IT3160E : INTRODUCTION TO AI

Artificial Intelligence for Gomoku

Team:

Manh Duong NGUYEN	20210243	duong.nm210243@sis.hust.edu.vn
Nam Hai NGUYEN	20214894	hai.nm214894@sis.hust.edu.vn
Sang HUYNH	20214930	sang.h214930@sis.hust.edu.vn
Jean ROLAND-GOSSELIN	2022T010	roland-gosselin.jjmo22T010@sis.hust.edu.vn

Supervisor:
Dr. Nhat Quang NGUYEN

Semester 2022.1, Class 136462
January 1, 2023

ĐẠI HỌC
BÁCH KHOA

Contents

Abstract

For years, the performance of AIs in games have been considered a useful benchmark of the progress of AI. Crucial programs and events are often marked as milestones for game AI, from DeepBlue v. Kasparov (1997) to AlphaGo v. Lee Sedol (2016). Nowadays, strong AIs in Chess (Stockfish, Alpha Zero) are even used by professional players to prepare for over-the-board matches.

Compared to other board games like Go and Chess, Gomoku is a much simpler game with an easy-to-implement set of rules. Hence, we choose this game to bring the most common adversarial search algorithm to the test - Minimax Algorithm.

1 Introduction

Gomoku is a simple board game: Two people in turn place "stones" on a board, and whoever got five stones of their color in a row first wins. However, it has many interesting variants [1] [2], namely:

- Freestyle Variant: Played on a 20-by-20 lattice, win by getting at least five stones of our color in a row.
- Standard Variant: Played on a 15-by-15 lattice, win by getting exactly five stones of our color in a row.
- Renju: Played on a 15-by-15 lattice, however, Black cannot make 4x4 or 3x3 forks and cannot win by overlines (but 4x3 forks are allowed). White has none of the above restrictions.
- Caro: The Vietnamese variant of the game. Played on an infinite lattice, and this is one of the most balanced variants of the game, since it does not allow for five in a row with two blocked ends.

The Freestyle and Standard Variants were already solved: Allis [3] found out that the first player would always win with optimal play from two players using his Threat-Space Search theory. Free Renju was also solved [4] with Allis' theory. For this project's report, we are going through our implementation of an intelligent agent for the Freestyle Variant. Originally, we set out to implement for the Standard Variant, but due to some difficulties that will be discussed later on in this report, we cannot achieve a stable version on time.

When we started this project, our expectation is that our AI could defeat some baseline AIs, and to be able to hold a game against the average human player, and it did achieve that (more on that later). Although our AI is far from perfect, we are pleased with the result and have gained a lot of valuable experience in the research and coding process.

2 Related works

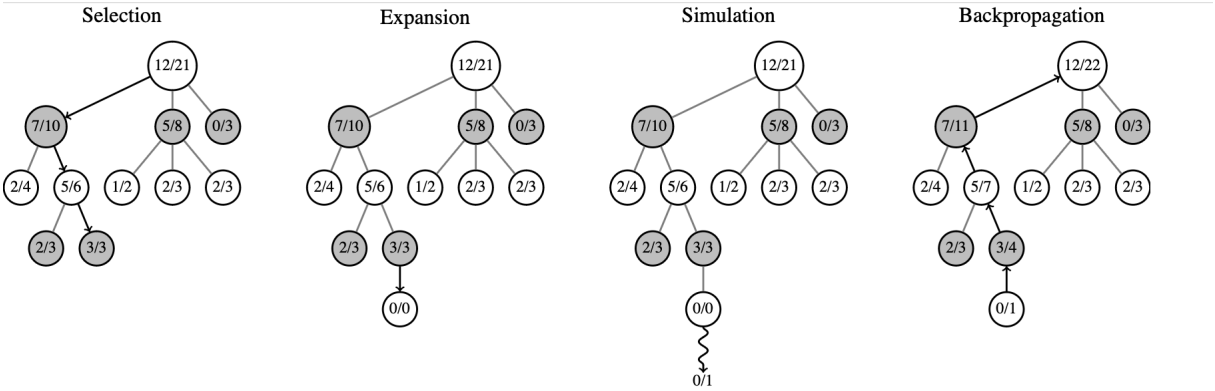
Many works have been done on algorithms for two-player games in general and the game of Gomoku in particular. This section will mention some important algorithms of the game.

2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [5] is a heuristic best-first search algorithm based on random playouts. It has yielded breakthroughs in computer Go and has succeed in several other games.

MCTS consists of four strategic phases, repeated as long as time is left:

- Selection Phase: The tree is traversed from the root node until it selects a leaf node that is not added to the tree yet.
- Expansion Phase: A leaf node is added to the tree.
- Simulation Phase: Moves are played until the end of the game. The result is either 1, 0 or -1.
- Backpropagation Phase: The results are propagated through the tree.



It takes a lot of time to simulate in the traditional MCTS tree. This is a huge flaw of MCTS, since the result is usually not much better than Alpha-Beta pruning. The technique of Upper Confidence bounds on Trees (UCT), a tree selection policy, is invented to tackle this problem. Together with Neural Networks, Reinforcement Learning and a few additional techniques, MCTS has proven its practicality in Game AI.

2.2 Minimax and Variants

Minimax algorithm is one of the most basic yet powerful algorithm in turn-based game. The algorithm is based on backtracking and is frequently used in zero-sum games, and could be modify to work with games involved randomness (such as 2048, Backgammon, etc.). Our goal is to determine the score of a position after a certain number of moves, with best play according to a predefined evaluation function. To achieve this, we have to also assume our opponent always choose the best move. Thus, for each position, we will have to determine which variation would happen if both players play optimally.

In the below example, we are at position A and would want to maximize our score, while our opponent wants to minimize it. Here we have three options, to either play B1, B2 or B3, and for each options, our opponent also has three choices. We observe, if we want to achieve position C1 or C9, we have to go with option B1 or B3, suppose we choose B1. Now, our opponent have the choice to get to C3, which it is not so desirable for us.

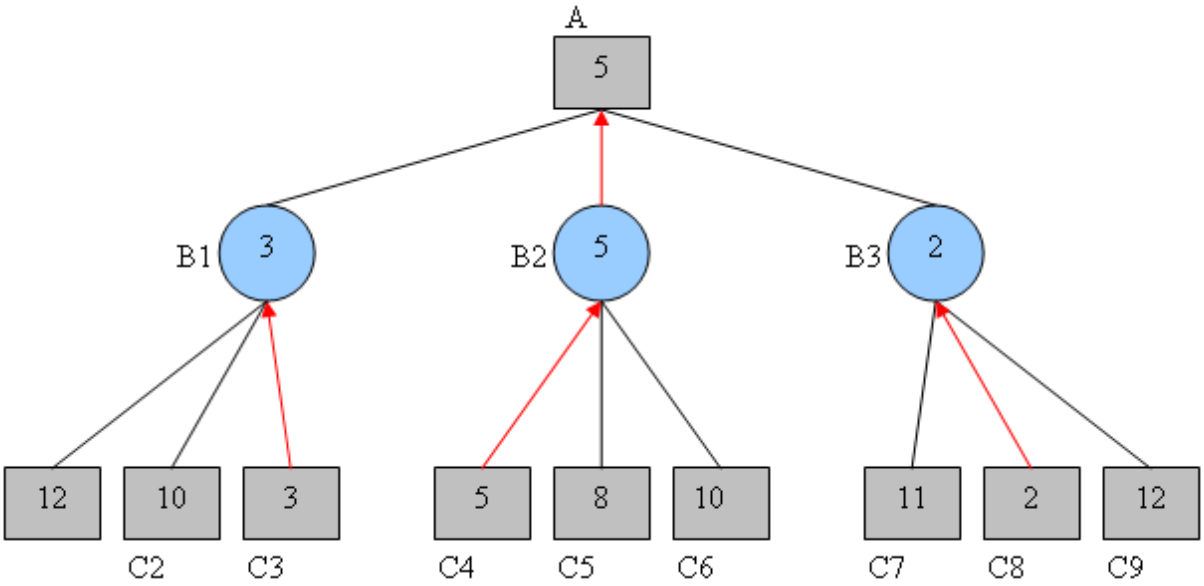


Illustration of minimax algorithm

In fact, we would have to assume that our opponent always tries to minimize the score while we also maximize it. Looking at the tree, we would see that it would be best if we get to position B2, then in that case we would get to a position of at least 5 points.

Concretely, we could formulate a recursive formula for the algorithm[6]:

$$F(p) = \begin{cases} f(p) & \text{if } b = 0 \\ \max\{G(p_1), ..., G(p_b)\} & \text{if } b > 0 \end{cases} \qquad G(p) = \begin{cases} f(p) & \text{if } b = 0 \\ \min\{F(p_1), ..., F(p_b)\} & \text{if } b > 0 \end{cases}$$

where p is the current position, b is the number of legal moves, p_i is the i-th child of p, and $f(p)$ is the numerical value computed when computing position p. Note that this is a brute-force algorithm to try all possible variations, and is not practical in almost every game tree. Hence, it is crucial that the algorithm stops when a terminating condition is reached, e.g. reaches depth limit or time limit. Here is a pseudocode for the depth limited version:

```
function MAXI(p, depth):
    if depth = 0 then return evaluate()
    else
        max := −∞
        for each child p_1, ..., p_n of p do
            max :=
     max(max, MINI(p_i, depth − 1))
        end for
    end if
    return max
end function
```

```
function MINI(p, depth):
    if depth = 0 then return evaluate()
    else
        min := +∞
        for each child p_1, ..., p_n of p do
            min :=
     min(value, MAXI(p_i, depth − 1))
        end for
    end if
    return min
end function
```

From this concept, many variants/improvements of minimax algorithm is invented: Alpha-Beta Pruning reduces the size of the search tree, SCOUT & NegaScout apply the concept of a null window to further enhance the pruning process, etc.

The hardest thing about minimax algorithm is not in the algorithm itself, but in the design of the evaluation function, i.e. how do we get the score of any position. We would discuss our implementation of this algorithm in details in this report.

2.3 Solving Gomoku

L. Victor Allis proved the game is solved using his theories of Proof-Number Search and Threat-Based Search by his computer program Victoria [3]. His Proof-Number Search aims to prove that Black (the first player) always wins with optimal play, and requires treating a game tree as an AND/OR tree, and would reduce the branching factor in the OR nodes by a heuristic ordering function. Since all the AND nodes need to be proved, the OR nodes should contain threats to also reduce the number of AND nodes.

The details of his optimizations is in his paper in our bibliography section.

3 Code Architecture

3.1 Algorithms

3.1.1 Negamax

Negamax is a variant of minimax algorithm in zero-sum games, based on the observation: $\max(a, b) = -\min(-b, -a)$. The above mathematical formulation can then be collapsed into one function [6]:

$$F(p) = \begin{cases} h(p) & \text{if } b = 0 \\ \max\{-F(p_1), ..., -F(p_b)\} & \text{if } b > 0 \end{cases}$$

where

$$h(p) = \begin{cases} f(p) & \text{if depth of } p \text{ is even} \\ -f(p) & \text{if depth of } p \text{ is odd} \end{cases}$$

And since practically we cannot search the whole game tree, usually we would just let the function recurse to an even depth and not worry about the function h at all.

Overall, the algorithm is much easier to implement. Here is the pseudocode of the algorithm:

```
function NEGAMAX(p, depth):
    if depth = 0 then return evaluate()
    else
        value := −∞
        for each child p_1, ..., p_n of p do
            value := max(value, NEGAMAX(p_i, depth − 1))
        end for
    end if
    return value
```

end function

3.1.2 Alpha-Beta pruning

Alpha-Beta Pruning is an essential technique in the class of minimax algorithms. It is a branch-and-bound algorithm that will eliminate a large portion of the game tree, while it ensures the same outcome as the traditional minimax/negamax algorithm.

The algorithm maintains two values, α and β. α serves as a lower bound for the minimax value of the root node, and *beta* serves as an upper bound. If a node's score is lower than α (i.e. fails-low), then it is too bad; if it is higher than β (i.e. fails-high), then it is too good that the opponent will most likely not allow it.

```
function ALPHABETAMAX(p, α, β, depth):
    if depth = 0 then return evaluate()
    else
        for each child p_1, ..., p_n of p do
            score := ALPHABETAMIN(p_i, α, β, depth-1))
            if score ≥ β then return beta // fail-hard beta cut-off
            else if score > α then α := score
            end if
        end for
    end if
return α
end function
```

```
function ALPHABETAMIN(p, α, β, depth):
    if depth = 0 then return evaluate()
    else
        for each child p_1, ..., p_n of p do
            score := ALPHABETAMAX(p_i, α, β, depth - 1))
            if score ≤ α then return alpha // fail-hard alpha cut-off
            else if score < β then β := score
            end if
        end for
    end if
return β
end function
```

The two function can also be merged in a single function, using the Negamax framework:

```
function ALPHABETA(p, α, β, depth):
    if depth = 0 then return evaluate()
    else
        for each child p_1, ..., p_n of p do
            score := -ALPHABETA(p_i, −β, −α, depth - 1))
            if score ≥ β then return beta // beta cut-off
            else if score > α then
                α := score
            end if
        end for
    end if
    return α
end function
```

3.1.3 Null Window

Null Window is a technique for enhancing Alpha-Beta, first proposed by Judea Pearl [7] when introducing his SCOUT algorithm. Scout was originally introduced by a recursive function called EVAL, with MAX, MIN-parameter (i.e. alpha and beta). A boolean function called

TEST was used to prove all siblings of the first brother were either below or equal to MAX so far, or above or equal to MIN. If a condition did not hold, a re-search was necessary to get the real new MAX or MIN value. The idea is that we hope the saved nodes of TEST would outweigh time for re-searches. Pearl expected this would perform better than traditional Alpha-Beta pruning in practical game trees, which was confirmed in 1985 by Rajjan Shinghal and Agata Muszycka-Jones.

3.1.4 NegaScout Algorithm

NegaScout [8] is an Alpha-Beta enhancement and improvement of Judea Pearl's SCOUT algorithm. It assumes our first move in the list of candidate moves belongs to the principal variation, i.e. the variation that both players play optimally. Then it perform tests on other moves with a null window centered around α to see if those moves can be better. If that is the case, we would have to do a proper re-search on that variation. To make NegaScout reaches its potential, an efficient move ordering is required, so as to ensure the number of re-searches is not very large and nodes are pruned aggressively.

Here is a pseudocode for the algorithm:

```
function NegaScout(p, α, β, depth)
    if d = 0 or p is a terminal node then return evaluate(p)
    end if
    for each child p_1, ..., p_n of p do
        if i = 1 then
            score := - NegaScout(p_1, −β, −α, depth - 1)
        else
            score := - NegaScout(p_i, −α − 1, −α, depth - 1)
            if α < score < β then
                score := - NegaScout(p_i, −β, −score, depth - 1) // Re-search
            end if
        end if
        α = max(α,score)
        if α ≥ β then
            break // Beta cutoff
        end if
    end for
    return α
end function
```

3.1.5 Iterative Deepening Search

Iterative Deepening Search [9] is a state space search strategy, in which a depth-limited version of depth-first search is run repeatedly with increasing depth limit until the goal is found. It has been adopted as the basic time management strategy in depth-first searches, but has proven itself to be extremely efficient when used with move ordering techniques, Alpha-Beta pruning, hashing, etc.

Most time-limited algorithm has an iterative framework, for example:

```
function iterativeFramework(root):
    let score
```

```
    depth = 2
    while there is time left do
        temp :=NegaScout(root, −∞, ∞, depth)
        if time is up then
            break;
        else
            score := temp // Register the minimax score up to depth
            depth := depth + 2
        end if
    end while
end function
```

Note that, our implementation had to include a procedure to register the best move for our intelligent agent to actually play it.

3.1.6 Transposition Table

A transposition table is a database that stores results of previously performed searches. It associates a game board with useful informations that could be reused. We have two types of transposition table, denote TT (transposition table) and ST (state table). Our transposition table entry consists of:

- The depth that the node is searched upto
- The true/upper bound/lower bound value of the node

and our state table consists of the evaluated value of the position.

Our implementation uses Zobrist's technique [10] to form a hash function, which takes a game board and outputs a bit-string of 32 bits accordingly. First, we populate a table of size 2 x BoardSize with random bit-strings. Then, we get a hashed value of a board by calling the getHash function, and update the hashed value by calling the updateHash function.

```
function getHash(board, player)
    hash = 0
    for each cell do
        if cell is not empty then
            hash = hash ⊕ table[cell][player]
        end if
    end for
    return hash
end function
```

```
function updateHash(hash, player, cell)
    hash = hash ⊕ table[cell][player]
    // Ensure O(1) update
    return hash
end function
```

We neglect the probability of hash collisions. In practice, we have not encountered such behaviors.

3.1.7 Evaluation Function

In the family of minimax algorithm, the importance of a good evaluation function is oftenly emphasized. Here we discussed qwertyforce's evaluation functions [11], whose source code was utilized and improved extensively in our project.

Let the following patterns be defined:

- "LIVE x" (x from 1 to 4): pattern of x consecutive stones of the same color, with two unoccupied ends.
- "DEAD x" (x from 1 to 4): pattern of x consecutive stones of the same color, with one occupied ends.
- "FIVE": pattern of 5 or more consecutive stones of the same color.

To each pattern, we should assign a weight accordingly.

For a game board, let the score of player c be defined as:

$$\text{score}_c = \sum_{\text{all patterns } p} w(p) \times cnt(p, c)$$

where $cnt(p, c)$ is the number of occurrences of pattern p of color c across all rows, columns and diagonals, and $w(p)$ is the weight assigned to pattern p. Finally, the score of the board from a player's perspective is $\mathbf{score_{player} - score_{opponent}}$.

By testing, we eventually came up with the weights for "LIVE" and "DEAD" patterns. It is quite simple:

$$w(p_{live-x}) = 10^x \text{ and } w(p_{dead-x}) = 10^{x-1}.$$

3.1.8 Move Ordering

To achieve the best result with NegaScout, a good move ordering heuristic is indispensable. Hence, we define a seperate function to evaluate a move's heuristic value. This heuristic value represent if this move is desirable by either player, by counting patterns on only the four directions that go through the cell. Specifically, if we are inspecting a move (x,y), then we retrieve the four arrays:

[(x-4,y), (x-3,y), (x-2,y), (x-1,y), (x,y), (x+1,y), (x+2,y), (x+3,y), (x+4,y)]
[(x,y-4), (x,y-3), (x,y-2), (x,y-1), (x,y), (x,y+1), (x,y+2), (x,y+3), (x,y+4)]
[(x-4,y-4),(x-3,y-3),(x-2,y-2),(x-1,y-1),(x,y),(x+1,y+1),(x+2,y+2),(x+3,y+3),(x+4,y+4)]
[(x-4,y+4),(x-3,y+3),(x-2,y+2),(x-1,y+1),(x,y),(x+1,y-1),(x+2,y-2),(x+3,y-3),(x+4,y-4)]
(These are arrays of size 9. If (x,y) is near the border, the array might not be of full-size)

For each of these directions, we check all of its subarrays of size five and count the number of our stones and the enemy stones. The heuristic value of one subarray is defined as:

$$V(subarray) = \begin{cases} 0 \text{ if in subarray there are both our stones and enemy's stone} \\ Y[\#ours] \text{ if there are only our stones} \\ E[\#enemy's] \text{ if there are only enemy's stone} \\ 10 \text{ if there are no stones} \end{cases}$$

where, $Y = \{0, 35, 800, 15000, 800000\}, E = \{0, 15, 400, 20000, 100000\}$. The heuristic value of a move is the sum of value of all subarrays across all directions.

4 Results

4.1 Benchmarking

4.1.1 Depth Evaluated

We consider the depth that our algorithm can reach in a certain amount of time in sample games versus a human player.

Time/Move	1 - 4	5 - 8	9 - 12	13-16
1s	6	5.5	4	4
2s	6	6	4	4
5s	6	6	5	4

The more stones are on the board, the shallower our algorithm could reach. This is due to the search space increases as the playing field increases in size. Typically, the AI will take all of its time to calculate for a move, and would generally begin to search at depth = 6.

4.1.2 Effectiveness of Transposition Table & State Table

We also consider the number of nodes that we have to evaluate, and the number of lookups from transposition table and state table from this sample game (time limit set to 2s per move):

Move	1	2	3	4	5	6
# Evaluated Nodes	404115	749718	1112458	1470052	1852018	2383451
# TT hash hits	31224	75822	134075	190354	265391	418070
# ST hash hits	3873	10867	13991	22306	29160	31741
% hits	8.68%	11.56%	13.30%	14.46%	15.90%	18.87%

Through the data, we could see that the percentage of hash hits ranges from 8% to 20% and increases as more nodes are evaluated.

4.1.3 Effectiveness of move ordering

We benchmark the effectiveness of move ordering by observing the number of cutoffs in each case: with and without move ordering. In these 2 sample games (time limit set to 1s per move), the result shows with great certainty that with our move ordering heuristic, the NegaScout algorithm produces more cutoff, as expected.

Move	1	2	3	4	5	6	7	8
# with move order	1643	2372	3583	4886	6108	7610	9624	11485
# w/o move order	382	1364	1928	2027	2052	2244	2478	2773

4.2 Versus AIs

We upload our AI to Piskvork Gomoku Manager (Petr Lastovicka) [12] to provide a graphical user interface for easier testing process.

The AI showed better results than we originally expected. We downloaded 6 AIs [13] on gomocup.org: Mushroom, Valkyrie, PureRocker, Pisq7, Noesis and Pela, ranked from low to high, and one AI from Yuqing Zhang's Github [14] (we refer to this AI as aaazyq's), whose YouTube video inspired us to take on this project. We played against each opponents 20 times; 10 as Black and 10 as White, and the time limit was set to 5 seconds per move.

AI name	Our wins
Mushrooms	20(10+10)
Valkyrie	20(10+10)
aaazyq's	20(10+10)
PureRocker	20(10+10)
Pisq7	17(9+8)
Noesis	2(1+1)
Pela	0

Our results against the 7 AIs, denoted by TotalWins(WinsWithBlack + WinsWithWhite).

4.3 Versus Human

Our group also tried to play with it and would occasionally lose to it. In this game, we played X and our AI played O. The AI took the chance when we blunderingly overlooked the straight three and wins.

5 Conclusion

5.1 Implications of Our Results

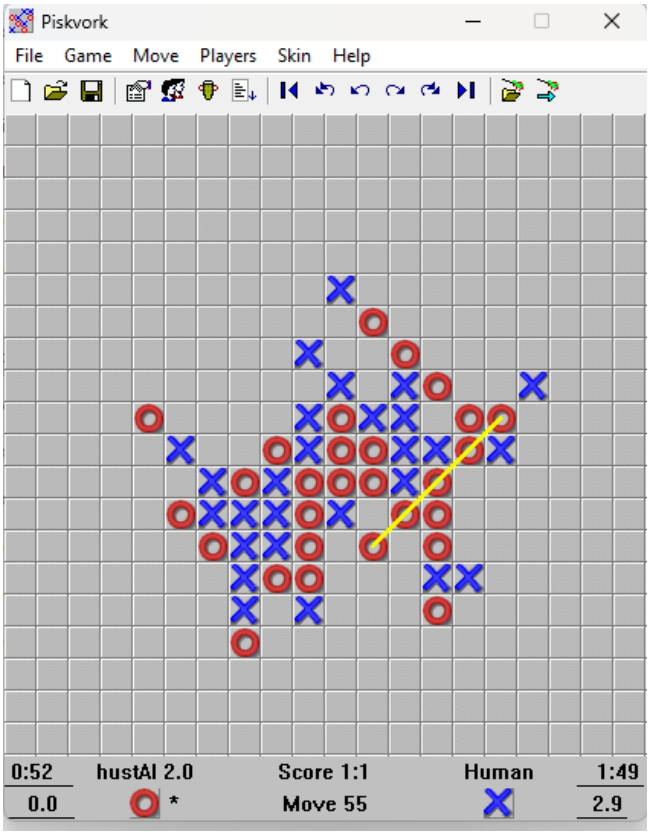
For low-level AIs, our AI had a clear edge over them. However, as we went up the ranks, stronger mid-level AIs such as Noesis and Pela showed their advantage: even when we allowed the time limit to 10 seconds per move, they usually do not need more than 2 seconds. We did not bother to test our AI against top-level ones.

Upon analysing our games, we do notice that our AI often overlooks defensive moves, therefore sometimes it loses unexpectedly. It also happened that there was a bug in our restriction array that makes our AI played clumsily near the edge of the board. We fixed that and then tinkered with the parameters ourselves, and did get slightly better results.

5.2 Our Limitations

5.2.1 Choice of Programming Language

Our first idea was to code our project in Python since everyone was familiar with it and we saw that several AIs were implemented in Python so it was doable. Hence, we have developed our AI in Python and have succeeded to achieve a functional one, with our own GUI implemented on tkinter.



However the performance was not good enough; the program was really slow so we switched in C++ our project to improve it. The result is that, for the same depth (when we have not implemented iterative deepening), the running time per search is roughly 400 times faster in C++ than in Python.

5.2.2 Choice of Variant

Originally, in the project proposal, we set out to implement the Standard Variant of the game together with Swap2[15] tournament rule. However, we encountered some minor problems that make us cannot deliver these features on time, namely:

- We had trouble adapting our evaluation function to fit with the Standard Variant, and we do not have a deep enough understanding about special defensive strategies of the variant; ultimately we chose to go back to the easier Freestyle Variant instead.
- For the Swap2 opening rule, we had retrieved a set of balanced opening from an opening book[16] , but there are only a number of very strong AIs have this feature implemented, and not AI from our level. We figured it would be uneasy for us to test and assess this feature, thus decided to drop it.

5.3 Future Improvements

Of course, there are much room for improvements for our AI. We could either implement enhancements on our search algorithms, such as MTD(f), SSS* Search, etc., or improve further our evaluation function and add additional support for different variants.

5.4 Final Remarks

This is the first time we work as a team together in HUST, and even with an exchange student. The project has provided us with not only hands-on experience on coding an actual, working intelligent agent, but also allowed us to learn new things, e.g. do citation. In the process of teamwork, miscommunication and mistakes is unavoidable; nevertheless, the result in the end is satisfying.

References

[1] "Detail Information," https://gomocup.org/detail-information/, [Online; accessed 2023-01-02].

[2] "Caro (aka Gomoku)," https://learnplaywin.net/caro/, aug 26 2016, [Online; accessed 2023-01-02].

[3] L. Allis, "Searching for solutions in games and artificial intelligence," [Online; accessed 2023-01-02].

[4] J. Wágner and I. Virág, "Solving RENJU," *ICGA Journal*, vol. 24, no. 1, pp. 30–35, mar 1 2001, [Online; accessed 2023-01-02].

[5] "Monte-Carlo Tree Search," https://www.chessprogramming.org/Monte-Carlo_Tree_Search, [Online; accessed 2023-01-02].

[6] H. Tsan-Hseng, "Alpha-Beta Pruning: Algorithm and Analysis," https://homepage.iis.sinica.edu.tw/~tshsu/tcg/2021/slides/slide6.pdf, [Online; accessed 2023-01-02].

[7] J. Pearl, "Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties," *Proceedings of the First Annual National Conference on Artificial Intelligence*, pp. 143–145, [Online; accessed 2023-01-02].

[8] A. Reinefeld, "An Improvement to the Scout Tree Search Algorithm," *ICGA Journal*, vol. 6, no. 4, pp. 4–14, dec 1 1983, [Online; accessed 2023-01-02].

[9] "Iterative Deepening," https://www.chessprogramming.org/Iterative_Deepening, [Online; accessed 2023-01-02].

[10] H. Tsan-Hseng, "Transposition Table, History Heuristic, and other Search Enhancements," https://homepage.iis.sinica.edu.tw/~tshsu/tcg/2021/slides/slide8.pdf.

[11] qwertyforce, "Github - qwertyforce/gomoku_ai_c: gomoku_ai in C++," https://github.com/qwertyforce/gomoku_ai_c, [Online; accessed 2023-01-02].

[12] "Download Gomocup Manager," https://gomocup.org/download-gomocup-manager/, [Online; accessed 2023-01-02].

[13] "Download Gomoku Artificial Inteligence (AI)," https://gomocup.org/download-gomoku-ai/, [Online; accessed 2023-01-02].

[14] aaazyq, "Github - aaazyq/Gomoku: Gomoku in MiniMax with a-b pruning." https://github.com/aaazyq/Gomoku, [Online; accessed 2023-01-02].

[15] "Gomoku - Swap 2 Rule," https://www.renju.net/rule/11/, [Online; accessed 2023-01-02].

[16] "15x15 Gomoku Opening Book," https://www.crazy-sensei.com/book/gomoku_15x15/, [Online; accessed 2023-01-02].