Coursework Report

Alexander Barker
40333139@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Abstract

The aim of this project is to research, design and implement a checkers game, using any programming language, to demonstrate understanding of both theory and practise in relation to algorithms and data structures. The goal is to research software programming best practices in order to produce a functioning and efficient game with a number of additional features.

Keywords – C#, Visual Studio, SET09117, Algorithms, Data Structures, Napier, Checkers, Draughts, 40333139

1 Introduction

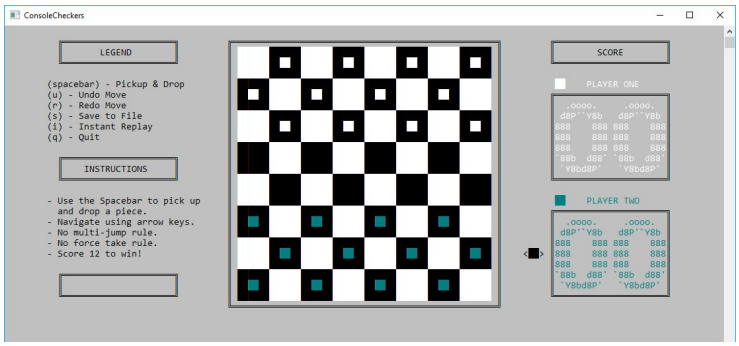
This report will describe the design, implementation and justification for a standalone application of checkers. The task of this project is to accurately represent a game of checkers in software and allow the user to interact with the application. Using knowledge gained from lectures and self-study, appropriate algorithms and data structures are chosen to represent the game board, piece positions, players and other game-state data. This will then be used to enable game modes such as Player vs Player, Player vs Computer and Computer vs Computer with additional features such as recording of play, undo move, redo move, instant replay, saving to file and loading from file. A simple, custom AI was created to enable matches against the computer.

The application was created with the programming language C# in the Visual Studio development environment and runs from the command line console.

2 Design

The user interface is text-based with restricted user input via the keyboard. The design of user interface elements are drawn at run-time and re-drawn after each user input. The program has a class structure with each class performing a specific function such as; the Score class is responsible for storing and displaying the score designs.

Fig 2.0 - Full Game Screen



2.1 Menu

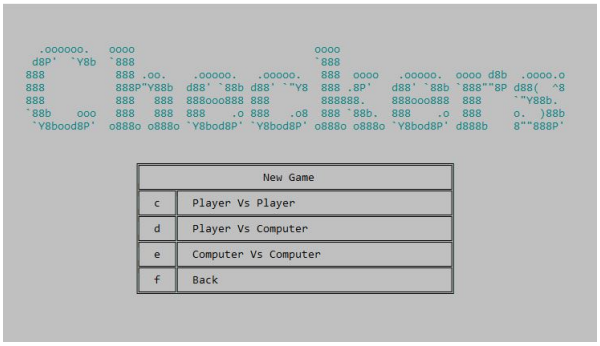
The starting point of the application is the main menu where the user is presented with a title screen and menu options. The user can choose to start a new game, load a game or quit the application using the specified keyboard inputs. Upon choosing an option the next menu screen will then change to display options for game modes. This was implemented using the switch-case statement to enforce selection control.

Below is a list of menu options with their assigned key:

- "a" - New Game
- "b" - Load Game
- "c" - New Player vs Player
- "d" - New Player vs Computer

- "e" - New Computer vs Computer
- "f" - Back
- "g" - Load Player vs Player
- "h" - Load Player vs Computer
- "q" - Quit

Fig 2.1 - New Game Screen



2.2 Board

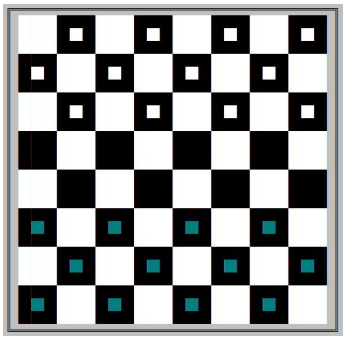
The game board consists of 8x8 white and black alternating squares drawn at run-time. This is achieved by hard-coding cursor positions into two arrays to represent the (x,y) coordinates with a multi-dimensional array to identify the colour.

Listing 1: Storing the board design

```
1 public int[ ] piecePositionsX = { 46, 52, 58, 64, 70, 76, 82, 88 };
2 public int[ ] piecePositionsY = { 3, 6, 9, 12, 15, 18, 21, 24 };
3 public Board(){
4     squares = new int[,]{ { 0, 1, 0, 1, 0, 1, 0, 1 },
5                           { 1, 0, 1, 0, 1, 0, 1, 0 },
6                           { 0, 1, 0, 1, 0, 1, 0, 1 },
7                           { 1, 0, 1, 0, 1, 0, 1, 0 },
8                           { 0, 1, 0, 1, 0, 1, 0, 1 },
9                           { 1, 0, 1, 0, 1, 0, 1, 0 },
10                          { 0, 1, 0, 1, 0, 1, 0, 1 },
11                          { 1, 0, 1, 0, 1, 0, 1, 0 } };}
```

Using the piece position data, the cursor can be placed, offset and iterated upon to produce solid black or white squares. The colour is achieved by changing the text colour depending on the value held within the 2-D array.

Fig 2.2 - Board Design



2.3 Pieces

Pieces are stored and drawn in a similar fashion to board squares however their values can be updated depending on user input. When the user picks up or drops a piece, the 2-D array that holds the value is updated by cross-referencing its position with the cursor. Cursor movement is restricted to the centre of black squares. This allows the (x,y) coordinate arrays to equal the exact location in the multi-dimensional array matrix. Re-drawing the pieces with this new data will simulate a physical pick-up, move and/or jump.

Below is a list of piece values:

- "[0]" - Empty
- "[1]" - White
- "[2]" - Black
- "[3]" - White King
- "[4]" - Black King

Listing 2: Storing the piece values

```
1 public int[,] pieceValues = new int[8, 8];
2 public Piece(){
3     pieceValues = new int[,]{ { 0, 1, 0, 1, 0, 1, 0, 1 },
4                               { 1, 0, 1, 0, 1, 0, 1, 0 },
5                               { 0, 1, 0, 1, 0, 1, 0, 1 },
6                               { 0, 0, 0, 0, 0, 0, 0, 0 },
7                               { 0, 0, 0, 0, 0, 0, 0, 0 },
8                               { 2, 0, 2, 0, 2, 0, 2, 0 },
9                               { 0, 2, 0, 2, 0, 2, 0, 2 },
10                              { 2, 0, 2, 0, 2, 0, 2, 0 } };}
```

Note: Black pieces are represented by the Dark Cyan colour.

2.4 Moves

Navigation around the board is done via the arrow keys with each key-press moving the cursor to the next hard coded (x,y) coordinate. Moves are made using the spacebar key which will trigger the AllowPVPMovement() function within Move class. This function is responsible for determining if you are holding a piece, the piece type, starting position and movement position. Using this data, validation checks are performed to ensure the following:

- You are not trying to pick up an empty piece.
- You are not trying to pick up the opponents piece.
- You are not trying to place a piece on top of your own.

- You are not trying to place a piece too far away from the starting position.

If this validation process comes back "true", the pieceValues 2-D array will be updated by making the starting position equal to zero and the movement position equal to the piece type.

Listing 3: Updating pieceValues 2-D array

```
1 // example of changing values based on user input.
2 piece.pieceValues[startingPositionY, startingPositionX] = 0;
3 piece.pieceValues[movementPositionY, movementPositionX] = ↩
      pieceType;
```

Each move is then added to a dictionary data structure by using a dictionaryIndex variable as a key value, paired with the current pieceValues array. This dictionary will receive a copy of each move to record game play

At the same time, the game state is also recorded in a separate dictionary. This includes; scores, turn, current player and movement position coordinates.

Listing 4: Recording of play

```
1 public Dictionary<int, int[,]> moveList = new Dictionary<int, ↩
      int[,]>();
2 public Dictionary<int, int[]> gameState = new Dictionary<int, ↩
      int[]>();
3 // ...
4 piece.moveList.Add(dictionaryIndex, (int[,])piece.pieceValues.↩
      Clone());
5 gameData[0] = playerOneScore;
6 gameData[1] = playerTwoScore;
7 gameData[2] = turn;
8 gameData[3] = player;
9 gameData[4] = movementPositionX;
10 gameData[5] = movementPositionY;
11 piece.gameState.Add(dictionaryIndex, (int[])gameData.Clone());
```

Normal pieces have restricted movement in the direction they are supposed to go, however king pieces can make moves in both directions. The restriction in movement is performed by the validation algorithm which only returns true when the movement position values are either -1 or +1 of the starting position values.

Kings are crowned at the post validation stage where the move is valid and hits the corresponding y coordinate.

2.5 Jumps

Jump moves can only be performed if the ValidateJumpMove() function returns true. This validation algorithm first checks to see if the movement position will be outwith the bounds of the board. The next check performed is whether the piece to be taken is of the opposite colour. The third validation checks to see if the space in the desired direction is empty. If all these conditions are met, the new data is added to the dictionaries and the board is updated to display the new data. Jump moves are restricted in movement similar to normal moves, however the valid conditions return true when the movement position values are either -2 or +2 of the starting position.

2.6 Scores

Scores are incremented post-validation of jump moves. To display the scores the ScoreUpdater() function within the Score class is called. This function stores the score designs and is responsible for drawing the design on the screen. Score designs are stored within a string array for each score and consist of ASCII characters.

Listing 5: Example of score design

```
1 string[] four = new string[] { "  .oooo.              .o      ",
2                                " d8P''Y8b           .d88      ",
3                                "888     888      .d'888      ",
4                                "888     888    .d'  888      ",
5                                "888     888 88ooo888oo ",
6                                "'88b   d88'         888      ",
7                                " 'Y8bd8P'         o888o    "};
8 // Display the score design
9 for (int i = 0; i < 7; i++)
10    {
11        Console.SetCursorPosition(104, (i + 7));
12        Console.ForegroundColor = ConsoleColor.White;
13        Console.Write(four[i]);
14    }
```

The above score design will be displayed if the current player is one and their score is four. This is performed by querying a switch-case statement.

2.7 AI

The AI used for the application is a custom AI, written based on a random move basis. This allow the user to play a full match against the computer or watch the computer play itself. For each move the AI performs, a number of list data structures are used. The first action the AI performs is to generate a list of pieces on the board and their locations where the list stores their starting (x,y) coordinates. An algorithm is used to filter out each piece by colour. For each piece in the list a new list is created for all possible and valid moves and a separate list for valid jump moves.

The AI will generate a random number based on the number of items in the list. This number is then used to pull out a possible move from the starting

position list by taking the (x,y) coordinates, using that to determine the piece type and the matched movement position. The move goes through one more validation check and if it returns true, then the while loop is broken and the move is recorded and displayed.

Listing 6: Example of AI

```
// This code refers to a White King, normal move.
if (aiValidWhiteJumpMoves.Count == 0){
    int chosenMove = rnd.Next(aiValidWhiteMoves.Count);
    int[] temp = (int[])aiValidWhiteMoves[chosenMove].Clone();
    movementPositionX = temp[1];
    movementPositionY = temp[0]; // Flatten the data structure.
    int[] temp2 = (int[])aiValidWhiteStartingPositions[chosenMove].Clone();
    startingPositionX = temp2[1];
    startingPositionY = temp2[0];

    pieceType = piece.pieceValues[startingPositionY, startingPositionX];

    if (pieceType == 3) {
        valid = ValidateNormalMove(piece.pieceValues, player, pieceType, holding, playerOneScore, playerTwoScore, turn, movementPositionX, movementPositionY, startingPositionX, startingPositionY);
            if (player == 1 && valid == true){
                dictionaryIndex++;
                player++;
                turn++;
                piece.pieceValues[startingPositionY, startingPositionX] = 0;
                gameData[0] = playerOneScore;
                gameData[1] = playerTwoScore;
                gameData[2] = turn;
                gameData[3] = player;
                gameData[4] = movementPositionX;
                gameData[5] = movementPositionY;
                piece.gameState.Add(dictionaryIndex, (int[])gameData.Clone());
                piece.moveList.Add(dictionaryIndex, (int[,])piece.pieceValues.Clone());
                valid = false;
                break;}}
```

2.8 Player vs Player (PvP)

The PvP game mode allows two users to play a game of checkers against each other. The turn indicator will change position after a valid move or jump is performed. There is no force jump or multi-jump rule enforced in this game mode. The first player to score twelve wins the match.

2.9 Player vs Computer (PvC)

The PvC game mode allows one user to play against the computer. The AI player will use the white pieces and will perform a move or jump six hundred milliseconds after the user has successfully completed a turn. The AI player will be forced to take a piece if a jump is available otherwise it will move around the board based on a random choice of valid moves held within a list data structure.

2.10 Computer vs Computer (CvC)

The CvC game mode allows the user to watch the AI play a game of checkers by itself. Each turn is delayed by six hundred milliseconds to make it easier to see the moves take place. The match progresses by performing a while loop over the PickAWhiteMove() and PickABlackMove() functions depending on which player turn it is. The match can be interrupted by pressing any key.

2.11 Undo/Redo

The undo and redo features can be accessed during a PvP or PvC match by pressing the "u" or "r" key. The variable dictionaryIndex is responsible for tracking where in the dictionary a set of game states or piece positions exist. To implement the undo and redo features the dictionaryIndex is either incremented or decremented by 1 during a PvP match or 2 in a PvC match. The undo/redo code is also responsible for adjusting game state data, so if a player has just scored, the score will be adjusted along with other game data.

Listing 7: Truncated code for undo (PvC)

```
case ConsoleKey.U:
// ...
foreach (KeyValuePair<int, int[,]> pair in piece.moveList)
    {
    if (piece.moveList.ContainsKey((dictionaryIndex - 2)) == true)
        {
        piece.pieceValues = (int[,])piece.moveList[(dictionaryIndex - 2)].Clone();
        }
    }
foreach (KeyValuePair<int, int[]> pair in piece.gameState)
    {
    if (piece.gameState.ContainsKey((dictionaryIndex - 2)) == true)
        {
        gameData = (int[])piece.gameState[(dictionaryIndex - 2)].Clone();
        playerOneScore = gameData[0];
        playerTwoScore = gameData[1];
        turn = gameData[2];
        player = gameData[3];
        movementPositionX = gameData[4];
        movementPositionY = gameData[5];
        }
    }
// ... Turn indicator code lives here
dictionaryIndex++;
dictionaryIndex++;
piece.moveList.Add(dictionaryIndex, (int[,])piece.pieceValues.Clone());
piece.gameState.Add(dictionaryIndex, (int[])gameData.Clone());
score.ScoreUpdater(player, playerOneScore, playerTwoScore);
board.ReDrawBoard();
piece.SetPieces();
// ...
break;
```

2.12 Instant Replay

The instant replay feature can be accessed during either a PvP or PvC match by pressing the "i" key. This will trigger a for-each loop to retrieve every set of data held within the moveList and gameState dictionaries, one at a time. Each pair of data sets will be displayed sequentially with a delay of six hundred milliseconds and cannot be interrupted.

2.13 Save/Load

The PvP and PvC game modes allow the user to save their current game to two .CSV files by pressing the "s" key. This feature will open up two Streamwriter operations; One for the moveList dictionary and one for the gameState dictionary. These save files will be stored in the directory location where the application is run from. e.g. If you launch the application from the desktop, the files will be saved to the desktop.

Loading files can be accessed via the main menu. Selecting "Load Game" will display a second menu where you can choose to load the data into either a PvP game or a PvC game. The Streamreader takes in each line of 64 values for a game board and then the code re-formats the values into the 8x8 multi-dimensional array. This array is then added to the moveList dictionary, one by one. The same is done for the game state data where 6 values of gameData is cloned to the gameState dictionary.

2.14 Other Features

To provide feedback to the user about the actions they perform, a feedback box has been implemented. This will display information such as "Replaying...", "Saving...", "Player One Wins!" etc.

Fig 2.3 - Feedback Box

3 Enhancements

The application operates in an efficient and robust manner, however, with more time, additional features and improvements could be achieved.

The flickering nature of the game board is a result of refreshing the entire board and pieces after every key press by the user. This could be improved upon by only refreshing the affected areas in the text-based graphical user interface. This would also be more efficient on the system and provide a smoother user experience.

The basic game logic for PvP matches does not include force-take or multi-jump rules. As most rule sets for a game of checkers involves these rules it would be better to include them to more accurately represent the game.

The AI is based on a random number generator on the number of available moves or jumps. This could be improved drastically by implementing a more intelligent AI based on the Minimax and Alpha-Beta pruning methods. Weighting each move by its importance to achieving the goal of tracking a piece across the board or becoming a king piece would increase the difficulty of gameplay.

4 Critical Evaluation

The undo and redo feature is limited in scope. The undo only allows for one step back and the redo must be used in conjunction with undo otherwise it will undo one step back also. The feature would be better served by allowing the user to return back to any point in the match.

The custom AI that was developed for this application consists of many, many lines of code when compared to other solutions in the industry. The code for collecting valid moves and jumps works well but is long-winded and inelegant. When compared to AI designs that take full advantage of data structures such as stacks and trees, have the capacity to make more intelligent moves over a random-based implementation. Writing the AI from scratch resulted in approximately 2000 lines of code dedicated to coming up with moves and jumps for both white and black pieces.

5 Personal Evaluation

This project was fun and a great experience as I have now become comfortable with a new programming

language. This was the first application I have written using the C# language. This was also the first project where I used a version control application during development, SourceTree. I have included a history of my GitHub commits via SourceTree in the appendix. I have also included my first use of a documentation method that outputs a .XML file. The syntax used for documentation was new to me and I have included it in the project folder and appendix.

I chose to take on the task of creating basic game logic for the AI used in this application. I feel like I got it to a good place, where a user can enjoy playing against the computer and the CvC mode does not take too long or tax the system to hard. It operates quickly and prefers to take pieces rather than ignoring jumps for moves.

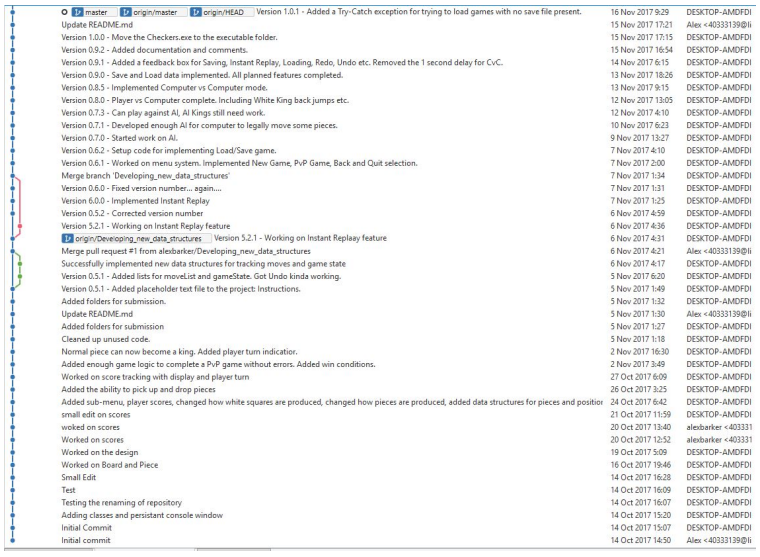
I believe I have included a number of good additional features and I am happy with how I was able to make a command line, text-based user interface look like and behave like a checkers board.

Appendices

A GitHub History

Direct link to commit history (GitHub): https://github.com/alexbarker/barker_alexander_set09117/commits/master

Version Control



B Documentation File

Direct link to the file (GitHub): https://github.com/alexbarker/barker_alexander_set09117/blob/master/Documentation.xml

XML Documentation