**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.

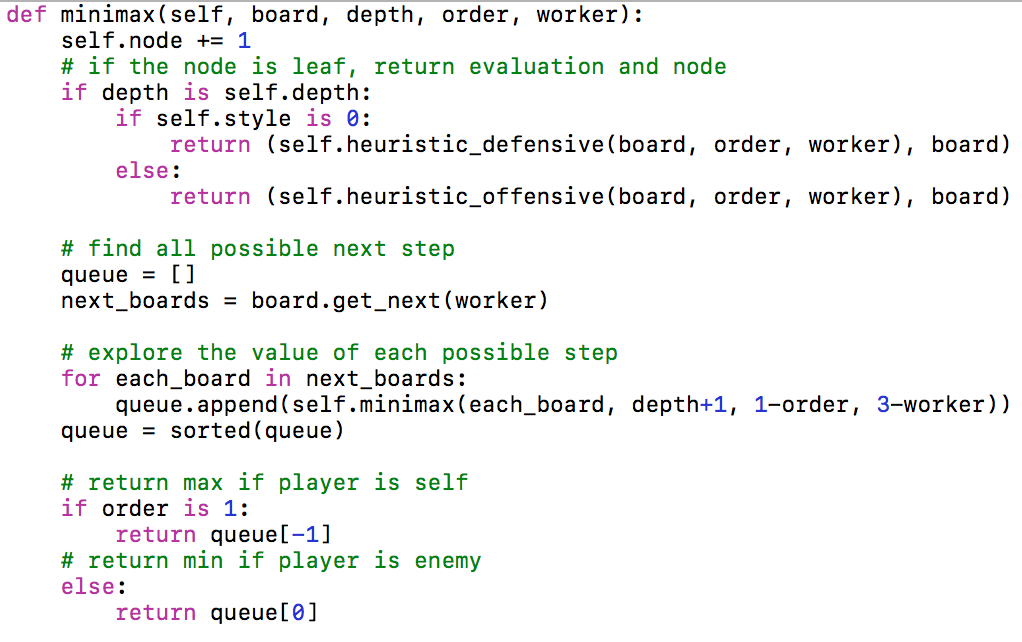


figure 1 minimax search python code

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.

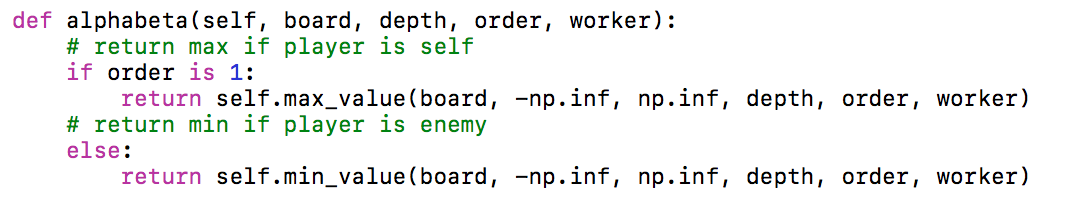


figure 2 python code for alpha-beta search

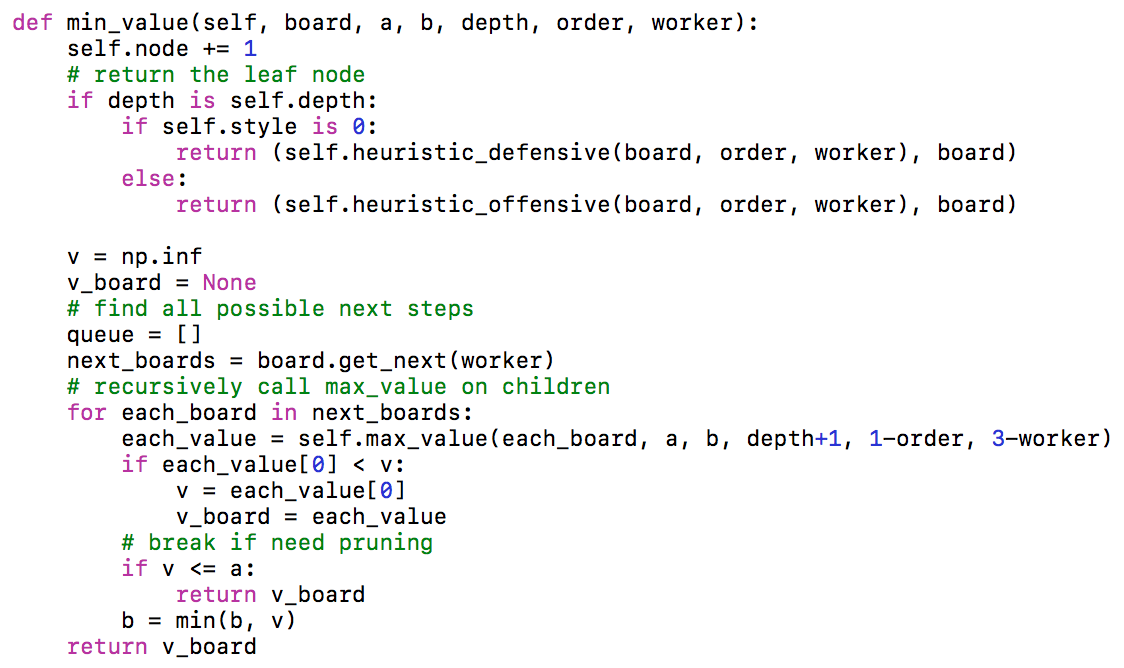


figure 3 python code for min\_value()

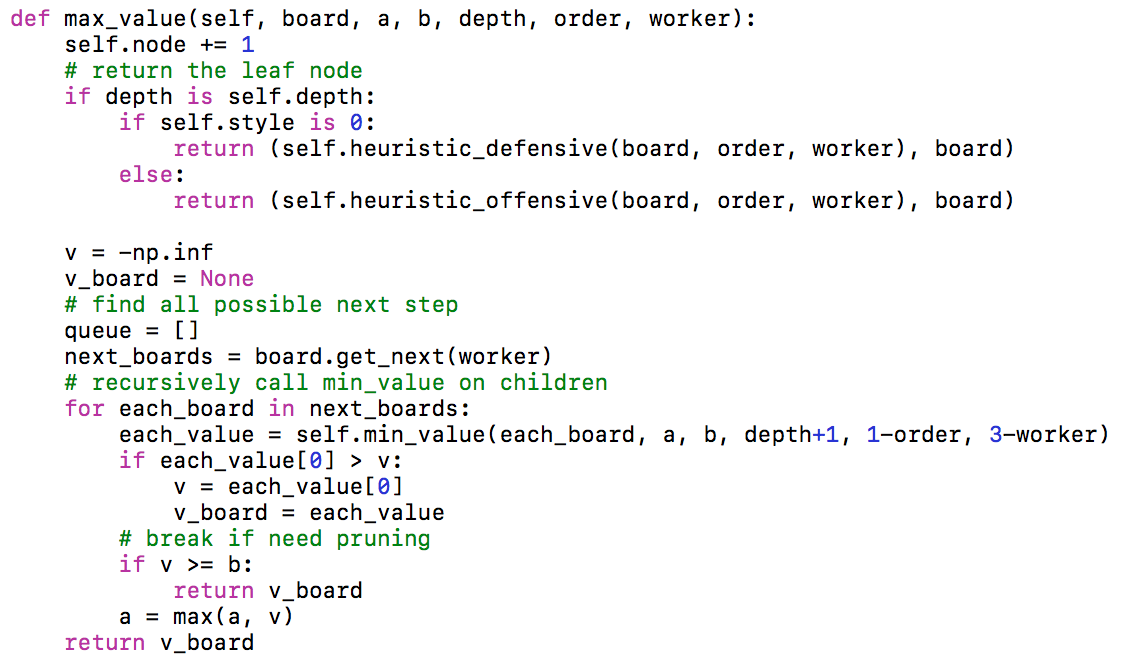


figure 4 python code for max\_value()

**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.

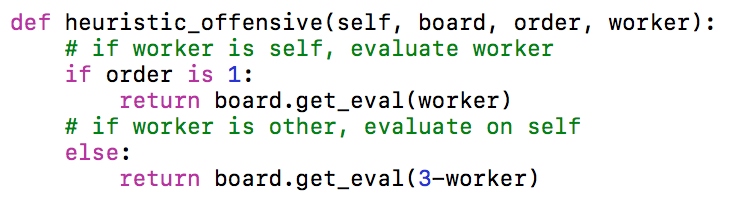
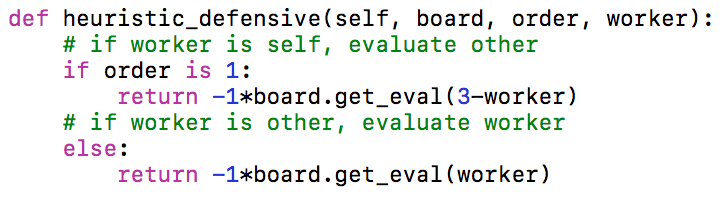
 

figure 5 python code for heuristic

**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.

|  |  |  |
| --- | --- | --- |
| Minimax  vs.  Minimax | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| Offensive vs. Offensive | Defensive vs. Defensive |
|  |  |

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.

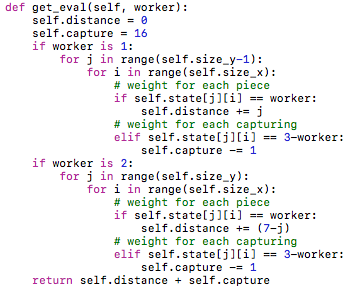


figure 6 python code for evaluation function

**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.

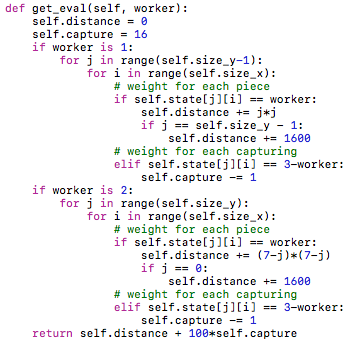


figure 7 python code for evaluation function

|  |  |  |
| --- | --- | --- |
| Alpha-beta  vs.  Alpha-beta | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

**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 | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

|  |  |  |
| --- | --- | --- |
| Alpha-beta  Depth = 4  vs.  Alpha-beta  Depth = 4 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

|  |  |  |
| --- | --- | --- |
| Minimax  Depth = 3  vs.  Alpha-beta  Depth = 4 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

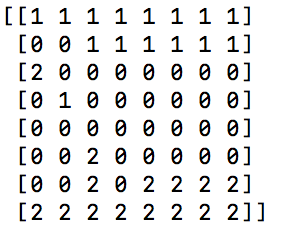
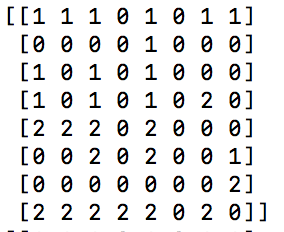
|  |  |  |
| --- | --- | --- |
| Alpha-beta  Depth = 4  vs.  Minimax  Depth = 3 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

**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.



|  |  |  |
| --- | --- | --- |
| Minimax  Depth = 3  vs.  Minimax  Depth = 3 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

|  |  |  |
| --- | --- | --- |
| Alpha-beta  Depth = 4  vs.  Alpha-beta  Depth = 4 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

|  |  |  |
| --- | --- | --- |
| Minimax  Depth = 3  vs.  Alpha-beta  Depth = 4 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

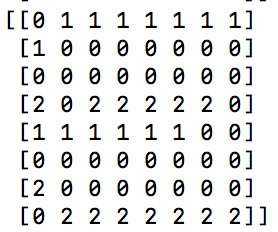
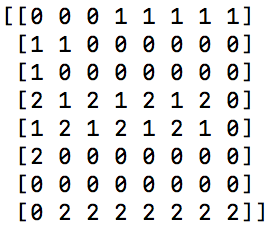
|  |  |  |
| --- | --- | --- |
| Alpha-beta  Depth = 4  vs.  Minimax  Depth = 3 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

**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.

Offensive Defensive

**Four Credit**

**Rectangular Board**

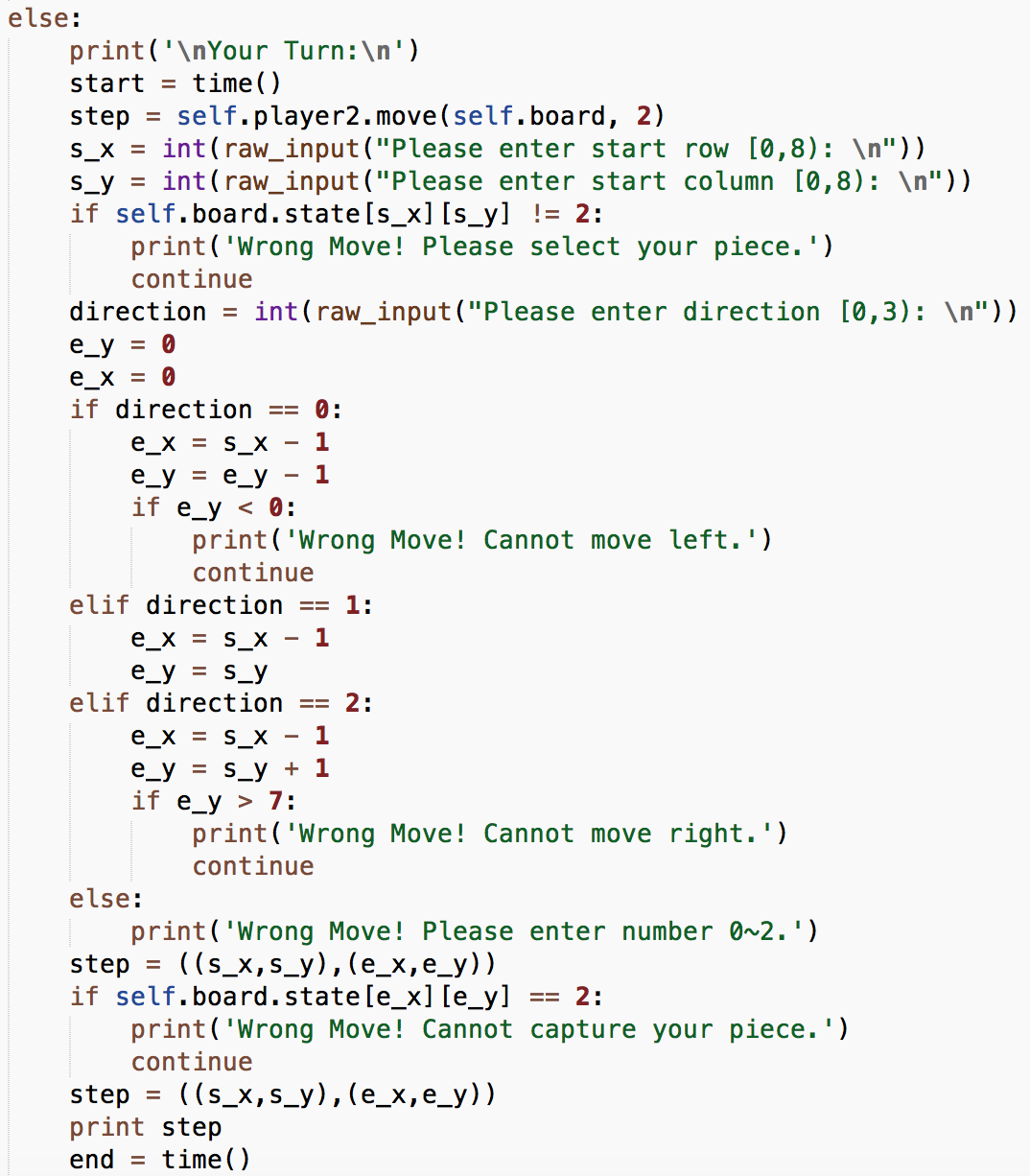
In this part, the number of pieces will be less and the distance to win will be longer. In this case, I will increase the weight of each pieces and the weight of distance. Therefore, the agent will avoid its pieces being captured and prefer to moving forward. The deepest depth that can solve the problem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

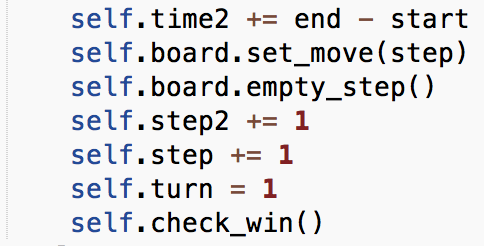
|  |  |  |
| --- | --- | --- |
| Alpha-beta  Depth = 5  vs.  Alpha-beta  Depth = 5 | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| **Offensive vs. Offensive** | **Defensive vs. Defensive** |
|  |  |

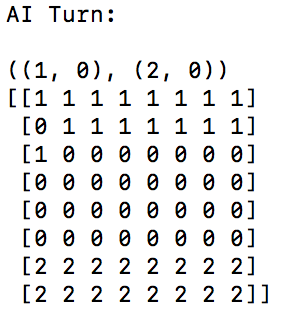
**Extra Credit**

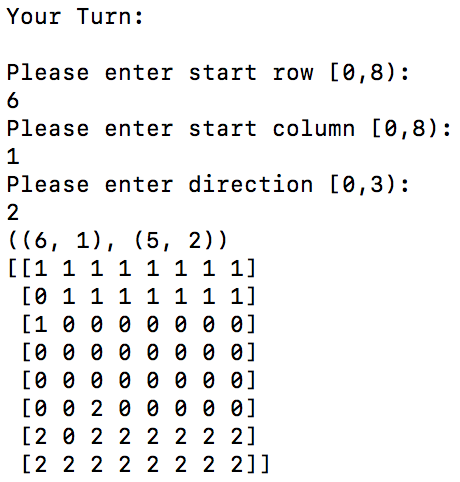
**AI vs. Human**

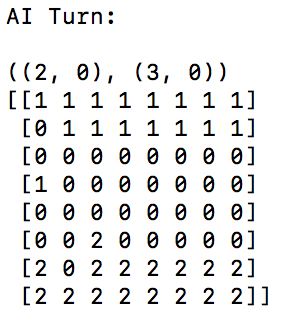
Please run start.py and compete against AI.











**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.

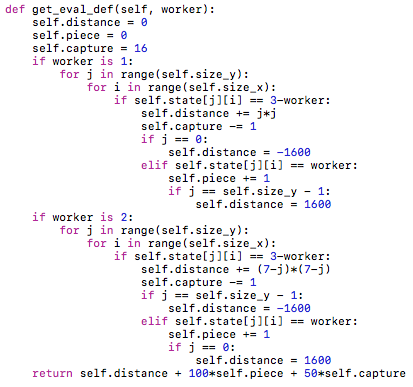


figure 8 python code for defensive evaluation function

|  |  |  |
| --- | --- | --- |
| Alpha-beta  vs.  Alpha-beta | Offensive vs. Defensive | Defensive vs. Offensive |
|  |  |
| Offensive vs. Offensive | Defensive vs. Defensive |
|  |  |

**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.