Edinburgh Napier UNIVERSITY

Coursework Report

Sam Jones
40227806@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

1 Introduction

In this document the developed system 'CheckersCLI' will be discussed. This system was started on Oct 16, 2017 and finally finished on Nov 12, 2017. The functional requirements as listed by the coursework specification document[1] are as quoted:

"Your task is to implement a checkers game in a language of your choice. Using your knowledge both from taught interactions and self-directed learning you should identify appropriate data structures to represent at least the game board, players, positions and anything else you identify that is necessary so that a game can be played between two people, a person and a computer, or between two computers. Your game should record play history, i.e. the sequence of moves that the players make during a game, so that each game that is played can be recorded and replayed. Your game should support undo and redo, again selecting the most appropriate data structures to enable these functionalities. Finally, your game should implement an algorithm that enables the computer to choose which moves to make during their own turn, i.e. a simple AI player."

1.1 Functional Requirements

From this the functional requirements can be gathered, these are:

- A working checkers board game following traditional rules.
- The ability to play against a friend or an AI.
- The ability to watch two AI play against each other.
- Have all games display a replay of the game after they are complete.
- The option to undo actions.
- The option to redo actions after undoing them.

1.2 Non-Functional Requirements

It is also stated that the program must run on from the command line using a text based interface. Anything further than this is not a requirement and therefore an extension of the actual project. Due to this 'CheckersCLI' is strictly a command line application.

2 Design

Throughout this section the design and choices behind these designs will be discussed. There will be a particular focus on the data structures in this section due to the nature of the course. The functional requirements will be used as a map for what design will discussed. Any addition design that needs to be referenced will proceed. Throughout this section 'the big O notation'[2] may be referenced. This is used to describe the performance or complexity of a specific algorithm.

All 'Code Examples' are invented for the purpose of this document and are for the purpose of representing a very basic version of the code implemented in the 'CheckersCLI' project.

2.1 Checkers Board Game

The game board is stored on a 2D array of 8 x 8 integers, this allows all tiles on the board to be assigned a specific value. The array is then easily updated. The game itself runs on a simple while loop. This is extremely basic but also extremely efficient. The game will continue until one condition is met, allowing for a turn based system to be easily implemented with basic if statements. This can be described as $O(1)$ meaning that the performance of the algorithm stays constant regardless of any other factors. Because of the simplicity and level of performance this algorithm is a great choice to meet the requirement.

Code example:

```
1   while(!winner)
2   {
3       if(playerTurn == 1)
4       {
5           ...
6       }
7       else
8       {
9           ...
10      }
11
12      if(piecesGone)
13      {
14          winner = true;
15      }
16  }
17
```

2.2 AI

In order for the user to play against an AI one must be implemented. Due to time constraints a simple linear search algorithm was implemented. This algorithm runs through the board checking for all pieces that the AI can move, this requires two for loops indented within each other.

The efficiency of this algorithm is $O(n^2)$ where n is the number of items in the array. This is normally a very inefficient method when combined with large input numbers, yet it does provide a low level of complexity and time to implement. Since the conditions under which it is used in this project the inefficacy is not a problem.

The use of for loops is therefore well justified in this scenario as it provides a working solution in a very short amount of time. Due to the simplicity of linear search selecting the first piece that can be moved there is very little variance in the AI movement. This is most visible when watching the AI match itself, the same game is played every time based on which AI goes first resulting in only two possible move sets being simulated.

Despite these drawbacks the AI still serves its purpose of being able to play against a human player and holds the ability to win the game meeting the functional requirement.

Code example:

```
    for(int i = 0; i < 8; i++)
    {
        for(int j = 0; j < 8; j++)
        {
            if(location[i, j] == player1)
            {
                ...
            }
            if(location[i, j] == player2)
            {
                ...
            }

            ...
        }
    }

```

2.3 Replay Games

Another function of the checkers board game is the function to be able to replay games after they are complete. This can be done simply by using a queue to hold the data allowing the program so dequeue all items from the queue one by one when the game is complete.

A queue of type 2D array of integers was implemented, whenever a move was taken by one of the players or the AI an instance of the array storing the board was queued onto the back of the queue. When the game is finished a foreach loop would then dequeue the items one by one from the queue and then proceed to display them on screen for the user to see. The efficiency of a queue can be evaluated in two ways; insertion and deletion.

Both inserting values onto the end of a queue and removing values from the front can be described as $O(1)$ using O notation. Removal and insertion is extremely fast when using queues making it the perfect data structure to hold the replay of the game for this project. Another plus side to using a queue in 'CSharp' is the lack of memory constraint at run time, the queue can hold a dynamic number of objects depending on physical memory held on the computer.

These reasons make this data structure a solid choice to implement the replay function.

Code example:

```
    //Initiation
    Queue<int[,]> replay = new Queue<int[,]>;

    //Add values to queue
    replay.Enqueue(board.Clone as int[,]);

    //Display values after game finish
    foreach(int[,] i in replay)
    {
        draw(i);
        System.Threading.Thread.Sleep(2000);
        Console.Clear();
    }

```

2.4 Undo and Redo

Being able to both undo and redo in a virtual board game are core functions as mistakes can easily be made especially in a text based interface, therefore the implementation of these data structures was a high priority when creating 'CheckersCLI'. In order to undo and redo actions those actions must first be stored on a stack so that they can be reverted allowing the user to carry on with their game from where they left off. And then once an undo has been carried out the redo stack must be populated if the user wishes to go back to their furthest point of progress.

In order to achieve this objective a stack of type 2D array of integers was used, this allows the program to simply add an instance of the board every time the player takes a move much like the replay function; Once a user typed the key phrase undo the board would return to the instance where it was one turn ago or, in the case of playing against an AI, two turns ago. In the case of redo every time the user types undo the current instance of the board would be added to the redo stack, once the user then typed redo the top value from the stack would be popped and pushed onto the top of the undo stack. It is important to flush the redo stack every time the user takes a new move in order to prevent redoing to an old move.

Both pushing values onto the top of a stack and pushing values from the top can be described as $O(1)$ using O notation. Removal and insertion is extremely fast when using stacks making it the perfect data structure for the undo and redo function of the game for this project. Another plus side to using a queue in 'CSharp' is the lack of memory constraint at run time, the stack can hold a dynamic number of objects depending on physical memory held on the computer.

Code example:

```
    //Initiation
    Stack<int[,]> Undo = new Stack<int[,]>;
    Stack<int[,]> Redo = new Stack<int[,]>;

    //Add values to stack
    Undo.Push(board.Clone as int[,]);

    //Undoing values
    Redo.Push(Undo.Pop);

    //Redoing values
    Undo.Push(Redo.Pop);

```

2.5 Enemy Pieces That Can Be Taken

In order to follow the official checkers game rules it is important that a function ensuring that if an enemy piece can be taken the player must take it. This is was added to 'CheckersCLI' in the form of a linear search algorithm. The AI shares the same method as the user for checking to see if there are any possible pieces that they can take on their turn, this is simply done by running two for loops indented within each other that search through the entire board array to find any pieces that meet the criteria provided.

This is not a very efficient algorithm but it is the most basic one which is all that is required for this scenario as all pieces on the board must be checked. The efficiency of the algorithm when used in a 2D array can be describes as $O(n^2)$ when using big O notation, this is because as the size of the array to be searched increases the higher the search time becomes. This algorithm is fine for this situation as the array will only be a size of 8 x 8 and will never change making the search instantaneous to the human eye with no load times.

Code example:

```
for(int i = 0; i < 8; i++)
{
    for(int j = 0; j < 8; j++)
    {
        if(location[i, j] == player1 & location[i + 1, j + 1] ↩
    == player2 & location[i + 2, j + 2] == 0)
        {
            location[i, j] == 0;
            location[i + 1, j + 1] == 0;
            location[i + 2, j + 2] == player1;
        }
        if(location[i, j] == player2 & location[i + 1, j + 1] ↩
    == player1 & location[i + 2, j + 2] == 0)
        {
            location[i, j] == 0;
            location[i + 1, j + 1] == 0;
            location[i + 2, j + 2] == player2;
        }

        ...
    }
}

```

2.6 Win Condition and King Creation

Much like the AI and the function that searches for pieces that can be taken the win condition and king creation algorithms used in 'CheckersCLI' are also linear search algorithms; Running at an efficiency of $O(n^2)$ when used in a 2D array, searching within a 2D array is very inefficient but much like in the prior problems n is extremely low, only being an 8 x 8 array, so again this very simple algorithm does the job perfectly. The king creation algorithm is sightly different in the fact that only 16 of the values in the array are checked, the top and the bottom most row of the board is checked for checkers pieces that did not start on that side of the board, if one is there then it is turned into a king and granted greater movement permissions.

The win condition, being very similar to the other algorithms, searches the entire board to see if there are at least one of each piece remaining, if not then the other player wins.

Code example(KingCreation):

```
int c = 0;
for(int i = 0; i < 8; i++)
{
    for(int j = 0; j < 8; j++)
    {
        c++;
        if(c < 8)
        {
            if(location[i, j] == player2)
            {
                //Turn into king
                ...
            }
        }
        if(c > 55)
        {
            if(location[i, j] == player1)
            {
                //Turn into king
                ...
            }
        }

        ...
    }
}

```

3 Enhancements

In this section features that could be improved upon given more time will be discussed. Due to the time constraints on this project there are many things that could be improved or added to make 'CheckersCLI' that much higher quality.

3.1 Object Oriented Language

The first and biggest thing that could be improved upon would be making the code a lot more object orientated. Since 'CSharp' was used for this program it is natural that the structure of the program would be of an object oriented manner but this was not greatly achieved when writing 'CheckersCLI' due to a lack of time and experience.

Without the right experience to instinctively know how to write the code needed in an object oriented way it would have taken a lot longer than writing in a mainly functional style, the time required to gain the knowledge to allow the writing of a purely object oriented approach would take more time than available to finish before the deadline. Writing the program in a more object oriented way would have most likely increased efficiency and performance, reduced size and amount of repeated code and added a higher level of maintainability.

3.2 AI

The AI in 'CheckersCLI' is extremely basic but does what it needs to do; Given more time the AI could be much more complex and engaging to play against allowing for different levels of difficulty with thousands of possible AI vs AI outcomes rather than the two present in the current game. Currently the AI runs on a linear search algorithm meaning so long as the AI cannot take a piece that turn it will always move the highest piece on the board that it can, this creates a very repetitive and predictable play style.

Given time a function could be created which sorts and stores all possible moves that the AI could take this turn and then the following turn after and then the AI could make a decision on which piece to move knowing what could happen next turn; Much like most humans who think many turns ahead when playing a board game the AI would critically evaluate what move would best benefit its objective based on play styles that could be set before the game. Defensive, offensive or hybrid play styles could be chosen to add variety to playing against the AI.

3.3 Undo and Redo Functions

The undo and redo functions are very basic functions that work and perform well under the given circumstances yet they are not entirely efficient. The undo and redo functions have no cap on the amount of moves that they can store meaning that thousands of moves could be stored. This is obviously not going to be the case in a normal game under normal conditions but it does not mean that it makes the code fully reliable.

Depending on the amount of main memory someone has they could potentially create a stack overflow exception if they intentionally took thousands of moves.

Now this seems like something highly unlikely to ever happen but it could easily be stopped by adding a function that copies all but the least recent move over to another stack and makes that the new undo stack. With the right research a number could be decided for what point the stack should start doing this to create optimal performance in the game rather than having performance degrade the longer the game ran.

3.4 User Interface

In the current year text based interfaces are not common to the general public nor are they very user friendly. 'CheckersCLI' is built in a console environment and designed to run on a text based interface due to this being one of the project requirements. Within text based environments it is possible to make very ascetically pleasing game screens but this requires a large chunk of time to design and implement. In 'CheckersCLI' windows character map was used in order to make drawing the board a little more readable and easier to understand.

Given time more text based visuals could be added that make the game more understandable or add a personal feel to the environment. Things such as randomised backgrounds or loading screens with nice ASCII visuals could be added giving the game a more professional look. Something like this would have made the project more fun and added a large sense or pride to the project upon completion.

Another option would be to add the extension of a graphical user interface; GUI's create a friendly easy to interact with environment for the users and is generally what the larger body of the public is used to dealing with today. The addition of clickable objects would make moving the pieces on the board a large deal easier and more enjoyable. A GUI would also add the option for images and nice graphics to be added on menu screens, in the background or loading screens.

3.5 Replaying Games

The function of replaying games was a requirement for this project and was therefore implemented to it's required level of functionality; Replaying games after the game had ended. This role is fulfilled with a stack, this is fast efficient and there is not another algorithm or data structure that could be used to provide any real benefit in the current instance.

In contrast there are improvements that could be made; Although not part of the requirements the replay function could be made to store all the past games in a file and then provide the player with an option to view any of them from the main menu.

This function would allow the player to watch past games and track their skill and growth as a checkers player over time. This could be coupled with statistics recordings displaying how many moves were taken, which player won and other things to provide a professional feel.

4 Critical Evaluation

Throughout the critical evaluation section all features and whether they worked well or poorly will be discussed. Most of the features worked well for this project because of the small scope but they would not work well on a larger scale and that will be discussed.

4.1 Checkers Board Game

As discussed before the board for the game is stored on a 8 x 8 2D array of integers, this is effective in this situation as the array is small so not many values can be stored; In terms of O notation the performance for searching through a 2D array being $O(n^2)$ is 'horrible' [3] which is obviously not a positive. Critically this means the data set is not effective or efficient but in this situation it does the job and nothing else would do the job with a noticeable increase in efficiency due to the low number of inputs.

In conclusion the use of a 2D array is effective in this instance but in large instances where more data has to manipulated it should be avoided.

4.2 AI

The AI is implemented as a way for the program to take control of the pieces on the board and play against the human; An effective AI would play with the intention to win, this is not the case of the developed AI in 'CheckersCLI' where it only follows a specific pattern. This does not however mean that the AI cannot win, it does provide a relatively close game when paired with someone of low skill, of course if the AI was to play against someone who knew the algorithm and how to play against it then they could predict every move with 100% accuracy.

It was not a full requirement to create a intelligent AI or even one that knew how to win so the AI is effective at filling the requirement. However it could be done much better without loss of performance and without needing a large amount of time, therefore the AI is poorly implemented and is only there to fill a role rather than bring any replay value or interest to

the game after playing against the AI a small number of times.

4.3 Replay Games

Replaying games allows the user to evaluate how the last game went and view the moves that were made, this a very basic function to use and a very basic function to implement. Therefore there isn't many places to go wrong or improve on, a basic stack provides the means to store all moves taken and then display them at the end of the game.

The replay stack provides all that is required of a replay function and therefor is effective in this instance and many other instances, there are data structures that could replace it but they provide no reasonable advantage to achieve the goal of replaying games.

4.4 Undo and Redo

Undoing and redoing are core functions of a computerised board game, if the user messes up moving a piece to where they wanted to, or they regret a choice then undo can be very useful; Of course redo just tags along as undoes partner whenever it is needed. In 'CheckersCLI' the undo and redo functions are created using a stack, this is a basic data structure but provides all that is needed for an undo and redo function. The user is given the option to type 'undo' or 'redo' at the start of their turn, this is both easy to do and if they miss-type it nothing goes wrong. The algorithm accepts both lowercase and capitalised versions of the word also.

The stacks used for replay and undo are implemented in the most simple and basic way to achieve their functions, this method is fast reliable and there is not another data structure that could be implemented that would make a notable difference in any way. In conclusion the undo and redo functions are very effective at their job in 'CheckersCLI'.

4.5 Enemy Pieces That Can Be Taken

A method is provided for ensuring that whenever an enemy piece can be taken it is taken by the user, as such is a rule of checkers. To do this a linear search algorithm is used, this algorithm searches through the entire 2D array that is the board to find any pieces that fill the criteria to take an enemy piece; Although searching through an array is of efficiency $O(n^2)$, horrible, this is not the fault of the linear search algorithm. The linear search algorithm has an efficiency of $O(n)$, this is a 'fair' speed but this does not matter as faster searching algorithms such as trees are not applicable in this scenario.

The linear search algorithm used to find moves in 'CheckersCLI' is fast and does the job effectively, it could not easily be replaced with a superior algorithm to achieve the required goal.

4.6 Making Kings and Winning

Both the algorithm to find and make kings and the algorithm to search the board for a winner use linear search much like in the prior subsection. These algorithms are a lot smaller and have a lot less conditions than the previous one but still run at the same efficiency. In relation to 'Enemy Pieces That Can Be Taken' this algorithm also cannot be replaced by a faster one in this scenario.

The user is notified as soon as they win, there are no bugs or issues creating problems, all they must to is remove all of the other teams pieces.

The king function changes the users piece to a king as soon as they reach the enemy's back row as is the rules of checkers.

In conclusion these algorithms are implemented on the fastest and most effective way possible, they do not require the user to do anything extra and they to not take any noticeable time to complete.

5 Personal Evaluation

The personal evaluation section allows me to reflect on what I learned, challenges I faced, how I overcame those challenges and how I performed. This section will be first person from the point of the developer and writer of this report.

5.1 Motivation

All people are affected by motivation throughout their lives in one way or another, people are driven by different things and that is their motive; Not all motives are good but all motives provide a person reason to do something, most people will not do things without a personal reason and it can be seen when large amounts of people work jobs they hate for no motive other than making enough money to live.

When working on a project that is not overseen motivation is a huge guiding factor in performance and time management, when making 'CheckersCLI' my obvious motivation was to complete this project to pass the unit; Goals without immediate self gain or clear rewards can be weak driving forces, this was a problem that I faced daily. Without a strong motivation I found it hard to concentrate on work and not procrastinate, this affected the speed of work stopping me from reaching daily goals pushing back weekly goals and so forth.

The fear of an ever approaching deadline is what provided me with enough motivation to get everything done, as the deadline drew closer my motivation grew greater, fear of failure and fear of disappointment are the two factors that provided me with the most motivation and helped me complete this project before the deadline.

5.2 Fatigue

Sleep impacts your brain greatly, your brain impacts your concentration, your cognitive functions and overall your performance. Lack of sleep is a big factor in getting work done on time and to a good standard; Not only does lack of sleep affect this but so does boredom, spending too much time on something that does not provide you with enjoyment can leave you feeling tired and fatigued.

My work rate and quality was often impacted by lack of sleep or working too long without any breaks, I find it easiest to motivate myself to get stuff done if I do it all in one go, this meant I was doing the same task for eight hours straight some

times and by the end my performance would have greatly dropped. This drop in performance would arrive even faster if I had not slept enough the prior night.

To overcome this problem I made some time schedules to go by so I could start work early enough to finish in the early evening to allow for leisure time. I also made sure to take regular breaks to rest my mind and stop the effects of boredom occurring.

5.3 What I Learned

When working in this project I learned a great deal about object oriented language and how things can be implemented in an object centred way, this did not translate into my code as I would learn these things after creation or I would not understand them enough to safely implement them into the program.

I learned that time management is key to getting things done, projects should be started early not on time to allow for overflow. Objectives should be set along with short and long term goals to provide a sense of progression and accomplishment. Reward should be given for goals to provide motivation ensuring you meet them.

I learned a great deal about algorithms and data structures and how they can be implemented in different ways with the vast uses that they have. Research helped me find less common data structures that provide useful advantages for specific implementations.

5.4 What I Done Well

I think I done a good job at completing the work before the deadline and submitting it all in time. I made sure to include all of the required functions. I ensured to test all of the functions of the program and fix any bugs or errors that occurred. I created a robust program that functions well and does not crash. I also used effective and relevant data structures.

5.5 What I Could Improve on

Improvements could be made in the quality of some of the code written, repetition in code could be removed, a more object oriented approach could be taken. Time management could be improved on and having a schedule from day one could help that. More complex data structures and algorithms could be used to improve performance, functions or show off knowledge of the subject. A more complex AI could have been developed and having more passion for the project could have increased my work load and my quality of work.

5.6 Overall Performance

Looking at the project as a whole, all requirements were met; There was not much experimenting or many additional features added to show passion for the subject. I put in enough effort to complete the project on time and to the required standard. Appropriate data structures were used although more complex or efficient ones could have been implemented in some situations. The game plays well and is easy to pick up.

From all this and the prior sections I would rate my overall performance at 7/10; I made sure to complete all that was required on time and to a good standard, I did not push myself to do extra work or add features that may have improved game play or attraction.

In conclusion I think this project was an overall positive success and I performed to an appropriate standard.

References

[1] S. Wells, "Coursework assignment."

[2] Wikipedia, "Big o notation."

[3] M. People, "Big o cheat sheet."