Theory of Computer Game

Name: Warron Yiang Wai Loon
Student Number: B09902078

Due Date: 6th Jan 2023
Subject: Final

Data Structure

Modified MyAI

Since this final homework has provided a sample code, with fully implemented basic function of the AI agent. Thus, I decided to use it with modified and added some function to handle the algorithm/heuristic implemented in this homework.

MyAI extra attribute

In order to handle transposition table and Evaluation heuristic function, I decided to added two extra attribute below:

```
class MyAI
{
        //...
        //...
private:
        // ...
        // added two array to record the position of the piece
        int red_cube[PIECE_NUM];
        int blue_cube[PIECE_NUM];
        // ...
        // added two key for transposition table
        uint64_t key1; // blue piece keys
        uint64_t key2; // red piece keys
        // ....
        // added functions of search
        bool isEnd();
        double eval(const int my_color);
        std::pair<double, int>F4( double alpha, double beta, int depth,
                        const int limit_depth, const int my_color );
        std::pair<double, int>G4( double alpha, double beta, int depth,
                        const int limit_depth, const int my_color );
        std::pair<double, int>Star1_F4( double alpha, double beta,
                int depth, const int limit_depth, const int my_color );
        std::pair<double, int>Star1_G4( double alpha, double beta,
                int depth, const int limit_depth, const int my_color );
        std::pair<double, int>IDAS( const int limit,
                                    const double threshold );

        // Modified Basic functions
        int Take_move(const int piece, const int start_point,
                                const int end_point);
        void Undo_move(const int piece, const int start_point,
                                const int end_point, int enemy_piece);
        void setup_state();        // setup hash + attribute
};
```

Take move and Undo move

As the original Make move doesn't record/store the history situation, we are unable to undo a move in constant time due to lack of history information. However, each recursion in search will modified the object we used, which cause incorrect when we retrograde back to the original call function and do another recursion. There is two possible way to handle this:

1. Create a new object and copy all the attribute from the parent to it, and use this new created object to do the recursion search. This method takes $O(sizeof(MyAI))$
2. Implement new make move with recorded history information and use this information to do the undo move. This method takes $O(1)$.

Obviously, the second method is way better than the first method, as we will do huge of the search, it will be waste of time and space if we keep create the object and do the copy action in each recursion search. Thus, I decided to implement the second method.

Basically, take move do the same thing in the make move, but to maintain the key and the array which recorded position of piece to ensure we holds the correct information. In extra, we have to return the enemy piece which captured by us in that move, if we don't captured any pieces, then just simply return -1.

Then, undo move take the enemy piece from the return value of take move, and recover the enemy piece back if necessary. Same, we have to maintain the key and the array which recorded position of pieces to ensure we holds the correct information when undo-ing the move.

Setup State

This function is to re-initialize the object based on the board situation. That is, it re-compute the key, position of cube, and other information in MyAI class.

Transposition Table

This homework required us to implement the transposition table.

Transposition table attribute

```
typedef struct node{
    uint64_t key1;          // blue piece key
    uint64_t key2;  // red piece key
    int dice;       // the dice number
    int depth;      // the node's depth
    int flag;       // to record the value is exact value
                  // or lower bound
                  // or upper bound
    bool valid;                // record the entry is valid or not
    std::pair<double, int>value; // the value and the best move
}node;
```

Handle Collision?

When designing the transposition table/hash table, the collision problem is the first factor to considered in the design. There exists many probing method for resolving collisions in hash table/transposition table.

In my design, I choose to implement Double Hashing in Open addressing with limited trials. The function to compute the index with the provided keys as below:

$$index = h(key, i)$$

where h is the hash function defined as following:

$$h(key, i) \equiv [h1(key) + i * h2(key)]\ MOD\ m$$

where m denote the size of transposition table, and the $h1, h2$ is two different hash function defined as follow:

$$h1(key) \equiv key\ MOD\ m$$

and

$$h2(key) \equiv (R - key)\ MOD\ m$$

where R be the random fixed prime with $R < m$. Note that key is defined as following:

$$key = (key1 \oplus key2) \oplus pr\_dice[dice]$$

In traditional design of double hashing, its will keep increment the value i until found the empty space or realising the hash table is full, which might make the search time increased. In this case, I decided to limit the trials, and do the Replacement Policy to handle the worse case.

Replacement Policy?

First, we use the h hash function mentioned above with the limited trials to find the empty space. If there is a empty space, then just store the information in the empty space and mark it as valid.

However, if the worse case happend, which doesn't have any empty space within the limited trials, then we have to pick the victim to replace the data. The way to choose victim is to find the entry with the most lowest depth (which is most nearest to root) to be the victim.

Unique key?

There exists probability that both different positions meet the same index, and we have to identify them by using key or comparison. If we use the comparison method to identify the positions, the time complexity will become $O(N)$, where N is the size of the board. This might effect the performance as we have to lookup the table in each recursion search and might waste the space to store the board.

In my case, I just simply uses both key1 and key2 to identify the positions. However, with using this method, I have to ensure the uniqueness of either key1 or key2. Thus, I wrote the script which brute force the permutations of [0,6] pieces in 25 different position with random value in zobrist hashing. It is brute forceable as there is only $25P6 + 25P5 + 25P4 + 25P3 + 25P2 + 25P1 + 25P0 = 134205626$ iterations. After we found the list of random value in zobrist hashing which mapping the all permutations of [0,6] pieces in 25 different position to different values, we take it to the design.

Combine with NegaScout

This part is implemented according to the course. Note that we can only uses the information store in the entry if and only if the entry holds the depth with not shallower.

```
F4(...){
        // Look up table here
        const uint64_t fixed_key1 = this->key1;
        const uint64_t fixed_key2 = this->key2;
        const int fixed_dice = this->dice;
        const int fixed_color = this->color;
        const int idx = lookup_entry(fixed_key1, fixed_key2,
        fixed_dice, fixed_color);
        if( idx != -1 &&
        hashtable[fixed_color][idx].depth >= (limit_depth - depth) ){
                debug_use_hash++;
                if( hashtable[fixed_color][idx].flag == EXACT ){
                        return hashtable[fixed_color][idx].value;
                }
                else if( hashtable[fixed_color][idx].flag == LOWER ){
                        alpha = std::max( alpha,
                        hashtable[fixed_color][idx].value.first );
                }
                else if( hashtable[fixed_color][idx].flag == UPPER ){
                        beta = std::min( beta,
                        hashtable[fixed_color][idx].value.first );
                }

                if( alpha >= beta ) {
                        return hashtable[fixed_color][idx].value;
                }
        }
        // NegaScout below...

        for(int i = 1; i < move_count;i++){
                //....
                if( m >= v_max || m >= beta ) {
                        update_entry( idx, fixed_key1, fixed_key2,
                        fixed_dice, (limit_depth - depth), LOWER,
                        res, fixed_color );
                        return res;
                }
                //...
        }
        if( m > alpha )
                update_entry( idx, fixed_key1, fixed_key2,
                fixed_dice, (limit_depth - depth), EXACT,
                res, fixed_color );
        else
                update_entry( idx, fixed_key1, fixed_key2,
                fixed_dice, (limit_depth - depth), UPPER,
                res, fixed_color );
        return res;
}
```

Note and Benefit

The reason where I choose to implement Double Hashing in Open addressing is due to double hashing is the one which closer to uniform probing compare to another probing method. We always hope to use the transposition table fully without wasting any space, and ensure the lookup and insert operations performances at the same time. That is, with using Double Hashing probing method, and limit the trials of collisions, we are able to take the benefit from balancing the lookup and insert operations performance.

Note that even I design the replacement policy to handle the worse case, there is very rare that to happend the worse case in real cases. With the searching depth 7 and using transposition table with the design mentioned above, the worse case, which exceed the limited trials, is just about 0.0005% of total lookup operations. That is, with the design above, we can ensure that the utilization of transposition table is at the good rate as the target.

After using the transposition table, the program are able to search up to 7 depth from only 5 depth maximum, which is the huge improvement for the search algorithm.

Algorithm and heuristic function

NegaScout + Star1

NegaScout is the main search algorithms of this homework. Star1 is the search algorithms to handle searching the chance nodes. This part is basically implemented by following the pseudo codes provided in the pdf on the courses. However, the psuedo code provided is based on F3, which is Fail Soft version Alpha beta, so we have to modified it to become NegaScout version.

Since I am more comfortable and more understand the concept of NegaScout in alpha beta version (not the negamax), so I decided to implement NegaScout with alpha beta version.

Note that since every nodes after moving according to the dice is chance node, this implies that all of the recursion next search will be Star1 searching algorithm in NegaScout searching algorithm, which is different to chinese dark chess (example provided at course) as chinese dark chess has deterministic move.

```
Algorithm F4(position p, value alpha, value beta, integer depth){
        determine the sucessor positions p1,...pb
        if( b == 0 or depth == 0 or time is running up or is terminal )
                return f(p)
        m = -INF
        // the first branch with Star1 Chance node searching
        m = max(m, Star1_F4(p1, alpha, beta, depth - 1))
        if(m >= beta)
                return (m)
        for(i = 2;i <= b;i++){
                // Null window search
                t = Star1_F4(pi, m, m + 1, depth - 1)
                if( t > m ){
                        if( depth < 3 or t >= beta )
                                m = t
                        else{
                                // re-search
                                m = Star1_F4(pi, t, beta, depth - 1)
                        }
                }
                // beta cut-off
                if( m is max possible or m >= beta ) return m;
        }
        return m;
}
```

```
Algorithm G4(position p, value alpha, value beta, integer depth){
        determine the sucessor positions p1,...pb
        if( b == 0 or depth == 0 or time is running up or is terminal )
                return f(p)
        m = INF
        // the first branch with Star1 Chance node searching
        m = min(m, Star1_G4(p1, alpha, beta, depth - 1))
        if(m >= beta)
                return (m)
        for(i = 2;i <= b;i++){
                // Null window search
                t = Star1_G4(pi, m - 1, m, depth - 1)
                if( t < m ){
                        if( depth < 3 or t <= alpha )
                                m = t
                        else{
                                // re-search
                                m = Star1_G4(pi, alpha , t, depth - 1)
                        }
                }
                // beta cut-off
                if( m is min possible or m <= alpha ) return m;
        }
        return m;
}
```

Note that pseudo code above doesn't shows the transposition table part, the details will be in the code submitted.

Benefit

By using the property of Failed-High and Failed-low in alpha beta, we can do a null window search first to determine the result to be either failed-hight or failed-low, then using the result to adjust the searching windows to be more smaller, which will improved the searching speed as the alpha beta performances also depends on the size of searching windows.

Besides, NegaScout results better when the transposition table is used, as each null windows search will also store the cut off information inside the transposition table, or even store the exact value inside the transposition table, which reduce the number of re-searching.

Iterative Deepening with Aspiration Search

This is the heuristic to improve the NegaScout. The concept is to do the NegaScout search in iterative deepening which starts with limiting the depth in 3 first. Then, using the results return from the first NegaScout search as the consideration of next NegaScout search, with smaller searching windows compare to the original searching windows.

However, if the results return from the NegaScout is out of the windows (either Failed-low or Failed-high), we have to do the re-search again, also with the return value back from the previous NegaScout search to be the searching window bound.

Note that we have to increased the limit depth by 2 in each NegaScout search, until the limited depth of Iterative Deepening is reached or the remaining time is not enough for the next search. The reason that to increased the limit depth by 2 as we hope to do the NegaScout search with the limit depth of odd value, as even depth will be too optimistic for the root player which might results in bad effect.

Pseudo code

```
Algorithm IDAS(position p, integer limit, value threshold){
        best = F4(p, -INF, INF, 3);
        current_depth_limit = 5;
        while(current_depth_limit <= limit){
                m = F4(p, best - threshold, best + threshold,
                current_depth_limit);
                // Failed-low
                if( m <= best - threshold ){
                        m = F4(p, -INF, m, current_depth_limit);
                }
                else if( m >= best + threshold ){
                        m = F4(p, m, INF, current_depth_limit);
                }
                best = m;
                if( remaining time is not enough ) return best;
                current_depth_limit = current_depth_limit + 2;
        }
        return best;
}
```

Benefit

Since I have used the transposition table in my design, the Aspiration search with iterative deepening will be better and faster than the normal NegaScout as the information stored in transposition table in the previous search with lower depth can be reused in the next search.

Besides, we ‘guess’ the searching result to be lied in the threshold searching windows to speed up the searching. If we ‘guess’ correctly with the previous result, we are able to obtain the benefit which has better performance than the normal NegaScout search. However, even if we failed to ‘guess’, we can do the re-search again with trivial penalty as the information of the previous search can be reused again.

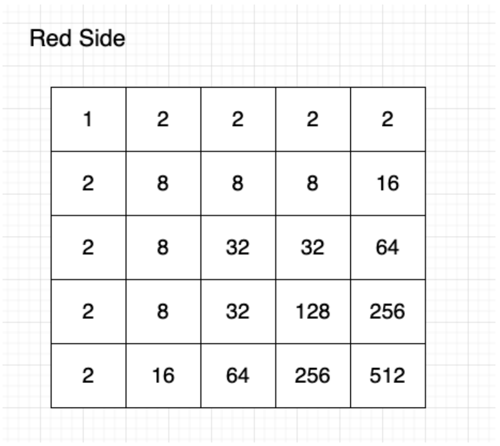
Evaluation heuristic

The evaluation heuristic is also the main part of this homework as the NegaScout required the evaluation score when we cannot effort to completely search all the game tree. We can win EWN in two way, which is:

1. One of the pieces reached the end first.
2. Captured all of the enemy pieces.

Note that its different with homework 2, which we can win the game with ANY pieces reached the end first, so the material values of each pieces will not different in too much. However, if the pieces is being captured, its material value will ‘transfered’ to the movable pieces when the dice is roll to the captured piece in EWN. Therefore, the amount of pieces should not be the main consideration in the evaluation heuristic as there might be more benefit when there is less pieces alive, as the probability that the target piece can be moved will be higher.

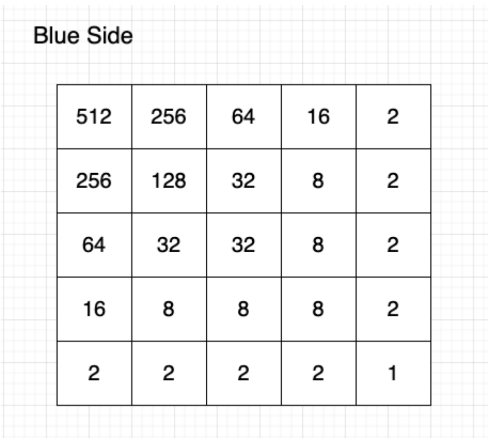
Therefore, our main focus is on designing the material values. I decided to defined the material values based on the positions of the pieces. Each pieces can gains the following values if it located in the corresponding position:



Red Side

1	2	2	2	2
2	8	8	8	16
2	8	32	32	64
2	8	32	128	256
2	16	64	256	512

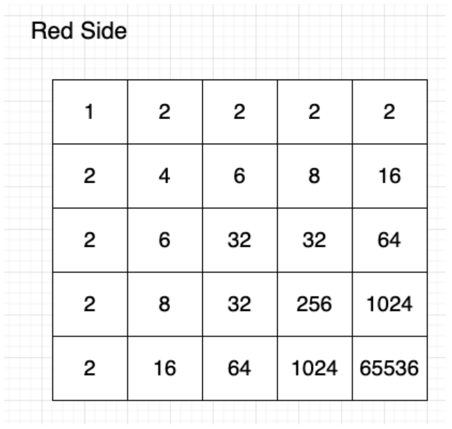
Figure 1: The evaluation of Red side


Blue Side

512	256	64	16	2
256	128	32	8	2
64	32	32	8	2
16	8	8	8	2
2	2	2	2	1

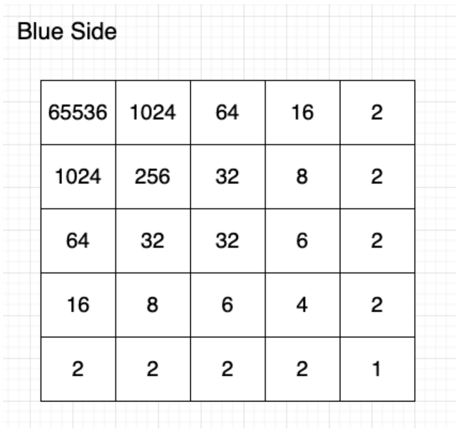
Figure 2: The evaluation of Blue side

Besides, I also design another table for the evaluation with more aggressive style:


Red Side

1	2	2	2	2
2	4	6	8	16
2	6	32	32	64
2	8	32	256	1024
2	16	64	1024	65536

Figure 3: The red side


Blue Side

65536	1024	64	16	2
1024	256	32	8	2
64	32	32	6	2
16	8	6	4	2
2	2	2	2	1

Figure 4: The red side

Except focus on material values, I decided to add the threat values to be the part of evaluation score. The threat values that each pieces obtained will be the maximum of the enemy piece's material values if the piece can capture the corresponding enemy pieces.

However, the total threat values should be lower than total material values as the probability with winning as reached the end first is higher than captured all of the enemy pieces (which is more difficult).

In conclusion, the evaluation score will defined as:

$$var = \sum_{i=1}^{6} Score(p(i)) + \sum_{i=1}^{6} Threat(p(i)) * c$$

where the $p(i)$ denote the position of the pieces and $Score(p(i))$ be the material values obtained for the piece based on its positions. The $Threat(p(i))$ here denoted the threat values obtained for the piece based on its positions, and the c be the constant value to control the play style. The bigger c will more focus on defense, and smaller c will more focus on attacking.

Note that when the i piece is captured, $p(i)$ will be $max\{p(j)\}$ where j is the moveable pieces if the dice rolls on the captured pieces.

Time control

Since the program cannot search up to 9 depth, I decided to use a limited depth of 7 as the time control of each plys. I have tried to allocate the time to each plys with aspiration search iterative deepening, but always reuslts that it only can search up to 7 depth and always time out at 9 depth. Furthermore, the searching result of 9 depth cannot be used as all of the branch are incompletely searched.

Therefore, I decided to just set the limited depth to 7 of aspiration search iterative deepening and do the 'best effort' to do the NegaScout.

Experiment Result

Version Log

```
ver0 -> F3 (Fail hard) + Star0 + Evaluation + Fixed Depth
ver1 -> F3 (Fail hard) + Star1 + Evaluation + Fixed Depth
ver2 -> NegaScout + Star1 + Evaluation + fixed depth
ver3 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
ver4 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
ver5 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
ver6 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
ver7 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table + Aspiration search
--- Optimized function ----
ver8 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
ver9 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
ver10 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table + Aspiration search
ver11 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
--- DAY1 ----
ver12 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table
--- Optimized function ----
ver13 -> F3 + Star0 + Evaluation + fixed depth + Take_move
                + Transposition table (BUG)
ver14 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table (fixed bug version of ver12)
--- 7 depth ---
ver15 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table + Referee improved
                + Transposition Bug fixed
ver16 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table + Referee improved
                + Aspiration search
ver17 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
                + Transposition table + Referee improved
--- DAY2 ---
ver18 -> NegaScout + Star1 + Evaluation + fixed depth + Take_move
```

```
                + Transposition table + Referee improved
                + Aspiration search
ver19 -> NegaScout + Star1 + Evaluation Aggresive + fixed depth
                + Take_move + Transposition table + Referee improved
                + Aspiration search
```

Local Versus Result

The following Experiment will let both AI play up to 100 rounds.

Ver12 (depth 6) vs Ver0 (depth 5)

1. 47W - 53L

Ver12 (depth 6) vs Ver3 (depth 5)

1. 52W - 48L

Ver18 (depth 7) vs Ver0 (depth 5)

1. 67W - 33L

Ver18 (depth 7) vs Ver3 (depth 5)

1. 79W - 21L

Ver18 (depth 7) vs Ver12 (depth 6)

1. 84W - 16L

Ver18 (depth 7) vs Ver19 (depth 7)

1. 54W - 46L

The results above shows that my version 12 program has bug which make the difference after implemented those heuristic and transposition table doesn't make any improvement to star0 version and F3 alpha beta search version, as they all search only up to 5 depth or 6 depth. Besides, searching up to 6 depth seem to be optimistic for root player(which is us), which make the situation more worse.

However, after fixed those bug and make version 18 bug-free, the program seem to be has a lot of improvement as its is able to search up to 7 depth. The results also shows that searching more depth will make the AI more powerful.

The last Experiment is the interesting found, which shows that the aggressive evaluation heuristic seem to be not better than version 18 in most case.

Competition Versus Result

Day1

Since it become practice day due to server bug, I only play to two players and have the results:

1. 4W - 2L
2. 2W - 4L

This day using the version 12 which has the transposition table bug and unoptimized Referee function. These bug and optimization will be done in ver18 completely.

Day2

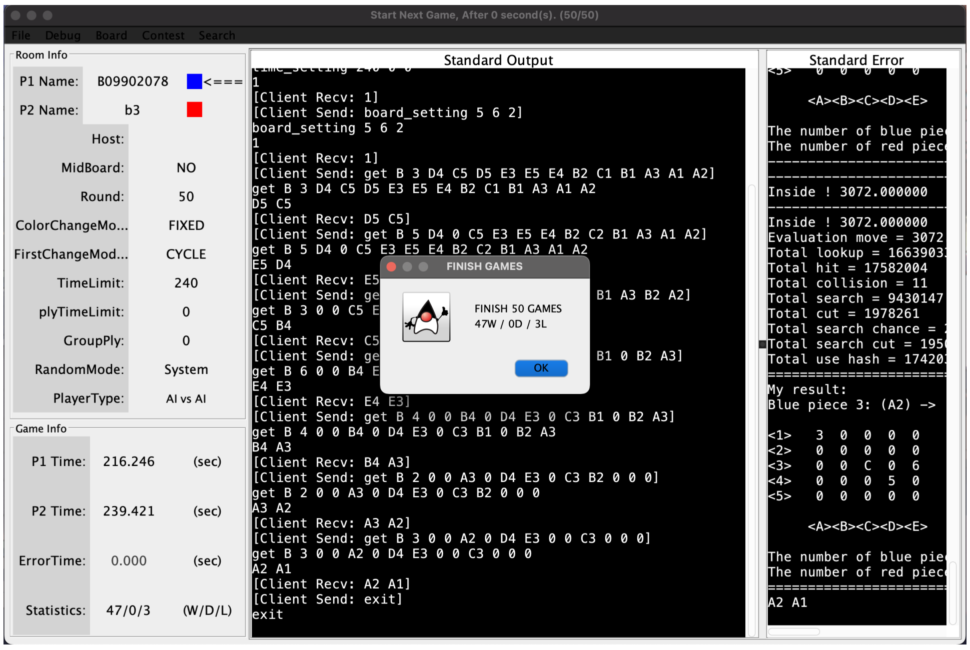
This day using the version 18 which has fully implemented and bug free. After fixed those bug mentioned in Day1, the program successfully able to search up to 7 depth, which is a huge improvement and finally get Rank 4 in last day!

I have total played 5 rounds which has the results of

1. 4W - 2L
2. 4W - 2L
3. 2W - 4L
4. 5W - 1L
5. 4W - 2L

It was fun and happy when watching my program finally bug-free and able to win most of the student in the class, which give me a lot of confidence. However, I should able to win more as the bug make me loses two round in second round QQ.

The following will be the result of versus AI:



P1 Name: B09902078 🟦 <===
P2 Name: b3 🟥
Host:
MidBoard: NO
Round: 50
ColorChangeMo... FIXED
FirstChangeMod... CYCLE
TimeLimit: 240
plyTimeLimit: 0
GroupPly: 0
RandomMode: System
PlayerType: AI vs AI

P1 Time: 216.246 (sec)
P2 Time: 239.421 (sec)
ErrorTime: 0.000 (sec)
Statistics: 47/0/3 (W/D/L)

FINISH 50 GAMES
47W / 0D / 3L

```
1
[Client Recv: 1]
[Client Send: board_setting 5 6 2]
board_setting 5 6 2
1
[Client Recv: 1]
[Client Send: get B 3 D4 C5 D5 E3 E5 E4 B2 C1 B1 A3 A1 A2]
get B 3 D4 C5 D5 E3 E5 E4 B2 C1 B1 A3 A1 A2
D5 C5
[Client Recv: D5 C5]
[Client Send: get B 5 D4 0 C5 E3 E5 E4 B2 C2 B1 A3 A1 A2]
get B 5 D4 0 C5 E3 E5 E4 B2 C2 B1 A3 A1 A2
E5 D4
[Client Recv: E5
[Client Send: ge                                    B1 A3 B2 A2]
get B 3 0 0 C5 E
C5 B4
[Client Recv: C5
[Client Send: ge                                    B1 0 B2 A3]
get B 6 0 0 B4 E
E4 E3
[Client Recv: E4 E3]
[Client Send: get B 4 0 0 B4 0 D4 E3 0 C3 B1 0 B2 A3]
get B 4 0 0 B4 0 D4 E3 0 C3 B1 0 B2 A3
B4 A3
[Client Recv: B4 A3]
[Client Send: get B 2 0 0 A3 0 D4 E3 0 C3 B2 0 0 0]
get B 2 0 0 A3 0 D4 E3 0 C3 B2 0 0 0
A3 A2
[Client Recv: A3 A2]
[Client Send: get B 3 0 0 A2 0 D4 E3 0 0 C3 0 0 0]
get B 3 0 0 A2 0 D4 E3 0 0 C3 0 0 0
A2 A1
[Client Recv: A2 A1]
[Client Send: exit]
exit
```

```
     <A><B><C><D><E>

The number of blue pie
The number of red piec
----------------------
----------------------
Inside ! 3072.000000
----------------------
Inside ! 3072.000000
Evaluation move = 3072
Total lookup = 1663903
Total hit = 17582004
Total collision = 11
Total search = 9430147
Total cut = 1978261
Total search chance =
Total search cut = 195
Total use hash = 17420
======================
My result:
Blue piece 3: (A2) ->

<1>   3  0  0  0  0
<2>   0  0  0  0  0
<3>   0  0  C  0  6
<4>   0  0  0  5  0
<5>   0  0  0  0  0

     <A><B><C><D><E>

The number of blue pie
The number of red piec
======================
A2 A1
```

Figure 5: The result of AI round.

Compilation

```
$ make
# or go to version 18
$ cd ver18
$ make
```

References

1. https://stackoverflow.com/questions/29990116/alpha-beta-prunning-with-transposition-table-iterative-deepening
2. https://blog.csdn.net/qq_40178533/article/details/110423147
3. https://en.wikipedia.org/wiki/Transposition_table
4. https://gamedev.stackexchange.com/questions/89719/negascout-with-zobrist-transposition-tables-in-chess
5. https://en.wikipedia.org/wiki/Double_hashing
6. The pdf of courses.