Report III

09016428 SongZixing

1 Review

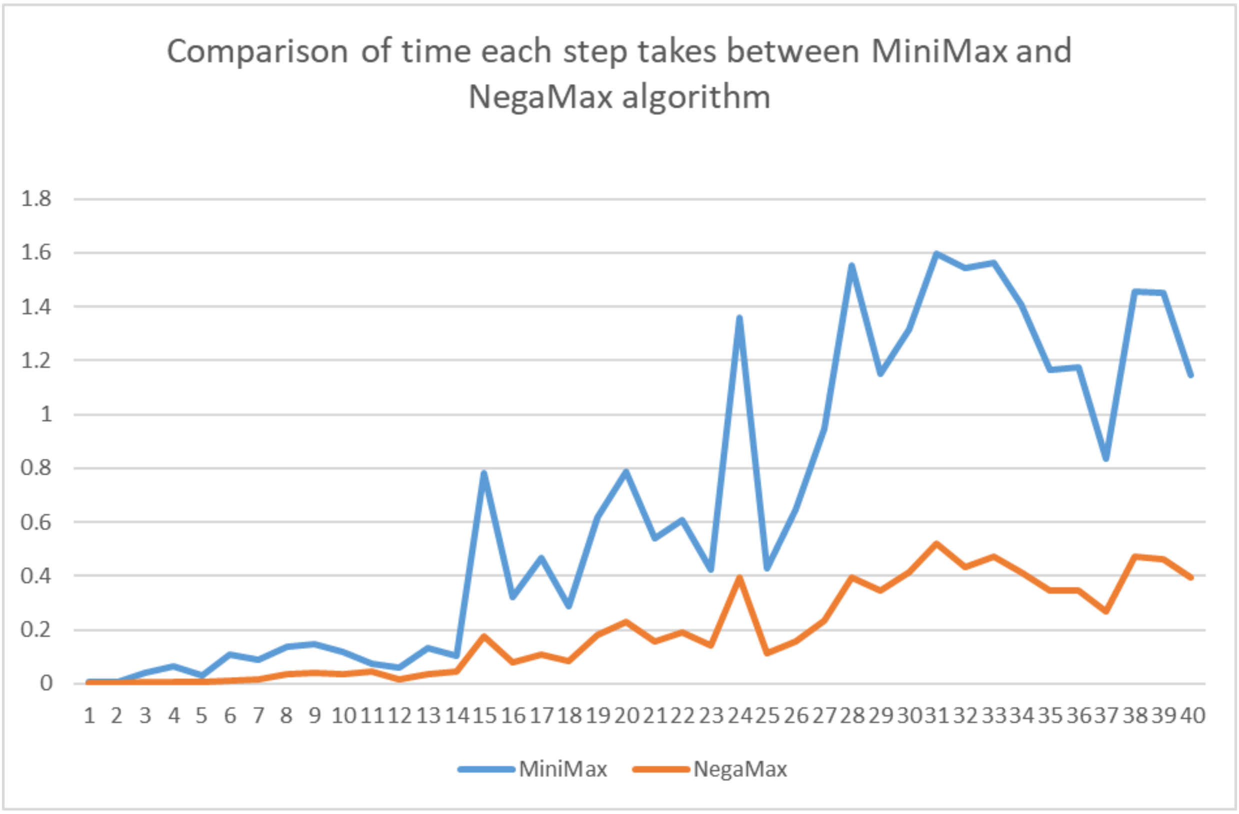
In the last report, I mainly cover the NegaMax algorithm implemented in my Gomoku AI. So I have managed to implement two main heuristic search algorithms, which are MiniMax Algorithm and Negamax Algorithm. However fail to compare these two algorithms in the term of time consumption, which is apparently critical, due to the limited time I have. During the past two weeks, I did some experiments to find which algorithm has some advantages in the term of time consumption acoording to some statistics without Mathematical prove. In the meanwhile, I further polish the alpha-beta pruning algorithm by tactfully using heuristic search algorithm, which significantly reduce the time the original algorithm took by approximately 40 percent. Finally, I focus on transposition table in order to improve the performance of AI to a higher level.

2 Comparison between NegaMax and MiniMax

The first point I want to stress is the fact that both algorithms return the same move provided that the board state is. the same. However, NegaMax takes slightly less consumption of space and equal consumption of time when compared to MiniMax bacause the former one has fewer calls of recursive functions.

To test the performance of these two algorithms, I make a comparison experiment. In this experiment, two AI engines are created separately based on the same algorithm, for example, MiniMax at first, with the same search depth of two. Then, I make them compete against each other and meanwhile the programme records the time each step of each engine takes. After that, I re-excute the programme three times to calculate the average time each step takes. Finally, I switch to the NegaMax algorithm and repeat the above-mentioned procedures.

The result is shown as follows.



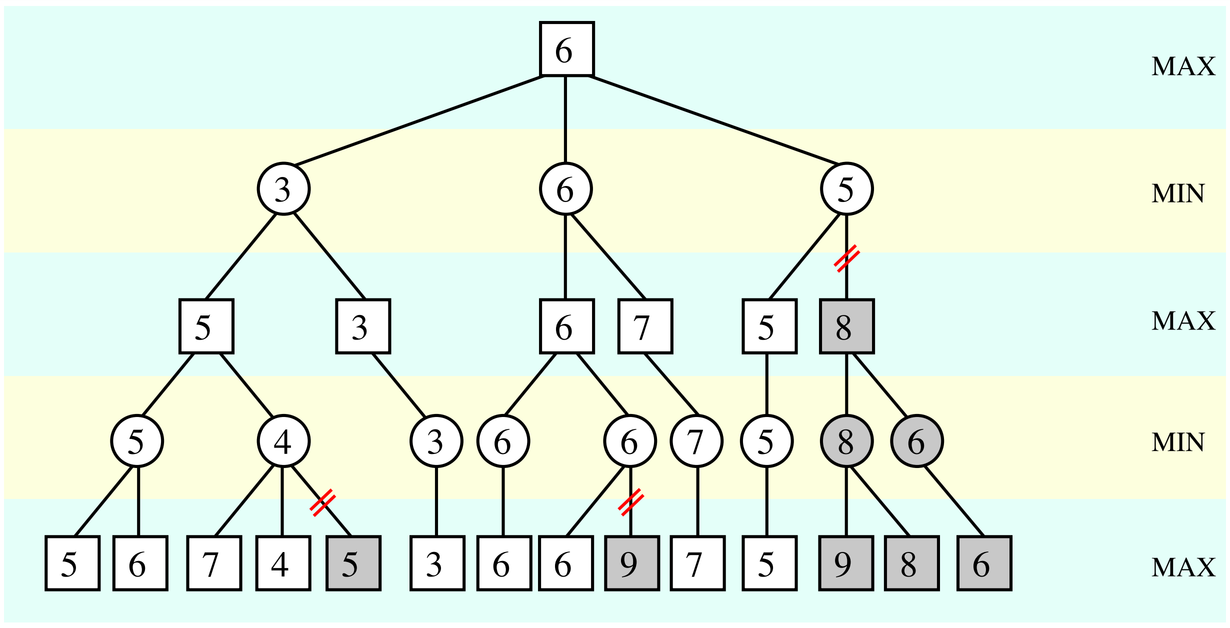
The x-axis means the number of steps and y-axis means the time(second) that each step made by AI takes. As it is shown above, we can clearly see that NegaMax does have some advantages in the term of time consumption. Furthermore, even though Negamax performs slightly better than MiniMax during the very first steps, it dramatically reduces the time consumption as the game goes on, in other words, as the number of moves the algorithm searches become more and more large.

3 Heuristic Search Algorithm

3.1 Introduction

To improve the alpha-beta pruning algorithm, I change the original structure of the search tree a bit. In the original version, each node of the same level of the search tree is placed at random. But we notice that if each node of the same level of the search tree is placed in the ascending or descending order, things will be quite different. As we all know, even if the parent node cannot be calculated ahead without getting the value of its child nodes due to the recursive function we use, the situation will change a lot if each node of the same level of the search tree is placed in order approximately.

Let's come back to the original version of alpha-beta algorithm first.



On the MIN level, the first child node is '3' and also it is the minimum value of all nodes of this level. So according to alpha-beta algorithm, the next two nodes of this level will be cut off. So this is our perfect situation. Likewise, on the MAX level, if the maximum node is place exactly the first place, all the following nodes of this level will be cut off as well. Theses two situations are the best ones.

Unfortunately, it is impossible to get the value of the node without knowing all of its child nodes. So I have to be short-sighted at first by setting the search depth to only one in order to quickly place all the nodes in ascending or descending order approximately. This could never equal to the best situation as I mentioned above, but it still turns out to be fine.

3.2 Implement

To implement heuristic search algorithm, I change the `vector<MOVE> GenerateLegalMoves();` to `vector<MOVE> GenerateSortedMoves(int color);`

And the source codes are as follows.

```
vector<MOVE> Gomoku::GenerateSortedMoves(int color)
{
    vector<MOVEwithValue> pq;

    int i, j;

    for (i = 0; i<15; i++)
        for (j = 0; j < 15; j++)
        {
            if (HasNeighbour(i,j) ){
                    MOVE newMove(i, j);
                    MOVEwithValue newMOVEwithValue;
                    newMOVEwithValue.candidate_move = newMove;
                    newMOVEwithValue.eval_value = EvaluateCandidatePoint(i, j);
                    pq.push_back(newMOVEwithValue);
            }

        }

    if (color == AIColor % 2 + 1) { sort(pq.begin(), pq.end(), compareInterval); }
    if(color==AIColor){ sort(pq.rbegin(), pq.rend(), compareInterval); }

    vector<MOVE> res; MOVE temp;
    for (int k = 0; k < pq.size(); k++) {
        temp = pq[k].candidate_move;
        res.push_back(temp);
    }

    return res;
}
```

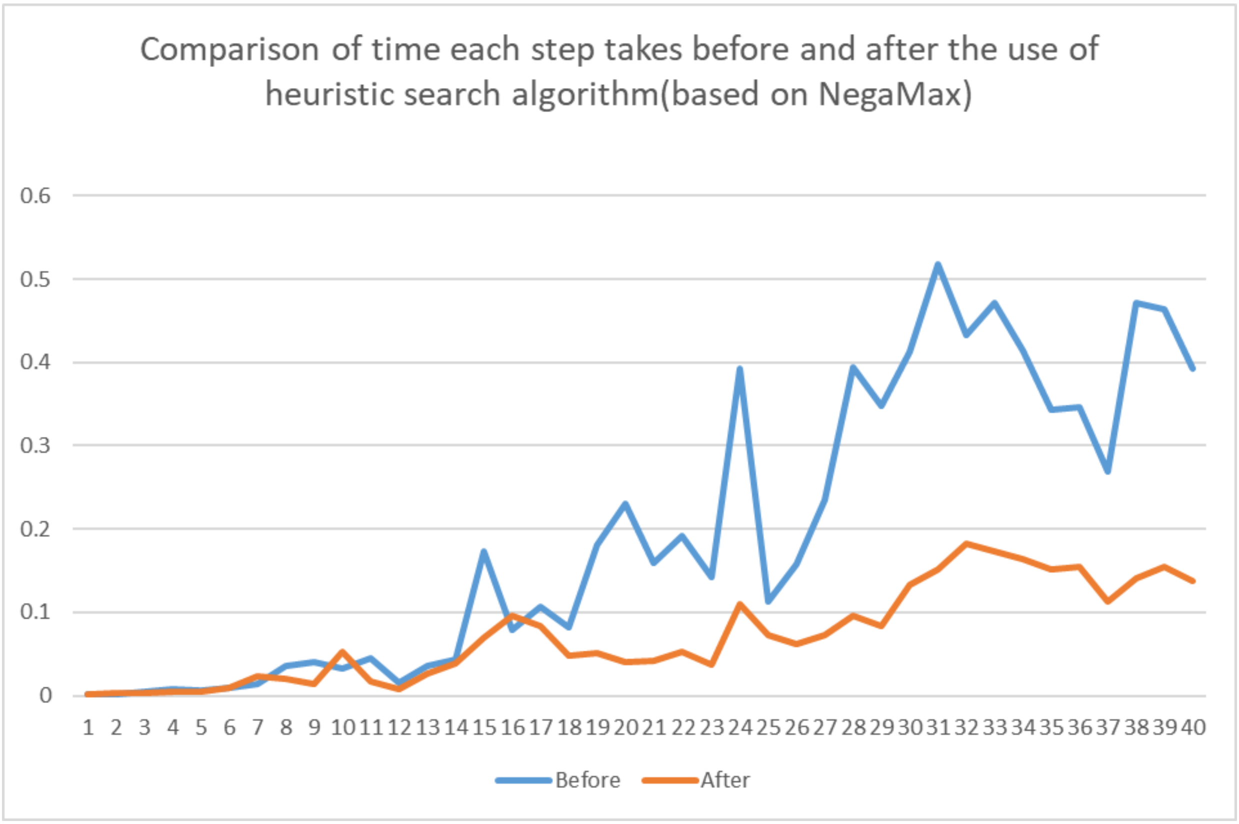
We notice that I change `double EvaluatePoint(int i, int j);` to `double EvaluateCandidatePoint(int i, int j);`. Because the former is used to evaluate the existed moves while the latter one is used to evaluate the candidate moves.

3.3 Performance Analysis

In this part, I still use experiment results to show how well the heuristic search algorithm has imporved the alpha-beta pruning algorithm without providing a rigid mathematical prove.

As part II points out, NegaMax out-perform MiniMax, so I will use NegaMax algorithm to illustrate the effectiveness of heuristic search algorithm.

In this experiment, two AI engines are created separately based on NegaMax without heuristic search algorithm (search depth is set to 2 in order to decrease the test time). Then, I make them compete against each other and meanwhile the programme records the time each step of each engine takes. After that, I re-excute the programme three times to calculate the average time each step takes. Finally, I switch to NegaMax with heuristic search algorithm and repeat the above-mentioned procedures. Also a line chart is made to illustrate the result as follows.



The x-axis means the number of steps and y-axis means the time(second) that each step made by AI takes. As it is shown above, we can clearly see that heuristic search algorithm does make some improvements in the term of time consumption. Furthermore, even though it may perform slightly better or even worse than the original version during the very first steps, it dramatically reduces the time consumption as the game goes on, in other words, as the number of moves the algorithm searches become more and more large.

4 Transposition table

4.1 Introduction

Transposition table is a database that stores results of previously performed searches. It is a way to greatly reduce the search space of a search tree with little negative impact.

When the search encounters a transposition, it is beneficial to 'remember' what was determined last time the position was examined, rather than redoing the entire search again. For this reason, we can have a transposition table, which is a large hash table storing information about positions previously searched, how deeply they were searched, and what we concluded about them. Even if the depth of the related transposition table entry is not big enough, or does not contain the right bound for a cutoff, a best move from a previous search can improve move ordering, and save search time. This is especially true inside an iterative deepening framework, where one gains valuable table hits from previous iterations.

4.2 Hash functions

Hash functions convert move positions into an almost unique, scalar signature, allowing fast index calculation as well as space saving verification of stored positions.

4.3 Address Calculation

The index is not based on the entire hash key because this is usually a 64-bit number, and with current hardware limitations, no hash table can be large enough to accommodate it. Therefore to calculate the address or index requires signature modulo number of entries, for power of two sized tables, the lower part of the hash key, masked by an 'and'-instruction accordantly.

4.4 Collisions

The surjective mapping from chess positions to a signature and an even more denser index range implies **collisions**, different positions share same entries, for two different reasons, hopefully rare ambiguous keys, or regularly ambiguous indices.

4.5 Implement

The hash table source codes are as follows.

```
enum ENTRY_TYPE{exact,lower_bound,upper_bound};
//哈希表中元素的结构定义
typedef struct HASHITEM
{
long long checksum;         //64位校验码
ENTRY_TYPE entry_type;//数据类型
short depth;          //取得此值时的层次
short eval;          //节点的值
}HashItem;
```

The following codes are still being written. It will be shown in the next report.

5 Expectations

All the data can be found in the file `Result.xlsx` and I will still focus on how to reduce the time of the algorithm takes by using some optimization algorithms to make some adjustments on alpha-beta pruning algorithm, especially transposition table.