Five-In-A-Row (Gomoku)

Contents

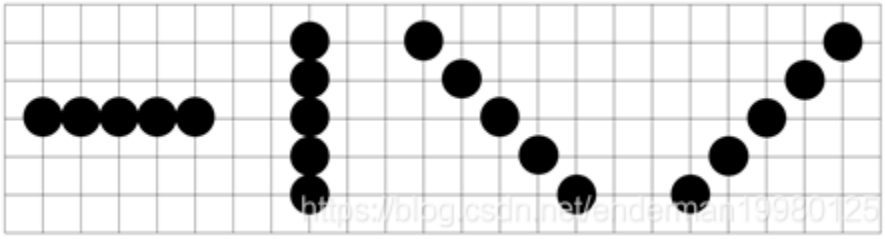
1. Introduction

In this paper, we work on implementing Gomoku based on a game tree, with Minimax algorithms and alpha-beta pruning.

First, we will give a brief view of how to play Gomoku. Then, we will also look at different approaches to implementing Gomoku and compare the advantages and disadvantages between them. Next, we will focus on the classical methods for implementing board games, the minimax algorithm and alpha-beta pruning and the related code implementation on this game. Finally, we will conclude the whole project and reflect on what could be done to improve the gameplay.

2. Gomoku Background

Gomoku, (five-in-a-row), is a board game of abstract strategy. The basic rule of the game is that two players compete to first obtain five consecutive pieces horizontally, vertically or diagonally. (Lasker, 2012)



Research in the field of Gomoku AI has been proliferating in recent years. In 1994, L. Victor Allis, (1994) proposed proof-number search (pn-search) and dependency-based search (db-search) algorithms that are effective in helping a player to have a winning strategy in a no opening restrictions Gomoku board. However, for opening rules in Gomoku, such as Swap2 (Gomoku World, 2018), these algorithms will no longer be applicable. Thus, the topic of Gomoku AI remains a challenge for the field of computer science, such as how to solve problems to improve Gomoku algorithms.

Previously, most of the advanced Gomoku algorithms are based on the alpha-beta pruning. However, some new techniques are also used in Gomoku AI recently, such as machine learning and genetic algorithms.

Deep Learning

With the huge success of AlphaGo, deep learning has become a common approach to chess game AI (Chouard, 2016. the strategy network employed by AlphaGo was trained to use supervised learning to predict the moves of opponent players, which largely reduces the breadth of the

search and reduces the time required through powerful arithmetic. Inspired by this, K. Shao et al. (2016) attempted to improve Gomoku agent intelligence by using Tensorflow and Keras as an experimental platform to use deep learning for move prediction.

By using the move data of expert Gomoku players in the RenjuNet dataset as input, the trained neural network model was able to identify some of Gomoku's game features and select the most optimal next move. The results show that the trained model achieves a move prediction accuracy of approximately 42% on the RenjuNet dataset, which is at the level of Gomoku experts.

Genetic Algorithms

Genetic Algorithm (GA) is a heuristic search method proposed by Holland (1992). In recent years, GA has often been applied in the field of games to optimise algorithms.Junru Wang & Lan Huangpresent (2014) proposed a genetic algorithm for solving Gomoku. They investigate a general framework for applying genetic algorithms to strategy games and design fitness functions in terms of various game-related aspects, including the value of layout patterns, the value of distinguishing between offense and defence, adjustments for early and near victory, and the overall trend of the solution.

Experimental results show that the proposed genetic solver can perform a deeper search than traditional game-tree-based solvers, resulting in better and more interesting solutions. Compared to tree-based algorithms, GA-based solvers do not need to enumerate all possibilities. As a result, they can search deeper steps and complete them in less time.

Minimax + Alpha-beta Pruning

Minimax search and its derivative algorithm Alpha-beta search (Schaeffer, 1989) are the most classical and commonly used game tree based methods for solving the field of board games. In each round of the game, these methods select the moves that are most favourable to the solver and the moves that are least favourable to its competitors. The alpha-beta pruning method further improves the efficiency of the search and speeds up AI decision time. In this paper, due to time constraints, we have chosen this framework to complete this Gomoku game.

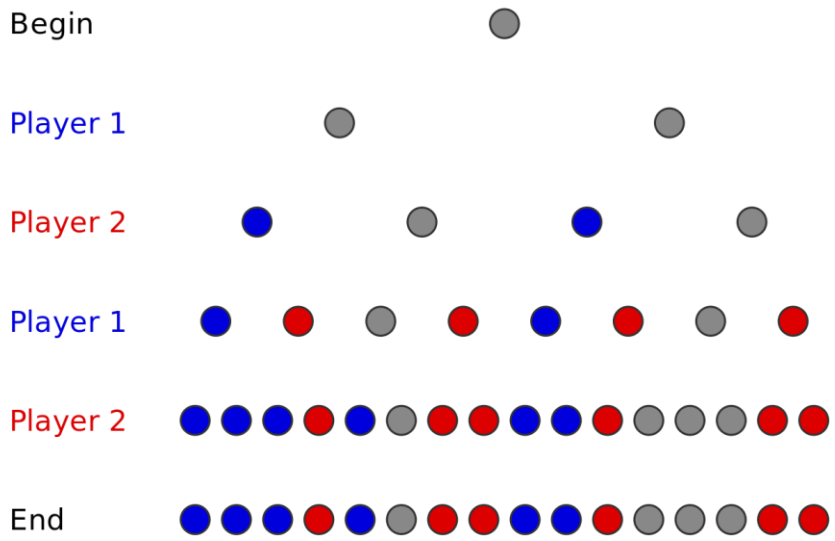
3. Game Tree Search

Experts in game theory introduce a definition of a game as “a zero-sum game between two players with complete information, deterministic, and taking turns to act.” (Russell Stuart & Norvig, 2009)

Gaming algorithms have never stopped improving since the advent of computers. The earliest AI to beat a top human player was Deep Blue.

In 1997 Deep Blue defeated the king of chess, Kasparov by a single point on aggregate, the first time a computer had beaten a top human player in chess, and the world witnessed the power of artificial intelligence. The technique used by Deep Blue at that time was to violently traverse the game tree by powerful arithmetic. (Higgins, 2017)

For a standard 15*15 Gomoku board, on the next move the machine has 15*15 moves, and underneath all these moves there are 15*15 move possibilities. This results in a huge tree-like network which is called a game tree.



For our game algorithm, all we need to achieve is to find the optimal path from a huge game tree. Generally speaking, the most commonly used algorithms are the Minimax algorithm and the Alpha-Beta pruning algorithm. (Campbell & Marsland, 1983)

Here is a general game tree we write in python. It traverses all the nodes of the tree and finds an optimal position to drop a piece.

```
def GameTree1():
    global alpha
    alpha=-100000
    if(grid[8][8]==0): #drop a piece in the middle of the board if there has not a piece
        return dropPiece(8,8)
    keyi=-1; keyj=-1
    #Iterate over x, y coordinates
    for x in order:
        for y in order:
            if(not checkDrop(x,y)):
                continue
            grid[x][y]=ai
            #use evaluation functon to get a score accourding to the coordinates
            tempp=evaluation(x,y)
            if(tempp==0):
                grid[x][y]=0; continue
            if(tempp==10000):
                return dropPiece(x,y)
            #enter into gametree depth 2
            tempp=GameTree2()
            grid[x][y]=0
            if(tempp>alpha): #get max value
                alpha=tempp; keyi=x; keyj=y
    dropPiece(keyi,keyj)
```

Helper Function

Here we define a number of helper functions to ensure that my game runs successfully. The relevant function descriptions have been written within the code comments so we will not be described here.

```
#check the mouse click point are in the available zone
def checkInBoard(x,y):
    if(x>=0 and x<=15 and y>=0 and y<=15): return True
    else: return False
#Check the mouse click point in the board and no yet to drop a piece
def checkDrop(x,y):
    if(checkInBoard(x,y) and grid[x][y]==0): return True
    else: return False
#check the mouse click point if it is equal to i
def checkSameColor(x,y,i):
    if(checkInBoard(x,y) and grid[x][y]==i): return True
    else: return False
#The number of points with the same value as the key in the z-direction
def samekeyNumber(x,y,z,i,key,sk):
    if(i==1):
        while(checkSameColor(x+directionX[z]*i,y+directionY[z]*i,key)):
            sk+=1
            i+=1
    elif(i==-1):
        while(checkSameColor(x+directionX[z]*i,y+directionY[z]*i,key)):
            sk+=1
            i-=1
    return sk,i
#Count the number of pieces of the same color in the z direction
def pieceinLine(x,y,z):
    i=x+directionX[z]; j=y+directionY[z]
    s=0; ref=grid[x][y]
    if(ref==0): return 0
    while(checkSameColor(i,j,ref)):
        s,i,j=s+1,i+directionX[z],j+directionY[z]
    return s
#to see if the gave is over
def gameOver(x,y):
    global gaveover
    for u in range(4):#if the pieces is morethan 4, gameover
        if((pieceinLine(x,y,u)+pieceinLine(x,y,u+4))>=4):
            gaveover=True
            return True
    return False
```

InitBoard function used to draw the user interface by using a third-party library-graphics.

(*https://mcsp.wartburg.edu/zelle/python/graphics.py)*

```
#draw the gomoku board
def initBoard():
    #Set back ground color
    window.setBackground('#f8df70')
    #draw lines vertical
    for i in range(0,451,30):
        line=Line(Point(i,0),Point(i,450))
        line.draw(window)
    #draw lines vertical horizontal
    for j in range(0,451,30):
        line=Line(Point(0,j),Point(450,j))
        line.draw(window)
    message.draw(window)
    lastMove_AI.draw(window)
    lastMove_human.draw(window)
```

4. Minimax Algorithms

The Minimax algorithm is an algorithm that finds the minimum of the maximum probability of failure and is commonly used in games and programs where two players are competing, such as chess. There appear to be a variety of moves in Gomoku, but in reality each move is expanded to form a huge game tree.

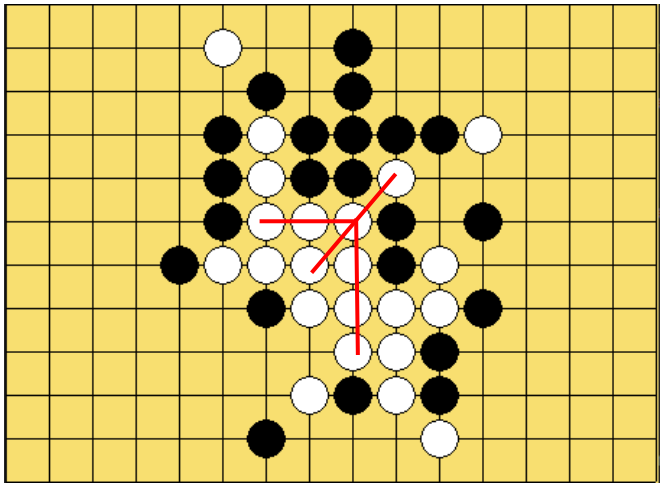
The algorithm uses Depth First Search to traverse the decision tree to populate the nodes in the middle of the tree with interest values for the leaf nodes, which are usually calculated using an interest evaluation function. The layer where the computer moves is what we call the MAX layer, this is the layer where the computer wants to ensure that it maximizes its interest, then it will pick the node with the highest score. The player moves on a layer we call the MIN layer, this is the layer where the player wants to ensure that he maximizes his interest, so he picks the node with the lowest score.

Minimax assumes that every move made by our opponent leads us in the direction of the pattern that is theoretically least valuable from the present point of view, i.e., the opponent has perfect decision-making ability. Our strategy should therefore be to choose the best of the worst-case scenarios that the opponent can achieve, even if the opponent causes the least damage to me in perfect decision-making.

Game Play

At the beginning, our game tree traverses all the points on the board to assess where the next move will be made, which results in very slow moves, and even in the middle and late stages of

the game, we have to wait four minutes to make a move. So, we need to improve the algorithm. One effective way to improve it is to evaluate the local refresh of the function.



In fact, if a piece is added to the board, as in the diagram, then only the scores of a few positions in the direction of its Interaction direction will change, most of the rest of the scores will not change at all. Through observation and experimentation, I have found that the following sixteen points are the most used in a 15*15 board. By traversing only these points, we will reduces the amount of computation significantly.

```
order=[8,7,9,6,10,5,11,4,12,3,13,2,14,1,15,0]
```

Evaluation Function

So how can we know which branch is optimal? We need an evaluation function that evaluates the entire current situation and returns a score. We specify that the more favorable to the computer, the higher the score, and the more favorable to the player, the lower the score. We simply use an integer to represent the current situation, the higher the score the greater the advantage, the lower the score the greater the advantage of the opponent, a score of 0 means that the situation is equal.

```
#Evaluation function to get a score
def evaluation(x,y):
    global gaveover
    if(gameOver(x,y)):
        gaveover=False
        return 10000
    score=openFour(x,y)*1000+(continuousFour(x,y)+openThree(x,y))*100

    for u in range(8):
        if(checkInBoard(x+directionX[u],y+directionY[u]) and grid[x+directionX[u]][y+directionY[u]]!=0):
            score=score+1
    return score
```

Open Four

```
#The number of openFour situations in the four directions of the landing point
def openFour(x,y):
    key,number=grid[x][y],0
    for u in range(4):
        samekey=1
        samekey,i=samekeyNumber(x,y,u,1,key,samekey)
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        samekey,i=samekeyNumber(x,y,u,-1,key,samekey)
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        if(samekey==4):
            number=number+1
    return number
```

If one side is blocked but the other side is not, the score is reduced by one notch, i.e. dead four and alive three are the same score

Continuous Four

Continuous four is a position where, in addition to a "open four", one more move can form a five-row, and a five-row is possible.

_●●●●○	●_●●●	●●_●●

```
def continuousFour(x,y):
    key=grid[x][y]; number=0
    for u in range(8):
        samekey=0; flag=True; i=1
        while(checkSameColor(x+directionX[u]*i,y+directionY[u]*i,key) or flag):
            if(not checkSameColor(x+directionX[u]*i,y+directionY[u]*i,key)):
                if(flag and checkInBoard(x+directionX[u]*i,y+directionY[u]*i) and grid[x+directionX[u]*i][y+directionY[u]*i]!=0):
                    samekey-=10
                flag=False
            samekey+=1
            i+=1
        i-=1
        if(not checkInBoard(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        samekey,i=samekeyNumber(x,y,u,-1,key,samekey)
        if(samekey==4):
            number+=1
    return number-openFour(x,y)*2
```

Open Three and Dead Three

By traversing the pieces in the direction of the interaction of the played pieces, we can calculate the number of Open tree and Dead Three situations.

Open three

_●●●_	_●_●●_

Dead three

_●●●○	_●_●●○
_●●_●○	●_ _●●
●_●_●	○_●●●_○

```
#The number of open threes in the four directions and the number of dead threes in the eight directions of the point
def openThree(x,y):
    key=grid[x][y]; number=0
    for u in range(4):
        samekey=1
        samekey,i=samekeyNumber(x,y,u,1,key,samekey)
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        if(not checkDrop(x+directionX[u]*(i+1),y+directionY[u]*(i+1))):
            continue
        samekey,i=samekeyNumber(x,y,u,-1,key,samekey)
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        if(not checkDrop(x+directionX[u]*(i-1),y+directionY[u]*(i-1))):
            continue
        if(samekey==3):
            number+=1
    for u in range(8):
        samekey=0; flag=True; i=1
        while(checkSameColor(x+directionX[u]*i,y+directionY[u]*i,key) or flag):
            if(not checkSameColor(x+directionX[u]*i,y+directionY[u]*i,key)):
                if(flag and checkInBoard(x+directionX[u]*i,y+directionY[u]*i) and grid[x+directionX[u]*i][y+directionY[u]*i]!=0):
                    samekey-=10
                flag=False
            samekey+=1
            i+=1
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        if(checkInBoard(x+directionX[u]*(i-1),y+directionY[u]*(i-1)) and grid[x+directionX[u]*(i-1)][y+directionY[u]*(i-1)]==0):
            continue
        samekey,i=samekeyNumber(x,y,u,-1,key,samekey)
        if(not checkDrop(x+directionX[u]*i,y+directionY[u]*i)):
            continue
        if(samekey==3):
            number+=1
    return number
```

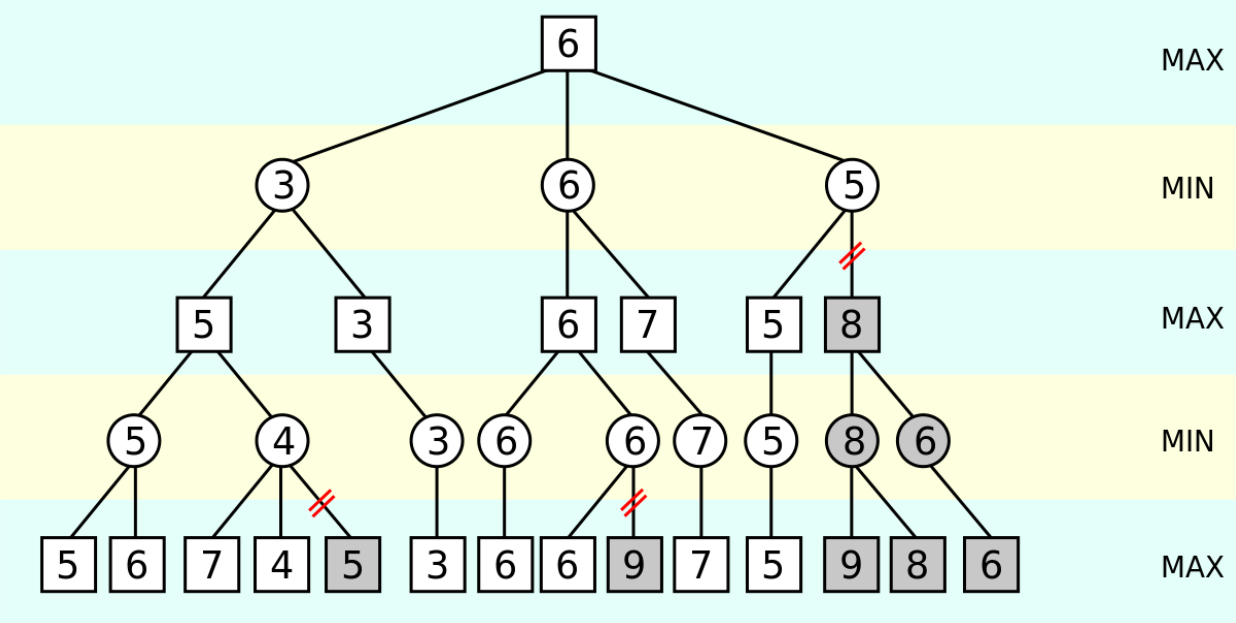
5. Alpha-beta Pruning

The alpha-beta pruning algorithm is a safe pruning strategy that does not have any negative impact on the strength of the game. Its main strategy is to determine if a node is a disadvantageous node then it can be pruned, thus saving time and increasing the efficiency of the algorithm.

Typically, we place the AI in the MAX layer and the player in the MINI layer, then the AB algorithm runs logically as follows.

If a maximum value X has been searched for in the MAX layer, then this value is stored and compared with the value generated by the next node found in the MINI layer, and if this value is less than X, then the node is simply cut. In simple terms, in the MINI tier the player will definitely choose the minimum value in order to win. This means that the player will not score more than Y at this node, so there is obviously no need to compute for this node.

If in the MINI layer we as well search for a minimum value Y, and if we find that the next MAX layer produces a value greater than Y, then we cut this node. This is because if the player makes this move, it will be more favourable to the AI for the situation, then we must not make this move.



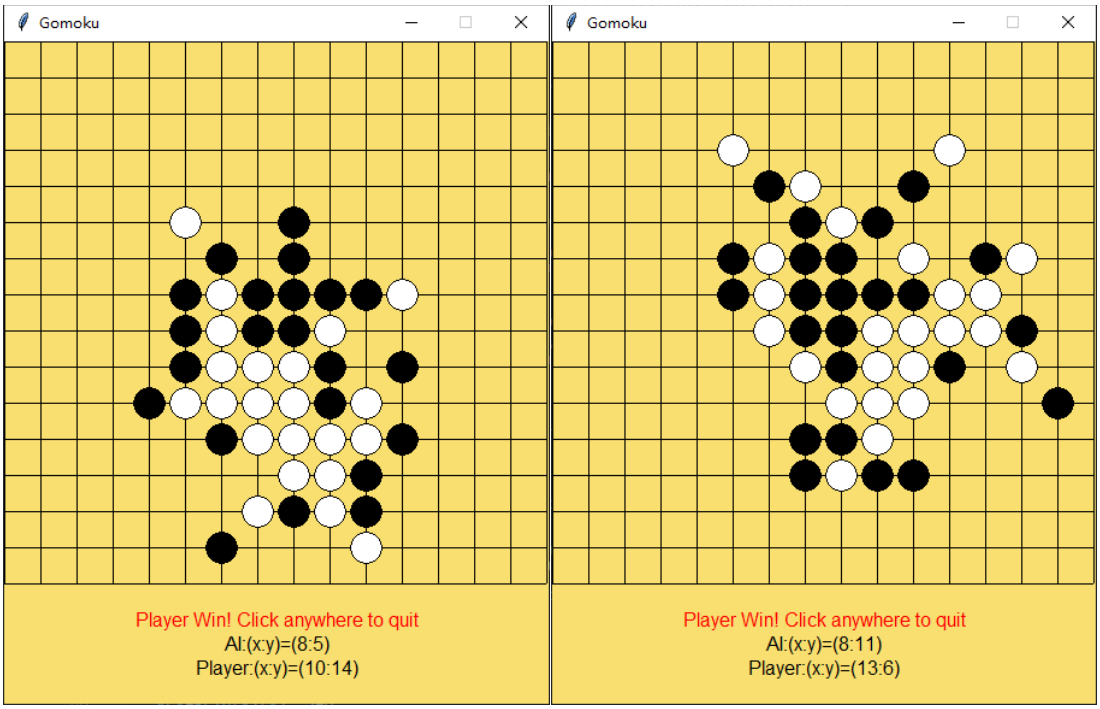
In order to implement our algorithm we will add an alpha and beta parameter to the playbook respectively. The values of both parameters are set to a larger value, in our case 100000 and -100000. If a child node is found in the max layer with a value greater than alpha, the descendant node is no longer counted. Here is the code implementation in our game.

```
def GameTree2():
    global beta
    beta=100000
    for x in order:
        for y in order:
            if(not checkDrop(x,y)):
                continue
            grid[x][y]=3-ai
            tempp=evaluation(x,y)
            if(tempp==0):
                grid[x][y]=0; continue
            if(tempp==10000):
                grid[x][y]=0; return -10000
            tempp=GameTree3(tempp)
            if(tempp<alpha): #Pruning level 1
                grid[x][y]=0; return -10000
            grid[x][y]=0
            if(tempp<beta): #get minimum value
                beta=tempp
    return beta
```

```
def GameTree3(p2):
    keyp=-100000
    for x in order:
        for y in order:
            if(not checkDrop(x,y)):
                continue
            grid[x][y]=ai
            tempp=evaluation(x,y)
            if(tempp==0):
                grid[x][y]=0; continue
            if(tempp==10000):
                grid[x][y]=0; return 10000
            if(tempp-p2*2>beta): ##Pruning level 2
                grid[x][y]=0; return 10000
            grid[x][y]=0
            if(tempp-p2*2>keyp): #get max value
                keyp=tempp-p2*2
    return keyp
```

6. Conclusion

After some experimentation tweaking the parameters, the AI system of three-level game tree has the best gameplay, although somewhat it is still difficult to win, but not invincible. Players will need to think hard to beat the AI in the mid to late game, but if they make a mistake early on, the AI will quickly gain the victory



Although we have made great progress with AI so far, there is still a distance between us and the real masters. There are still a lot of areas that need to be optimized.

Firstly, it is still slightly lacking in terms of running speed. Later on we intend to improve the speed of AI thinking in two ways. The first is to speed up the computation by means of multi-threading or Improving our search algorithm. If the search starts at the beginning of the search, it is possible to find relatively large maxima and relatively small minima as quickly as possible by searching around the point from the previous step, thus allowing for faster pruning. This is because the neighboring points are the most likely ones. The second one is to have an opening

book.(De Firmian, 2008) which is a feature set that allows the program to "learn and correct itself" from repeated opening changes that always lead to disadvantages (failures). Just as one remembers the opening book. This is a major help for the program to save thinking time. In addition, we can use the previously mentioned GA algorithm to increase the speed or use deep learning to create an AI that can play chess like alpha go.

7. References

Allis, L. V. (1994). Searching for solutions in games and artificial intelligence. Ponsen & Looijen Wageningen.

Campbell, M. S., & Marsland, T. A. (1983). A comparison of minimax tree search algorithms. Artificial Intelligence, 20(4), 347-367.

Chouard, T. (2016). The go files: AI computer wraps up 4-1 victory against human champion. Nature News.

De Firmian, N. (2008). Modern chess openings: MCO-15. Random House Incorporated.

Gomoku World. (2018). Opening rules |GomokuWorld.com. http://gomokuworld.com/gomoku/2

Higgins, C. (2017). A brief history of deep blue, IBM's chess computer. Mental Floss,

Holland, J. H. (1992). Genetic algorithms. Scientific American, 267(1), 66-73.

Junru Wang, & Lan Huang. (2014). Evolving gomoku solver by genetic algorithm. Paper presented at the - 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), 1064-1067. https://doi.org/10.1109/WARTIA.2014.6976460

K. Shao, D. Zhao, Z. Tang, & Y. Zhu. (2016). Move prediction in gomoku using deep learning. Paper presented at the - 2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC), 292-297. https://doi.org/10.1109/YAC.2016.7804906

Lasker, E. (2012). Go and go-moku. Courier Corporation.

Russell Stuart, J., & Norvig, P. (2009). Artificial intelligence: A modern approach. Prentice Hall.

Schaeffer, J. (1989). The history heuristic and alpha-beta search enhancements in practice. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(11), 1203-1212.