Chess Project

By: Ep16fb and Pj16gf

Introduction

This program runs the normal game of chess, a turn based game where two players face off and try to obtain each other's King piece. The first to obtain the opposing side's King piece wins the game.

Pieces

There are 6 different types of pieces on the board which can be played with. These pieces include, the King, Queen, Bishop, Rook, Knight, and Pawn. Each player has 1 King, 1 Queen, 2 Bishops, 2 Rooks, 2 Knights, and 8 Pawns to play with. The King is the main piece and can only move one space in any direction at a time. The Queen piece is able to move in any single direction for as many spaces as desired. A Bishop can only move in a single direction diagonally for as many spaces as desired. A Rook only move in a single direction either horizontally or vertically for as many spaces as desired. A Knight must always move two spaces horizontally or vertically and then one space to the left or right. A Pawn can only move forward by one space, and can only kill pieces which are one space to the front and diagonal of itself. There are special rules that allow moves not mentioned above for the King and Pawn.

Basic Rules

A player is not able to bypass their own piece if it is blocking their path. A piece is not allowed to move outside the board. If a move requires the piece requires the piece to go outside the board, then the move is considered illegal and cannot be made. A player can however capture the opponent's piece if it blocking their path. But once a piece is captured the turn is over. Once a piece is captured, it is removed from the board, and cannot be brought back. When the king is place in a position where it can be captured but still be saved, it is in a state called "Check". Player's are not allowed to do moves which will result in their King being in "Check". The player whose King is in "Check" is required to save it, either by moving the King or a piece in the attacker's path. When the King cannot be saved it is in a state called "Checkmate", which means that the game is over, and the player whose King is in "Checkmate" has lost the game.

Special Rules

If a Pawn is at at starting position, it is allowed to move either one or two spaces forward, however afterwards it is only allowed to move one space forward. When a pawn reaches the other end of the board it is able to get promoted to either a Queen, Knight, Bishop or Rook. In the case where the opponent Pawn moves by two spaces from its starting position, where the Player's Pawn would have been able to capture it if it were to move my a single space, the Player's Pawn can capture it as if it moved by a single space, this is called "En Passant". If a player has not moved their King and either Rook from their starting position, then the King is is allowed to move two spaces towards the Rook that has not moved and the Rook moves to the opposite square beside the King. The King may not perform this if any piece is threatening either of the two squares the King moves over. A draw can occur in several ways. One of the ways a draw can occur is if both player agree to a draw. A draw can also occur if a checkmate is not possible and there has been no captures or movement of a pawn after 50 moves. The threefold repetition rule can cause a draw, the rule can be applied when the same positions have been repeated three times consecutively. The last way a draw can occur is when there is a stalemate, a stalemate occurs when the player in turn only has a King, which is not in check and cannot make a legal move, and Pawns which cannot move or capture.

Algebraic (Chess) Notation

Algebraic notation is used for readability and logging of chess moves. The chess pieces King, Queen, Bishop, Rook, Knight and Pawn are denoted as "K", "Q", "B", "R", "N", and "" (pawns do not have a symbol), respectively. Columns are indicated using letters between a to g, while rows are indicated using numbers 1 to 8. Moves are simple denoted with their symbol and the position they moved to, for example Qf2 (Queen to f3) and d2 (Pawn to d2). Captures are denoted with "x", for example, Qxf3 (queen captures the piece on f3). Pawn promotions are denoted with the location followed by the symbol of the piece the Pawn was promoted to, for example a1=Q (pawn at a1 promoted to Q). Castling is denoted by "0-0" for King side and "0-0-0" for Queen side. When a move places the King in check, the move is appended with a "+". When a move checkmates the King, the move is appended with a "#". The end of game is denoted by "1-0" to indicate White has won, "0-1" to indicated Black has won, and "½ -½" to indicate a draw. En passant is denoted with either "e.p." or simply the same as a normal capture.

Use

The player inputs a start and end location using letters a-g to indicate vertical movement and numbers 1-8 to indicate horizontal movement. A log will be outputted using Chess Notation with the resulting outcome of the move. En Passant is denoted the same way as a capture.

Design

Piece

The piece is an abstract class to represent all 6 pieces of chess, and generally what a piece can do on the board. The class has parameters like piecetype, colour, and weight to determine what type of piece it is. It contains methods to determine if the piece itself is being threatened, the valid moves it can perform, the best move it can perform at the moment, and the how good its current position is. In addition to these method, a piece can compare itself to other pieces, and is able to print itself to log or to the board.

PieceType

This is an enumerable to differ between the 6 types of pieces which can be played with on a chessboard (king, queen, rook, bishop, knight, and pawn).

Board

This class is used to represent the chess board. On initialization, it sets the the chessboard. This class contains method to print the board to console, the current board value (how good the the board look) for a colour, convert code indexes to board indexes and vice versa, as well as perform log actions. Log actions include, print to log and print the log to console.

Player

This class represents the players who will be playing chess. It contains variables to indicate the colour of the player, the number of piece the player has in the center (used for heuristics), the last move they performed, the number of times they repeated a move (used to track a condition for a draw), where the player can attack, and whether or not the player has lost the game. This class also contains methods to move a piece on the board, they calculate where the player can move, determine which actions the move resulted in, check for repeated moves, and the number of

pieces in center of the board. This class also has getter method for last moved piece, and setter and getter methods for if the player has lost.

Colour

This is an enumerable to make it easily readable to perform conditional statements that involve checking the player or current turn. The two colours are Black and White.

Game

This class manages the full game state containing both players, the board, and the turn order to ensure the game operates according to the game rules. It has methods to parse user input such as from the console to allow easier board modification, and it has methods to change the board state which then changes the turn. Validation of the next board states are performed in the board class, and some part by the piece itself.

Move

This class stores the information related to performing an action on the board that can be used in place of managing the individual start and next positions a piece moves on the board. This class is also used to store the full action as a log string in chess notation, and it stores the piece it captured for undoing actions.

Action

This is an enumerable used to differentiate the actions that can be performed by a piece. This is to make it simpler to do conditional statements that would be used for logging the move and action. The actions that can be performed by a piece are: Move, Capture, Promotion, Check, Checkmate, Stalemate, Castle King Side, Castle Queen Side, En Passant

Log

This class is used to keep track of all moves (done my any piece) on the board. It contains a move list, which is a list of all the moves performed so far. Moves can be added to the the move list to represent a move that has been perform, and the last move added can be removed to represent an undo. The class also has a variable to determine if its the master log file, this is important only the master needs to track actions, which is needed for the final log outputs.

GameTree (Algorithm)

This class is used to perform a game tree algorithm on a current game state to determine the next best move to be performed by each player and using a Min and Max approach to determine the best-worst case action. This class performs alpha-beta pruning to reduce the nodes expanded on to reduce memory and improve performance, and this is doing iterative deepening search.

Node

This class is used to store a move that is performed during a chess game and a value associated with that action, which the value is determined by the move itself or cumulative actions for the GameTree A* expansion. This has a variable amount of children which depends on how many valid actions can occur at a given board state.

Games

Games are played with adversary, this means there is a strategy or a plan to win. Due to this nature of games, adversarial search algorithms are used. Adversarial search algorithms are designed to return the best winning strategy through game trees, under the assumption that the player are adversaries (rational, self-interested, and play to win). Adversarial search algorithms use a static evaluation function which returns a numeric estimate of the state of the board in the perspective of the player; it can be positive for winning, or negative for losing.

Game Trees

A game tree is a tree which each node represents a board configuration, and branches are board transitions. In a game like chess, a 2-player turn based game, each level of the tree represents all possible moves by one player, followed by the other player, and so on; therefore each transition is a move. In most nontrivial games, such as chess, do not permit exhaustive searches as the trees are too large, and/or pure goal reduction as it is difficult to convert board positions into a simple sequence of steps. Therefore a proper search and heuristic is needed.

Searches

A*

A* search tries to avoid expanding paths which are already expensive. It does this by using an evaluation function which checks the cost (so far) to reach the current node, in addition to the

estimated cost to reach the goal from the current node. For A* to work, the heuristic must be admissible, as in the heuristic must never overestimate the cost to reach the goal.

BFS

BFS stands for Breadth-First Search. Breadth-first search expands the shallowest unexpanded node. However with this search, over time, space becomes an issue.

DFS

DFS stands for Depth-First Search. Depth-first search expands the deepest unexpanded node. DFS solves the space problem found in BFS, however it is not optimal as it can spend more time expanding deeper without exploring other solutions.

IDA*

IDA* search is a combination of A* and iterative deepening search. Iterative deepening search performs a depth-first search with a depth limit l. As in it will perform a depth-first search, with the same root, but the layer will increase at each iteration. This will allow searching of all nodes at each layer, solving the optimality problem found in DFA*. The IDA* search is used in our chess program.

Min max

Minimax is a search tree algorithm usually used on games like chess. The two players are represented as MAX or Maximizer and MIN or Minimizer. where MAX moves first, followed by MIN; they takes turns until game is over. When examining the game tree, MAX wants to obtain the highest static evaluation score at each level, under the presumption that the opponent has access to the same evaluation scores and will prevent MAX from obtaining the best score. This is the opposite for MIN, where MIN will try to obtain the lowest static evaluation score. Thus the game tree consists of alternating Maximizer and Minimizer layers, where each layer presumes that the player desires the best score for themselves. In minimax, the search tree is expanded ply (total number of levels in tree, including the root) p deep. A static evaluation is done on all expanded leaf configurations, and are then selected under the presumption that the opponent will force you to make the least desirable move for yourself, while making the best for themselves. In the case that the opponent (human) does not play optimally, then it will be even more advantageous to the computer. However a drawback to minimax is that it requires the entire subtree of ply depth p to be generated, so the number of game states is exponential to the number of moves, which can be really harsh time wise. For example, in a “reasonable” game of chess, the time complexity for minimax is about 35^{100}.

Alpha-Beta Pruning

Alpha-Beta pruning is the solution to the time complexity problem of the minimax procedure. Pruning refers to the deletion of unproductive searching in search trees. Pruning involves ignoring entire branches, where a qualitative decision about searching a branch is made with regards to a search strategy, such as the evaluation function. Alpha-Beta pruning builds on the minimax procedure, as it adds two additional variables called Alpha and Beta and prevents the unnecessary evaluation of whole branches. Alpha tracks best choice found so far for MAX. Beta tracks the value of best choice found so far for MIN. The decision to ignore a branch is based on knowing how your opponent will react if a clearly good move is available for them. As in the decisions are made under the presumption that the opponent is as intelligent as we are, and will make the moves which are advantageous for themselves, and disadvantageous for us. Since pruning does not affect final results, entire subtrees can be pruned, and with good move ordering the effectiveness of pruning can be improved. In the best case scenario, with "perfect ordering", alpha-beta can cut exponential growth rate by half. For example, in chess the time complexity would be $35^{100/2}$. This means that Alpha-Beta pruning can look twice as far in the same amount of time as minmax. Alpha-Beta pruning is used in our chess program.

Heuristic Evaluation Function

Heuristic evaluation function produces an estimate of the expected utility of the game from a given position. For example in chess, how good does the board look from a player's perspective. In our chess program, the evaluation function checks, what can be threatened, considers pieces which are threatened, and control of center board. What can be threatened, use the weight of each piece, King is 2^{31}-1 (Integer.Max_Value), Queen is 9, Rook is 5, Bishop and Knight are 3, and Pawn is 1. Evaluating what can be threatened, allows for more aggressive playing to encourage taking pieces. Considering which pieces are threatened, using the weight of the pieces, allows to avoid very aggressive actions so it does not needlessly sacrifice pieces. Control of center is a general rule of thumb in chess which just states that the more pieces a player has in the center of the board, the easier it is for the player to win as they are capable of threatening more.