Computer Checkers - Turn-Based Board Game

Gordon Swan
40202556@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Abstract

Checkers - or English Draughts - is a well-known strategy game played on a black and white checkerboard. Each player has twelve pieces, which they can use to move across the board, and to capture opponent pieces. Movement of pieces is limited to only the black spaces on the board, meaning that all movement is diagonal. Certain challenges are faced when implementing a board game in digital form, and Checkers is no exception. Implementing Checkers as a computer program requires numerous data structures to represent the board state and various complex algorithms to provide consistent and accurate gameplay.

Keywords – checkers, game, board, computer, c++, cplusplus, turn, play

1 Introduction

In addition to the standard rules of checkers, this program will include numerous additional features such as the ability to undo and redo moves, display all of the past moves for the current game, play against a computer-controlled opponent and have the computer act on the player's behalf. Effective implementation of these features requires concise and robust design of algorithms designed to interpret the data representation of the board and it's current state.

2 Design

Point Type Definition One of the fundamental structures of Checkers is the location of game pieces on the board state, these pieces need to be represented on a two dimensional plane, which requires storage of an X co-ordinate and a Y co-ordinate. A decision was made to implement a custom type that allowed for both an X and Y coordinate to be stored and referenced as such. This type consists of two integer variables, appropriately labeled 'x' and 'y', which together represent a board location in two dimensional space.

Representation with OO Concepts An object-oriented solution seemed the obvious choice when tasked with implementing a game that originally existed as a real world object, with the board and each piece being it's own separate game object. This design choice makes it simple to project the physical attributes of each of these parts into software attributes. In addition to the physical attributes, the rules and instructions associated with these parts can be represented as object methods in object-oriented software.

Board Representation In contrast to the real-world counterpart, the program does not represent the entire board as a singular object but rather a collection of objects which represent 'positions.' Positions are the locations on the board, which can hold a game piece and are stored in a two dimensional vector data structure - a vector of vectors essentially. This allows for pieces to easily be placed and moved on the board representation. Position objects have defined methods, which ensure the validity of the board, for example, only black-attributed locations may actually host a piece, as Checkers is played on the black squares only.

Defining and Representing Pieces As mentioned previously, pieces are represented as software objects, containing their own defining attributes and rules. Pieces are defined by their colour, and by their status, which may be either standard or king. King pieces may move in the relative backwards direction as well as forwards, whereas standard pieces may only move relative forwards.

Flow of Control with Nested Loops The primary function of this program - from which all other functions stem from - is implemented using two infinite' loops. These loops ensure that the next turn is requested and played sequentially, and that a new game is created should the current game reach completion. Infinite loops are a fundamental component of all video games as they ensure that new information is consistently requested so the game may progress.

Initialization of the Game In order for a new game to be created and played, the board state must first be set up with the correct number of pieces for each side. The board state is first initialized by allocating a new 'Position' instance to each board location. Pieces of appropriate colour may then be located to the correct starting positions. The primary design decision here was to populate the board with pieces using an algorithm, as opposed to 'hard-coding' the starting positions on the board representation. The initialization function can be seen in Listing 1.

Listing 1: Game Initialization - C++

```
1  void Game::initGame()
2  {
3      // Initialise the gamestate
4      board.resize( 8, vector<Position>(8));
5      p_turn = 0;
6      // Init all board locations with the correct colour
7      for(int i = 0; i < 8; i++){
8          for(int j = 0; j < 8; j++){
9              if(i % 2 == 0){
10                 if(j % 2 == 0){
11                     // 0
12                     board[j][i].setColour(1);
13                 }else{
14                     // 1
15                     board[j][i].setColour(0);
16                 }
17             }else{
18                 if(j % 2 == 0){
19                     // 1
20                     board[j][i].setColour(0);
21                 }else{
22                     // 0
23                     board[j][i].setColour(1);
24                 }
25             }
26         }
27     }
28     // Init board with pieces on the first and last three rows.
29     for(int i = 0; i < 3; i++){
30         for(int j = 0; j < 8; j++){
31             if(board[i][j].getColour() == 0){
32                 point loc;
33                 loc.x = j;
34                 loc.y = i;
35                 board[i][j].setPiece(new Piece(0, loc));
36             }
37         }
38     }
39     for(int i = 5; i < 8; i++){
40         for(int j = 0; j < 8; j++){
41             if(board[i][j].getColour() == 0){
42                 point loc;
43                 loc.x = j;
44                 loc.y = i;
45                 board[i][j].setPiece(new Piece(1, loc));
46             }
47         }
48     }
49 }
50
```

3 Enhancements

Given more time, there are numerous additional features which would have greatly benefited the functionality of this program.

Graphical User Interface One of the most important aspects of this particular program is a user's ability to understand and use it. Therefore, by adding a graphical user interface, allowing for more natural interaction - such as drag and drop - would greatly enhance the user's ability to play and enjoy the game itself. Given more time, a simple user interface could be implemented using such a library as SDL (Simple DirectMedia Layer)[1].

Save / Load In addition to a graphical user interface, additional functionality, such as saving and loading game states could provide a more casual aspect to the game, as users would not be tied to playing an entire game in one sitting. Given more time this feature could easily be implemented using file reading and writing, and some simple changes to the existing functions already contained within the program, which are used for undo and redo functionality.

Artificial Intelligence More adept Chess players will find the included computer opponent to be too simple as the computer does not consider or evaluate the consequences of it's actions. Given more time, the program would benefit from the implementation of a full min-max artificial intelligence script, which would pose as a much harder computer opponent than the one currently included.

4 Critical Evaluation

4.1 Current Key Features

Undo and Redo Functionality One feature, which this program implements well is the ability to undo and redo moves during the game. The program is able to undo all of the played moves, taking the board state all the way back to it's original starting position. and can redo all of the moves again back to the most recent move. In addition, the program will clip the stored list of moves if a user rolls back their moves an plays a different move instead. This solves a key problem where a user would otherwise have been able to redo moves that were no longer valid as the board state has changed considerably since the move was first saved.

Move Recording and Replay In addition to the undo and redo functionality, the program is able to display all of the played moves from the beginning of the game up to the current state, as all of the moves are saved sequentially in a vector. The replay functionality is powered by the same vector of moves which supports the undo and redo functionality as mentioned previously.

4.2 Possible areas of Improvement

Memory Management One considerable flaw with this program is memory management. As C++ does not have an automatic 'garbage collector' instances of moves and pieces remain in memory long after their relevant lifetime has ended. This is a considerable flaw as it is considered as a classic memory leak, and should the game be played extensively without a restart, it would consume all of the available memory. This however, is not a huge issue on modern computers which have large amounts of memory, but still a considerable concern that should be fixed given more time.

User Input and User Experience As mentioned previously, the program could benefit from the addition of a proper graphical user interface. The current user experience is extremely limited due to the entire program relying on the standard input and standard output of the terminal. This means that although the application features robust and concise menus, the general user experience feels clunky and counter-intuitive. The user input interpretation for the main menu can be seen in Listing 2.

Listing 2: User Input Interpretation using Select Case in C++

```
1   switch(stoi(usrInp, nullptr, 10)){
2       case 2: cout << "<---\n" << endl;
3               m_game->rewindState();
4               m_input = true;
5               break;
6       case 3: cout << "--->\n" << endl;
7               m_game->redoState();
8               m_input = true;
9               break;
10      case 4: cout << "...asking Siri" << endl;
11              m_game->autoSelect();
12              m_input = true;
13              break;
14      case 5: cout << "...playing all moves" << endl;
15              m_game->autoPlay();
16              m_input = true;
17              break;
18      case 6: cout << "Quitting..." << endl;
19              m_input = true;
20              m_active = false;
21  }
22
```

5 Personal Evaluation

In general this program is a strong attempt at implementing a feature-rich computerized version of the Checkers board game. However, producing the final product has not been without numerous challenges.

5.1 Learning Experiences

Data Representation with Vectors Learning to use existing data structures that exist in C++ has been cumbersome, however in the long run has lead to a better understanding of how data can be represented in software. C++ vectors have provided a simple and consistent method of storing and representing the game board and all of it's data.

Optimization Creating a software solution is one thing. Doing it well is another. A key aim for this project in general has been to solve all of the problems and challenges with the simplest and most efficient way possible. This can be seen above in Listing 1, as the program uses an algorithm to generate and populate the initial board state, rather than a hard coded list or string. In addition, care has been taken to find the most optimal way to detect moves as well as collecting user input without sacrificing too much playabillity.

5.2 Challenges and Solutions

Diagonal Move Detection One considerable challenge faced during development was the detection of possible moves for all pieces. This is a function which returns a vector of all the possible moves for all of the pieces that belong to the current player. This challenge was overcome by defining the diagonal offsets as points, then implementing custom modifiers for multiplying, adding and subtracting points from one another, as well as comparisons such as greater than or less than. This makes it simple to not only get the diagonal offset positions, but also to ensure that they are valid, within the bounds of the game board, as not to cause errors or false positives when detecting moves.

User Interaction and User Experience Another considerable challenge faced during development was providing full-feature control of the program using only the standard input and standard output of the terminal. This program can only print one line at a time and can only read one line at a time, implementing game controls using only these input-output methods has been difficult. This challenge has been overcome by implementing menus that are displayed to the user each turn.This has made the user experience a bit more cumbersome but has exponentially improved the ability to control the program.

6 Conclusion

In conclusion, this program effectively achieves it's goal as a computerized implementation of the well-known Checkers - or Draughts - board game. The program has implemented numerous additional features such as reversing and replaying moves, displaying all previous moves and automated play against a computer opponent. In addition to the standard requested features, this program can even use the automated play function to act on behalf of the player, even during a standard player versus player match. These features, coupled with the robust implementation of the classic rules makes for a very usable, and entertaining game of Checkers.

References

[1] libsdl.org, "Simple directmedia layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via opengl and direct3d."