Edinburgh Napier UNIVERSITY

Coursework Report

Paulius Bieksa
40216180@napier.ac.uk
Edinburgh Napier University - Computer Graphics (SET08116)

Keywords – OpenGl, GLSL, Phong, Graphics, Portal, Stencil buffer, Colour correction

1 Introduction

Project Aims The aim of this project is to show a theoretical and practical understanding of algorithms and data structures by implementing a checkers game. Large emphasis is placed on choosing the correct data structures to represent data in the program, as well as designing efficient algorithms to facilitate gameplay and other features of the program.

Russian checkers variant The program implements the Russian checkers variant of the rule-set. There are four main differences from the standard rule-set of international checkers: pawns can capture diagonally backward as well as forward; kings (promoted pieces) move any number of spaces diagonally forward or backward, and can capture any number of spaces diagonally forward or backward as well; it is not mandatory to make the longest possible jump; If a pawn reaches the last row while jumping, it promotes to a king and continues jumping immediately. The main way this impacted the task is having the modified movement of kings in this variant of the game increase the complexity of the algorithms used to find possible moves.

Features The program can accommodate games for player versus player(Figure 1), player versus AI and AI versus AI in any configuration of the two. The game itself has an option to undo any number of moves made and redo any undone moves. The game provides choices for the for valid pieces and moves for the player to avoid having to input each move and thus speed up the game. The program can also replay the last completed game.

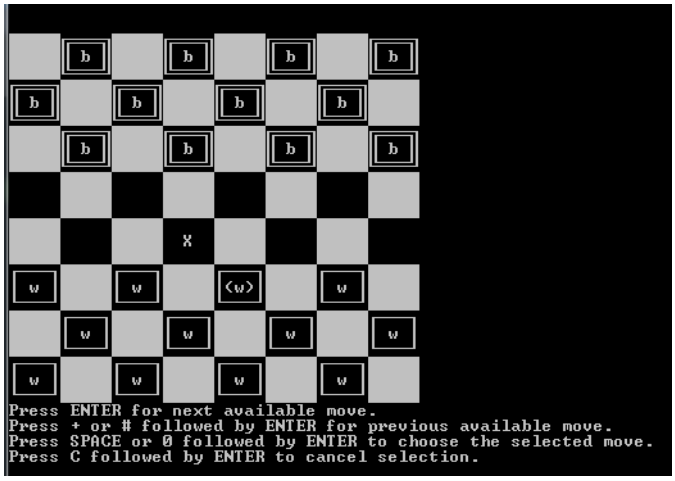

Figure 1: **Player versus player game**

2 Design

2.1 Game logic

Data structures The main game states are represented by a 2-dimensional array of enumerator values to represent the game-board and two lists of 'piece' structures to keep track of pieces still in play for each player. The 'piece' structure contains the position of a piece and whether or not he piece is a king, while the enumerator representing the spaces on the board contains values that can represent any possible state a space on the board can be in.

To represent move history the program uses a deque of 'move' structures. This data structure is used because it only allows access to elements at the ends of the deque. This way it ensures that moves insied are used in the correct order. The reason a deque is used instead of a stack is that in addition to being used to undo moves the history container is also used to replay previous games and needs to be accessed for both ends. The The undone moves are stored in a stack for the reasons mentioned above.

Algorithms The game logic class has methods that generate all available choices for pieces that can move this turn and all valid moves the current player can take. The class has it's own internal player state and therefore does not need to use different function calls for different players, and the player state value is updated whenever the next players turn is reached. Each move in the game logic class is consider to be a single hop of a single piece with a possible capture. A turn, which can be comprised of multiple moves (when a piece can perform chained captures), is not defined in the game logic class, but the endpoint of a turn is detected and the player state updated accordingly. While the class allows for direct move input this is only used for AI, because making moves in the game can be made more intuitive and sped up considerably when choosing form all available pieces with minimal button clicks. The chosen move is executed with the 'executeMove' function which returns an integer that represents

the outcome of that move (turn not complete, turn complete, game over). Undo and redo functions undo/redo an entire turn by going through each 'move' entry in their respective containers until it finds that a player state has to be toggled to the other player.

2.2 AI

Data structures The AI class has game logic object which is used to calculate multiple permutations of future turns without changing values on the game logic object that the actual game is running on. The AI uses an n-tree (Figure 2) structure to generate all possible permutations a single turn from a predetermined starting move. Besides n-trees the AI uses a combination of lists and resizable arrays (c++ vectors) for its intermediate operations when the best move to make is being calculated. Lists are preferred to arrays do to the cost of recreating the array when adding elements, but in cases where specific members of the container need to be accessed arrays are used.

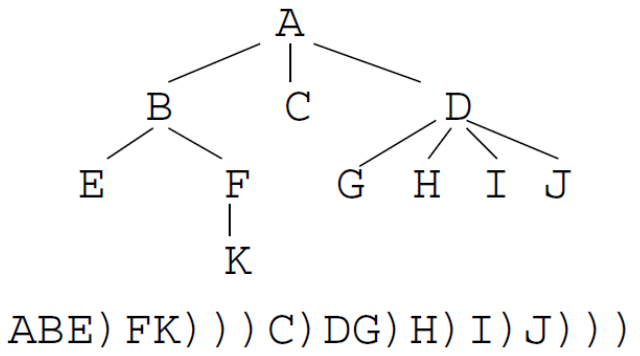

Figure 2: **N-tree**

Algorithms The AI generates all possible turns up to a given depth (depth being the number of consecutive turns) and scores all of those turns based on pieces captured/lost and own/opponent pieces. These scores are calculated recursively for each possible first turn and the turn with the highest score is taken.

3 Enhancements

GUI While the game was designed to be played on command line, the game logic is encapsulated in it's own class and has methods that facilitate play regardless of input/output method. Thus, the most obvious choice is adding a graphical user interface. A GUI would streamline the game inputs even further and overcome the issues of displaying the game-board using only monochrome ASCII characters.

AI The AI could also be improved by tweaking some values or perhaps adding some sort of heuristic algorithm instead of the deterministic all available move approach.

Additional rule-sets The game could have options to use different rule-sets, most of which would require minimal additions to the code. Most of the different variants of checkers rely the basic rules like moving pieces diagonally, which is already implemented and could be easily adapted to suit various variations on the core concept.

4 Critical evaluation

Predetermining choices Giving the player choices that have already been calculated removes the need for sanitizing the users input. The validation for moves is unnecessary because it is already built in to the functions that generate those lists. If the programs interface would be changed and moves were selected directly those same functions could be used to validate those inputs.

AI algorithms The algorithm used to calculate the best moves was created without consulting any materials relating to AI programming or AI approaches in regards to checkers specifically. The algorithm is not mathematically proven or even extensively tested. As such, some moves are sub-optimal. While from testing I found that the AI plays better than just moving a random piece, it has a lot of space for improvement.

5 Personal evaluation

All paths AI The AI in this project uses an all possible paths approach to get the best possible move. This is very inefficient, because even with the average of about 5-6 possible moves per turn the exponential nature of the algorithm ends up with very large numbers of permutations with even a small depth. A better approach would be to use some sort of heuristic evaluation that uses predetermined roles of thumb to get the best moves available.