Final Design Document – Chess

By: Wenchao Du and Hao Xue

UML Class Design

See UML in uml.pdf

Outline of source files

All classes depicted in the UML have their own respective *.cc and *.h files. In addition to these files, we have main.cc, which will call Game to initialize the game, *.xbm files which are images files for the chessboard and chess pieces, and a Makefile, and ai.cc/h that contains some helper functions for AI Level 1-3.

Overview of Features

This section will highlight the main features of the chess game and describe their implementation. This is in an order similar to the requirements specified in the chess.pdf instructions document for the project.

Rules of Chess

All rules of chess are implemented in this program, including proper castling, en-passant, pawn promotion, check and checkmate detection. The implementation details of all rules with regard to the movement or capture of a piece are put inside each individual piece subclasses, such as castling in King (king.cc) and promotion in Pawn (pawn.cc). When a Player registers a move with ChessBoard, ChessBoard will determine if the move is legal by calling the moving piece's move method, which follows a template method pattern, and registers the move if it is legal, or reject if illegal. (See piece.cc:move() for template method pattern)

Displays

There are graphics and text displays in the program. The TextDisplay (textdisplay.cc) is straightforward, and print out messages on checks and checkmates, as well as AI behaviour for AI level 3. The GraphDisplay (graphdisplay.cc) is a rewritten version of the provided Xwindow class to handle .xbm image loading. Information on image loading using <X11/Xlib.h> was scarce to find, therefore I used a lone example of implementation using Pixmaps to implement this class. It did not integrate well with the existing Xwindow class, so the Xwindow class was removed because it was redundant. In essence, GraphDisplay loads images of chess pieces in to a window and the individual Cell objects will notify the graphical display when they are changed. This is an example of the Observer pattern.

Players

We have all player features described in the instructions document. In other words, the game supports human vs human, computer vs human, computer vs computer. HumanPlayer is straightforward. The class's move method takes moves from stdin until it receives a legal move or "resign". ComputerPlayer (computer.cc) has 4 difficulty levels. A computer player with a higher difficulty level is usually able to defeat a player of lower level.

<u>Level 1</u> – Randomly select a move

<u>Level 2</u> – Prefers capture and checking moves

<u>Level 3</u> – Prefers avoiding capture, capture, and check moves. This is the best AI that computes its move in a near instant amount of time. It is implemented using statistics kept track by the game, in particular, whether a cell is a reachable by each piece on the board. Using this information, it is able to efficiently compute how it can safely capture a piece or perform a check.

<u>Level 4</u> (aitree.cc) – Make moves using concepts from game theory (game tree and the minimax algorithm):

1. The AI first builds a game tree by playing out the best moves chosen by a heuristics algorithm for 4 consecutive turns (by making moves and undoing them). In essence, the AI first adds to the tree the best possible moves for itself during every turn, and then after each of these moves, attaches the opponent's best possible moves for its turn to themselves in the tree, and so on for 4 turns. This tree building process is obviously slow because it's exponential ($O(m^n)$, where m is the number of "best moves" and n is the depth of the tree)
2. After the tree is built, I do a post-order traversal on the tree that uses a minimax algorithm to find the move gives the greatest net gain

This version of the ComputerPlayer is significantly slower than all previous counterparts because it thinks 4 moves ahead, but will defeat all AI of previous difficulty levels on a consistent basis, especially in late game situations with fewer pieces on the board and less probability for heuristics to go wrong. How much slower depends on the system. When ran locally on an i7 CPU, it takes 2 to 4 seconds for it to compute each move. However, I did notice over ssh on school servers, it can take well over 10 seconds.

Command Interpreter

The game supports all commands described in the instructions document (game, resign, move, setup, etc). In addition, the game supports "undo", which will undo a player's move. The human player can undo an unlimited number of times, but he or she must use caution because he or she must manually undo twice in order to return control back him or herself because otherwise, it is the opponent’s turn and the player will have input control. The command interpreter will check for input validity and the program does not break when an invalid command is issued. However, this does not apply to setup mode in that only the rules stated in the instructions document are implemented.