Chesskell

By Alen Kubati, Sanjay Paul, and Jiten Suthar

1. Project Description

For our final project, we developed a complete chess-playing game engine that we have affectionately dubbed *Chesskell*. The engine is capable of simulating two types of games: human vs. human and human vs. machine. The more interesting of the two, of course, is the human vs. machine capability and it manages to challenge the human player at a reasonably effective skill level.

1. Running and Playing Chesskell

In order to play Chesskell, simply run the main class, called ‘Chesskell.hs’. You will be prompted for the type of game you’d like to play (there are two options: human vs. human and human vs. machine). From there, the game is on and you and your opponent alternate inputting moves until the game is won. Moves are input using the standard chess representation in which a space is specified using a letter followed by a number (don’t worry, the board is labeled!). A complete move is formed by separating two such space declarations by a dash (e.g. ‘e4-e5’ will move the piece at e4 to e5). Of course, only legal chess moves are accepted! The exception to this is the very last move of the game. Due to the design, a checkmate is not recognized by the engine (see ‘Gameplay Disclaimer’ for further details). The game also supports non-standard chess moves such as castling, *en passant*, and piece promotion. See the instructions printed at the start of the game for additional details.

1. Under the Covers

Developing Chesskell was a fantastic exercise of Haskell coding ability that necessitated the coupling complex data structures as well as the careful consideration of control flow mechanics.

Fundamentally, the game progresses recursively towards a ‘base case’ of a king capture and proceeds indefinitely until this is achieved. Further, we needed a relatively efficient workaround in order to update the game state. This was achieved by maintaining a (hashed) mapping of board locations to pieces as well as an inversion on that map that yielded the space occupied for each non-empty board piece. The engine solicits the player for an action that is then subsequently processed in stages. The action is extensively validated and, for the majority of cases, this is an invariable and trivial procedure; however, the case of a king being in check meant that only a subset of the usual moves were valid. Move validity is determined by first checking the piece’s possible moves. Next, the spaces between and within a piece and its destination are checked and validated. Assuming the action is legal, the piece is moved and the game engine updates the ‘state’ by making replacements in the stored maps.

Artificial intelligence presented the need for further complexities in the design of the game engine. To implement a reasonably effective AI, we made use of the recursive (two-player) minimax algorithm, which combines action weightings of potential moves with a predictive capability to generate moves that more closely approximate optimality than an otherwise naïve implementation (e.g. nondeterministic selection, ‘singly-weighted’ selection, etc) . The algorithm evaluates the game tree by replicating it in the system – it essentially considers every possible player move and opponent reaction for successive turns. Naturally, the expansion is approximately exponential and thus only a certain number of levels may be parsed given finite resource availability and the need for responsiveness. In our implementation, the AI looks only two moves ahead to make its decision as this seemed to be the tipping point between acceptable gameplay and utter computational chaos.

The AI relies on predefined weightings for possible moves. For example, claiming an opponent’s piece without any immediate repercussions would be assigned a very high weighting whereas needlessly sacrificing a piece would result in the lowest.The outcome of this design decision is that the AI is skewed towards defensive play. This strategy is favored because of the limitation on the AI’s ability to look ahead in the game tree, which prevents it from executing more complicated plays. As well, the AI is also programmed to assign a greater priority to controlling the center of the board as opposed to the edges. Finally, the AI’s opening gameplay was far from ideal and, thus, this phase utilizes somewhat altered mechanics.

1. Gameplay Disclaimers

Sadly, Chesskell’s AI is far from perfect. There are times at which it will be locked into making seemingly random and otherwise trivial moves (one that we noticed in testing is an oscillation of the rook back and forth across the topmost row). In addition, the AI struggles to achieve a checkmate and did not manage to do so in our testing when we attempted to lose intentionally (unless we gave it a one or two move checkmate, which the game tree will actually find). We suspect that it can only actually achieve checkmate via serendipity and we attribute this to the overall defensive character of its play-style. Chesskell’s AI is truly most effective at capitalizing on its opponent’s mistakes, playing relatively defensively, and attempting to achieve and retain control of the central portion of the board.

It should also be noted that a checkmate can occur and will not result in game termination. This is due to a particular quirk of our recursive implementation, which has the check for game termination placed *before* a player’s move is executed. In order to work around this, simply capture the king piece in the move after placing an enemy king in check. Once the king is actually off the board, the game will end and the winner will be declared.

1. Lessons Learned

At a fundamental level, chess facilitates the use of recursion rather well. In fact, many issues, such as devising a game control flow and implementing the minimax algorithm were almost trivial on account of the ‘naturalness’ of recusion. By contrast, the absence of state management resulted in complications that would otherwise be avoided in imperative languages. Chess is a very naturally stateful game and was thus resistant in this regard to a Haskell implementation. For example, we found it necessary to pass the board (double hashmap, which is part of the game state) as a parameter to function calls. This makes every node of the game tree rather big, which we believe limits the effectiveness of our minimax implementation. However, the use of hash mapping and thoughtful and extensive logical branching ultimately allowed us to deliver a complete implementation.

1. Futurespect

There are a large number of refinements that could be made to the AI to improve its effectiveness. Perhaps foremost among these would be the use of alpha-beta pruning to improve the efficiency of game tree parsing. Additionally, some type of learning technique could be used to optimize the weightings on piece moves. Also, the evaluation function that we have right now is rather naïve. It just assigns the standard values to the pieces, with a few more points awarded if a piece is in the center. A more sophisticated evaluation function might make the program more aggressive. Finally, the program is completely deterministic, meaning that given a position it will always respond the same way to that position. Thus, one final improvement could be to add some degree of randomness.