Project 4 Final Writeup

Yuzhou Gu, Haoran Xu, Yinzhan Xu, Chengkai Zhang

December 14, 2016

For the final writeup, we first present the optimizations we have made, and their performance, then we give the optimization without much performance gains, as well as our final ranking.

1 Optimizations made

In this section, whenever the speedup is referred to without mentioning the metric, it is measured in terms of nps (nodes per second). That is, if nps before optimization is 1, and after optimization is 2, we say we have 100% speedup.

1.1 Bottleneck improvement: `scout_search`

According to our profiling, the function `scout_search` takes up to 32.55% of the total time, and it does the following: for a given node, it retrieves a list of all possible moves, sorts these moves, checks each move in order and performs a recursive search if necessary. To improve the performance of `scout_search`, we first parallelized it, which gives about 120% speedup. Furthermore, we made optimizations based on our observations below:

1. We noticed that hash table move from pre-evaluation and killer moves are returned with high probability, so we check them first before generating the move list

2. Since a move is ignored when the node is quiescent and there is no victim, so we try to predict whether there is a victim without running the move. Since we are certain that if both positions involved in the move are not on the current laser path, no new victim can show up, so in this case we exclude the move from the move list. This is a conservative prediction, but it decreases the number of moves in the move list by 2/3.

3. We noticed that about half of the moves have a sort key of 0, and thus they are checking the last in the move list, and the relative order among these moves are not important. Therefore, we directly move them to the end of the move list and exclude them from the sorting procedure.

4. Since maintaining the node count introduces true sharing between threads, we directly removed it.

1.2 Openbook

1.2.1 Motivation

The idea of generating an openbook is based on the assumption that good AI makes similar moves and the number of possible good moves at each given state is limited.

To justify our assumptions, we downloaded 83000 games from Scrimmage and split them into a training set of 39000 games and a test set of 44000 games. For the first 5 rounds of games, 782 (2%) openings in the training set occurred at least twice. When comparing with the test set, we noticed that around 30000 (2/3) of the games are covered by these 782 openings. Therefore, it suffices to say that an openbook would be useful.

On the other hand, the advantages provided by openbook is quite tempting: we can search very deep for a good move in the openbook, giving optimized moves for our first several moves, at the cost of *no* time at all. Quantitatively, the time saved by openbook under the default timing strategy is 20s in Regular, 8s in Blitz for hitting 5 rounds, and is 38s in Regular and 15s in Blitz for hitting 10 rounds.

Considering the justification and advantages above, we decided to calculate the openbook.

1.2.2 Implementation

We first generate all popular openings. For this step, the 83000 downloaded games are used, and frequent openings were stored into MySQL. The search depth for each opening move varies from 9 to 11, with deeper depth used for openings with higher number of occurrences. Over 100000 openings are generated in this step.

To calculate all these openings, we used distributed computing with LAMP (Linux+Apache+MySQL+PHP) web server to distribute down tasks and collect up results, and each client only needs `wget` to interact with web server. Overall, over 150 CPUs in Microsoft Azure were used, with over 15000 CPU hours in total.

1.2.3 Improvements based on test results on ReferencePlus

We tested our openbook on one of our bot that has a 50% winrate against ReferencePlus. When running our bot with openbook with the same bot without openbook, we get a winrate of 61%. However, the winrate against ReferencePlus decreases to 45%. Therefore, we made further improvements on our openbook based on these results.

For the decrease in winrate, two reasons are possible:

1. The opening patterns of ReferencePlus were not captured in the data (at the time we capture data, ReferencePlus is still not available).

2. Deeper search doesn't guarantee a better move, but just a good move with higher probability. So an unlucky bad move in the hotspot of openbook may actually degrade performance.

To address the first possibility, we add the games the bot played against ReferencePlus into the openbook. We note that this process is very similar to how an actual human player learn from a skilled player:

- S/he first plays with the skilled player in a competitive setting (in our case, playing with ReferencePlus).
- After the matches, s/he replays the game, and spend time thinking about which moves could have been better (in our settings, replay the open game, and search each step with a high depth).
- Then s/he learns those moves, so when the same situation is met, the same mistake won't be made again (in our case, add those better moves into openbook so we can react better the next time we met this situation).

It turns out that this strategy increases winrate steadily, as shown by Table. 1.

# games learned	0	1500	3000	4500
winrate	45%	50%	56%	61%

Table 1: A table of winrate of our bot against ReferencePlus, before and after adding games played against ReferencePlus to openbook.

To address the second possibility, we noticed that our AI has a distinctly different win rate between moving first (30%) and moving second (60%). We conjectured that the first move "h4g5" generated by the Bot is actually a bad move. We experimented with move "h0L", which rotates the king. Amazingly, despite that we essentially waste a move, and this move is considered bad by the Bot, the win rate increased drastically to 60%. Together with the boosted openbook, the win rate becomes 69%.

Based on this idea, we experimented several other opening moves. It turns out that the king move "h0h1" has the highest win rate. As a side note, among the 140000 games we downloaded, the starting move "h0L" and "h0h1" are never used by anyone but us. And it turns out that this move drastically increases the win rate. This suggests that heuristic function value and search depth are not absolute measurement citeria, especially in start game.

1.2.4 Summary

After all the improvements, our openbook contains about 200000 game states arisen from 140000 games. It almost always hits 6 rounds, and can hit 7 or 8 rounds with good probability. Sometimes, even 10 rounds or more are hit in games.

The winrate against ReferencePlus is also listed in Table. 2. It clearly shows that our openbook leads to a significant increase in winrate.

1.3 Constant optimization

We also made constant optimization to improve our code performance. They are listed below:

	Blitz	Regular
Before	74.6%	67.6%
After	82.5%	82.3%
Improvement	7.9%	14.7%

Table 2: A table of winrate of our bot before and after adding the openbook, for both Blitz and Regular games.

1. We used a uint64_t to store the cells on board that are lasered. This replacement saves some scans of the whole board, and also saved some memory space. Also, in `eval.c`, the laser was computed several times on the same board; since we are using a bitmap, we can simply use a bitmap to store the laser and use this bitmap for all the computation. It gives about 20% speedup.

2. We also use a bitmap to store the cells on the board that are white pieces and another bitmap to store the cells that are black pieces (it is supplementary and the original representation is still stored). This helps to reduce some blind scan of the whole board. It gives about 15% speedup.

3. We change ARR_WIDTH from 16 to 10. Therefore ARR_SIZE decreases from 256 to 100. This gives about 30% speedup.

4. We used some constant tables to reduce work. The `pcentral` function in `eval.c` repeats calculation a lot of times. We precompute the results and stores the result in a constant table. We use constant table to remove many divisions in the code. These optimizations give about 10% speedup.

5. We changed a lot of small functions to inline functions or macros. This gives about 20% speedup.

6. We found that in some places, it is unnecessary to use int. We replaced them with appropriate smaller types such as uint8_t and uint16_t. This gives about 10% speedup.

7. `subpv` in `searchNode` is only used to store the best moves up the search depth, but we really only need the first move. Thus, we deleted the array and replaced it with a variable. This saves memory and thus improves the speed. It gives about 20% speedup.

8. We modified some logic in `eval.c` while keeping the result the same. For example, we merged the case for king and for pawn in the switch struct, and then minus score for king out of the loop. Such improvements give a 20% speedup in total.

9. We found that it is unnecessary to store the victim pieces; instead, we only need to record some necessary information. Thus, victims_t can be packed into a int16_t. This gives about 5% speedup.

10. We further made some functions into macros. This gives negligible speedup.

11. We changed the set in transposition table to be 4-way set-associative, which gave a significant speedup.

2 Optimization without performance gain

2.1 Generating Closebook

We generated a closebook for cases with two kings and only one pawn. However, this does not give much performance gain, since most games end with more than one pawn. In this subsection we list details of our closebook.

As suggested in the handout, we generated a closebook for cases where the total number of pawns is no more than one. There are 64 positions on the chessboard and each piece has 4 directions, so there are 256 possibilities for each piece. Two kings and a pawn gives us about $256^3 * 2 \approx 3.2 \times 10^7$ possible chessboard states (the pawn may belong to either side). Our goal is to calculate precisely which states are winning states and which states are losing states. The huge data scale, not acyclic transition graph and KO rule (which limits possible moves according to the previous move) complicate the problem. Below we will explain our solution to those complications.

- KO rule: KO rule prevents a player to swap back two pieces that are just swapped by the opponent if no pieces are zagged. Since there are only 3 pieces on the board, only 3 swap moves are possible. Therefore, we add into our chessboard state an extra variable, denoting if the previous move was a swap move, and the type of swap move. This enlarges the number of possible states to $256^3 * 2 * 4$.

- Storing the transition graph: we compressed a board state into an int, which takes 4 bytes. There are 17 possible moves on average, so it takes about 8GB memory to store the graph.

- Since the graph contains cycles, we cannot use simple DFS to determine the winning states like what we do for decision trees. Instead, we need to do it in the other direction. We first figure out the nodes that can be won/will lose in the next move, record the result for those nodes, and put them into a queue. Every time we pick out a node and remove the node from graph. If the node is a losing node, we mark all nodes that can reach it in one move as winning and put them into queue. If all moves of a node lead to a winning node, we mark this node as losing and put it into queue. The nodes that are not determined in the process are draw nodes.

Note that KO rule does affect which opponent wins in some states. In that case we just record those states as "uncalculated". (The number of such cases is very small). To conclude, there are 4 possiblities for each state (win, lose, draw, determined by KO rule), which can be recorded in 2 bits. The total size of the output table is $256^3 * 2/4 = 8$ Megabytes, which is acceptable.

2.2 Other optimizations

During our beta I submission, we used a range tree to replace the incremental sorting used in `scout_search`. However, since the range tree implementation is not compatible with parallelization, we removed the range tree and used the original incremental sorting, with optimizations mentioned in the bottleneck section.

3 Addressing MITPOSSE's comments

We addressed MITPOSSE comments. We

1. removed useless code;
2. addressed some code style problems, particularly in the closebook generator;
3. replaced magic numbers with macros;
4. made long functions shorter;
5. added comments for auto-generated code.

We

1. did not attempt to improve the performance of the closebook generator because performance is not important here.