit += 1**

**Node.Black\_win\_value += Black\_win\_value**

**Node.White\_win\_value += White\_win\_value**

**while (Node.parent != None):**

**Node.parent.visits += 1**

**Node= Node.parent**

**Node.Black\_win\_value += Black\_win\_value**

**Node.White\_win\_value += White\_win\_value**

**FunCTION Populate\_tree(Current\_NODE) #this reads our current Training File and populates the existing Tree**

**#model file example**

**‘’’’**

**rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1,rnbqkbnr/pppppppp/8/8/8/N7/PPPPPPPP/R1BQKBNR b KQkq - 1 1**

**.**

**.**

**8/8/8/8/1k3K2/8/8/8 b - - 98 233,8/8/8/k7/5K2/8/8/8 w - - 99 234**

**8/8/8/k7/5K2/8/8/8 w - - 99 234,8/8/8/k7/4K3/8/8/8 b - - 100 234**

**8/8/8/k7/4K3/8/8/8 b - - 100 234,8/8/8/1k6/4K3/8/8/8 w - - 101 235**

**50TurnDraw this is the result of the game**

**‘’’’**

**with open("TRAINing.model") as data:**

**For Each LIne in Data.Read:**

**if Line[0] = game.RESULT.WhiteWIN**

**BackProgogate(self.Current\_node,BLACKLOSSSCORE, WhiteWINSCORE)**

**if Line[0] = game.RESULT.BLACKWIN**

**BackProgogate(self.Current\_node,BLACKWINSCORE, WhiteLOSSSCORE)**

**if Line[0] = game.RESULT.STALEMATE**

**BackProgogate(self.Current\_node,BLACKSTALEMATESCORE, WhitESTALEMATESCORE)**

**if Line[0] = game.RESULT.50TURNDRAW**

**BackProgogate(self.Current\_node,BLACKDRAWSCORE, WhitEDRAWSCORE)**

**if LINE[0] == CURRENT\_NODE.GAME.STATE**

**for child in Current\_node.Child**

**Found = false**

**if CURRENT\_NODE.GAME.STATE CHILD\_NODE = NODE(LINE[1]**

**Then current\_Child = NODE(LINE[1]**

**Found = true**

**if found is false:**

**Current\_NODE.add CHILD (NODE(LINE[1])**

**#start a new branch**

**#if there is a winning path for White, its win value is higher than an unexplored node and it will keep going down that path (edited)**

**#when its Black's turn, black will try to explore an alt Leaf Node so it will explore unvisited nodes, it can end in black win, stale mate or 50 turn #draw or another white win (edited)**

**#White will only start exploring unexplored nodes, when Black found a leaf node that end in Black win, at which it will try to find unexplored #node instead**

**#Both players will try to avoid 50 turn draw unless all nodes are explored and has no winning paths, at which it's just random again**

**FunCTION ucb1(self,current\_player):**

**if current\_player == "b":**

**exploit = (self.Black\_win\_value-self.White\_win\_value)**

**explore = 1 \* (sqrt(log(self.parent.visits +1+ (10 \*\* -6)) / (self.visits + (10 \*\* -10))))**

**return exploit + explore**

**if current\_player == "w":**

**exploit = self.White\_win\_value-self.Black\_win\_value**

**explore = 1 \* (sqrt(log(self.parent.visits +1+ (10 \*\* -6)) / (self.visits + (10 \*\* -10))))**

**return exploit + explore**

### Problems:

### - incredibly training Intensive. Obviously, a simpler game is easy to train. However, in chess, it isn’t enough to just train to win, it will also actively avoid taking paths that can give the opponent winning moves. I found that during training, wins tend to be grouped. When white wins, white will generally win in a series with some draws in between. Then Draws will happen a lot more. Finally black will start winning and slowly increase until white force draw to increase again.

### Simulation Playing against itself - 270 times:

|  |  |  |  |
| --- | --- | --- | --- |
| DRAWBY50M | WHITE WIN | BLACK WIN | STALE MATE |
| 177 | 26 | 32 | 40 |

Results after playing against a generic AI(white) as Black(MCTSAI)

{'White': {'Wins': 4, 'Loses': 10, 'Draws': 36}, 'Black': {'Wins': 10, 'Loses': 4, 'Draws': 36}}

It is significant, although small increase over the generic AI. Most of the time it will ends in a Draw. Increasing in training should Improve this.

Unfortunately there was no time to simulate MCTSAI against MinMaxAI