Report for CS4386 Assignment 2 (Semester B, 2019 - 2020)

Abstract

Dots-And-Boxes is an interesting game played on a rectangular grid of dots. There are two players in the game, they can draw a horizontal or a vertical line between the adjacent dots. The last one who completes a square (or two squares) will gain one point (or two points) and get another turn. The game is finished once all the squares are completed, and the player with the highest score will be the winner. [1] In this report, I will introduce several AI algorithms to play the game with a size 5-by-5. I have tried different algorithms, including simple if-else logic (also can be treated as a 2 layer min-max tree), Nega-max search, Nega-max with α-β pruning (ab-Nega-max), Monte Carlo Tree Search (MCTS). I also tried other schemes including changing the data structure of the game state, using the transposition table, tuning the evaluation function, randomizing the final output, and using the dynamic depth of search. Finally, I decide to use an algorithm as an MCTS guided ab-Nega-max with simulation-evaluation function and dynamic depth (dynamic depth is the most important part of the algorithm) since it has an acceptable success rate (96% against *housebot-competition*), speed of computation, and a random initialization state. All the codes for the algorithms described above can be provided upon request. However, only the file *MCTS_ABNEGAMAX_BOT.py* is in the submitted zip file. All the experiments are conducted on the AI GAMING website [2].

Challenges

There are several challenges to implement a feasible AI algorithm to play this game, including time limit, memory limit, the complexity of the game state, evaluation function, and player switching.

Time Limit On the AI Game website, the AI algorithm will be called when the AI player on our side needs to make a move. For each decision, there will be a deadline given in the UNIX timestamp format. The performance of the tree search algorithms depends on the depth searched. Algorithms with a larger search depth will have better performance (higher rate of winning). However, under the time limit constraint, I cannot have a depth of searching larger than two for Nega-max and ab-Nega-max. And for MCTS, the simulation can only be conducted less than 295 times.

Memory Limit I did not find the exact memory limit for this experiment environment. However, for each new game, the memory used in the previous game will be released. Therefore, I cannot reuse the computation result of the previous round, which makes the transposition table approach less efficient. Indeed, I have tried to combine the transposition table with Nega-max. The result is not acceptable with even lower accuracy and no obvious speed enhancement. The reason might be the depth limit makes the evaluation score of the previous round less accurate. When we reuse the score, it is no longer representative.

Game State Complexity Game state complexity is also an important issue, which will affect the computation speed. For the game *Tic-Tac-Toe*, regardless of whether the state is legal, there are only $3^9 = 19683$ states. For *Dots-And-Boxes*, $2 * 4 * 5 = 40$ edges can be selected by two users, which will construct 3^{40} states including all valid or invalid states. Therefore, it is more complex to build an AI to play this game than *Tic-Tac-Toe*.

Evaluation Function As a consequence of the previous challenges, we can only search for a limited number of layers. Therefore, it is difficult to give an accurate evaluation function for Nega-max and ab-Nega-max. We need to find a heuristic function to evaluate the current state.

Player Switching In a normal zero-sum game, two players take turns to play the game. However, a player can take one more move once it completes a square in *Dots-And-Boxes*. This rule will change the logic of switching players in Nega-max and ab-Nega-max. Besides, in the MCTS algorithm, the new state given in the next move will be one of the offspring nodes of the node selected in the previous round. For a normal zero-sum game, we

can simply search all the child nodes. But for *Dots-And-Boxes,* we need to search for all the offspring nodes since the opponent might take more than one step.

Algorithms

As stated in the *Abstract* part, I have tried many algorithms. Theses algorithm can be classified into three categories including strategy-based search, evaluation-based search, and probability-based search. There are two metrics to evaluate the algorithm:

$$formula\ 1\colon \frac{w}{w+f}$$

$$formula\ 2\colon \frac{w}{w+d+f}$$

Where w is the number of winning, f is the number of losing, and d is the number of draws. Table 1 shows the two metrics of the algorithms when playing against the *housebot-competition* on the AI GAMING website. Notice that the final proposed algorithm is mixed, therefore all the algorithms involved in the design of the final algorithms will be introduced.

Table 1: The Evaluation of the Algorithms

Algorithm	if-else rule	Ab-Negamax			MCTS	MCTS + Ab-Negamax	
Scheme	None	Counting "3"s Evaluation (depth = 3)	Simulation Evaluation (depth = 2)	Simulation Evaluation with randomization	None	Same as best one for Ab-Negamax	Dynamic Depth
Formula1	0.55	0.85	0.86	0.55	0.03	0.84	**0.96**
Formula2	0.43	0.72	0.73	0.53	0.03	0.76	**0.94**

Strategy-Based Search

According to [1], a possible strategy for *Dots-And-Squares* is to prioritize moves that can gain more scores as well as guard against the opponent using the same strategy.

Strategy based search is implemented by if-else rule, the algorithm will enumerate all possible moves in two steps and choose the one results in the game state that minimizes the score: $score = \Delta Myscore - \Delta Oppscore$. It is equivalent to a two-layer min-max tree using the evaluation function as the score taken in the two steps by me minus the score take in the two steps by the opponent. This method yields a success rate of 0.55 despite draw cases.

Evaluation-Based Search

Although the strategy-based search is better than the random approach, the success rate is not acceptable. Therefore, I tried to use other approaches to deal with this problem.

Mini-max/Nega-max Mini-max is a widely used tree search algorithm for board games. In this algorithm, each state of the game is represented by a node, each child node represents a possible state after one of the legal moves at the current state. Once we find the finish state or the maximum depth, an evaluation score will bubble up according to the minimax rule which is determined by the player changing. As a variant of mini-max, Nega-max reverses the sign of the score to make the implementation more intuitive [3]. In this specific problem, an important issue is that a player can take more than one move. Therefore, instead of directly changing the sign of the score, we need to check whether the next move is taken by another player.

Inspired by page 17 of the CS4386 lecture note [3], I implemented a Nega-max search algorithm with the same evaluation function. However, the success rate is not improved and the computation speed is reduced. The reason is the strategy-based search algorithm is equivalent to a minimax search tree with depth two.

```python
def negamax(gamestate, maxDepth, currentDepth):
    if gamestate['GameStatus'] == 'STOPPED' or (maxDepth == currentDepth) :
        score = gamestate['MyScore'] - gamestate['OppScore']
        return score if gamestate['IsMover'] else -score, None

    bestMove = None
    bestScore = -100

    legal_moves = find_legal_moves(gamestate["Grid"])
    for move in legal_moves:
        newState = simulate_move(move, gamestate)

        (recursedScore, currentMove) = negamax(newState, maxDepth, currentDepth+1)
        currentScore = -recursedScore if (gamestate['IsMover'] ^ newState['IsMover']) else recursedScore

        if currentScore > bestScore:
            bestScore = currentScore
            bestMove = move

    return bestScore, bestMove
```

Figure 1: Implementation of Nega-max refer to [3]

Data structure As stated in the previous parts, I tried different data structures including a transposition table and a different structure for game state. It turns out that the transposition table does not have a good performance. For the game state, I used the default structure at first, which is slow. After experimenting, I found that the slow execution is caused by the deep copy of the game state and searching in the Grid/Square list during the simulation process. Therefore, I changed the data structure from list to dictionary for Grid and Square list and wrap the game state into a *Board* class.

```python
def from_list_to_dict(gamestate):
    gamedict = {}
    gamedict['Dimensions'] = gamestate['Dimensions']
    gamedict['IsMover'] = gamestate['IsMover']
    gamedict['ResponseDeadline'] = gamestate['ResponseDeadline']
    gamedict['MyScore'] = gamestate['MyScore']
    gamedict['OppScore'] = gamestate['OppScore']
    gamedict['InvalidMove'] = gamestate['InvalidMove']
    gamedict['OpponentId'] = gamestate['OpponentId']
    gamedict['GameStatus'] = gamestate['GameStatus']
    gamedict['Grid'] = {}
    gamedict['Squares'] = []

    for line in gamestate['Grid']:
        move = (line[0], line[1])
        gamedict['Grid'][str(move)] = line

    for square in gamestate['Squares']:
        lines_already_complete = list(filter(lambda l: l[2], square[0]))
        num_lines_complete = len(lines_already_complete)
        gamedict['Squares'].append(num_lines_complete)
```

```python
class Board:
    def __init__(self, gamestate):
        # change the data structure of the gamestate to the dictionary
        self.state = gamestate
        self.dimensions = gamestate['Dimensions']
        self.is_mover = gamestate['IsMover']
        self.res_ddl = gamestate['ResponseDeadline']
        self.my_score = gamestate['MyScore']
        self.opp_score = gamestate['OppScore']
        self.game_status = gamestate['GameStatus']
        self.grid = gamestate['Grid']
        self.squares = gamestate['Squares']

    def finished(self): ···

    def isMover(self): ···

    def take_move(self, move): ···

    def isWin(self): ···

    def isEqual(self, board): ···

    def take_move_chain(self, maxDepth): ···

    def take_random_move(self): ···

    def evaluate(self): ···

    def evaluate_1(self): ···

    def evaluate_2(self): ···
```

Figure 2: Implementation of from_list_to_dict and Board class

In the *Board* class, *finished(), isMover(), isWin(),* and *isEqual()* function will return the corresponding boolean value. *Take_move(), take_move_chain(), take_random_move()* will return a new *Board* object. And there are three different evaluation functions. There will be a more detailed introduction to these functions. Notice that for the square list, I change it to a list consists of only numbers that represent the number of edges completed for a specific square. The mapping between the list index and the left-top corner of the square can be represented as $index = (n-1) * x + y$, where n is the number of rows and x, y are two coordinates of the left-top dot. By using the new data structure, the execution speed of the program is significantly enhanced.

AB-Nega-max As stated in [3], Nega-max with α-β pruning maintains a changeable interval of searching. The upper-bound β is defined by the maximum value of $\Delta Myscore - \Delta Oppscore$ to get considering searched nodes and current node. While α is the symmetric value represents the minimum value of $\Delta Myscore - \Delta Oppscore$ to get. It is obvious that α is controlled by the current player, while β is controlled by the opponent. The implementation is inspired by the pseudo-code on page 21 of [3]. Notice that we need to change the rule of updating α according to the player switching property. The success rate is not improved, but the computation speed is enhanced at least two times.

```python
def abnegamax(board, maxDepth, currentDepth, alpha, beta):
    if board.finished() or (maxDepth == currentDepth) :
        score = board.evaluate_2()
        return score, None

    bestMove = []
    bestScore = -INFINITY

    legal_moves = board.get_legal_moves()
    for move in legal_moves:
        newBoard = board.take_move(move)
        recursedScore = 0
        currentScore = 0
        currentMove = None

        # calculate the bestscore for the newstate and store
        if board.isMover() == newBoard.isMover():
            (recursedScore, currentMove) = abnegamax(newBoard, maxDepth, currentDepth+1, max(alpha, bestScore), beta)
            currentScore = recursedScore
        else:
            (recursedScore, currentMove) = abnegamax(newBoard, maxDepth, currentDepth+1, -beta, -max(alpha, bestScore))
            currentScore = -recursedScore

        # Update the best score
        if currentScore > bestScore:
            bestScore = currentScore
            bestMove = []
            bestMove.append(move)

        # If we're outside the bounds, then prune: exit immediately
        if bestScore >= beta:
            return bestScore, bestMove

        # store all the moves with the same score
        if currentScore == bestScore:
            bestMove.append(move)


    return bestScore, bestMove
```

Figure 3: Implementation of ab-Nega-max refer to [3]

Counting "3"s evaluation Previous schemes only increase the execution speed based on the Nega-max approach. To increase the success rate, we need to come up with a better evaluation function. Counting "3"s evaluation will add the number of squares that has three occupied edges to the score of the current state and return it. This new evaluation score is easy to be calculated and gives a reasonable prediction of future income. By using this method and ab-Nega-max with the new data structure, the depth of searching is at most three. This method yields a success rate of 0.85 despite draw cases.

```
def evaluate_1(self):
    # Heuristic evaluation based on the current state and number of completable squares

    current_score = self.evaluate()

    # Chain Cases
    completable_squares = list(filter(lambda square: square == 3, self.squares))
    num_completable_squares = len(completable_squares)
    score = current_score + num_completable_squares
    return score
```

Figure 4: Implementation of the counting "3"s evaluation

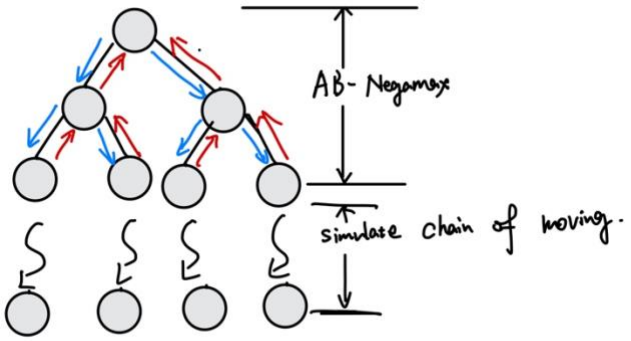

Figure 5: The process of simulation evaluation approach

Simulation evaluation Although the counting "3"s evaluation yields a better success rate, the evaluation score is not accurate enough. Compared to counting "3"s evaluation, simulation evaluation has better accuracy and slower execution speed. The idea of the simulation evaluation is inspired by the Monte Carlo Tree Search (MCTS) approach. For MCTS, we randomly simulate the game until the game is finished for many times under some constraints, and select the move based on the result of the simulation. Similar to MCTS, the new evaluation will continue simulating the state when the maximum search depth is reached. However, the search is not fully random. The algorithm will repetitively select all squares with three edges completed and randomly choose one square to complete. The simulation process will be stopped when there are no more squares with three edges completed, and the evaluation score equals to the difference between the score of the current player and the score of the opponent. Notice that since there is no constraint on changing players, the simulation will consider the current player's moves as well as the opponent's moves. Due to the time limit, the maximum depth for simulation evaluation is at most two. This method yields a success rate of 0.86 despite draw cases.

```
def evaluate_2(self):
    # Heuristic evaluation based on simulation
    # There will be a chain of simulation until the player is changed (i.e. 0 score exists)
    return (self.take_move_chain(15)).evaluate()
```

Figure 6: Implementation of Simulation Evaluation

```python
def take_move_chain(self, maxDepth):
    # take a chain of moves as long as there are completable squares
    m = self.dimensions[0]; n = self.dimensions[1]
    current_board = self
    completable_squares = [i for i, square in enumerate(current_board.squares) if square == 3]
    if len(completable_squares) == 0:
        return current_board

    currentDepth = 0
    while len(completable_squares) > 0 and currentDepth < maxDepth and (not current_board.finished()):
        currentDepth += 1
        idx = random.randint(0, len(completable_squares)-1)
        x = completable_squares[idx] // (n-1)
        y = completable_squares[idx] % (n-1)
        line1 = ([x, y], [x + 1, y])
        line2 = ([x, y], [x, y + 1])
        line3 = ([x + 1, y], [x + 1, y + 1])
        line4 = ([x, y + 1], [x + 1, y + 1])

        grid = current_board.grid
        if grid[str(line1)][2] == -1 :
            current_board = current_board.take_move(line1)
        elif grid[str(line2)][2] == -1 :
            current_board = current_board.take_move(line2)
        elif grid[str(line3)][2] == -1 :
            current_board = current_board.take_move(line3)
        elif grid[str(line4)][2] == -1 :
            current_board = current_board.take_move(line4)

        completable_squares = [i for i, square in enumerate(current_board.squares) if square == 3]
        num_completable_squaresd = len(completable_squares)
    return current_board
```

Figure 7: Implementation of the function take_chain_move()

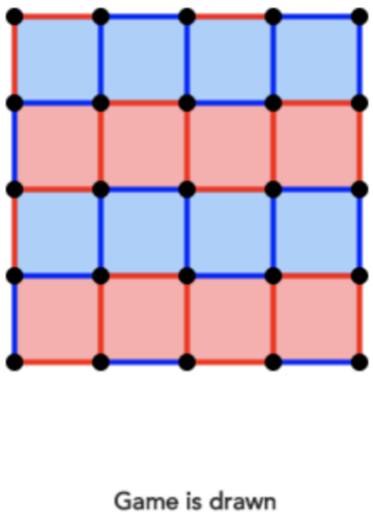

Figure 8: The draw case without randomization

Randomization I tried to let two algorithms play against each other during the experiment to find out which one is better. I found that two different Nega-max based algorithms will always yield a draw as shown in Figure 8. The reason is that in the early stage of the game, all the moves will change the current state to a new state without getting points. And this stage of the game will be extended when the players are rational. At this stage, players will always choose the first move stored in the grid list. To improve the diversity of the AI algorithm, I try to record a list of best moves and randomly select one move from the list. However, the experiment shows that the success rate is only 0.55 against *housebot-competition* which is the same as the if-else rule. It seems that choosing the first one in a list of best moves somehow is beneficial to the current player, although it always yields draw with Nega-max based algorithm. This problem will be solved by probability-based search.

Probability-Based Search

MCTS Monte Carlo Tree Search (MCTS) is a probability-based search algorithm. As stated in [4], MCTS has the advantage that it is capable of any kinds of board games. We do not need to design a specific evaluation function to execute the algorithm, the selection of move will only be based on the probability of success. For each decision, MCTS will randomly simulate the game by many rounds, each round will be ended only the game is finished. For each round of the simulation, there are four operations including selection, expansion, simulation, and backpropagation. The select operation can be only applied to nodes that are fully-expanded (all possible child nodes are created), it will select a child node with the maximum UCB score, which is defined by the following equation:

$$UCB_i = \frac{W_i}{N_i} + C * \sqrt{\frac{lnN}{N_i}}$$

Where i represents the i-th child of the current node, W_i and N_i represent the number of winning and the total number of simulations respectively for child i, N is the total number of simulations for the current node, C is a parameter to control the weight of the second term. Intuitively, the first term represents the probability of success, while the second term represents the degree of ignorance for this child node. Therefore, a node with a higher possibility of success, less simulation frequency will be preferred.

The expansion state simply generates a new child node by selecting a move that has not been used. The simulation state will be activated when the algorithm finds a node with a zero number of total simulations. The simulation will be ended only when the game is finished. Once the simulation state is ended, the backpropagation state is activated. The backpropagation state will update the information from the leaf node to the root node. It will increase the number of simulations for all relevant nodes by one and increase or decrease (depends on the player of that node) the total number of winning for all the relevant nodes by one. Once the algorithm is finished, there will be one more selection with $C = 0$, apply to the root node. The implementation of this algorithm is referring to [5].

The problem of MCTS is that the performance of the algorithm depends on the sample size of the simulation. Due to the time limit, the maximum number of simulations applicable is 295, while the number of all possible final states is 2^{40}. It makes the result of the simulation less accurate. The player switch limitation also makes it difficult to use MCTS. Since it is possible for a player to take more than one move, it is not applicable to memorize the Monte Carlo tree during one round. Therefore, the performance of this approach is not ideal with a success rate of 0.03.

Final Algorithm

After the experiment, the final algorithm is selected. The idea of this algorithm consists of MCTS guided ab-Nega-max and dynamic depth for tree search (the most important part of this algorithm).

MCTS guided ab-Nega-max Although the performance of pure MCTS is not acceptable, it still outperforms the random selection algorithms. Therefore, instead of maing a move from the best moves in the early stage of the game, we can use MCTS to select a move from the best moves provided by the ab-Nega-max algorithm. The condition of activating the MCTS is that the number of best moves is larger than 35. This method yields a success rate of 0.84 despite draw cases. This method is better than the ab-Nega-max algorithm with simulation evaluation since it has a similar success rate and a greater degree of randomness.

Dynamic Depth The dynamic depth scheme can significantly increase the success rate of the AI algorithm. When I published the MCTS guided ab-Nega-max to the AI GAMING website to play against my classmates, I noticed that there was a player can win almost all the games against my bot. By observing the games played against the top players and doing online research, I found that my AI algorithm has missed an important trick to play this

game called *The double-cross strategy* [6]. To illustrate this strategy in detail, I will introduce several terminologies.

- ***Chain State***: Assume each one of the two players is rational, they will try their best to prevent the opponent get the first point. Therefore, the game will be played until the existence first square with three edges occupied. The chain state is defined as the state that will result in the first square with three edges occupied by taking any possible move from that state. For example, in Figure 9 the left game state is a chain state, while the right game state is not a chain state since you can take the move ((2,3), (2,4)).

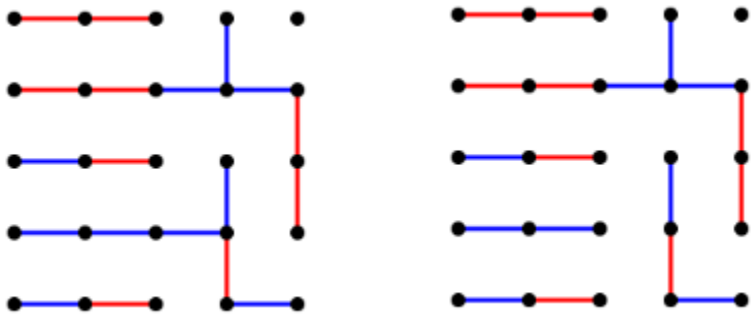

Figure 9: Chain State VS Non-Chain State

- ***Connectivity*:** Two squares are defined to be directly connected if they share an edge that is not occupied. Two squares (says A, B) are said to be connected if we can construct a sequence of squares satisfying:
  - Adjacent squares are directly connected.
  - Square A and B are included in this sequence.
- ***Chain / Region***: In the chain state, all the squares on the game board can be divided into subsets of squares. All the squares within the same subset are connected, and there are no two squares connected if they belong to different subsets. The subsets are defined as chains or regions.

It is obvious that in the chain state, if player1 occupies an arbitrary edge within a region, player2 can take a series of moves to complete all the squares in that region. However, for player2, sometimes it may not be a good idea to complete the whole region. Since a player must make a move after getting a point, if player2 completes all the squares, it needs to occupy one edge in another region. We assume all the players are rational, the player1 will occupy an edge in the region containing the minimum number of squares, if player2 completes all the squares in this region, it will lose a larger region to player1. Therefore, in this situation, a better strategy is to complete the region until there are two consecutive squares, the player2 form a 2-by-1 triangle to give the turn to the player1 (as shown in Figure 10).

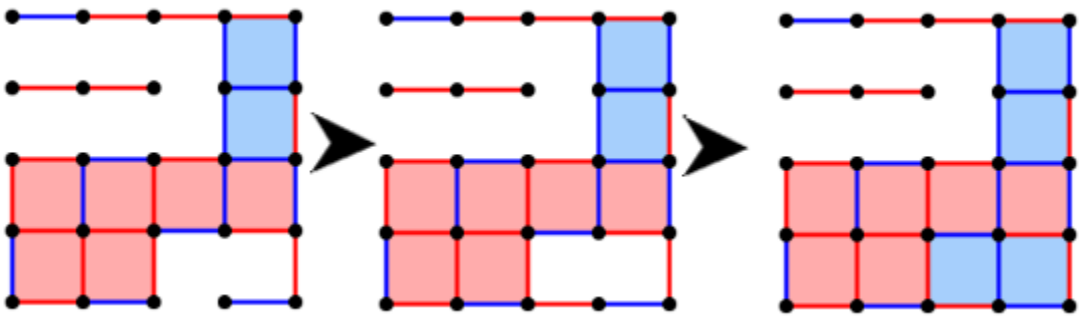

Figure 10: The double-cross strategy

In the original design of my algorithm, the maximum search depth is two. If the game is in a chain state, the algorithm with search depth two cannot find out *the double-cross strategy*. Since the algorithm only applies the simulation evaluation to all possible states two moves later from the current state, and the simulation will only be performed when there is a square with three edges occupied, my algorithm can only find out the interest of completing the last two squares in a region without foreseeing a great loss in the future. Unfortunately, if I increase the search depth to three, timeout error will exist.

To solve this problem, I use a dynamic depth scheme. This scheme will increase the search depth to three when the total number of the unoccupied edges is less than a threshold since the game will only get into a chain state when most of the edges are occupied. For each region, the number of unoccupied edges is the number of squares in the region plus one. Therefore, the total number of unoccupied edges for the whole game board is roughly equal to the number of regions plus 16 (there are 16 squares in total). Since the minimum number of regions is one, and the maximum number of regions is impossible to be 16, the interval for the threshold is $[17, 32)$ theoretically. However, during the experiment, I found that the double-cross strategy can be applied later than the first chain state appears, so I set the threshold to be 18. Besides, I also increase the depth when the number of remaining legal moves is getting smaller.

```
# function will be called
def calculate_move(gamestate):
    gamestate = from_list_to_dict(gamestate)
    board = Board(gamestate)
    move_num = len(board.get_legal_moves())

    depth = 2
    if move_num <= 6:
        depth = move_num
    elif move_num <= 8:
        depth = 5
    elif move_num <= 10:
        depth = 4
    elif move_num <= 18:
        depth = 3
    (bestScore, bestMoves) = abnegamax(board, depth, 0, -INFINITY, INFINITY)

    idx = random.randint(0, len(bestMove) - 1) if RANDOMIZE == True else 0
    action = bestMoves[idx]

    if len(bestMoves) > 35:
        mcts = MCTS(bestMoves)
        action = mcts.choose_action(board)
    return action
```

Figure 11: Implementation of the Final Algorithm

The implementation of the final algorithm is shown in Figure 11 (the number of simulations for MCTS is set to be 40). The final algorithm yields a success rate of 0.96 despite draw cases.

Possible Improvements

Although the success rate against *homebot-competition* is high enough, considering my classmate may use similar or better algorithms, I come up with two possible improvements to the current algorithm which are not implemented.

Chain State Evaluation If we assume two players are rational, they both use the double-cross strategy. In this case, an important issue is to control the chain state. For example, if there are fewer chains (which means a larger number of squares within each chain), the first player who takes move on the chain state is likely to lose since the second player will use the double-cross strategy on the first chain. While if there are more chains (says the size of the smallest chain is one), the first player who takes move on the chain state is likely to win by using the double-cross strategy on the second chain with length larger than one. If this chain-state-level information can be expressed as a formula and plug in the evaluation function, the performance might be better.

Transposition Table I tried to use the transposition table for ab-Nega-max, but I did not try it for MCTS. Maybe we can use it to store the location in the Monte Carlo tree for each game state, then the player switching problem will be solved.

References

[1] "Dots And Boxes," 2019. [Online]. Available: https://www.aigaming.com/Help?url=game-help/dots-and-boxes. [Accessed April 2020].

[2] [Online]. Available: https://www.aigaming.com/.

[3] H. Leung, "CS4386 AI Game Programming," City University of Hong Kong, Hong Kong, 2020.

[4] P. Muens, "Game AIs with Minimax and Monte Carlo Tree Search," 3 Apr 2019. [Online]. Available: https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0. [Accessed Apr 2020].

[5] R. Yuan, "蒙地卡羅搜索法," 13 Aug 2019. [Online]. Available: https://medium.com/@skywalker0803r/%E8%92%99%E5%9C%B0%E5%8D%A1%E7%BE%85%E6%90%9C%E7%B4%A2%E6%B3%95-ee7de77940ef. [Accessed Apr 2020].

[6] "Dots and Boxes," [Online]. Available: https://en.wikipedia.org/wiki/Dots_and_Boxes.