Milosh Zelembaba
Cameron Porteous

Written Component for Assignment 5

Question: Chess programs usually come with a book of standard openings…

Answer: In our implementation, the physical chess board along with the pieces on it are represented by a 2D array of ChessPiece* (a super class representing each chessman). First off, a list of standard openings implies that the board is in it's standard starting state, so in the rest of the discussion the case of when "setup mode" modifies the board does not have to be considered (since the standard openings won't apply anymore). So with that, openings and how to move based off of the opponent's moves are dependant on the *exact* state of the board (not only the opponent's previous move). This means that in our code we have to consider the exact state of the board and make a move accordingly. We could create a map that takes the board's state as the key and the corresponding value could be a second map of size ~10, with suggested moves as the key and the corresponding value being the predetermined "strength" of that move. So for the first dozen moves or so, the computer player will take the state of the board, check if the board state is a key in the map of opening moves, then choose a move from that state's map of suggested moves (taking the moves "strength" into consideration). It is important to note that our algorithm for choosing opening moves wouldn't always pick the same move given the same board state. If it did, the program would become predictive and thus less challenging of an opponent.

If we wanted to save some more memory, instead of using the state of the board as the index (which is a 2D array of ChessPiece*), we could instead convert the board state to a 2D array of strings (or even 1D) and use this string representation of the state of the board as the index. This conversion would be O(1), so it wouldn't have an efficiency cost but memory would be saved.

Opening lines have been rigorously studied for centuries, so it makes sense to design the algorithm analogously to how humans think in the early game. The main strength a machine player has over a human player is in the "middle game", so until we get to that stage (~dozen moves), referring to a book
of standard openings is optimal.

Question: How would you implement a feature that would allow a player to undo....

Answer: Well in the first case, since we're only looking back one move into the past, all we need is a variable that stores the previous state of the board. Since an undo would set the state of the board back to the previous state when it was that player's turn, we could implement two of these variables (each keeping track of the board state when it was that player's previous turn). If a player tries to undo their move, the board state would simply be set to variable that was just talked about. However if we wanted to implement undo so that it would support unlimited undos, we would instead use a stack. This is beneficial for the fact that a stack is sorted in LIFO order. Each element in the stack would contain the previous state of the board. Whenever a move is made, the board state would be pushed on top. Whenever the undo is called, two elements would be popped off the stack (to get to the state of when it was that player's previous turn) and the state of the board would be set the second element that was popped.

Question: Variations on chess abound...

Answer: So in four-handed chess, the obvious differences are that the board is different and that there are four players in total. So our representation of the board would have to change to a 14x14 2D array where the 3x3 squares in the corners would be considered illegal or invalid spots. The next obvious step would be that we would have to keep track of the pieces of four players instead of just two, and in the controller we would have to cycle through four players instead of just two. In regards to the movement of each piece, we would have to consider more possibilities. For example a pawn can only move in the direction it's orientated in, so modifications would have to be made to adjust for how pieces move according to their colour (orientation comes with colour). There would be more restrictions on what a legal move is as well due to the 'cut out' corners. Now, if we were playing singles, the capturing of pieces would be the same as your still only capturing pieces that aren't your colour. However if it was teams, then we would have to make sure that the method knows which teams consist of which colours and to only attack colours that are not on your team. While with regular chess we might have been able to get away with some efficiency gripes, in four-handed chess, the number of everything increases (board size, number of pieces etc...) so methods and functions and maybe even the design of the program would have to be revisited in order to be as efficient as possible for a smooth gameplay. Next, while in regular chess, the game is over when someone is checkmated, in this game the game is over when there's only one player remaining. Since when a player is checkmated, they can either remove their pieces or the player that did the checkmating can play those pieces, the appropriate switch of who has what pieces would need to be accounted for.

Question: Assuming you have already implemented all the commands specified...

Answer: A saved game session can be thought of simply as a custom board state. With that said, that means that we could use our already implemented "setup" mode command to modify the board until it reaches the state specified by the saved session. Once this is achieved, setup mode would terminate and it would be put into "game" mode in which the game can commence.

Plan of Attack:

The first step to the project consists of a debate as to which project we should choose to do. We want to get this done on day one as it helps provide motivation to start the project immediately and also due to the fact that there's no benefits in pushing it back. With this we chose PP9k and on the same day we should start talking about a general design to the project and generate rough UML ideas for how the program should function/be designed.

I think it's important that no coding be done in this first day due to the fact that we haven't had enough time to analyze how the game is going to function and any code that is written would probably be garbage, and just a waste of time. Within the first 2-3 days we should have all the important classes working in cohesion (not perfectly implemented, just working) so we can develop a deeper understanding of how methods are going to be implemented. We will also come across problems that we might not have been able to think of otherwise.

It'll also be important that within this deadline we get a working textual display along with "setup mode" functionality. As of course we want there to be ideally no bugs, having these two tools helps us more efficiently manually test our game and come across bugs faster. The textual display provides us with an easy interface to use and the setup mode helps us test tricky situations immediately (rather than playing out the game until the situation we want to test occurs). Before the weekend (hopefully Saturday, maybe Sunday) we should have a fully implemented version of the game working minus any computer player functionality. This includes graphics, and at this stage the command "game human human" should be able to be carried out without flaws.

In terms of how coding responsibilities will be divided up in this first week, it'll ideally be pretty even. Since (through experience) people work better when they focus on one task fully instead of multiple tasks with partial focus, code will be divided up in terms of specific functionalities instead of both of us working on the same code. For example I might work on fully implementing graphics while my teammate will work on implementing how pieces move. This lets each of us get immersed in our area and as a result a greater depth of understanding is developed for what we are working on and can thus produce better functioning, less buggy code.

The second major section of our plan is week two (starting Monday or Sunday), which will consist solely of computer player implementation and ideally bug fixes. With the knowledge of how we have implemented our game so far, we will first brainstorm ideas for how all the computer players should be implemented along with how they will be integrated into the game we already have. Within the first two days (ideally one) we should have the framework and structure for how computers will blend into our code and also have level 1 and level 2 computers completed. With the time that is remaining, our first priority would be to get a working version of the level 3 computer and then spend half a day or so thoroughly testing our code to achieve the highest degree of correctness. Once that is finished, with all the remaining time that is left, we should dedicate our efforts to implement a working version of a level 4 computer player to the best of our abilities. During this second week code will be divided up in the same way as it was in week one, however much more discussion will be needed between us to be as clever about our computer implementations as possible. Of course along the way, at every step, we should be commenting our code to help us question design details more and also to avoid wasting time doing that at the end.

By the end of this two week period we will hopefully have a fully working chess game to walk away from.