Draughts

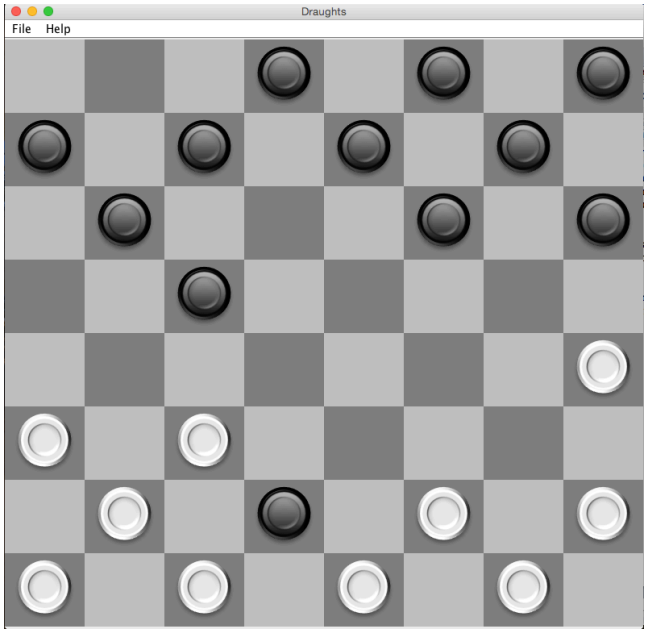
An interactive, seemingly intelligent game.

Lewis Lloyd

Chapter 1: Introduction

The challenge for this project was to create an interactive draughts (or "checkers") game in which you, the user, could play against a computer. Interaction was to happen through a graphical user interface (GUI), with a board appearing on your screen, complete with pieces that jumped and stepped to legal positions when either you or the computer made a move. The computer was, if possible, to have variable levels of artificial intelligence which could be adjusted by the user according to how bold or inept they were feeling.

My approach to implementing the game involves four classes. The first – the Game class – initialises the graphical user interface (creating the frame, adding a menu bar to it, and retrieving a board display from the Board class), pops up an introductory dialogue when the game is first opened (which, among other things, asks the user to choose how good at the game they would like the computer to be), has methods allowing the user to "quit" the game or read the rules from the GUI, and loops through the process of getting an AI move, then a human move, until someone wins.



The second class – the Board class – is by far the most complicated and important. The board display for the GUI is implemented here (see the screen shot to the right), as are the majority of the "Game Internals". Primarily, it contains methods to validate and carry out any moves that the user tries to make, and evaluates all the possible moves the AI could make (using a minimax algorithm with Alpha-Beta pruning) before returning the best – given any limitation placed on the computer's intelligence. The code is heavily commented, but there is also more detail on all of this below.

The other two classes are very simple. The StatesAndScores class allows possible next game states to be stored along with their evaluated scores from the Board class, together in individual objects. The Square class, meanwhile, is specific to the GUI. Each of its objects is a square on the board display (in the form of a JButton), and the class allows the icons displayed on these JButtons to be updated to reflect whether a given position is occupied by a piece in the current game state.

Chapter 2: Description of programme functionality

State representation, successor function, and validation of moves

The state representation I opted for when developing the internals of my game was a 2D, 8x8 int array. Each element in the 2D array is intended to represent a square on the board, with the element at [0][0] being the top left-hand corner, [7][7] the bottom right, and so on. The value given to each element represents whether or not the corresponding square is occupied by a piece in the current game state. If the value is 0, the square is unoccupied. The value 1 indicates the presence of a black piece, with 2 for a white piece, 3 for a black king, and 4 for a white king.

The successor function fell quite naturally from this – although it quickly became complicated, with methods multiplying, as I added functionality for multi-step moves, and backwards moves for kings. For the purposes of evaluating the next best move to make on the AI's part, I decided to represent the results of any change in a piece's location – the square it was in no longer being occupied, the square it has moved to now being occupied, any intervening square that was jumped over no longer being occupied, with the piece that was there being removed form the board - as a whole new state. The getPossibleStates() method in the Board class, then, was supposed to cycle through all the possible moves that could be carried out from a given game state, given the locations of all of the pieces in that state, with the results of each of those moves registered as a new, separate state. The method would then return an ArrayList of all of those possible next states.

To force takes, that method split into a process of determing whether or not any jumps were possible for the player from the given state, then whether or not any steps were possible. If jumps were possible, only those states resulting from a jump were to be added to the ArrayList of possibleStates to be returned. Whether or not any steps were possible would only be determined if no jumps were possible.

Naturally, this split into two further methods: getJumpStatesState() and getStepStates(). The former cycles through all of the positions on the board, identifies any position which is occupied by the player, then determines whether a jump is possible in each of the four diagonal directions. This last assessment is made by a call in each case to the jumpPoss() method, which goes through a list of rules to check that a jump by a certain piece from one square, over another square, onto another square, given a certain starting state is legal. If a jump is found to be legal, it is carried out virtually in a new state (using the makeJump() method), which is then returned as a possible next state following on from the original.

An extra step then had to be introduced to take into account multi-step moves. For each of those possible next states resulting from a jump, a check had to be made to see whether or not another jump was possible. My solution to that was to draw up another method, takesPossPosition(). This returns whether or not

any takes are possible, in any direction, from a given position. If they are, those jumps are then carried out virtually, with the new resulting states saved and added to the list of possible states, rather than the state resulting from the first jump.

I opted to process user requests in a separate method, but using the same tools. Keeping this separate was largely on account of two things. Firstly, with the JButton approach I had chosen to use for the GUI, the selection of a piece to move (or square to move from) and square to move to would have to be monitored separately, the brought together. I thought it would be easier to do this without muddling the process up with the validation of AI moves and next states for the minimax algorithm to search through. Secondly, the need to offer some explanation for the rejection of invalid user moves made it seem sensible to keep this process separate.

The moveRequest() method processes user moves accordingly. In the first instance, it checks and monitors whether or not a piece has already been selected to be moved. If it has not, when a square is clicked on, that square is registered as the square a desired move will be from (following validity checks concerning whether a take is possible from the current state (takesPossState()), and if so whether a take is possible from the selected position – or, if no takes are possible, whether the selected square is actually occupied by the player). Any subsequent click on a square is then treated as a request to move the piece in the previously clicked on square to this new square. A series of checks on the validity of the requested move are carried out again, using some of the methods outlined above. If the move is valid, the current game state is updated. If not, some explanation and a hint as to what to do instead is printed to the terminal.

I added a kingCheck() method to check for whether or not a piece has reached the end of the board, and implement this in the makeJump() and makeStep() methods to allow the conversion to happen automatically. Throughout the checks described above (applying to both user and AI moves) as to whether a jump or step is possible at any given point, the possibility that a piece might be a king is always taken into account and monitored, allowing for legal king moves. Takes are forced, when available – including when they are multi-step.

Minimax algorithm

The minimax algorithm employed in the game builds on the above. It is always called initially from the getAIMove() method, and the idea is that it should populate an ArrayList of StatesAndScores, with each object in the ArrayList encompassing a possible next state reachable from the current state and a value representing the utility of that state (how much, in short, said state will improve the AI's chances of winning the game).

The process of getting the possible states given the current state is as already discussed. The more interesting questions relate to evaluation, and depth. Evaluation is a concern, in the first instance, because draughts is too complex a game for the minimax algorithm to go all the way to the bottom of the search

tree from the game's start state until it reaches a point where one or other side has won, and then assign a value to the states higher up the tree accordingly. As a result, a simple evaluation – with a score of 1 returned when the player represented by MAX (in our case, the computer) wins, and a score of -1 when MIN (the human) wins, with 0 otherwise – is not an option. Some sort of heuristic has to be used to approximate to the utility of any given state reached when the algorithm gets as far down the tree as it is able to go (with the depth limit in general terms set by time and computing power, but in our case by the difficulty level selected by the user at the start of the game).

The heuristic I have used balances the number of black pieces on the board with the number of white pieces. For every black piece, the score goes up one; for every white piece, it goes down one. To encourage the AI to recognise the higher value of kings, the corresponding values for black kings and white kings are +2 and -2. When the algorithm reaches its depth limit, if the state it arrives at does not have either side as the victor (very high values are given for these eventualities if they do appear), it evaluates the state according to this heuristic, and passes that value back up the tree. When a value is retrieved from each recursive call to minimax within the loop, values of alpha and beta, initialised as the worst possible in each case, are also updated, allowing for the monitoring of which branches of the search tree are still worth exploring (while alpha is less than beta), and the pruning of any that are not.

As hinted at above, the cleverness of the AI is varied by adapting the depth the minimax algorithm is allowed to reach in the search tree. At the lowest difficulty level, the AI actually picks a next state at random from all the possible next states (see implementation in getAIMove() method in the Board class), but thereafter limits are set at depths of 1, 2, 5 and 10.

Summary

The GUI is not quite as error-free, but is not far off. Sometimes the updating of the piece locations on the board can be slow, or can require clicking on something else on the screen – a bug I have not managed to get to the bottom of – although usually this is not an issue. The GUI pauses to show intermediate states in multi-step user moves, but does not do the same for multi-step AI moves, and, unfortunately, squares do not change colour when they contain movable pieces. That will all have to wait until version 2!

I hope you enjoy the game.