Mini Project 3 – Mini Chess AI

`greedy_pss` Greedy + Piece Score Sum

State Value Function

Just adding up all the piece scores (positive for player's, negative for opponent's). Score of each piece is overall the same as the tie rule.

Pathfinding Algorithm

Extending 1 level from the current state, we choose the possible next state with the highest score.

Remark

Actual code choose the one with lowest score instead, since the evaluation is based on the opponent side. Such phenomenon can be contributed to the result of the localized calculation of state value.

`minimax_pss` Minimax + Piece Score Sum

Pathfinding Algorithm

With the minimax (MM) algorithm, which is basically a deeper version of the greedy algorithm, we can have a better analysis on the current state. We choose the best board for the player's round, and assumes that the opponent would choose the worse board (for the player.)

Remark

1. If we're dealing with the minimizing player at `depth=0`, the state value calculated should be negated. (The reason is similar to the greedy algorithm's negated state value.)
2. A depth-1 MM is effectively the greedy algorithm.
3. We're actually implementing the decision process with a depth-2 MM, then depth-4, and finally depth-6. Thus, if the deeper MM can't calculate the output on time, we can still have some not-so-bad move sent.
4. `baseline2` is probably a depth-2 minimax, with a state function similar to ours. Suspection is based on their similar actions.

`abprune_pss` Alpha-Beta Pruned Minimax + Piece Score Sum

Pathfinding Algorithm

Even though a deeper MM emits better result, it would take lots of time & space to compute all the possible states (depth-8 MM would run out of time). Therefore, we utilize the alpha-beta pruning (ABP), which can avoid evaluating unnecessary branches. Here we have $\boxed{\alpha}$, the maximum score possible for us, and $\boxed{\beta}$, the minimum score possible for the opponent. When $\boxed{\alpha >= \beta}$, it means that the opponent would always have a better choice than ours, which is the situation we won't take into account.

Remark

1. Similar to `minimax_pss`, here we're running four iterations of ABP, with an increasing depth (6→8→10→11). Benefit from the efficiency, we can search way deeper than `minimax_pss`.
2. As the search depth gets deeper, it's difficult to have any more improvements because of both the time limit and the limited view of our evaluation of a board. Also, even with deeper search, the outcome would probably stay same.

`abprune_hce` ABP + Enhanced Handcrafted Evaluation

State Value Function

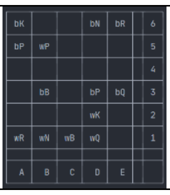
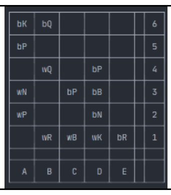
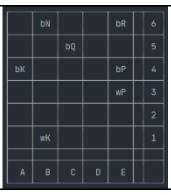
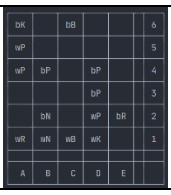
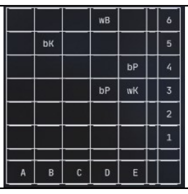
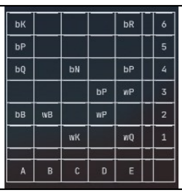
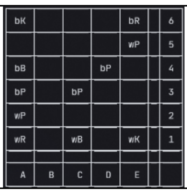
As the search tree approaches its limit, it's about time we make a better evaluation function. Here we're utilizing two other evaluation techniques on top of the original piece score sum algorithm:

1. Mobility: According to the Internet, the more possible moves you have, the more probable your victory is. We can just simply obtain mobility value from the size of `legal_actions`.
2. King Threat: The king piece is clearly the most important of all pieces, since you'd lose if your king is captured. We can start from the king, and see if any opponent pieces threats our king.

Trial Result

Here we play with mobility (MB) and king threats (KT) with various weights. The opponent is `abprune_hss`, which is a tough enemy to fight with.

*Games that reached max-step limitation are marked by an asterisk.

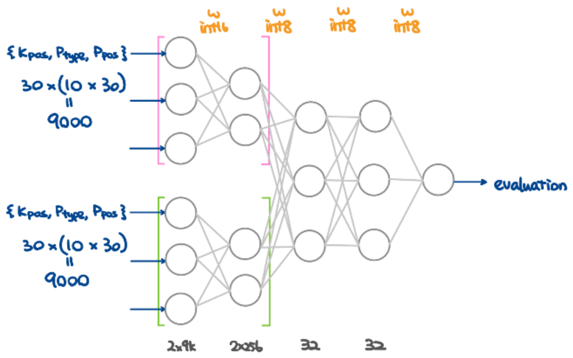
	MB/24	MB/16	MB/8	MB/2
WHITE				
	LOSE	WIN*	LOSE	LOSE
BLACK				
	WIN	WIN*	WIN*	WIN
	KT*1	KT*4	KT*16	KT*64
WHITE				
	WIN*	LOSE*	LOSE	WIN*
BLACK				
	WIN	WIN*	LOSE	DRAW*
	MB/16 + KT*1 fin. impl.	MB/2 + KT*1	MB/16+KT*64	MB/2 + KT*64
WHITE				
	WIN*	LOSE*	WIN*	LOSE
BLACK				
	WIN*	WIN*	LOSE*	LOSE*

The outcome of `MB/16` and `KT*1` is actually quite good. If we keep tuning the weight and adding other evaluation techniques, (e.g. PST, pawn patterns, etc.) we can probably get an even better outcome. Spending days and days playing with these is not so realistic, though. If only we can make computer do things for us…

abprune_nnue ABP + Efficiently Updatable Neural Network (WIP)

Originates from the Shogi game, Efficiently Updatable Neural Network (NNUE), developed by Yu Nasu, has been well known for its calculation efficiency and capability for many other games, including chess. Actually, the famous chess engine Stockfish has already implemented NNUE into its newest version, but for our mini chess game, it sure needs some modification.

As the name suggests, NNUE is basically a specialized neural network, with the capability of fast computation on CPU. The main purpose of this function is to have a better evaluation of a given game state. It has been proven that NNUE is way better than any handcrafted evaluation functions, with overall capped execution time and outstanding performance.



Network Structure

The above image is the basic structure of our modified NNUE. For the input layer, we use the HalfKP representation, which is parameterized by the king's position, another position (actually can be same as king's position for the sake of simplicity), and if any other piece is at the position, what type it is (ours or opponent's & piece type). The input would be 1, as long as the specified piece exists at the location. Even though there are better input structure existing, this is by far the easiest to implement.

The hidden layers are conceptually the same as a normal neural network, except that when implementing, the weight and bias (except the weight of the first layer) are stored as 8-bit integers, and all the neuron data are 32-bit integers. Such implementation, in addition to other optimizations which we'll talk about later, enables the super power of "efficiently updatable" of NNUE.

Acceleration Techniques

1. Incremental Computing

Thanks to the input structure of NNUE, the number of changed input nodes would be small (at most 3 if moved normal piece, at most 31 if moved king.) Therefore, with some simple matrix manipulation, we can reduce the calculation of the first layer from a 9000x256 matrix multiplication to just 3~31 row vector addition. To achieve efficient row addition, though, we have to store the matrix in a row-major structure.

2. SIMD (Single Instruction/Multiple Data)

Since matrix multiplication and vector addition is so useful and common, modern CPU usually provides hardware to compute multiple data parallelly at the same time. With the GCC compiler arguments `-maltivec` and `-mabi=altivec`, we can enable various “vector” operations, including addition and multiplication, on a fixed-byte space. For example, suppose we’re two vectors containing 8 16-bit integers, on an Intel CPU, we can use the following code:

```
#include <dvec.h>
void VectorAddition(Is8vec16 a, Is8vec16 b, Is8vec16 res) {
    res = a + b;
}
```

As shown above, with the operator overloading functionality of C++, we can treat the vectors as simple as any other primitive, without the need of memorizing numerous unreadable function calls.

Incomplete Works

1. Actually implement SIMD into the network.
2. The method and necessary functions for training the network.

`mcts` Monte Carlo Tree Search

In contrast to the MM algorithm, which does a thorough analyze until a given depth and spits out the “best” move, the Monte Carlo Tree Search method (MCTS) goes for a entirely different approach. Instead of assuming that the opponent would always choose the best possible move, MCTS does an (actually not quite) random sampling on all the possible states, and returns the best performed move according to statistics.

Sampling Process

The MCTS algorithm is basically repeating the following 4 steps, so called simulation cycle, until we run out of time, that is,

1. Selection

We first select a node to start with by a selection policy. The policy can be purely random, but we can actually utilize a better stratergy: Upper Confidence Bound (UCB). The formula for UCB of a node is `Quality + C * sqrt(log(Parent.Visit) / Visit)`, where `Quality` is the simulated win rate of the node, `Visit` is the time we've visited this node, and `C` is the exploration factor (in theory it should be `sqrt(2)`.) This, so called "exploration and exploitation" model, is the essence of MCTS.

Overall, we want to explore the node with better score (exploitation), but the possibility of other nodes shouldn't be ignored (exploration). Therefore, as the number of visits to a node increases, we are less likely to explore that node, unless it truly outperforms other possible nodes. Also, as the number of visits of the parant grows larger, the importance of exploration is decreased, thus make the final result converge to the best we can expect.

2. Expansion

After selecting the node we're exploring, we want to extend further from the selected node for further exploration. Here we're just extending one additional child node per simulation.

3. Rollout

Since we want to analyze the winning rate of a given node, the most straightforward method of sampling is to randomly "rollout" from the original state, that is, have a randomized simulation of the game, and see who wins.

4. Backpropagation

After the rollout, we would have the result of the simulation, either win or lose (or even some more sophisticated value e.g. PSS as in the implementation.) The result would then be propagate through the tree till the root, updating the nodes' data along side.

Flaws

In theory[1], if given infinite time and computation resources, the result of MCTS should eventually converge to be the same as which of MM.

[1] http://old.sztaki.hu/~szcsaba/papers/cg06-ext.pdf

Nevertheless, the actual performance of our `mcts` is not quite good, as it can't even win `baseline2` when playing as black, as shown in the Final Result section.

During the test, an interesting phenomenon is observed, though. the convergent speed of MCTS is actually quite fast – it generally stablizes within 10^6 simulation cycles, although converging to a completely different result to `abprune_hss`. The reason is still unknown, but it's likely that there's some mistake in the implementation of `mcts`. The information available largely varies, even disagree on the exploration factor `C`. In fact, a source[2] even states that MCTS is "not currently very good on chess or chess-like games."

That being said, it's known that Stockfish also implements MCTS, paired with a well evaluation function. Perhaps with some better evaluation function such as NNUE, the win rate of our MCTS could be increased, but it remains unknown unless we try it. For now, the result of `mcts` is still very limited.

`mixed` MCTS + ABP Minimax

Poorly as the MCTS algorithm worked in the mini chess game, I still wanted to try it out and see its better applications. In `mixed`, we utilize both MM and MCTS, and apply them under different circumstances, hopefully we can get some different result. However, the result is not quite good (compared to `abprune_hce`,) as shown in the following section. To sum things up, our MCTS either requires a thorough re-examine, or need some more advanced evaluation function.

[2] https://homepage.iis.sinica.edu.tw/~tshsu/tcg/2016/slides/slide9_1.pdf

Final Result

*Games that reached max-step limitation are marked by an asterisk.

B1	baseline1	P3	abprune_pss
P1	greedy_pss	P4	abprune_hce
B2	baseline2	P5	mcts
P2	minimax_pss	P6	mixed

WHITE (player 1) vs BLACK (player 2)

W \ B	B1	P1	B2	P2	P3	P4	P5	P6
B1	W	B	B	B	B	B	B	B
P1	W	W	B	B*	B	B	B	B*
B2	W	W	—	—	B*	B*	W	B*
P2	W	W	W*	W	B*	B*	W	B*
P3	W	W*	W	—	B	B*	W	W
P4	W	W	W*	—	B	—	—	—
P5	W	W	W*	B	B	B	W	B
P6	W	W	W*	B*	B*	B	B	W*