Minimax Search with Alpha-Beta pruning for Gomoku

***Abstract*—In this project, we used Minimax algorithm and Alpha-Beta pruning to implement an Agent for Gomoku. Due to the limitation of computation power and time, we implemented the Truncated search based on evaluation functions, which is calculated according to the "distance" of current board state(the number of steps) to the goal state and thus is accurate, effective, and consistent. Inspired be Local search, to speed up our time for searching, we set a Branch Factor for our Minimax search, only expanding the notes that have higher score with respect to our evaluation function. This agent can defend against threatening situations or behave aggressively when we are on a good wicket with the professional concept of "VCT "(Victory of Continuous Threat) inspired by Threat Space search. It also worth to be noticed that our board representation enables us to update our board state, resulting in relatively deeper searching layers. In the end we can always beat MUSHROOM, and win the game most of the time when playing against FIVEROW.**

***Index Terms*—Minimax Search, Alpha-Beta pruning, Truncated search, Threat space search, Gomoku**

I. Introduction

The Gomoku game seems to be over 4000 years old, and its rules have been developed in China. The goal is to align 5 checkers of the same color, vertically, horizontally, or diagonally, at the central intersection of the board. Minimax algorithm is a recursive or backtracking algorithm which is used in zero-sum games. It provides an optimal move for the player assuming that opponent is also playing optimally, minimizing the possible loss for a worst case (maximum loss) scenario. However, as William says [1] , it is impossible to complete a search completely. Fortunately, alpha-beta pruning can be used to speed up Minimax search tree [2], since it can avoid access to a state that is already worse than the known state. In this project, we have also adopted the following methods to speed up the search speed and make the program run effectively: limitations on search depth, setting of the branch factor, evaluation of each move, and dynamical update of the score of the board. Organization for the following report:

- Section 2 Minimax with Alpha-Beta Pruning
- Section 3 Evaluation Function
- Section 4 Implementation Details (representation, optimizations(branch factor, sort, move evaluation, VCT detectation))
- Section 5 Experimental Results
- Section 6 Conclusion and Future Word

II. Minimax with Alpha-Beta Pruning

A. Minimax Search

In general **Zero-sum** games, the maximin value of a player is the largest value that the player can be sure to get without knowing the actions of the other players; equivalently, it is the smallest value the other players can force the player to receive when they know his action [3]. Notice that it is based on the assumption that our opponent is rather "rational".

The state space size of Gomoku is too much large, so our agent is based on the Truncated Minimax search with pre-specified maximum depth of searching and it will use evaluation functions to score each Board state.

B. Alpha-Beta Pruning

Alpha-Beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the Minimax algorithm in its search tree. It will cut off those branches that are impossible to affect the decision, let the agent acts more effective. A simple illustration for Minimax search coombined with Alpha Beta Pruning is shown below.

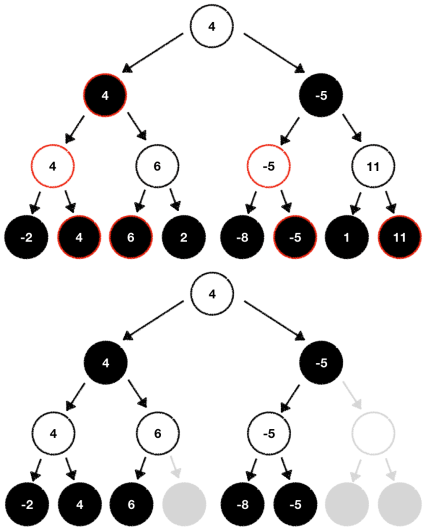

Fig. 1. Example of Minimax Search with Alpha-Beta Pruning

The algorithm maintains two values, alpha and beta, which represent the minimum value guaranteed by the maximizer and the maximum value guaranteed by the minimizer. At the beginning, both players start with their worst score. When the maximum score obtained by the minimizer is less than the minimum score obtained by the maximizer, the maximizer does not need to consider the subsequent nodes of this node, as they will never be reached in the actual game [2].

Note that the larger the branching factor of the tree, the higher the amount of computations we can potentially save through this technique. The pseudo-code is shown below.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Fig. 2. Pseudo-code of Minimax Search

III. Evaluation Function

In truncated search, the evaluation function performs utility value calculations on non-terminated nodes. Here we use the Expert Scored features and carefully designed linear combinations of features to evaluate the non-terminal states.

A. Basic Ideas: Possibility of Winning

An intuitive way to determine the probability of winning from the current situation is to count the number of all potential(or threatening) patterns on the board and assign different patterns different values. Then, we will calculate the linearly weighted sum of these patterns of both sides, taking their numbers of appearance into consideration as well, as an estimate. of the probability of winning in the current state. The determination of 'potential patterns' and the assignment of their corresponding scores use expert Gomoku knowledge. The pattern-value table is listed below.

B. Assigning Weights

In the world of Gomoku, we want to win the game as soon as possible, but we don't want to lose as well. So for the same pattern the black side and the white side will be assigned different weights. In particular, we multiply some of the patterns of the white by a factor k, which means we will consider more about the possible threat from the opponent side. A direct consequence of this is that our agent will behave more conservative.

TABLE I
PATTERN-VALUE TABLE

Key	Pattern	Value
WIN	-ooooo-	200000
H4	-oooo-	10000
C4	-oooox-	200
H3	-ooo-	200
M3	-oox-	50
H2	-oo-	5
M2	-ox-	3
S4	-xoooox-	-5
S3	-xooox-	-5
S2	-xox-	-5

Assuming that we count 2 "H3" or better patterns, (we only consider it if the number of the appearance of these patterns is more than 1), this results in an additional score of $2 \times bonus$, where $bonus$ is a pre-determined super-reference. Here we set $bonus = 5000$, between the values of "H4" and "C4". In addition, as mentioned earlier, our AI is on the conservative side. This is also reflected in the bonus value, which will adjust the opponent's values as four times higher than the values from our side.

C. Combined Patterns: More Detailed Evaluation

We noticed that there are some special cases that should be put more emphasis on. For example, sometimes the next move we take will form a double "H3" or a "C4" and "H3" at the same time, in which cases the opponent will have to block one of them and we can win with the other, or vice versa. Therefore, on top of the aforementioned evaluation function, we have added an extra part to deal with this situation.

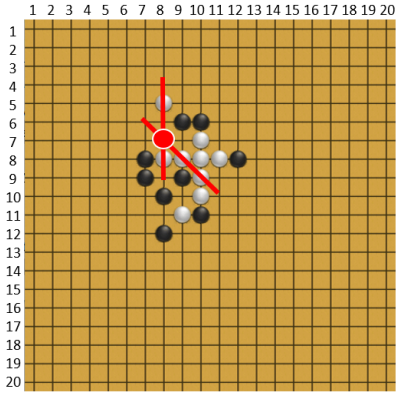

Fig. 3. Example for Combined Pattern of Double "H3"

If it is our agent's round, i.e. in the max layer, the agent will check whether this combined pattern will be formed if itself places a piece here, and whether it will be formed if the opponent takes the move. If so, we will **add** some points accordingly.(**minus** some points if it's the min layer). This strategy will suggest our agent to block the best position of the opponent player within current layer of simulation, so it's a preferable strategy especially when the maximum depth

is relatively small. However, when we take deeper iterative depth, we changed our strategy to come across the problem mentioned above.

IV. Implementation Details

A. Node Representation of the Tearch Tree

The nodes of the search tree in our implementation of the minmax algorithm contain properties like $successors$, $isLeaf$(whether it's the leaf node), $value$, $action$, where action indicates the move that will take us from the previous state to current state (i.e. where to place the next piece).

B. Representation of the board

1) Representation with Dictionaries: Since each move will possibly change the chess pattern only along 4 directions during the simulations in a search, we re-represent the board (originally a 2-dimensional array) as a dictionary, with key being the hash value of each line (which will be explained in details later) and value being a list, including the string corresponding to the direction and a score calculated by the evaluation function.

Also, in order to quickly update the calculation of the evaluation function later in the tree building process, we have calculated separate scores of each line black and white sides respectively. Thus, assuming that the size of the board is N x N, the number of key-value pairs of will be

$$2x(N + N + (2N - 1) + (2N - 1)) = 12N - 4 \quad (1)$$

which is 236 in this project especially), the contents of which are ($player$, hash value of this line):[string of this line, $utility$].

What's more, our opponent's moves will also change patterns along 4 directions according to our board representation throughout the game, we set this representation as a global variable, like the board in the original code, and update it in our $brain_turn()$ and our opponent's $brain_opponents()$.

2) Calculation of Hash Values: For rows and columns, this is their corresponding row and column numbers. Also, given (x, y), its corresponding index in row dictionary is (player, x) and (player, y) in column dictionary. It is then obvious that points on the same line in the subdiagonal direction have the same $x + y$ values, different from the value of other lines, so we use $x+y$ as the hash value for the index in the subdiagonal direction dictionary. A similar approach can be taken for the index in the primary diagonal dictionary, i.e., $width-1-x+y$.

For example, the first main diagonal of the board below will be represented in dictionary $diag\_1$ as $\{(1, 19)$:['00000000110200000000',5]$\}$ for the black side and $\{(2, 19)$:['00000000110200000000',0]$\}$ for the white side.

C. Optimizations

The number of possible successors depend on how we select the expanding strategy, while the how many nodes are taken into the fringe is influenced by our Generating strategy. If we were to use all the empty positions on the board as candidates for the next level, there would be too many possible next states and require too much calculation. What's more, the search order of successors will affect the performance of **Alpha-Beta** pruning, so the order of successor nodes is also important. Therefore, we adapted following strategies to speed up the searching process.

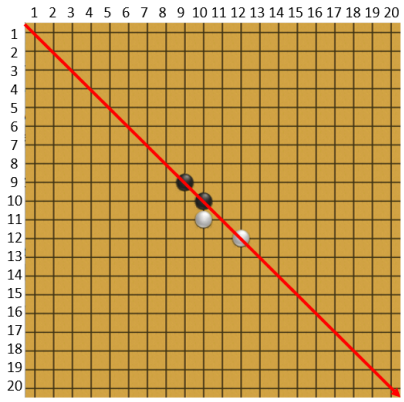

Fig. 4. Representation Example

1) Expand: Observe neighbourhood Nodes: According to the expert knowledge of Gomoku, an intuitive optimisation is to consider only the empty positions around pieces already on the board. In the implementation, we expand empty points with a distance of one square around non-empty ones for the next layer, at which point there would be dozens of or $100+$ search nodes per layer. As the game continues, however, this number can be very large, becoming about the same as searching the entire board.

2) Generate: Sorting and Selecting Successors: For each layer of the search tree, we specify the maximum branching factor N, and, since in min-max search using alpha-beta pruning, in the max layer it is more beneficial to sort the nodes from largest to smallest according to their values before pruning, and vice versa in the min layer. In the implementation we first calculate the value of each node we expanded with evaluation function as if we've already placed a piece on this position. Then we sort the nodes by the value, generate the nodes with higher rankings and keep the number of successors no more than the branching factor N.

3) Threat Patterns Detection: When certain patterns appear on the board, such as when the opponent has "C4" or when we have "H4", our optimal solution is unique and easy to determine. Therefore, making these judgements before performing a min-max search can, on the one hand, prevent the Agent from making low-level mistakes, and on the other hand, reduce the Agent's search time. If neither of these two situations is present, we then test from our own side to see if we can form a "WIN", and if so, just place our piece here; if all of the judgement were failed, then we finally perform a min-max search. In other cases, for example, where we can form a "H4" but the opponent already has a "H3", we will not make a quick decision but let this node be evaluated and ranked as others.

V. Experiment Results

In the matching, our AIs will compete with 12 AIs from Gomoku website, under 3 fixed openings with exchanges of hands. All players should limit the time of each step within 15 seconds and the entire game time should not exceed 90 seconds.

We've implemented several agents with either different maximum depth for iteration, branch factors or the scope that influence the number of successors we take into consideration. Here's the results of two of the agents, one with maximum depth of 3 and the branch factor of 6, the other with maximum depth of 5 and the branch factor of 4, which are denoted as AI3 and AI5 respectively. Their results are listed as follow.

TABLE II
Gomoku Competition Results

Opponent	**AI3**	**AI5**
YIXIN17	0:12	0:12
WINE	0:12	0:12
PELA17	0:12	0:12
ZETOR17	1:11	1:11
EULRING	1:11	2:10
SPARKLE	0:12	0:12
NOESIS	1:11	0:12
PISQ7	2:10	1:11
PUREROCKY	8:4	3:9
VALKYRIE	8:4	4:8
FIVEROW	10:2	6:6
MUSHROOM	12:0	12:0
Ratio	48%	25%

Notice that though the overall performance of AI3 is better than AI5, AI5 can also 100% defeat the MUSHROOM in the competition. Actually when running the competition on our own computer with unlimited time rather than on the website, we find given enough time and AI5 can defeat other AIs more times than AI3, which means the relatively low elo score may due to the excessive time it needs to search with deeper layers.

VI. Conclusions and Future Work

In this project, we implemented Gomoku Agent based on minimax searching algorithm with alpha-beta pruning. Our representation of the board state, sorting and selecting strategies to expand and generate successors, and detailed evaluation function all contributes to the performance our Agent. (The highest result of the elo score is 1173.) In the future, we intend to improve our AI in the following fields:

A. Better Pattern Detection Strategy

For truncated minimax search, as our instructor mentioned in the class, **Table-based** evaluation function can be more effective in the opening and closing phases. We want to improve our strategy to detect the patterns on the board to save more time, thus allowing us to iterate deeper during each search process.

B. Non-repetitive Representation of the Board

We have just set the representation of the board as a global variable, saving some time for updates. In fact, if the opponent plays a piece that has been modelled in our search tree before, we can just use this sub-tree directly as the new search state.

C. Combined Patterns: Threat Space Search

In this implementation, we just assign a specific coefficient to the combined patterns. However, we find this strategy won't always work well. So in the future, we want to introduce the concept of **VCT**(Victory of Continuous Threat) and **VCF**(Victory of Continuous Four) to our agent, which may improve the performance and reduce the time for computation when we reach some certain patterns.

References

[1] William Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. Journal of Finance, 16(1):8-37.q

[2] Knuth D E , Moore R W . An analysis of alpha-beta pruning. Artificial Intelligence, 1975, 6(4):293-326

[3] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach. Prentice Hall. 2009.