Beta I Writeup

Yuzhou Gu, Haoran Xu, Yinzhan Xu, Chengkai Zhang

November 18, 2016

1 Profiling Data

We tested the original program on the command "go depth 8" and recorded the profiling. Below is the result for the most costly six functions.

scout_search	pawnpin	h_squares_attackable	eval	square_of	make_move
14.59%	13.34%	9.71%	8.74%	6.18%	6.04%

2 The changes so far

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. In scout_search, an incremental sorting function is used. We found that this function is very slow so we replaced it with a more efficient sorting algorithm. First we tried quick sort but it did not help. Then we discovered that only the smallest items are used, so we find the smallest element each time. We used a range tree to maintain the smallest element in the array. It gives about 10% speedup.

8. 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.

9. 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.

10. We also created a closebook, which is discussed in detail in next section.

3 Generating 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 must 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.

4 Optimizations we plan to make

1. Parallelize the code. Chengkai Zhang and Haoran Xu peer programming.
2. scout_search and make_move are currently the most costly functions, so we may be able to optimize them in more detail. Chengkai Zhang and Haoran Xu peer programming.
3. During beta 1, we find that make searchNode smaller can give significant speed up, so we can still use this trick to improve, Yuzhou Gu and Yinzhan Xu peer programming.
4. We can also perform some strategy optimization. All team members can contribute to this (because it is currently unclear how to do this).