THINK OF A TITLE FOR REPORT

Marc Reid
03001588@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Abstract

1 Introduction

The purpose of this report was to document the process of implementing the game of Checkers with a command line interface in C #.

2 Design

During the project a test-driven development approach was used in conjunction with an agile methodology. This is where the code is developed incrementally in sprints with the project only moving on to the next sprint once the testing of the work done, has been passed. We will look at the following 3 main areas which demonstrate the design choices that were made during development.

2.1 Interface

The application runs from the command line as this is what was asked for in the coursework specification with the proviso that we could develop the application with a graphical interface later.

2.1.1 CLI Playing Board

To represent the board visually in the CLI, characters from the Windows character map application were used to create a custom design. The priority when designing the CLI board, was to ensure that it was easy to read with good spacing to stop the screen from becoming cluttered.

2.2 Data Structures Used

We highlight the essential data structures used and explain why we used these specific data structures in the circumstances which we did. The data structures used have allowed us to provide the features included in the application such as undo/redo and replay.

2.2.1 2-Dimensional Array

As the board for English draughts is an 8 x 8 chequered board, a 2 dimensional array[8,8] named *positionsArray* is used to represent the board and store the current locations of the playing pieces. The array is updated after every turn is made, with all the contents of each index of *positionsArray*, concatenated together separated by a comma into a single string with the last value either a 'X' or an 'O' to indicate who has just taken their turn. Every time the string is created it then gets popped on to a stack called *undoStack* and enqueued in to a queue called *replayQueue*.

2.2.2 Queue

The queue data structure was selected for the replay feature as it uses the First-In-First-Out method for storing and retrieving information. This meant that we could add (enqueue) the string comprised of the current playing piece positions to the back of the queue after every turn. When the user selects to replay the current match, the application enters a loop which will iterate through all the string stored in the queue. A pause has been added to the loop, so that the replay can be followed by the human eye.

2.2.3 Stack

The stack data structure is used to aid the implementation for both the undo and redo features. A stack uses the Last-In-Last-Out method for storing and retrieving information, which meant we could synergise the stacks together, to provide robust undo and redo features.
Both the queue and stack data structures selected offer improved performance over the other data structures available which could perform similar tasks, such as an array or a list.

2.2.4 List

The list data structure is used twice in the application, both times it has been used as the number of values it would store at any one time will vary from move to move. It was decided that as the size of the data structure needed could not accurately be predicted a list would be the best option as it adjusts its size as needed. This removes the potential for an out of range exception when accessing the data structure.

2.3 Algorithms Used

The application has many algorithms which when all combined help to produce a robust version of the game checkers. We will look at the algorithms that are essential for the application to simulate multiple games of checkers.

2.3.1 Valid Move Check

If a player has no valid moves available, then they lose the game. To implement this rule, we used a valid move check algorithm. The valid move check takes place before the user has entered the y co-ordinate of the piece they wish to move. The method *AreThereAnyValidMoves()* is called (from the Player class) which returns a

Boolean value depending on whether the current player has any available move to play or not.
The algorithm starts by:

- Looping through the textitpositionsArray checking every index for a string that matches the current players normal piece or a king piece.
- Once a piece is found, the algorithm first checks if that piece can capture an opponent's piece.
- If not, then the algorithm checks if the piece can complete a successful move.
- If after all the current players pieces have been checked and neither the capture or move checks return true, then the current player loses the game as there are no available moves to be made.

2.3.2 Force Capture

As stated in the rules of English draughts that if there is a capture available the player must make that move, unless there are multiple captures where the player can choose which capture to make. After the player has entered the y and x co-ordinates of the piece they want to move and the tile they want to move the piece to, the method *ForcedCaptureCheck()* is called (from the Player class).
This algorithm starts in the same way as the valid move check algorithm does by:

- Looping through the *positionsArray* checking every index for a string that matches the current players normal piece or a king piece.
- When a piece is found, the algorithm checks if the piece can capture an opponent's piece.
- If the piece can capture an opponent's piece, the algorithm takes the y and x co-ordinates and concatenates them together storing the value in a list called *listOfForcedMoves*.
- Once the algorithm has looped through all of the array, storing all the potential moves the current player can make.
- The algorithm takes the y and x co-ordinates that the player entered as the values for the piece they want to move, concatenates them together then compares them with the values stored on the list *listOfForcedMoves*.
- If the co-ordinates entered match any of the values stored on the list, then that is an allowed move.
- The algorithm passes the co-ordinates to the method *PlayerMove()* which allows the move to continue.
- If the co-ordinates entered do not match any of the values stored on the list, then the move the player wants to make is not allowed.
- A message informs the player that he must play the capture move available to them.

2.3.3 A.I

2.3.4 Win / Draw Conditions

For a game to end, certain conditions must be met which result in either one of the players winning the match or a draw between the two players. This application has two algorithms which handle potential game ending, checks. The first algorithm checks every turn if a player has won the game.
The algorithm does this by:

- Calling the method *IsThereAWinner()* (from the TheBoard class).
- The method adds the value for each players kings and normal pieces together, then checks if the total equals zero.
- If either player is left with zero playing pieces, then they have lost the game.
- The winning screen is shown indicating which player has won.

The second algorithm has only been implemented in the A.I. vs A.I game mode. It checks every turn if the game is a draw or not.
This algorithm works by:

- Calling the method *IsItADraw()* (from the AI class).
- This method checks if the variable *howManyMovesWithoutACapture* is equal to 20.
- The variable *howManyMovesWithoutACapture* acts as a counter and increases by 1 everytime either player makes a move but not a capture.
- The counter is reset to 0 every time there is a capture by either player.
- This means that if both players go a combined amount of 20 turns without a capture, the game is considered a draw.

3 Enhancements

During the implementation phase of this project, these were the features that were considered but either deemed to be too time intensive to implement and test before the submission date or a simplified version was implemented. These features may be implemented or improved as part of the software's evolution in the future.

3.1 Export/Import full game replay

The ability to export/import game replays, to/from a file format such as comma-separated values (CSV). The application would convert the string that are currently used to store the positions of all the playing pieces each turn, into the standardised notation for recording English draughts games.

3.2 Multiple rules sets available

Allowing the users to select which game rules they would like to play for the current session. Examples of different rule sets that could of been implemented:

3.2.1 International Checkers

The size of the board would have to be increased as for International Checkers the standard board size is 10 x 10 with 20 playing pieces per player. The 'flying kings' rule and the ability for 'normal' pieces to capture opponent pieces by jumping backwards would both require alterations to the movement and capturing logic. As it is a rule that the move which captures the highest number of opponent pieces must always be taken, a form of scoring system would need to be introduced to ensure that only the highest scoring move is taken by the player or AI.

3.2.2 Brazilian / Italian Checkers

Both of these rule sets are only slight modifications from either English draughts or International checkers. Brazilian checkers uses the same rule set as International checkers, but these games are played on a 8 x 8 board whereas Italian checkers while played on the same size of board with a similar rule set to English draughts, there is the additional rule that 'normal' pieces cannot capture king pieces.

3.2.3 Suicide Checkers

Suicide checkers, which is also known as Losing draughts, is a variation of checkers where the objective of the game is to lose all the pieces that belong to yourself before your opponent can lose all their pieces. The winner being the first player left with zero checkers or no available moves to make.

3.3 Improved AI

The current A.I. could be improved by implementing an algorithm which would search for each available move assigning them all a score, based on set criteria. The algorithm would then select and play the move with the highest score. The current A.I. only searches either one or two tiles away diagonally before making an available move. Increasing the search distance to 3 or 4 tiles from the selected piece would survey a larger portion of the board each move and would allow a score based algorithm to adapt sooner to the opponents move.

4 Critical Evaluation

Describe the features that I felt worked well or poorly and why I thought this.

5 Personal Evaluation

Reflecting on what I learnt, the challenges I faced, the methods I used to overcome these challenges and how I feel that I performed.

6 References

Links.....