CS440/ECE448 Fall 2016

Artificial intelligence
Assignment 2:
Constraint Satisfaction Problems

Work Distribution:
Part 1: Yifei Li
Part 2: Bangqi Wang
20th Oct, 2016

Part 1: CSP: Word Sudoku

In this assignment, we will implement a variation on Sudoku that we fill the grid with words. We will be given a word bank, which is a list of words that must be used exactly once(Or in input3, part of the list). The words must be arranged so that every cell contains a letter. The words can take up multiple cells in the grid and can be oriented either horizontally or vertically, and words are allowed to overlap. Following the rules of Sudoku, each row, column, and 3x3 cell cannot contain duplicate letters.

Constraint Satisfaction Problem Algorithm

In this part, our algorithm uses general backtracking algorithm for constraint satisfaction problem. Our implementation works for input1 and input2. However, input 3 requires some modification. We will talk about that in the later section.

Below is the pseudo code snippet of our backtracking algorithm:

```
def sudokuSolver(sudoku,wordbank,assignment):
    if not find_empty_location(sudoku):
        return Success
    curr_word = select_unassigned_variable(wordbank)
    for value in order_domain_value(sudoku,curr_word):
        if vertical_check(sudoku,curr_word,value) and vertical_constraint_satisfied(sudoku,value[1],value[0],curr_word):
            assign variable,value to assignment
            remove value from wordbank
            if sudokuSolver(sudoku,wordbank,assignment):
                return Success
            else:
                unassign the variable from assignment
        if horizontal_check(sudoku,curr_word,value) and horizontal_constraint_satisfied(sudoku,value[1],value[0],curr_word):
            assign variable,value to assignment
            remove value from wordbank
            if sudokuSolver(sudoku,wordbank,assignment):
                return Success
            else:
                unassign the variable from assignment
    return Failure
```

Constraint Satisfaction Problem Algorithm Code snippet

The idea here is that every time we select one most constrained variable from Wordbank. Then we select one least constrained value from the sudoku. We try to fill the word into sudoku and check if there's duplicate letter in same row, column or 3x3 grid. If the value is consistent with our constraint checking, then we can fill

it in the sudoku. After we finish assignment, the function go deep down one level and check next available variable. If there's no complete assignment returned by recursion call, we unassign this current variable and check next orientation or next grid. In this way, the function can finally determine whether the sudoku can be completely filled.

Note that for input1 and input2, there's no need for optimization using forward checking and arc consistency checking. Since our program finishes executing immediately, so we do see the needs to further optimize our code.

Variable, Domain, Constraints:

For choice of variable, we use the individual word in the wordbank as our variable. And we choose set of grid in sudoku as our domain. In this way, we have a recursion tree of depth equal to the length of wordbank and branching factor smaller or equal to numbers of grids in the sudoku. In this way, the computation time is largely reduced compared to the choice of selecting grids as variable and wordbank as domain.

Regarding to constraints selecting, we initially use length of word to select the most constrained variable since the longer word will be harder to fit in the empty grid. However, we don't really apply least constrained value heuristic to our solution because the benefit is trivial. We basically select the empty grid and the grid where the letter inside is same as first letter of the word we choose. Therefore, We have eliminated few grids from each iteration.

For general constraints, we exactly follow the rule of sudoku, that is, each row, column, and 3x3 cell cannot contain duplicate letters. We won't discuss the details of our constraint checking implementation because it is trivial.

Output

Input 1:

```
V,7,0: MARVELING
V,1,1: OUTRAGED
H,0,1: CONFUSE
V,0,2: SEMINAR
H,0,0: LIGHTEN
H,1,2: UPWIND
V,3,3: NIMBLY
V,8,0: PYTHON
H,1,3: TUNDRA
V,2,4: FOLKS
V,5,5: HUMP
V,8,5: NECK
V,6,5: POUT
H,3,4: ICKY
V,4,5: SAVE
LIGHTENMP
CONFUSEAY
SUPWINDRT
ETUNDRAVH
MRFICKYEO
IAOMSHPLN
NGLBAUOIE
AEKLVMUNC
RDSYEPTGK
Number of node Expanded without leaves: 82

real    0m0.066s
user    0m0.052s
sys     0m0.011s
```

The word sequence can be seen at the top of this image, where V and H indicate vertical and horizontal direction. The number after that is, X,Y location and the word that to be filled. The middle part is the filled sudoku and number of node expand. The running time is also included.

Input 2:

The word sequence can be seen at the top of this image, where V and H indicate vertical and horizontal direction. The number after that is, X,Y location and the

word that to be filled. The middle part is the filled sudoku and number of node expand. The running time is also included. (Note that the sudoku has different version because of choice of implementation: we check vertical first)

Part 2: Game of Breakthrough

-Bangqi Wang

In this part, we will implement the AI agents to play the game of breakthrough against each other or against human. The AI agents has two different strategies, minimax search and alpha-beta search. For each strategy, there are also two different styles, offensive and defensive. This report will show the implementations of each strategy and each style, as well as, the implementation of heuristic functions.

Part 2.1: Minimax and Alpha-Beta Agents

Classes

This part includes 4 files and 3 classes: main file, game class, board class, and player class.

Name:	**Game**	**Board**	**Player**
Attributes:	Self.board Self.player1 Self.player2 Self.turn Self.finish Self.winner Self.step Self.node	Self.state Self.size_x Self.size_y Self.step	Self.strategy Self.style Self.depth Self.node
Functions:	Self.check_win() Self.solve()	Self.get_copy() Self.set_move() Self.empty_step() Self.get_next() Self.get_eval()	Self.move() Self.minimax() Self.alphabeta() Self.min_value() Self.max_value() Self.heuristic_defensive() Self.heuristic_offensive() Self.empty_node()

Game class represents the processes of the game. Each game class contains one board class and two player classes. The attribute self.turn represents the player in each, self.finish and self.winner represent the finally output, and the self.step and self.node represent the steps and nodes expanded. The function self.check_win() check the winner of the game and set attributes self.finish and self.winner. The self.solve() function runs the game for two players.

Board class represents the board of the game. Each board class contains one initial game board as self.state, the size of board as self.size_x and self.size_y, and the step moved as self.step. This class contains several functions. Self.get_copy() will deep copy the board to avoid synchronization issues. Self.set_move() changes the game board according to the move. Self.empty_step() clears

the step list. Self.get_next() returns the list of all possible game boards with one move step. Self.get_eval() evaluates the score of the game board.

Player class represents the player. Each player class contains the self.strategy (minimax or alphabeta) and self.style (offensive or defensive) of the player. The attribute self.depth represents the depth limitation and self.node is the cumulative number of nodes expanded. The function self.move return the best next move that calculated by different strategies and different styles.

Strategies Implementations

This MP have two different strategies: minimax search and alpha-beta search. Those two strategies use same basic algorithm, minimax tree. The difference between the strategies is the number of node expanded. Minimax search traverses the entire minimax tree, while the alpha-beta search traverses the minimax tree with pruning.

Minimax Search

The minimax tree returns the minimum or the maximum values of the children. I implement the minimax search without using tree node. Instead, I use list to contains all children. For each node, I will pass the game board, the depth, the order, and the worker. If the depth is equal to the maximum depth, the node is a leaf and it will return the pair of heuristic value and node, (h(n), node) according to the player style (offensive or defensive). Otherwise, the algorithm will find all possible next moves by board.get_next() and push all game boards into children list. For each element in the children list, recursively call minimax on all children until leaves. The order parameter will specify the min or max mode of minimax tree and the parameter worker will tell the turn of player. The order parameter can be 0 or 1 with 0 for min() and 1 for max(). The worker parameter can be 1 or 2 with 1 for player1 and 2 for player2.

```
def minimax(self, board, depth, order, worker):
    self.node += 1
    # if the node is leaf, return evaluation and node
    if depth is self.depth:
        if self.style is 0:
            return (self.heuristic_defensive(board, order, worker), board)
        else:
            return (self.heuristic_offensive(board, order, worker), board)

    # find all possible next step
    queue = []
    next_boards = board.get_next(worker)

    # explore the value of each possible step
    for each_board in next_boards:
        queue.append(self.minimax(each_board, depth+1, 1-order, 3-worker))
    queue = sorted(queue)

    # return max if player is self
    if order is 1:
        return queue[-1]
    # return min if player is enemy
    else:
        return queue[0]
```

figure 1 minimax search python code

I use the list to store the children and sort the list to get the minimum and the maximum value quickly without storing current maximum and minimum values during the process.

Alpha-beta Search

The alpha-beta search is the developed version of the minimax search with pruning. The core algorithm is the minimax tree search. However, the pruning can improve the performance by expand less nodes. In this MP, the alpha-beta search can handle deeper searching depth and increase the probability of win. I tried the alpha-beta search with depth equal to 5 and win the game with around 1,600,000 nodes expanded.

The implementation of the alpha-beta search is based on the pseudocode on lecture08. The alpha-beta search will call the min_value() or max_value() function according to the order of the node (0 - min, 1 - max) and find the value based on the worker parameter (1 - player1, 2 - player2).

The min_value() and max_value() function will recursively call each other until the leaf node. Then it returns the pair of heuristic and node, (h(n), node). The algorithm will break the recursion if the values of the rest of the children are meaningless.

Ordering

The total number of node expanded may decrease when we calculate the evaluation function from pieces that are far away to the pieces that are close to the base. The pieces that are far away are more likely to have higher evaluation than the pieces in the base. Therefore, calculating the pieces far away first might prune more branches when calculating the minimax tree.

```
def alphabeta(self, board, depth, order, worker):
    # return max if player is self
    if order is 1:
        return self.max_value(board, -np.inf, np.inf, depth, order, worker)
    # return min if player is enemy
    else:
        return self.min_value(board, -np.inf, np.inf, depth, order, worker)
```

figure 2 python code for alpha-beta search

```
def min_value(self, board, a, b, depth, order, worker):
    self.node += 1
    # return the leaf node
    if depth is self.depth:
        if self.style is 0:
            return (self.heuristic_defensive(board, order, worker), board)
        else:
            return (self.heuristic_offensive(board, order, worker), board)

    v = np.inf
    v_board = None
    # find all possible next steps
    queue = []
    next_boards = board.get_next(worker)
    # recursively call max_value on children
    for each_board in next_boards:
        each_value = self.max_value(each_board, a, b, depth+1, 1-order, 3-worker)
        if each_value[0] < v:
            v = each_value[0]
            v_board = each_value
        # break if need pruning
        if v <= a:
            return v_board
        b = min(b, v)
    return v_board
```

figure 3 python code for min_value()

```
def max_value(self, board, a, b, depth, order, worker):
    self.node += 1
    # return the leaf node
    if depth is self.depth:
        if self.style is 0:
            return (self.heuristic_defensive(board, order, worker), board)
        else:
            return (self.heuristic_offensive(board, order, worker), board)

    v = -np.inf
    v_board = None
    # find all possible next step
    queue = []
    next_boards = board.get_next(worker)
    # recursively call min_value on children
    for each_board in next_boards:
        each_value = self.min_value(each_board, a, b, depth+1, 1-order, 3-worker)
        if each_value[0] > v:
            v = each_value[0]
            v_board = each_value
        # break if need pruning
        if v >= b:
            return v_board
        a = max(a, v)
    return v_board
```

figure 4 python code for max_value()

Style Implementations

The agent can have different styles, including offensive and defensive. Offensive agent focuses on maximizing its own score, while the defensive agent concentrates on minimizing other's score. Generally, the two styles have the same evaluation function.

Offensive

The offensive agent more focused on moving forward and capture enemy pieces. In other word, it more focused on its own distance moved and the number of other's pieces. The evaluation function is based on the distance to the furthest row and the number of enemies captured. The offensive agent will choose the step that will maximize evaluation function. For instance, the player1 with offensive style will calculate the steps moved by worker 1 and the number of worker 2 captured.

Defensive

The defensive agent more focused on preventing the enemy from moving into its territory or capture its pieces. In other word, it more focused on other's distance and its number of pieces. As we know, the evaluation of one agent is based on the the distance to the furthest row and the number of enemies captured. Preventing enemy from moving or capture will lower its evaluation. Therefore, the defensive agent is trying to minimize the evaluation value of the enemy. In this case, the algorithm will return the minimum evaluation of other by return the maximum negative evaluation. For instance, the player1 with defensive style will calculate the evaluation of worker 2 and return the maximum negative evaluation.

```
def heuristic_offensive(self, board, order, worker):
    # if worker is self, evaluate worker
    if order is 1:
        return board.get_eval(worker)
    # if worker is other, evaluate on self
    else:
        return board.get_eval(3-worker)
```

```
def heuristic_defensive(self, board, order, worker):
    # if worker is self, evaluate other
    if order is 1:
        return -1*board.get_eval(3-worker)
    # if worker is other, evaluate worker
    else:
        return -1*board.get_eval(worker)
```

figure 5 python code for heuristic

Evaluation Function

The evaluation function determinate the evaluation value of each step. The step with higher evaluation value has higher priority. The main purpose of this part is to compare between evaluation functions rather than the algorithm. Therefore, I will use only one algorithm in this part with same depth.

1st Vrsion Evaluation: Eval() = step + weight*capture

This evaluation simply add the distance moved and the number of enemy captured. This evaluation function has few difference with randomly moving, because player must move one pieces forward in each round and every moves have same additional weight as 1. The only differences are the weight of captured enemy and the weight of losing pieces. Step with more capture will have more weight. When losing piece further from the base, the loss is larger. In this case, this evaluation function is more likely to capturing enemy and avoid losing the pieces far away from the base.

	Offensive vs. Defensive	**Defensive vs. Offensive**
Minimax	[[0 0 2 1 0 0 0 0] [0 0 0 0 0 0 1 0] [0 0 0 1 0 1 0 0] [0 0 0 0 0 0 0 0] [0 2 0 0 2 0 2 2] [0 0 0 0 0 0 0 2] [0 2 2 2 2 0 2 2] [2 0 0 0 0 0 0 0]] Winner: Player_2 Strategy: MiniMax vs. MiniMax Style: Offensive vs. Defensive Node Expanded: 475846 vs. 486752 Node Exp/Move: 14870 vs. 15211 Avg Time/Move: 2.1918 vs. 2.2574 Oppo Captured: 12 vs. 3 moves to Win : 64	[[0 0 0 0 1 1 0 1] [0 1 0 1 0 1 0 1] [0 0 0 1 0 1 1 1] [0 0 0 0 0 0 0 2] [0 0 0 2 0 0 2 0] [0 2 1 1 0 0 2 0] [0 0 0 0 1 0 0 2] [2 0 0 0 0 1 0 0]] Winner: Player_1 Strategy: MiniMax vs. MiniMax Style: Defensive vs. Offensive Node Expanded: 475190 vs. 411392 Node Exp/Move: 15839 vs. 14185 Avg Time/Move: 2.4911 vs. 2.1081 Oppo Captured: 1 vs. 9 moves to Win : 59
vs.	Offensive vs. Offensive	Defensive vs. Defensive
Minimax	[[2 0 0 0 0 0 0 0] [0 0 2 0 0 0 0 0] [1 0 2 0 1 2 1 0] [0 1 0 0 0 1 2 1] [1 2 1 2 1 0 0 0] [2 0 2 0 1 0 0 2] [2 2 0 1 2 0 0 0] [0 0 0 0 0 0 0 0]] Winner: Player_2 Strategy: MiniMax vs. MiniMax Style: Offensive vs. Offensive Node Expanded: 550378 vs. 555742 Node Exp/Move: 12230 vs. 12349 Avg Time/Move: 1.8736 vs. 1.8258 Oppo Captured: 5 vs. 3 moves to Win : 90	[[2 0 0 0 0 0 0 0] [0 0 0 0 0 0 1 0] [0 0 0 0 0 0 0 0] [0 0 1 0 0 0 2 0] [0 0 2 0 0 0 0 0] [2 0 1 0 0 0 0 0] [2 0 2 1 0 0 0 0] [0 0 0 0 0 0 0 0]] Winner: Player_2 Strategy: MiniMax vs. MiniMax Style: Defensive vs. Defensive Node Expanded: 558699 vs. 583894 Node Exp/Move: 12145 vs. 12693 Avg Time/Move: 1.744 vs. 1.8553 Oppo Captured: 12 vs. 10 moves to Win : 92

The alpha-beta search has little difference with minimax search except the node expanded. I only run minimax search for this strategy and figured that the algorithm for offensive and defensive is not desirable. If one player with offensive style compete against the other player with defensive style. The offensive player will win the game and capture more opponent, which is reasonable. However, if two offensive players compete against each other, the total steps will raise and the opponents captured will remain in low level. The agent is trying to move further and avoid be

captured, which is not consistent with the requirement for offensive agent. Instead, the defensive agent is more likely to capture the opponents that are closed to the base. Therefore, the offensive agent focused on capturing the opponents and avoiding losing the piece far away from base, while the defensive agent focus on capturing the pieces that are closed to the base.

```
def get_eval(self, worker):
    self.distance = 0
    self.capture = 16
    if worker is 1:
        for j in range(self.size_y-1):
            for i in range(self.size_x):
                # weight for each piece
                if self.state[j][i] == worker:
                    self.distance += j
                # weight for each capturing
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
    if worker is 2:
        for j in range(self.size_y):
            for i in range(self.size_x):
                # weight for each piece
                if self.state[j][i] == worker:
                    self.distance += (7-j)
                # weight for each capturing
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
    return self.distance + self.capture
```

figure 6 python code for evaluation function

Analysis

I still believe that the offensive and defensive evaluation function are actually the opposite. In other word, I consider the situation as zero sum game. The offensive agent focused on its own evaluation value, while the defensive agent focused on the evaluation value of opponent. I simply used the minimax search on the negative evaluation value of the other player to find its own strategy. The minimum value of other player is the maximum value of itself. Therefore, I considered this game as zero sum game.

However, the offensive and defensive evaluation function focused on wrong part. In this case, in order to emphasis the desire to move forward, we should give different weight for moving forward. For example, we can square the weight of each step. Piece in row j will have j*j weight.

2nd Version

In this version, I give more weight to capture opponent and moving forward. I square the step as step weight and give the highest weight for capturing opponent.

Evaluation Function: Eval() = step^2 + weight*capture

In this evaluation function, the pieces far away from base are more likely to move forward. For example, for player 1, if the piece moves from row 2 to row 3, the additional evaluation value is 3*3 – 2*2 = 5. If the piece moves from row 5 to row 6, the additional evaluation value is 6*6-5*5 = 11. Therefore, the pieces far away from base are move likely to move further. The weight for capture is 100 and the weight for win is 1600. Therefore, the weight of capturing all opponent is the same as reaching the furthest row. In this case, the win has the highest priority. Capturing opponent has the second highest priority, and the piece far away from base moving forward has the third highest priority. I used alpha-beta search in order to save runtime.

```python
def get_eval(self, worker):
    self.distance = 0
    self.capture = 16
    if worker is 1:
        for j in range(self.size_y-1):
            for i in range(self.size_x):
                # weight for each piece
                if self.state[j][i] == worker:
                    self.distance += j*j
                    if j == self.size_y - 1:
                        self.distance += 1600
                # weight for each capturing
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
    if worker is 2:
        for j in range(self.size_y):
            for i in range(self.size_x):
                # weight for each piece
                if self.state[j][i] == worker:
                    self.distance += (7-j)*(7-j)
                    if j == 0:
                        self.distance += 1600
                # weight for each capturing
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
    return self.distance + 100*self.capture
```

figure 7　python code for evaluation function

	Offensive vs. Defensive	**Defensive vs. Offensive**
Alpha-beta	[[1 2 1 0 1 1 1 1] [0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 1] [0 0 0 0 1 0 1 1] [0 0 0 0 0 0 0 2] [0 0 0 0 0 0 2 2] [2 2 0 2 2 0 0 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Defensive Node Expanded: 27760 vs. 28357 Node Exp/Move: 1110 vs. 1134 Avg Time/Move: 0.2434 vs. 0.2483 Oppo Captured: 7 vs. 5 moves to Win : 50	[[1 2 0 1 0 0 1 1] [0 1 0 1 0 0 0 0] [0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 1 2 2 0 0] [0 0 0 0 0 0 0 2] [0 0 0 0 0 2 0 0] [2 2 2 2 0 2 0 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Offensive Node Expanded: 41646 vs. 37845 Node Exp/Move: 1487 vs. 1351 Avg Time/Move: 0.3378 vs. 0.3423 Oppo Captured: 5 vs. 8 moves to Win : 56
vs.	**Offensive vs. Offensive**	**Defensive vs. Defensive**
Alpha-beta	[[1 1 1 1 1 1 1 1] [0 2 0 0 0 0 0 0] [0 0 0 0 0 1 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 2 2 2 0 2 0] [2 2 2 2 0 2 1 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Offensive Node Expanded: 20199 vs. 16930 Node Exp/Move: 1262 vs. 1128 Avg Time/Move: 0.3832 vs. 0.2878 Oppo Captured: 5 vs. 6 moves to Win : 31	[[0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 1 1 0 0 0 0 0] [0 0 1 0 0 0 2 1] [1 0 0 0 1 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 1 0] [0 0 0 0 0 0 0 1]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Defensive Node Expanded: 145066 vs. 89527 Node Exp/Move: 2901 vs. 1827 Avg Time/Move: 0.5893 vs. 0.4482 Oppo Captured: 15 vs. 8 moves to Win : 99

Analysis

This evaluation is more reasonable than the previous one. According to the table above, two offensive agents can finish the game by 31 steps. One offensive agent with one defensive agent can finish game between 50 to 60 steps. Two defensive agent can finish game about 100 steps. We can see that the offensive agent trying to finish game with less steps, while the defensive agent will finish the game with more steps. What's more, the offensive agent captured more opponent when competed against the defensive agent.

However, we can see that the game between two defensive agents actually captured more opponent than the game between two offensive agents. I believe that the defensive agent tried to avoid being captured but when game get longer, capture is not avoidable. Therefore, we shouldn't compare the game between two offensive agents and two defensive agents and made the conclusion according to the data we got from those two games because the game lengths have significant difference. Indeed, we would better to compare the game between one offensive agent and one defensive agent. We can conclude that the offensive agent can capture more opponent than defensive agent.

Game Running:

According to the analysis above, the second evaluation function is more desirable and effective. Therefore, I will run the algorithm on the second version of evaluation function.

Minimax Depth=3

vs.

Minimax Depth = 3

Offensive vs. Defensive

```
[[1 1 1 1 2 1 0 1]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [2 2 0 0 0 0 0 0]
 [0 0 0 2 2 0 2 0]
 [0 2 2 0 0 0 0 0]
 [0 2 0 2 2 2 2 0]]
Winner: Player_2
Strategy: MiniMax vs. MiniMax
Style: Offensive vs. Defensive
Node Expanded: 440329 vs. 459010
Node Exp/Move: 17613 vs. 18360
Avg Time/Move: 2.5134 vs. 2.6833
Oppo Captured: 3 vs. 9
moves to Win : 50
```

Defensive vs. Offensive

```
[[1 0 0 0 0 1 2 0]
 [0 0 0 2 0 0 1 0]
 [1 0 0 0 0 1 0 0]
 [1 1 1 1 0 0 1 0]
 [2 0 0 0 0 0 0 1]
 [2 0 0 0 2 0 0 2]
 [0 0 0 0 0 0 0 0]
 [2 2 2 2 0 0 2 2]]
Winner: Player_2
Strategy: MiniMax vs. MiniMax
Style: Defensive vs. Offensive
Node Expanded: 510620 vs. 502469
Node Exp/Move: 17020 vs. 16748
Avg Time/Move: 2.7482 vs. 2.5621
Oppo Captured: 4 vs. 5
moves to Win : 60
```

Offensive vs. Offensive

```
[[1 1 1 1 1 0 1 1]
 [1 1 1 1 1 1 2 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [2 2 2 2 0 0 0 0]
 [2 2 2 2 0 2 1 2]]
Winner: Player_1
Strategy: MiniMax vs. MiniMax
Style: Offensive vs. Offensive
Node Expanded: 163638 vs. 148415
Node Exp/Move: 14876 vs. 14841
Avg Time/Move: 2.2143 vs. 2.1567
Oppo Captured: 5 vs. 2
moves to Win : 21
```

Defensive vs. Defensive

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 2 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 2 1 0 1 2 1]
 [0 1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]]
Winner: Player_1
Strategy: MiniMax vs. MiniMax
Style: Defensive vs. Defensive
Node Expanded: 559660 vs. 558321
Node Exp/Move: 11193 vs. 11394
Avg Time/Move: 1.715 vs. 1.7161
Oppo Captured: 13 vs. 8
moves to Win : 99
```

Alpha-beta Depth = 4

vs.

Alpha-beta Depth = 4

Offensive vs. Defensive

```
[[2 1 0 1 1 1 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 0 0 0 0 0]
 [0 1 0 1 0 0 0 0]
 [0 1 0 2 0 0 0 0]
 [0 2 0 0 0 0 0 0]
 [0 2 0 2 0 2 2 2]
 [0 0 0 2 2 2 2 2]]
Winner: Player_2
Strategy: AlphaBeta vs. AlphaBeta
Style: Offensive vs. Defensive
Depth: 4 vs. 4
Node Expanded: 148764 vs. 153419
Node Exp/Move: 8264 vs. 8523
Avg Time/Move: 3.3557 vs. 4.6097
Oppo Captured: 3 vs. 3
moves to Win : 36
```

Defensive vs. Offensive

```
[[2 1 0 1 1 1 1 1]
 [0 0 0 0 0 1 1 1]
 [0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 1 0 0 1 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 2 2 2 2]
 [0 2 2 2 2 2 2 2]]
Winner: Player_2
Strategy: AlphaBeta vs. AlphaBeta
Style: Defensive vs. Offensive
Depth: 4 vs. 4
Node Expanded: 109984 vs. 118415
Node Exp/Move: 5788 vs. 6232
Avg Time/Move: 3.1924 vs. 3.1124
Oppo Captured: 4 vs. 3
moves to Win : 38
```

Offensive vs. Offensive

```
[[2 1 0 1 1 1 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 2 2 0 2 2 2 2]
 [0 0 0 0 2 2 2 2]]
Winner: Player_2
Strategy: AlphaBeta vs. AlphaBeta
Style: Offensive vs. Offensive
Depth: 4 vs. 4
Node Expanded: 64795 vs. 65086
Node Exp/Move: 3239 vs. 3254
Avg Time/Move: 1.7297 vs. 1.7808
Oppo Captured: 5 vs. 5
moves to Win : 40
```

Defensive vs. Defensive

```
[[1 1 1 0 1 0 1 1]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 1 0 1 0 0 0 0]
 [0 1 0 1 0 2 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [2 2 2 2 2 1 2 0]]
Winner: Player_1
Strategy: AlphaBeta vs. AlphaBeta
Style: Defensive vs. Defensive
Depth: 4 vs. 4
Node Expanded: 373555 vs. 336290
Node Exp/Move: 12881 vs. 12010
Avg Time/Move: 6.6796 vs. 6.4026
Oppo Captured: 9 vs. 4
moves to Win : 57
```

Minimax Depth = 3

vs.

Alpha-beta Depth = 4

Offensive vs. Defensive

```
[[1 0 1 2 0 0 0 0]
 [1 0 0 1 0 0 1 1]
 [0 1 0 0 0 1 0 1]
 [0 2 1 1 0 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 2 2]
 [2 2 2 2 2 2 2 2]]
Winner: Player_2
Strategy: MiniMax vs. AlphaBeta
Style: Offensive vs. Defensive
Depth: 3 vs. 4
Node Expanded: 432658 vs. 136880
Node Exp/Move: 19666 vs. 6221
Avg Time/Move: 2.5345 vs. 2.7935
Oppo Captured: 4 vs. 4
moves to Win : 44
```

Defensive vs. Offensive

```
[[2 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 2 1 2]
 [0 2 0 0 0 0 0 0]
 [0 2 0 0 0 0 0 0]
 [0 1 0 0 2 0 0 0]
 [0 0 0 0 2 0 2 2]]
Winner: Player_2
Strategy: MiniMax vs. AlphaBeta
Style: Defensive vs. Offensive
Depth: 3 vs. 4
Node Expanded: 625441 vs. 267675
Node Exp/Move: 14545 vs. 6225
Avg Time/Move: 1.8687 vs. 2.5627
Oppo Captured: 7 vs. 14
moves to Win : 86
```

Offensive vs. Offensive

```
[[2 0 0 1 0 0 0 0]
 [0 1 1 1 1 1 1 1]
 [0 1 0 0 0 0 1 1]
 [0 1 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0 0]
 [0 0 0 0 0 2 2 2]
 [2 2 2 2 2 2 2 2]]
Winner: Player_2
Strategy: MiniMax vs. AlphaBeta
Style: Offensive vs. Offensive
Depth: 3 vs. 4
Node Expanded: 304307 vs. 48196
Node Exp/Move: 16905 vs. 2677
Avg Time/Move: 2.17 vs. 1.5266
Oppo Captured: 3 vs. 3
moves to Win : 36
```

Defensive vs. Defensive

```
[[2 0 0 0 1 0 1 0]
 [0 2 2 1 1 0 2 1]
 [0 1 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 2 0 0 2 1 0]
 [0 0 0 0 0 0 0 2]
 [2 2 2 2 2 2 2 2]]
Winner: Player_2
Strategy: MiniMax vs. AlphaBeta
Style: Defensive vs. Defensive
Depth: 3 vs. 4
Node Expanded: 626097 vs. 278437
Node Exp/Move: 24080 vs. 10709
Avg Time/Move: 3.309 vs. 6.0154
Oppo Captured: 1 vs. 7
moves to Win : 52
```

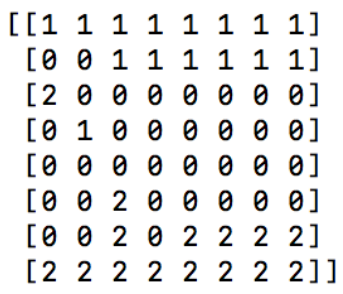
	Offensive vs. Defensive	Defensive vs. Offensive
Alpha-beta Depth = 4 ***vs.*** ***Minimax Depth = 3***	[[1 1 1 1 1 1 1 1] [0 0 0 0 0 1 1 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 2 0 0 0 2 2] [0 0 0 0 2 2 0 0] [0 0 2 2 2 2 0 2] [1 0 0 0 0 0 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Offensive vs. Defensive Depth: 4 vs. 3 Node Expanded: 83519 vs. 352083 Node Exp/Move: 3631 vs. 16003 Avg Time/Move: 2.062 vs. 2.1979 Oppo Captured: 6 vs. 4 moves to Win : 45	[[0 1 1 1 1 1 1 1] [0 0 0 0 0 0 0 0] [1 0 2 0 1 1 1 0] [0 0 0 0 0 0 0 0] [0 2 0 0 2 0 0 2] [2 0 2 2 0 2 2 2] [0 0 0 0 0 0 0 0] [1 2 0 0 0 2 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Defensive vs. Offensive Depth: 4 vs. 3 Node Expanded: 302336 vs. 561189 Node Exp/Move: 11197 vs. 21584 Avg Time/Move: 4.6252 vs. 2.7711 Oppo Captured: 4 vs. 4 moves to Win : 53
	Offensive vs. Offensive	**Defensive vs. Defensive**
	[[1 1 1 1 1 1 1 1] [0 0 0 0 1 1 1 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 0 0 2 0 0 0] [0 0 2 0 0 0 2 2] [2 0 2 0 2 0 2 2] [1 2 0 0 2 0 2 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Offensive vs. Offensive Depth: 4 vs. 3 Node Expanded: 42228 vs. 260945 Node Exp/Move: 2484 vs. 16309 Avg Time/Move: 1.383 vs. 2.0523 Oppo Captured: 3 vs. 3 moves to Win : 33	[[0 1 1 1 1 1 1 1] [0 0 0 0 0 0 0 0] [0 0 2 0 0 0 0 0] [0 0 0 0 2 0 2 0] [0 0 0 0 2 2 0 0] [0 1 2 0 0 1 1 0] [0 2 0 0 0 0 0 0] [0 0 1 0 0 0 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Defensive vs. Defensive Depth: 4 vs. 3 Node Expanded: 617442 vs. 600827 Node Exp/Move: 16687 vs. 16689 Avg Time/Move: 5.095 vs. 2.1341 Oppo Captured: 9 vs. 5 moves to Win : 73

Trends and Conclusions:

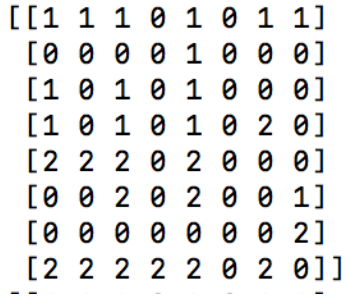
According to the table above, if the two players have different styles, the player going second are more likely to win the game because the offensive and defensive strategies contradict each other. The player going second can always counter the previous player. If the two players have same styles, the player going first are more likely to win the game. In this case, each player is only focus on themselves or only on others. They focused on different people and they didn't counter each other. In this case, the player going first have more advantage. I believe the reason is that I designed the offensive and defensive evaluation functions as zero sum game and the evaluation of defensive agent is the negative evaluation of opponents.

Offensive vs. Defensive

The game with two offensive agents will finish faster, and the game with two defensive agents will last much more longer. The length of the game with one defensive and one offensive agent are in between. The offensive agents are more likely to capture the opponent in most of the game. The offensive agents are more likely to moving forward one piece by on piece. However, the defensive agent is more likely to push all pieces alternatively and form a wall.



Offensive



Defensive

Depth

The agent with more depth are more likely to win the game. If both agents have the same depth, they have same chance to win the game if ignoring the difference of strategies and styles. According to the table above, when two player have different searching strategy, the player with alpha-beta search will always win the game because alpha-beta search can handle deeper searching than minimax search can.

Alpha-beta Pruning Ordering

The total number of node expanded may decrease when we calculate the evaluation function from pieces that are far away to the pieces that are close to the base. The pieces that are far away are more likely to have higher evaluation than the pieces in the base. Therefore, calculating the pieces far away first might prune more branches when calculating the minimax tree.

3rd Version: combination of two evaluation function

In this part, I will combine the two evaluation function and give them different weight.

Evaluation function: w_1 * (self_step + captured) + w_2 * (opponent_step + lost)

The offensive agent will choose the step will make their evaluation larger and the defensive agent will choose the step that will make opponent's evaluation smaller.

```
def get_eval_comb(self, worker):
    self.distance = 0
    self.danger = 0
    self.capture = 16
    self.lost = 16
    if worker is 1:
        for j in range(self.size_y):
            for i in range(self.size_x):
                if self.state[j][i] == worker:
                    self.distance += j
                    self.lost -= 1
                    if j == self.size_y - 1:
                        self.distance += 800
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
                    self.danger += (7-j)
    if worker is 2:
        for j in range(self.size_y):
            for i in range(self.size_x):
                if self.state[j][i] == worker:
                    self.distance += (7-j)
                    self.lost -= 1
                    if j == 0:
                        self.distance += 800
                elif self.state[j][i] == 3-worker:
                    self.capture -= 1
                    self.danger += j
    return (self.distance + self.capture) - (self.danger + self.lost)
```

	Offensive vs. Defensive	Defensive vs. Offensive
Minimax Depth = 3	[[1 0 0 0 0 0 2 0] [2 0 0 0 0 0 0 0] [0 2 0 2 0 0 0 0] [0 0 2 0 1 0 0 0] [0 0 1 0 0 0 1 0] [0 1 0 0 0 0 0 1] [0 0 2 0 0 0 0 0] [2 2 0 0 2 0 2 2]] Winner: Player_2 Strategy: MiniMax vs. MiniMax Style: Offensive vs. Defensive Depth: 3 vs. 3 Node Expanded: 459082 vs. 508654 Node Exp/Move: 11477 vs. 12716 Avg Time/Move: 2.8969 vs. 3.1354 Oppo Captured: 5 vs. 10 moves to Win : 80	[[0 0 0 1 0 1 0 0] [1 1 1 0 0 0 1 0] [0 1 1 0 1 0 1 0] [0 0 1 0 0 0 2 0] [0 2 0 0 0 2 0 0] [0 2 2 0 2 0 0 0] [0 0 2 0 0 0 0 0] [2 0 2 0 0 1 0 0]] Winner: Player_1 Strategy: MiniMax vs. MiniMax Style: Defensive vs. Offensive Depth: 3 vs. 3 Node Expanded: 432667 vs. 434101 Node Exp/Move: 14919 vs. 15503 Avg Time/Move: 3.6501 vs. 3.7986 Oppo Captured: 7 vs. 4 moves to Win : 57
vs.	**Offensive vs. Offensive**	**Defensive vs. Defensive**
Minimax Depth = 3	[[1 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 1 1 0 1 1 0 0] [0 0 0 1 0 1 0 0] [0 2 2 0 0 0 0 0] [0 0 2 2 0 2 2 0] [2 0 0 0 0 0 0 0] [0 0 0 0 0 1 0 0]] Winner: Player_1 Strategy: MiniMax vs. MiniMax Style: Offensive vs. Offensive Depth: 3 vs. 3 Node Expanded: 480306 vs. 423347 Node Exp/Move: 12315 vs. 11140 Avg Time/Move: 3.1445 vs. 2.7291 Oppo Captured: 9 vs. 7 moves to Win : 77	[[0 0 0 0 2 0 0 0] [0 0 1 0 0 0 0 0] [1 1 0 1 0 2 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 2 0 0 0 0 0] [0 0 2 2 1 0 0 0] [2 2 0 0 0 0 0 0]] Winner: Player_2 Strategy: MiniMax vs. MiniMax Style: Defensive vs. Defensive Depth: 3 vs. 3 Node Expanded: 469761 vs. 503013 Node Exp/Move: 13048 vs. 13972 Avg Time/Move: 3.1355 vs. 3.4589 Oppo Captured: 9 vs. 11 moves to Win : 72

	Offensive vs. Defensive	Defensive vs. Offensive
Alpha-beta Depth = 4	[[2 0 0 0 1 1 1 1] [0 0 1 0 0 0 0 0] [1 0 1 0 0 0 0 0] [0 1 0 1 2 1 2 0] [1 2 1 2 1 2 1 0] [2 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Defensive Depth: 4 vs. 4 Node Expanded: 402683 vs. 382292 Node Exp/Move: 14914 vs. 14158 Avg Time/Move: 5.0695 vs. 4.9512 Oppo Captured: 2 vs. 2 moves to Win : 54	[[0 0 1 0 1 1 1 1] [1 0 1 0 0 0 0 0] [2 0 0 0 0 0 0 0] [1 2 1 2 1 2 2 0] [0 0 2 1 2 1 0 1] [2 2 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [1 0 0 2 2 2 2 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Offensive Depth: 4 vs. 4 Node Expanded: 535234 vs. 480791 Node Exp/Move: 17841 vs. 16579 Avg Time/Move: 6.3712 vs. 5.9772 Oppo Captured: 2 vs. 2 moves to Win : 59
vs.	**Offensive vs. Offensive**	**Defensive vs. Defensive**
Alpha-beta Depth = 4	[[0 1 1 1 1 1 1 1] [0 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0] [2 0 2 2 2 2 2 0] [0 2 1 1 1 1 0 0] [0 0 0 0 0 0 0 0] [0 2 0 0 0 0 0 0] [1 0 2 2 2 2 2 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Offensive Depth: 4 vs. 4 Node Expanded: 619034 vs. 592218 Node Exp/Move: 22108 vs. 21934 Avg Time/Move: 6.8266 vs. 6.7798 Oppo Captured: 2 vs. 3 moves to Win : 55	[[2 0 0 1 1 1 1 1] [0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [2 0 0 1 2 1 2 0] [0 2 0 2 1 2 1 0] [1 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Defensive Depth: 4 vs. 4 Node Expanded: 369428 vs. 420396 Node Exp/Move: 12738 vs. 14496 Avg Time/Move: 4.5473 vs. 5.3953 Oppo Captured: 2 vs. 4 moves to Win : 58

	Offensive vs. Defensive	Defensive vs. Offensive
Minimax Depth = 3	[[2 0 0 1 0 0 0 0] [0 0 0 0 1 1 1 1] [0 0 1 0 0 0 1 1] [0 1 0 0 1 0 0 0] [0 2 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 2 2] [2 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: MiniMax vs. AlphaBeta Style: Offensive vs. Defensive Depth: 3 vs. 4 Node Expanded: 379586 vs. 159974 Node Exp/Move: 18075 vs. 7617 Avg Time/Move: 4.668 vs. 6.3362 Oppo Captured: 3 vs. 6 moves to Win : 42	[[0 0 0 2 0 0 0 0] [1 0 1 1 0 0 1 1] [1 1 0 0 0 0 0 1] [0 0 0 0 1 1 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 2 2] [2 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: MiniMax vs. AlphaBeta Style: Defensive vs. Offensive Depth: 3 vs. 4 Node Expanded: 414825 vs. 213368 Node Exp/Move: 17284 vs. 8890 Avg Time/Move: 4.0401 vs. 6.9108 Oppo Captured: 5 vs. 6 moves to Win : 48
vs.	**Offensive vs. Offensive**	**Defensive vs. Defensive**
Alpha-beta Depth = 4	[[1 2 1 0 1 0 1 0] [0 0 1 1 0 1 0 1] [0 0 1 0 0 1 0 1] [0 1 1 0 0 1 0 0] [0 0 2 0 0 0 0 0] [0 0 0 2 0 0 0 0] [0 0 0 0 0 2 2 2] [2 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: MiniMax vs. AlphaBeta Style: Offensive vs. Offensive Depth: 3 vs. 4 Node Expanded: 312732 vs. 87547 Node Exp/Move: 19545 vs. 5471 Avg Time/Move: 5.7285 vs. 5.8886 Oppo Captured: 2 vs. 2 moves to Win : 32	[[0 2 0 0 0 0 0 0] [0 1 0 2 0 0 0 1] [1 1 0 2 0 2 0 0] [0 1 0 0 0 0 0 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 2 0 0] [0 0 0 0 0 0 0 0] [2 2 2 2 2 2 2 2]] Winner: Player_2 Strategy: MiniMax vs. AlphaBeta Style: Defensive vs. Defensive Depth: 3 vs. 4 Node Expanded: 563082 vs. 333473 Node Exp/Move: 18163 vs. 10757 Avg Time/Move: 3.8673 vs. 7.0933 Oppo Captured: 3 vs. 10 moves to Win : 62

	Offensive vs. Defensive	Defensive vs. Offensive
Alpha-beta Depth = 4 ***vs.*** ***Minimax Depth = 3***	[[1 1 1 1 1 1 1 1] [0 0 0 1 1 1 1 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 2 0 0 0 0] [0 0 0 0 2 0 2 2] [0 1 0 2 2 2 2 2] [1 0 0 2 2 2 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Offensive vs. Defensive Depth: 4 vs. 3 Node Expanded: 69225 vs. 204888 Node Exp/Move: 4615 vs. 14634 Avg Time/Move: 3.8886 vs. 3.5464 Oppo Captured: 4 vs. 1 moves to Win : 29	[[1 1 1 1 1 1 1 1] [0 0 0 1 1 1 1 1] [0 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 2 2 2 0 2 2] [0 2 0 0 2 2 2 2] [2 1 0 2 2 2 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Defensive vs. Offensive Depth: 4 vs. 3 Node Expanded: 60148 vs. 177999 Node Exp/Move: 5012 vs. 16181 Avg Time/Move: 4.215 vs. 4.0581 Oppo Captured: 1 vs. 1 moves to Win : 23
	Offensive vs. Offensive	**Defensive vs. Defensive**
	[[1 1 1 1 1 1 1 1] [0 0 0 1 1 1 1 1] [0 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 0 2 2 2 2 2] [0 2 0 2 2 2 0 2] [2 1 0 2 2 2 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Offensive vs. Offensive Depth: 4 vs. 3 Node Expanded: 59869 vs. 177999 Node Exp/Move: 4989 vs. 16181 Avg Time/Move: 4.258 vs. 4.1841 Oppo Captured: 1 vs. 1 moves to Win : 23	[[1 1 1 1 1 1 1 1] [0 0 0 1 1 1 1 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 2] [0 0 2 2 2 0 0 2] [0 2 1 0 2 2 2 2] [2 1 0 0 2 2 0 0]] Winner: Player_1 Strategy: AlphaBeta vs. MiniMax Style: Defensive vs. Defensive Depth: 4 vs. 3 Node Expanded: 69357 vs. 211420 Node Exp/Move: 4623 vs. 15101 Avg Time/Move: 3.8173 vs. 3.6104 Oppo Captured: 3 vs. 1 moves to Win : 29

Trends and Conclusions:

The defensive agent is more likely to winning the game and the alpha-beta search always beat the minimax search because it has deeper depth.

Offensive vs. Defensive

The offensive agent is more likely to pushing pieces one by one and waiting in a safe area before start final attack. The defensive agent is more likely to push together and to find the opportunity to cross the defenses. The agent can be more offensive if we increase the weight of the capturing and the distance, i.e. w1. The agent can be more defensive if we increase the weight of pieces lost and opponent's distance, i.e. w2.

```
[[0 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [2 0 2 2 2 2 2 0]
 [1 1 1 1 1 1 0 0]
 [0 0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0 0]
 [0 2 2 2 2 2 2 2]]
```

Offensive

```
[[0 0 0 1 1 1 1 1]
 [1 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]
 [2 1 2 1 2 1 2 0]
 [1 2 1 2 1 2 1 0]
 [2 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 2 2 2 2 2 2 2]]
```

Defensive

Four Credit

Rectangular Board

In this part, the number of pieces will be less and the distance to win will be longer. In this case, I will increase the weight of each pieces and the weight of distance. Therefore, the agent will avoid its pieces being captured and prefer to moving forward. The deepest depth that can solve the problem in a reasonable time is depth 5.

	Offensive vs. Defensive	**Defensive vs. Offensive**
Alpha-beta Depth = 5	[[0 0 0 0 1] [0 1 1 1 0] [0 0 0 0 0] [2 2 2 0 0] [0 0 0 0 0] [0 0 0 0 0] [2 1 0 0 0] [0 0 0 0 0] [0 0 0 0 0] [1 0 2 2 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Defensive Depth: 5 vs. 5 Node Expanded: 810257 vs. 1028913 Node Exp/Move: 25320 vs. 33190 Avg Time/Move: 12.5387 vs. 14.3033 Oppo Captured: 3 vs. 4 moves to Win : 63	[[2 0 0 1 1] [0 2 1 0 0] [1 0 0 0 0] [0 0 0 0 0] [2 0 0 0 0] [0 0 0 0 0] [2 1 0 0 0] [0 0 0 0 0] [2 2 0 0 0] [0 0 0 2 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Offensive Depth: 5 vs. 5 Node Expanded: 736146 vs. 785807 Node Exp/Move: 23746 vs. 25348 Avg Time/Move: 11.4473 vs. 12.2848 Oppo Captured: 2 vs. 5 moves to Win : 62
vs.	**Offensive vs. Offensive**	**Defensive vs. Defensive**
Alpha-beta Depth = 5	[[0 0 0 1 1] [0 1 1 0 0] [1 0 0 0 0] [2 2 2 0 0] [2 0 0 0 0] [0 0 0 0 0] [1 1 0 0 0] [0 0 0 0 0] [0 2 0 0 0] [1 0 2 2 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Offensive Depth: 5 vs. 5 Node Expanded: 741825 vs. 913489 Node Exp/Move: 25580 vs. 32624 Avg Time/Move: 12.6358 vs. 15.1427 Oppo Captured: 2 vs. 2 moves to Win : 57	[[2 0 0 0 1] [0 2 1 0 0] [0 0 2 0 0] [0 0 0 0 0] [0 0 0 0 0] [0 0 0 0 0] [2 0 0 0 0] [2 0 0 0 0] [2 0 2 0 0] [0 0 0 0 2]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Defensive Depth: 5 vs. 5 Node Expanded: 743037 vs. 893398 Node Exp/Move: 21229 vs. 25525 Avg Time/Move: 6.2854 vs. 6.5944 Oppo Captured: 2 vs. 8 moves to Win : 70

Extra Credit

AI vs. Human

Please run start.py and compete against AI.

```
else:
    print('\nYour Turn:\n')
    start = time()
    step = self.player2.move(self.board, 2)
    s_x = int(raw_input("Please enter start row [0,8): \n"))
    s_y = int(raw_input("Please enter start column [0,8): \n"))
    if self.board.state[s_x][s_y] != 2:
        print('Wrong Move! Please select your piece.')
        continue
    direction = int(raw_input("Please enter direction [0,3): \n"))
    e_y = 0
    e_x = 0
    if direction == 0:
        e_x = s_x - 1
        e_y = e_y - 1
        if e_y < 0:
            print('Wrong Move! Cannot move left.')
            continue
    elif direction == 1:
        e_x = s_x - 1
        e_y = s_y
    elif direction == 2:
        e_x = s_x - 1
        e_y = s_y + 1
        if e_y > 7:
            print('Wrong Move! Cannot move right.')
            continue
    else:
        print('Wrong Move! Please enter number 0~2.')
    step = ((s_x,s_y),(e_x,e_y))
    if self.board.state[e_x][e_y] == 2:
        print('Wrong Move! Cannot capture your piece.')
        continue
    step = ((s_x,s_y),(e_x,e_y))
    print step
    end = time()
    self.time2 += end - start
    self.board.set_move(step)
    self.board.empty_step()
    self.step2 += 1
    self.step += 1
    self.turn = 1
    self.check_win()
```

```
AI Turn:

((1, 0), (2, 0))
[[1 1 1 1 1 1 1 1]
 [0 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [2 2 2 2 2 2 2 2]
 [2 2 2 2 2 2 2 2]]
Your Turn:

Please enter start row [0,8):
6
Please enter start column [0,8):
1
Please enter direction [0,3):
2
((6, 1), (5, 2))
[[1 1 1 1 1 1 1 1]
 [0 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0 0]
 [2 0 2 2 2 2 2 2]
 [2 2 2 2 2 2 2 2]]
AI Turn:

((2, 0), (3, 0))
[[1 1 1 1 1 1 1 1]
 [0 1 1 1 1 1 1 1]
 [0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0 0]
 [2 0 2 2 2 2 2 2]
 [2 2 2 2 2 2 2 2]]
```

3rd Version – Extra Credit

All evaluation functions above are based on the hypothesis that the game is zero sum game. In this version, I will develop two different evaluation functions for offensive agent and defensive agent.

Offensive

The offensive evaluation function is the same as the function above. The evaluation function will calculate the position values of each piece and the number of opponent captured.

Defensive

The defensive evaluation function is different from the function above. This function will calculate the position values of the opponent's pieces and give the most weight to the pieces in the base. Then the function will calculate the number its own pieces. The difference between this function and defensive evaluation function above is that the further opponent pieces have larger weight with equation j*j.

Evaluation function:
Eval_off() = step^2 + 100*captured;
Eval_def() = distance^2 + 100*pieces + 50*capture.
The offensive function is trying to move forward and capture opponents, while the defensive function are trying to block the way, keep opponent away from its base, and avoid being captured. The code for offensive evaluate function is the same, so I only show the python code for defensive.

```
def get_eval_def(self, worker):
    self.distance = 0
    self.piece = 0
    self.capture = 16
    if worker is 1:
        for j in range(self.size_y):
            for i in range(self.size_x):
                if self.state[j][i] == 3-worker:
                    self.distance += j*j
                    self.capture -= 1
                    if j == 0:
                        self.distance = -1600
                    elif self.state[j][i] == worker:
                        self.piece += 1
                        if j == self.size_y - 1:
                            self.distance = 1600
    if worker is 2:
        for j in range(self.size_y):
            for i in range(self.size_x):
                if self.state[j][i] == 3-worker:
                    self.distance += (7-j)*(7-j)
                    self.capture -= 1
                    if j == self.size_y - 1:
                        self.distance = -1600
                    elif self.state[j][i] == worker:
                        self.piece += 1
                        if j == 0:
                            self.distance = 1600
    return self.distance + 100*self.piece + 50*self.capture
```

figure 8 python code for defensive evaluation function

	Offensive vs. Defensive	**Defensive vs. Offensive**
Alpha-beta vs.	[[1 1 1 1 1 1 1 1] [0 0 0 1 0 1 0 0] [0 0 0 0 0 0 0 0] [0 2 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 2 0 0 0 0 0 0] [0 2 0 2 0 2 1 2]] Winner: Player_1 Strategy: AlphaBeta vs. AlphaBeta Style: Offensive vs. Defensive Node Expanded: 22355 vs. 45869 Node Exp/Move: 931 vs. 1994 Avg Time/Move: 0.2629 vs. 0.5839 Oppo Captured: 10 vs. 5 moves to Win : 47	[[1 2 0 1 1 1 1 0] [0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 1] [0 0 0 0 0 0 0 0] [0 0 0 0 0 0 0 0] [0 0 0 0 2 0 0 0] [0 0 0 0 0 0 2 0] [2 2 2 2 2 0 2 0]] Winner: Player_2 Strategy: AlphaBeta vs. AlphaBeta Style: Defensive vs. Offensive Node Expanded: 49910 vs. 24880 Node Exp/Move: 1919 vs. 956 Avg Time/Move: 0.5696 vs. 0.3168 Oppo Captured: 7 vs. 9 moves to Win : 52
	Offensive vs. Offensive	Defensive vs. Defensive

Alpha-beta

```
[[1 1 1 1 1 1 1 1]
 [0 2 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 2 2 2 0 2 0]
 [2 2 2 2 0 2 1 2]]
Winner: Player_1
Strategy: AlphaBeta vs. AlphaBeta
Style: Offensive vs. Offensive
Node Expanded: 20199 vs. 16930
Node Exp/Move: 1262 vs. 1128
Avg Time/Move: 0.3832 vs. 0.2878
Oppo Captured: 5 vs. 6
moves to Win : 31
```

```
[[0 0 0 1 1 0 1 0]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [2 0 0 0 0 0 0 0]
 [0 0 0 0 2 1 0 0]]
Winner: Player_1
Strategy: AlphaBeta vs. AlphaBeta
Style: Defensive vs. Defensive
Node Expanded: 47264 vs. 49378
Node Exp/Move: 1350 vs. 1452
Avg Time/Move: 0.351 vs. 0.3647
Oppo Captured: 14 vs. 11
moves to Win : 69
```

Analysis

The defensive evaluation function is the combination of defensive and offensive. The function is trying to keep the opponent far away from the base, remain as many pieces as possible, and capture the opponent if the opponent is close to the base. However, the effect of this defensive function is not desirable. When defensive agent competes against the offensive agent, the defensive agent lost every time. The reason is that the defensive function tries to block the opponent, but if it failed, it won't focus on the opponent passed through.