Checkers Game Report

Craig Cheney
40227803@napier.ac.uk
Edinburgh Napier University - Algorithms
Data Structures (SET09117)

1 Introduction

1.1 Problem Domain

The given problem domain, was to design and build a command line based Checkers game allowing users to a play game against the computer or against another human player. They should be able to undo and redo a move,Then watch a replay of an already played match.

The game Will need to Implement some Form of AI to enable Human vs Computer games. The game will be targeted for use with Windows 7 and Windows 10 based systems within the JKCC computing centre at Edinburgh Napier University The problem could be solved using any language, of which C# was chosen.

1.2 Features

The solution Should allow:

- Player to play a game of checkers.
- Play against human or computer.
- Record game history to allow games too be replayed.
- Undo/Redo feature for player moves.
- Built-in AI.

1.2.1 Additional Features

During development, additional features were added to the game making it more engaging for users.

- Player names .
- Player scores.
- Player scoreboard.
- Player turn counter.

1.3 Features Overview

1.3.1 Play against human or computer

Will allow a player to start a game against a human player or against the computer. This feature will be implemented by creating a main menu that will allow a user to select game type.

1.3.2 Undo/Redo

Will allow a player to undo a move, after moving a checker. The player will then be offered to redo the move they have just undone. Once the choice has been made game control will be passed to the next player.

1.3.3 Watch previous game

Once a game has been completed the player will then be given the chance to watch back the game they just played. By selecting the watch replay option in the main menu. It will automatically play the replay until finished.

1.3.4 AI

AI, will consist of a game object that will be able to make reasoned choices that allows it to play a game of checker against a human.

1.3.5 Player name

Used to identify different players during game, and save their games score to the leaderboard.

1.3.6 Player score

Will reward the player for making good moves e.g. taking other checkers. the points system will reward moves based on a 0 to 4 scale.

1.3.7 Player scoreboard

Players will be able to check how they compare to other players on their system. It will show them Player name and score of the top 10 scores.

1.3.8 Player turn counter

Let's the player know how many moves they have made during a game.

2 Design

The Build of this game will follow the agile development model.

2.1 Planning Stage

This stage began by researching the basic rules of checkers.[?]

Then the process of what would be needed to make a checkers game work. It started with looking at other solution developed looking most of the solutions found were built on C# using the WPF format[?]. However it gave a good start point for logic and some of the objects needed for instance, gameboard, player moves, force moves, how to make the indexes of the array user friendly. How to setup the features of the solution such as Undo/redo move, watch a replay of a game just played, AI player and scoreboard.

2.2 Implementation

Gives an Overview of how features are implemented and what data structures and algorithms they use.

2.2.1 Data Structures Used

- List

  Used in this solution to store strings for a number of functions that collect data, for player force moves, AI available move and also AI move scoring.

- Stack

  Used to store a snapshot of the array for the undo and redo move function.

- Queue.

  Used to store snapshot of the array for the watch replay section of the game.

- 2D Array.

  Used to store the game background that stores all the game objects and allows player to move around on the board.

- Array

  Used to split strings that have been stored when a available move is present for the AI player

- Data Dictionary

  Used to help make the gameboard more user friendly by changing the rows to letters A-F and columns to 1-8.

2.2.2 Gameboard

The gameboard was implemented using a 2D array of strings, the reason, this data structure was chosen is because the other options were using lists or 1D arrays. Which would not be practical, and it would of meant that accessing elements in the list would be difficult as it couldn't be found using a row and column index, using either of these data structures. The way a 2D array is setup it makes accessing elements inside using a row and column the most efficient way of drawing the gameboard and moving checkers. This data structure will also allow the program to access elements of the array in constant time. As the size of the input never changes.[**?**]

2.2.3 Player Moves

Player moves are not stored for the long term and as such are only stored for the duration of the players turn. The data is overwritten on each player turn.

2.2.4 Force Moves

Force moves uses 2 lists and 1 dictionary, the lists are used to store the row and col of a checker that must move and where it needs to move too. It's the converted into a user-friendly format. once the list has been populated the game logic then checks user input to confirm if that the move is valid. the lists are cleared after each player turn, The maximum size of a list is not expected to grow to beyond 10-15 elements. The algorithm will perform linearly as the list grows O(N). Meaning the effect on performance would be negligible.[**?**] The dictionary, is used because it stores a key value pair that enables values coming out of the force move list to be converted to a user-friendly format. it will run in constant time O(1) because of the dictionary's key value pair.

2.2.5 User friendly UI

User friendly UI is part of the way the gameboard is displayed, as the rows are represented by using A to H instead of 0-7. The columns are displayed by converting 0-7 to 1-8. The dictionary is used because of the key value pairs and performance over other key value pair data types. such as the associative array or linked list because the dictionary does not need to expand in anyway.

2.2.6 Undo/redo

Undo function works in the same way as the redo function. They both use a stack to store 2D string arrays which are snapshots of the gameboard at a given time. The stack is used because it's a Last In First Out data type. meaning the player's last move will always be at the top of the stack making it easy to access and it will run in constant time O(1). Other data types could be used like a dequeue. which although would work would of required more code than using the built-in stack, both will achieve O(1).[**?**]

2.2.7 Watch Replay

Watch replay uses a queue to take a snap shot of every move made during a game. it does this by storing a copy of the array before and after each move is made. the player then watches it by pressing enter until the queue no longer has any moves left. The queue makes the replay of the game simple. It's a First in First out data type that allows for the array screen shot's to be displayed in the correct order.[**?**]

2.2.8 AI Player

AI player uses 5 lists to store moves, the list stores moves that have score values from 0 to 4. Letting the AI player prioritise moves that have better out comes than others. For instance, if the move would get the AI player checker taken on the next move it will only make that move it has no other move it can make. when the AI finds a move, it can make. It adds the move to a list in the form of a string. These lists are then converted from user friendly to its format original format. Then it searches for a list with the highest score, and a move it will then randomly pick one of those moves. Using the random class built-in to c# to pick one of the moves.[**?**]

Lists are used not because of performance but because the AI player needs to randomly pick a move from the list. it could have used an array to do this, but a list can expand as required to allow for all possible moves and an array needs to be of a fixed size.

2.2.9 Leaderboard

The leaderboard is a list that stores strings with the player score concatenated on the end. The list is sorted by player score. The list is used because it makes the leaderboard dynamic and allows it to grow. The player can have as many scores per session as they wish. However, once the game is closed, all scores will be returned to default.

3 Enhancements

3.1 Features to Add

Features to Add given more time.

3.1.1 More Versions of Checkers

The solution would be improved by adding different versions of checkers of the world. giving it more playability, however the time involved to add these game types would be at least another 6 weeks. as there as over 16 national and regional variants due to this, the checkers type was kept to straight checkers.

3.1.2 Save/Load Game

The save game would allow a player to stop mid game save the game state, then allow that player to reload that game save next time it's is started.

3.1.3 AI levels

Given more time adding easy, medium and hard levels of AI. would allow the any player to pick up and play the game, and be challenged.

3.2 Improvements to Features

Improvements to implemented features given more time.

3.2.1 Better User Prompts

The user prompts could be improved, by making them more specific more some situations, or by having an error message class that handles all the error. Instead writing the same code over and over again.

3.2.2 Moves combo

Combo moves, would give a player more incentive to find a move that allows them to take more than one checker at a time. by adding a multiplier to a player score each time they take a checker in a single move.

3.2.3 Player Movement

The player movement could be made more generic and more efficient, the code does repeat itself by having movement for each situation for each checker e.g. O, X, kO, kX. making it slower than it could be, but not on a noticeable level.

3.2.4 Game Logic

Game Logic although works. it could be tidied to make it easier to maintain.This would also make it easier to add new features in future. The game logic also repeats itself and had to changed in-order for the AI to work with it.

3.2.5 Neural Network AI

Having the AI use a neural network would be a great improvement to the AI as it would enable the game to evolve around the player. giving the player an ever-changing opponent to play against. the AI could be handicapped if an easy, medium and hard mode was added. Allowing for less experienced players.

3.2.6 GUI

If given more time, this game could be rewritten in WPF format, using the mouse and click events, to move checkers. Instead of text input, timers to be displayed and used to help with player scoring.

4 Critical Evaluation

4.1 Worked Well

4.1.1 AI Player

Although the AI player is simple, it works well, because in a simple form it has the ability to pick the best move available to it. It does this by adding moves which are scored by how important that move would be, then adds it to a list with other moves of the same score. It then looks through the lists with the highest score until it finds a list containing moves. Once it finds a list with moves, it uses a random number generator to pick one of the moves. Although, the AI player does not look more than one move ahead it still plays a good game.

4.1.2 Undo/Redo Move

The Undo/Redo function uses the player logic to add an array to a stack, because it's a stack the last game Background array will always be on the top of the stack. making the code to access the last move short and quick. To show the player the move has been undone the gameboard is redrawn. if the player then changes their mind and want to redo the move undone another stack that loads the game Background array from after the player selected their first move, and redraws the gameboard.

4.1.3 Watch Replay

Watch replay is similar in its execution to the undo/redo move, in the way, that it saves a snapshot of the array every time a player makes a move. However, the watch replay uses a queue to store the arrays. the function then starts to dequeue arrays from the queue redrawing the board each time it does so. until the queue is empty, it then returns to the main menu. it works well because it uses a queue to save array of the gameboard, saving on extra code for finding the checker moved, where it moved to, what checker was taken and so on.

The whole watch replay class consists of around 55 lines of code. Watch replay has been further enhanced by removing the need to press enter for every turn. It now uses a 2 second timer to run through the queue.

4.2 Could Be Improved

4.2.1 Player Movement

The way that player movement was implemented meant that, code had to written for each player and this could have been cut down quite a lot. If the player movement was based on objects instead of basing it on finding a certain string and what's around it in the array. A further drawback of this algorithm, is maintaining it and adding further functionally would be quite difficult because how tightly coupled the player movement and player logic are.

4.2.2 Game Logic

The code became quite long had and to be repeated, such as when asking for the player to input their move. The code asking for user input could have been made into it's into its own function and called from there instead. There are also some errors that will be triggered for 2 different events meaning a targeted message can't be pushed to the user without a full reworking of the game logic. Game logic also must run more than once to hand over player control after they take a checker on the board.

4.2.3 User Input Validation

Validation of the user input has also been hampered by the setup of game logic, the algorithm must wait for both row and col to be entered before it can tell if the move is valid. So, when a player inputs an incorrect value for row they must wait until they have entered a value into col. Before an error message is displayed and the player is given the chance to re-enter the values for their move. this could be improved by adding a function that checks the input of pick row, pick col, move row and move column as they entered. so, if a user enters an invalid input. An error message will be thrown straight away. Stopping the algorithm from performing actions that are not needed. saving memory and time.

4.2.4 Leaderboard

The leaderboard does work, however it only saves the data while the game is running, once the game has been stopped all data is lost, this could have been better implemented if some form of text or XML file was used to store player names and their scores. the sort used on the list to set scores to descending order, is a quick sort which works well with this function, but because of the way the information is stored e.g. string it requires an additional function to compare the values of the string, and sort them into numeric order.

5 Personal Evaluation

5.1 What was Learnt

5.1.1 Preplanning

During the design phase, a lack of preplanning hindered develop of 2 classes specifically. Namely game logic and player movement, although the algorithm work's quite well. it's taken a lot of development to get it to that stage. Time, which could have been spent adding more features to the solution.

If more preplanning had taken place during the design phase. The solution could have used OOP principles. For example, polymorphism should have been used to treat, all players as one instead of having code to handle each object, polymorphism could have enabled the code to treat each class that inherits from the player base class as the same object until runtime. reducing the complexity of the code not just in run time, but also would make it easier to read and maintenance would be simplified.

5.1.2 Data structures

When making algorithms data structures used can have a real effect, with how Algorithms work, and the complexity of the code need to perform the task. For instance, using an array list over a list in c#, would require the further overhead of casting, more chance of runtime errors. As the array list is of a fixed size, and could throw errors when adding more values than accounted for.

In the first instance undo/redo had been implemented using a series of if statements that held the instructions for each type of move and each type of checker. This solution worked in some instances, however multiple moves with checkers would not. As such the undo method was modified to use a stack to store the snapshot of the array before and after each move. if the player then wants to undo the move the array is loaded into to the gameboard.

5.1.3 Algorithm Impact

While making algorithms, it was discovered that the impact of using certain structures while coding. while Seemly the same on the surface they do the same thing, however in practice work completely differently and have a great effect on the complexity of an algorithm.

For instance, the way the force move has been implemented could have been a lot more efficient if the for each loop had been a for loop. This is because, a for loop could be stopped once it had found the last checker of the player on the board, but instead uses a foreach loop that looks through the whole 2D array before stopping. taking the algorithm from O(N*N) to O(N).[**?**]

5.1.4 Using GIT

During this project, Git Hub was used for the first time to manage changes to the solution. It proved to be a great asset to the project. Helping to keep track of changes to the source code, allowing any changes that had been made to be reverted. GIT also highlights the lines of code that have been changed as of the last push to the repo making it far easier to spot any changes that need to be made to fix the bug, if the changes caused any bugs in the solution.

5.2 Challenges Faced

5.2.1 AI

The game AI needed to be able to play a game against a human and be able to make reasoned decisions in the move's it makes. The first challenge was to find a way to allow the AI player to find move on the board. Once it found the moves it would need to be able to pick one of those moves. while making sure it wasn't just moving a checker for the sake of moving a checker. the hardest challenge was finding a way to get the AI to pick a move, that gives it the best chance of winning the game.

In the end the solution to this problem was find all move and attach scores certain moves found with the best having the highest score, adding them to a corresponding list. It then looks for the list with the highest score and the randomly picks a move using the random class built- in with c#. although the AI player is simple it plays quite well. Probably being the solutions best feature. this could have been improved by also adding scoring for checker positioning and the ability to look more than move ahead

5.2.2 Game Logic

Game logic Proved to be more complex than first thought. the original implementation of the class only considered the human vs human game mode. this meant having to essentially having to copy the code for human vs human game, and changing small parts of it to accommodate the AI player. The computer vs computer mode was drop. as again it would have meant having to do the same as with Human vs Human code. If the project had more time.

A full reworking of the logic would of went a long way to solving this problem by treating objects with the same base class in a similar manner. thus, allowing the code for the logic to not have to be repeated. for each game (Player) type.

5.2.3 Watch Replay

Watch replay would require the player to be able to watch a match after it has been played, saving each move e.g. Row and Col for each move.

After researching the problem at looking at other people's solutions. It was changed to take a snapshot of the 2D array that stores the game objects. Add a clone of that array to a Queue, and when a player completes a game they are taken back to the main menu with the option to watch the last played match. This is where the 2D array taken after each move is dequeued from the queue from the first move to the end, without user input. The foreach loop has a 2 second delay that allows it to redraw each move after 2 seconds.

5.2.4 Undo/Redo

The undo/ redo moves in its first iteration was much like watch replay in the way it was set out. The row and column were taken in from the game logic and depending on what is present in the array that position. It would pick from a set of instructions, which told it were to redraw the checker. This meant it wouldn't work both directions on the board.

To fix this, the function was changed to work like watch replay by taking a snapshot of the 2D array before and after a player turn. after their turn they are then asked if they want to undo the move, if they decide they want to undo the move, they are then asked if they wish to redo the move they have just undone. It does this by using a stack to save the arrays after each turn, uses the values popped from the stack to redraw the gameboard.

5.2.5 Game Timer

The game timer is a feature that was never implemented, that main problem was not getting the timer to run but to display to the user. Mainly due to the way that the gameboard was drawn in to the console, the background array used to store game objects, would stop the gameboard from being fully drawn to the console stopping half way through. Until the timer had finished. the problem wasn't discovered until late on in the built. Forcing it to be drop and a reworking of the player scoring. The scoring should have been tied in with the time taken to make a move.

5.3 Overall Performance

5.3.1 Data structures Chosen

The data structures used in this solution for the most part match the requirements of the functions they implement; the main criticism would be that 2 of the class could be more performant. if when implementing more consideration had been taken. when looking for different moves and game logic could be less complex.

5.3.2 implemented Features

Almost all features of the solution asked for have been implemented, as well as other features added to try and add to the depth of the game. Such as player names, player scores, leaderboard, Move counter, a rules section. The only feature not implemented is the computer vs computer game mode.

6 References