Albert Tran (a233tran), Harrison Chiu (h28chiu), Ryan Li (r433li)
CS 247, Spring 2022

Plan of Attack

Phase	Task	Date	Assignee
General Structure	Textual board display	16/7/2022	Albert
	Setup mode	17/7/222	Harrison
	Piece object implementation (i.e. directions)	17/7/2022	All
	Simple input to move pieces	17/7/222	Ryan
	Move object and list	17/7/2022	Albert
Finding A Board's Legal Moves	Basic movement (includes captures)	19/7/2022	All
	Advanced pawn movement (capturing diagonally, beginning double move)	19/7/2022	Harrison
	Kingside and queenside castling (notation, looking for checks, unmoved rook+king)	19/7/2022	Albert
	En passant	19/7/2022	Harrison
	Pawn promotion (notation, display)	19/7/2022	Harrison
	inCheck() method for given board+side	20/7/2022	Ryan
	inMate() method (ability to find stalemates)	20/7/2022	Albert
Additional Features	AI levels 1-3	22/7/2022	Ryan/Harrison
	Human Resignation	22/7/2022	Harrison
	Graphical display (SDL)	22/7/2022	Albert
	AI level 4 (min-max using custom heuristics, deciding move depth, etc.)	24/7/2022	All
	Opening book (display current opening name)	26/7/2022	Albert
	Move suggestions (from book or AI)	26/7/2022	Ryan
	Undo command	26/7/2022	Harrison
	Three-fold repetition	26/7/2022	All
	50-move rule	26/7/2022	All

Project Breakdown

For our plan of attack, we plan to separate the development of the chess game into phases so that we first flesh out functionality that is useful for testing later development.

To begin, we will implement the basic structure described in the UML diagram. We will only test simple functionality for our necessary input flows, ensuring that we have a text display so we can test the game in the later phases. We will also create our setup method to populate the board.

When this is done well enough, we can then implement the details revolving around piece movement. There is basic movement for every piece that should be implemented first, including

captures. Once that works, more advanced movement will be added, including pawn captures, promotion, en passant, and castling.

By this point pseudo legal movement in the game will be implemented, but now the rules will have to be enforced. For this, we will implement methods to verify the state of the game (check, checkmate, stalemate) and also update our implementation with moves to keep track of more advanced concepts such as pins and blocks. Completing this step is sufficient for us to play an entire chess game between two humans with our program, so testing would revolve around all the normal cases for human moves.

Once complete, we can look toward additional features. We will first attempt the higher-level requirements for the program, including the basic AI levels and resignation. The table outlines our priority list of additional features; we may pick some to implement over others if time does not permit.

Questions

Question 1

As opening positions branch off of one another it would be sufficient to hold the data of a book of starting moves in a tree data structure. We would implement this using a tree of node objects. At each level, we would have a node that stores the move object being made, as well as a list of nodes corresponding to candidate response moves from the opponent. In this way, once a player has taken one move, we can quickly access the book's recommended moves after the move has been made by traversing to the appropriate child node. We could also store the name of the opening at a given move in the corresponding node.

Question 2

The simplest and most straightforward way to approach this would be to implement our stack of played moves, which keeps the initial and final state of the piece moved, and to then move the piece at the final position to the initial position when performing the undo.

However, this approach does not account for changes to the board that are a consequence of the move, as opposed to the movement of the piece itself. These possible changes include attacking, where a piece is destroyed; pawn promotion, where a piece is changed; and castling, where a piece other than the moved piece moves. To address this, however, we could add a value in the move object that flags for these cases. Considering that the move already contains information on which piece was taken, there is sufficient information to determine the previous state of the board.

To perform the undo, we would let our board object hold the list of played moves. We would first call the undo method on the game level, which both calls the board's undo method and also

revises game-state information such as the turn. The board's undo method would pop the most recent move off the stack and return the board to the state prior to this move.

Question 3

- Board shape
- Pawn promotion (promoting at a closer square/rank)
- 4 colours/players
- Have to physically capture king

These are the main concepts that would need to be adjusted for four-player chess. Board shape would be addressed by changing the size of the board data array; since the board resembles an unfolded box as there are no squares in corners, we would disallow pieces to be on some *n* x *n* square in each corner of the board by introducing a flag to the square class to determine whether it can be occupied.

Since legal pawn movements depend on which player the pawn belongs to, we would have to account for the new directions now possible after adding two extra players. This would be accounted for when initializing the game, as we create all the pieces with their move vectors (i.e. all the directions they are allowed to move in). We could also enumerate the player/piece colours arbitrarily to have some normalized method of determining which player the piece belongs to.

On captures, we would need to check whether a king has been captured (and of what colour). If so, we want to create some eliminateColour() method to remove all pieces of that colour from the board in accordance with standard four-handed chess rules.