Final Report

R09922057 張致強

January 27, 2022

1 Implementation

1.1 How to compile and run your code in linux.

To compile the code, run

```
$ make
```

1.2 Techniques

NegaScout

The negascout algorithm is implemented according to the lecture slides. However, one notable distinction is that when setting up the null window, a small fractional number is used instead of 1. This is because the range of the output value from evaluation is fractional numbers in the range [-2.6, 2.6]. If we use 1 as the null window, it would be too large, making the TEST part behave like a regular negamax search.

I used the inverse of the denominator of the evaluation function (i.e. the minimum possible score increment):

$$n = \max(alpha, m) + 0.00041491510090903424$$

Time control

I use the dynamic scheme introduced in lecture. I estimate the game length to be around 250-300 plys, and use this to estimate the time allowed for every ply. First, we can estimate the expected plys by the current number of plys, the number of chess alive, and a stalling penalty. The stalling penalty is to speedup the game to prevent wasting time when both players refuse to progress the game further. It is calculated as

$$StallPenalty := \text{number of consecutive plys that are not eat or flips} \tag{1}$$

$$NumPlys := \text{number of plys I played} \tag{2}$$

$$NumAlives := \text{number of alive chess} \tag{3}$$

$$N := NumPlys + NumAlives + StallPenalty \tag{4}$$

$$ExpectedPlys = \begin{cases} 125, & \text{if } N < 125 \\ 250, & \text{if } 125 \leq N < 250 \\ N, & \text{otherwise} \end{cases} \tag{5}$$

Then the time limit for the next ply is calculated as:

$$MaxPlyTime := 15000 \tag{6}$$

$$TimeLeft := \text{time left for me} \tag{7}$$

$$PlyTime = \min\left(MaxPlyTime, \frac{TimeLeft}{ExpectedPlys - NumPlys + 1}\right). \tag{8}$$

Transposition table

A transposition table is used. A chessboard with 8×4 values on it is encoded to a 128-bit integer key using the Zobrist's hash function.

The 128-bit key of the current board is first computed before the search, then the key of its descendants are computed incrementally during the search. The random numbers used in the Zobrist's hash are initialized at runtime using C++'s 64-bit Mersenne Twister engine. Each 128-bit integer is obtained by shifting a 64-bit random number and adding to another.

Next, for the underlying table storage, I used the robin-map[1], which is a C++ hash map with good performance. The table is cleared once per ply.

Each entry in the hash table consists of 1) the value, 2) the subtree depth, 3) the best/cutoff move, and 4) list of all legal moves. If the hash hit does not immediately return the value, then 4 above can be reused to skip move generation during search.

Dynamic search extension

I have implemented search extension by extending depth at the horizon when 1) number of moves less than three, 2) last move is capturing and 3) only one chess is left for this player and it is checked.

However, this is discarded in favor of quiescent search using SEE, detailed in experiments.

Aspiration search

I use aspiration search with an initial threshold of 0.005, and the window is widened 10 times into the appropriate direction each time failed-low/high occurs.

Chance search

I implemented both Star0 and Star1 for chance search. When using chance search, at most 2 flips are explored in each search PV path. When there are 2 flips in a search path, the search depth is reduced by 2.

Star1 takes the (α, β) window into account and use the chance node cutoff when appropriate. As experiment would show, Star1 is superior to Star0.

Killer heuristic

The killer heuristic table is implemented. A MaxDepth $\times 2$ table maintains 2 killer moves at each depth. More importantly, I shift the table up by 2 depth evey ply, in order to reuse the correct killer moves from previous ply's search.

[1] `https://github.com/Tessil/robin-map`

Opening

For opening, I do not use any prior knowledge. The first flip is completely random, and the rest is handled by Star1 search.

Evaluation with dark pieces

I use the evaluation from homework 2, which includes dark pieces' value in the evaluation function.

1.3 Other optimizations

Recycled previous implementation

Things recycled from previous homeworks include: The doubly-linked-list-like array structure used to maintain each color's pieces on board, which improves speed at endgame. The mobility score in evaluation. The penalty to prevent draw if winning is possible. SEE for finding quiescent position.

Pre-computation

Several frequently used values are pre-computed:

For each position in the board, there are 10 possible destinations for cannons (32×10 array).
The L1 distance between two positions in the board (32×32 array).
The check for whether a piece can capture another (14×14 array).

Optimize move generation

The sophisticated verification in the Referee() function may sometimes be redundant. For instance, the source is always a valid piece from the correct color if we use linked list; The target position will always be 1 move away from the source (except for cannon), because we only check the neighbors. The check for whether a piece can capture another can be pre-computed. These checks are either removed or pre-computed to speed things up.

Optimize draw detection

We can first observe that at least 12 consecutive non-eat-or-flip moves are required to trigger the position repetition draw. This is because for both players, their chess needs at least 2 plys to leave and return to a position, which yields 4 moves. The repetition limit is 3, thus $4 \times 3 = 12$.

Next, to detect repeated patterns efficiently, we can use the KMP algorithm, whose space and time complexity are both linear to the input sequence size. Specifically, given a list of history of moves, we start from the very **last** move, going back until the **last eat/flip move**, and we compute the Longest Prefix which is also Suffix (LPS) array. During computation, whenever the examined length (e.g. l) is a multiple of the difference between LPS value and the examined length (i.e. $l - LPS[l-1]$), we have detected a repeated pattern. If the multiple is 3, we have a draw.

$$\begin{aligned}
\text{Sequence}: & \underbrace{\text{ABCD}}_{\text{pattern}}\text{ABCDABCD} \\
\text{Matched Prefix}: & \quad\quad\text{ABCDABCD} \\
\text{LPS}: & \underbrace{0\ 0\ 0\ 0}_{\text{pattern}}1\ 2\ 3\ 4\ 5\ 6\ 7\ 8
\end{aligned}$$

Settings	fractional window	draw	move gen	aspiration search	table	killer	total (s)	speedup
baseline (1)							695.5	-
baseline (2)	✓						249.6	(1)×2.79
+draw	✓	✓					243.5	(2)×1.03
+move gen	✓		✓				209.8	(2)×1.19
baseline (3)	✓	✓	✓				203.4	(2)×1.23
+aspiration	✓	✓	✓	✓			177.7	(3)×1.14
+table	✓	✓	✓		✓		127.2	(3)×1.60
+killer	✓	✓	✓			✓	119.3	(3)×1.70
all	✓	✓	✓	✓	✓	✓	59.8	(3)×3.40

Table 1: Total time of 6 games and speedup using various techniques. Tested on the first 3 sample boards from homework 2 (thus 6 games). The max search depth is fixed to 8, and all randomness are turned off. All settings produce the same ply throughout 6 games.

We can identify the repeated pattern as **the segment from start up to the last occurrence of 0** in the LPS array. The length of this segment is $l - LPS[l-1]$. Therefore, to detect draw, we simply check if the length is a third of the currently examined length l:

$$\text{if } \frac{l}{l - LPS[l-1]} = 3 \text{ then draw.}$$

2 Experiments

2.1 Speedup effectiveness

Several techniques above contribute to reducing the computation time during the search. I conducted a experiment to compare them. I use the first 3 sample boards from homework 2, and use different settings to play against the baseline on these boards. All settings' max search depth is fixed to 8, and all randomness are turned off. All settings produce the same ply throughout 6 games. Table 1 shows the result. We can see that the fractional null window is very important for speed. The two extra optimizations combined (baseline (3)) can provide a ×1.23 speedup which is good. Out of the three required techniques, killer heuristic is most effective for speedup. Transposition table comes close to killer heuristic, and aspiration search's speedup is also decent. When combined, the three techniques yield a surprising ×3.4 speedup.

2.2 Time control strategy

I compared with another strategy: Fixed search depth of 10, and a maximum ply time of 20s. Both strategies are used against the baseline program for 50 random games. The dynamic time control achieves 47W3D0L. The fixed depth achieves 41W9D0L. As expected, the dynamic scheme can use the spare time to search deeper, so it is less likely to reach a position that is impossible to win and seek draw.

2.3 Search extension and quiescent search

I tested the dynamic search extension and the quiescent search using SEE, both combined with transposition table, against the baseline on the 10 sample boards from homework 2. The dynamic search extension results in 18W2D, while SEE results in 19W1D.

2.4 Star1 and Star0

I tested star1 versus star0 on 20 random games. Star1 wins 16 times, while star0 wins 4.

2.5 Final Setting

In the final setting, all the above are used, except the dynamic search extension, which is replaced by SEE.

3 Discussion

3.1 Compare effectiveness of using different heuristics

By default, we use knowledge heuristic for move ordering. The ordering priority is based on the following:

1. Capturing is considered first, then moving, and finally flips.
2. Moves that capture larger pieces are considered sooner.
3. Among moves that capture the same piece, smaller pieces are considered sooner.
4. If two moves are equal according to previous points, then randomize their order.

In this assignment, I tried the history heuristic and the killer heuristic. The history heuristic records the occurrence frequency of each move. In my experiments, the history heuristic is not beneficial to the move ordering, and even slows down the search. On the other hand, the killer heuristic is easier to implement. It stores at each depth the most recent 2 moves that causes cutoffs during search. As shown in Table 1, adding killer heuristic provides a 1.7× speedup compared to simply using knowledge heuristic (when there are no flips involved).

3.2 Methods introduced in alpha-beta to be used in MCTS

Move ordering heuristics

Methods that improve move ordering in alpha-beta search can be used in MCTS as a way to allocate resource towards more hopeful branches. For instance, to choose which node to expand, or how much will a node be simulated. These methods include the knowledge heuristic, refutation table, killer heuristic and history heuristic. For example, we can store the "killer" moves of each depth as the moves along the PV path. Then in the next ply, we can reuse the deeper killer moves from previous ply as the first choices to be simulated.

Transposition table

The transposition table is versatile and can be used in all sorts of search engines. In MCTS, we can use the transposition table to store statistics from previous plys' trees, and can greatly speedup the convergence of subsequent plys, if the opponent's ply is exactly as predicted by our PV path.

Search depth adjustments

During alpha-beta search, we can adjust the search depth dynamically based on the quality of the position. e.g. Extending doubtful positions or forced variations, and trim hopeless positions. These can be used in MCTS during simulation, then we no longer need to simulate to the end of game or a fixed depth, but adjust based on hopeful/hopelessness. These technique includes dynamic search extension, quiescent search, late move reduction.

Pruning for depth-i

We can also consider using alpha-beta pruning for the depth-i enhancement. The depth-i enhancement would exhaustively simulate the first i level of the tree, quite similar to mini-max trees. If the depth is greater than 3, then we can integrate some form of alpha-beta cut to save time. Note that we may need to take variance into account, similar to when we do progressive pruning.

3.3 Methods introduced in MCTS to be used in alpha-beta

Evaluation function

In MCTS, we use large amount of simulation of games to estimate the score as the an evaluation function for each position in the tree. In alpha-beta, we can replace (or somehow integrate) the original evaluation function with simulations. In endgame, the number of moves until game over is not much, so doing simulations as an evaluation function is feasible. The advantage of this is that its value is more accurate than handcrafted evaluation, because the value is actually the win rate. Which means it may detect some situations that the player can never win, but the material score is very high, in which case simple evaluation failed to detect.

Node expansion policy

We use node expansion policy in MCTS to decide which children gets expanded and which to ignore. This is very similar to move ordering in alpha-beta search. Essentially, we use heuristics to bias our algorithm to consider some hopeful positions first, and to save time from hopeless ones.

3.4 Different goals that a computer game programming can achieve

Finding the best strategy

Many games can be seen as real world problems at smaller or abstract scale. When we develop a good search engine for games, we essentially found a good strategy for these smaller problems. Therefore, it is possible to apply these strategy finding technique in the real world problems as well.

Training to make players better

Previously, a person needs to train with other peer with similar/higher level of skills to improve their own skill. Now, with a good performing computer program, we can design training course by using the program to train new comers. The advantage of this can be that computer programs are available 24/7, or that computers don't take much time to think, and that the level of hardness might be adjusted at will. Besides, the programs may provide strategies that are not obvious to human.

Fairness of games

Suppose we have designed a new game. Suppose that theoretical proof of its fairness does not exist. In order to determine if it is fair, we might need thousands or even millions of games played by different players in order to confidently decide if it is fair. This is because every human think or play differently. However, with the help of a well designed search engine, we can pit it against itself, creating a absolutely fair environment, then we can decide the fairness of a game with far less resource.