Implementation

For the game implementation, we using three classes to represent it. They are Player, Board and Piece. And in order to make the code clear, we let the minimax tree be a individual class.

The player class performs the functions including choose a solver, using the solver to decide move which piece and perform capture or retreat actions.

Board class stored all the piece objects data and the raw real board information which means all the pieces' position and color are directly stored in a two-dimensional array. When a player wants to move a piece, he needs to using board class's function to find which pieces are movable.

Piece class stored its position data and color. It also used to store all the actions which can be performed by the piece and the related pieces which may be removed when the piece performs such an action. Therefore, player can decide which piece to move and which action to do by accessing to piece's data.

For random solver, we just let player check which piece can move and randomly choose one of them. Then the player will randomly choose one of the available actions. And the algorithm will do the process again and again until there is only one color of pieces on the board.

For minimax brute-force solver, we use MinimaxTree class to perform its functions. Firstly, it will build the minimax tree by traverse all the possible actions of a movable piece. Each action will generate a new raw board data and add it to be the child of the action. The algorithm will do it recursively until reaching the depth limit. Then the algorithm will traverse the tree and run the evaluation function from bottom to top and assign the evaluation value to current tree node by minimax algorithm. Finally, when all the nodes in the tree are assigned an evaluation value. The algorithm will find the path by following minimax algorithm and choose the first one in the path to be the next action to perform. Generally, the solver will do this process until the board only has one color of pieces on the board.

For alpha-beta solver, we using the same tree building algorithm to build the minimax tree. However, it will traverse all the tree nodes using alpha-beta algorithm. Hence, it may prune some useless branches when running the alpha-beta algorithm.

Reasons

The reason why we need three classes is using the OOP concepts to decrease the difficulty of implement such a game. We can get the idea by simulating a real game. And distribute all the possible actions in a real game to different objects and pack

them into sever functions. Also, it can be much easier to extend the code to fit different solvers or test cases.

Evaluation Functions

First:

The evaluation value represents the difference between the number of white pieces and the number of black pieces. For player using white piece, he needs to let the evaluation value be as bigger as possible. And for its opponent, the value should be the opposite side.

Because rules of the game, if we can keep the number of allied pieces is more than opponent's pieces as many rounds as possible. The win rate will be as higher as possible.

Second:

The evaluation value represents the percentage of all the allied pieces for all the pieces on the board. That means the higher the value is, the higher win rate is. When the value is 1, the player wins the game or the player will fail when the value is 0.

Experiments

When the children of a states have the same evaluation value, it will select the first one fit the previous settled path value. Hence, sometime the algorithm will go to different branch even with the same starting configuration.

Test Case 1

Board: 3*3
First: Brute-force with large depth limit and zero evaluation function
Second: Brute-force with large depth limit and zero evaluation function

Result:

First: 11 rounds, white total states: 4691, black total states: 870
Second: 13 rounds, white total states: 4907, black total states: 1129

Analysis:

The two cases' results are almost the same. But sometimes they will be different because of the implementation details. We let the algorithm randomly choose one branch of several branches having the same evaluation value.

Test Case 2

Board: 5*5, 5*9
First: Random solver

Second: Brute-force with 2 depth limit and 1 evaluation function

Result:

Board: 5*5
First: 17 rounds, white total states: 9, black total states: 8
Second: 18 rounds, white total states: 334, black total states: 331

Board: 5*9
First: 45 rounds, white total states: 23, black total states: 22
Second: 127 rounds, white total states: 13408, black total states: 13524

Analysis:

Generally, random solver has better performance than Brute-force solver. However, the performance of random solver is not consistent and will be heavily affected by the board size. On the contrary, brute-force solver's performance is much more consistent.

Test Case 3

Board: 5*5, 5*9
First: Brute-force with 1 depth limits
Second: Brute-force with 2 depth limits

Result:

Board: 5*5
First: 18 rounds, white total states: 39, black total states: 53
Second: 13 rounds, white total states: 161, black total states: 134

Board: 5*9
First: 38 rounds, white total states: 171, black total states: 222
Second: 40 rounds, white total states: 1117, black total states: 1101

Analysis:

The solver with higher limit will traverse much more states than another solver. Generally, brute-force solver can have a better performance with lower depth limit.

Test Case 4

Board: 5*5, 5*9
First: Brute-force with 2 depth limits, evaluation function one
Second: Brute-force with 2 depth limits, evaluation function two

Result:

Board: 5*5
First: 16 rounds, white total states: 146, black total states: 115
Second: 17 rounds, white total states: 169, black total states: 139

Board: 5*9
First: 22 rounds, white total states: 264, black total states: 254
Second: 20 rounds, white total states: 248, black total states: 295

Analysis:

The two different evaluation function have a similar performance for current test environments. The games rounds are similar and the number of total traversed states are also similar.

Test Case 5

Board: 5*5, 5*9
First: Alpha-beta with 2 depth limits, evaluation function one
Second: Alpha-beta with 2 depth limits, evaluation function two

Result:

Board: 5*5
First: 19 rounds, white total states: 199, black total states: 218
Second: 19 rounds, white total states: 205, black total states: 167

Board: 5*9
First: 26 rounds, white total states: 344, black total states: 349
Second: 27 rounds, white total states: 255, black total states: 338

Analysis:

The results show the similar information just like the previous test case. The two different evaluation function have much similar performance.

Test Case 6

Board: 5*5, 5*9
First: Brute-force with 2 depth limits, evaluation function one
Second: Alpha-beta with 2 depth limits, evaluation function one

Result:

Board: 5*5
First: 19 rounds, white total states: 248, black total states: 285
Second: 19 rounds, white total states: 178, black total states: 183

Board: 5*9

First: 25 rounds, white total states: 339, black total states: 309
Second: 17 rounds, white total states: 255, black total states: 158

Analysis:

The results show that alpha-beta algorithm is more efficient than brute-force. When the game rounds are the same, it traverses about 2/3 state of brute-force algorithm.