Edinburgh Napier UNIVERSITY

Checkers

Richard Donaldson
40329070@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Keywords – java,checkers,algorithms,data structures

1 Introduction

I have been asked to implement a checkers game that demonstrates my understanding of both the theory and practice of Algorithms and Data Structures. To implement the game, I will be writing my code using Java, using a console based design to implement all the features of the game, and then if time permits, use an integrated UI Design to Java called Swing. It must use appropriate data structures to represent the key features of the game, such as the board to allow the game to be played between two players or a person and a computer.

The game should also record the sequence of moves that are played during the game, and the player should also be able to undo and redo moves. Finally, the game should also implement an algorithm to create a simple AI Player.

This report will detail the features of the game, as well as explain the algorithms and data structures used to create the game. It will also explain the successes of the project and any problems that arised during development.

2 Design

2.1 Class Diagram

To begin coding the game, the structure and design of the code needs to be thought about. A Model-View-Controller (MVC) Design has been used as a structure for the code[3]. This is a methodology that is used in Object-Oriented programming as it groups all the code together in its relevant packages, and allows the object code to be reused, which aids in faster and more efficient development. Each section only contains the code relevant to the section. e.g. the Model class does not contain any information about the User Interface. The design is broken down into its three sections:

- Model: Represents the logical structure of the data and the high-level classes within the application.
- View: Represents the User interface of the application
- Controller: represents the classes and acts as a go between for the model and view classes.

2.2 Data Structures Used

2.2.1 Board

To represent the board, a two-dimensional array was used, as the two dimensions would represent the Axis of the board. Where the first element represents the Y-Axis, and the second element represents the X-Axis. The elements within the array represent the pieces on the board where:

1. Black
2. White
3. Black King
4. White King

The board is initialized in a **Board** class, then an object is created within the **Game** class.

2.2.2 Game Pieces

To represent the pieces on the board, two ArrayLists were used to represent the two colours used in the game. An ArrayList is a re-sizable-array implementation of the List interface, which can store all elements, including null, which makes it ideal for storing objects.

```
public ArrayList<Checker> blackPieces = new ArrayList<↩
    Checker>();
public ArrayList<Checker> whitePieces = new ArrayList<↩
    Checker>();
```

The most important factor when deciding which data structure to use for the checker pieces is being able to re-size the array as pieces can be removed from the game.

2.2.3 Moves

To store the moves played within the game, two Linked Lists were used, one to store all the moves played and the other to act as a temporary storage to be used within the undo and redo method. A Linked List is a doubly-linked list implementation of the List and Deque interfaces, which allows for moves to be inserted and removed in constant time, which is ideal for the undo/redo feature as its purpose is to insert/remove moves.

```
LinkedList<Move> moves = new LinkedList<Move>();
LinkedList<Move> copy = new LinkedList<Move>();
```

When the undo/redo method is being called, all the moves are stored in the *moves* List. If the user wants to undo a move, it is then copied into the *copy* list and then removed from the *moves* list. If the user then decides to redo a move, it is copied from *copy* back into *moves*

2.3 Main Algorithms used

2.3.1 Validate Move

One of the most important methods in the application is the ***movePiece()*** method. This method takes in the users input for the piece they would like to move and where they wish to move it to. It will validate the input so that it is the correct format. The method will take the users input, using substrings to break the input down into its X and Y co-ordinates. It will check to see that the X-input is a letter, will then pass that X-input into the *convertXPosition()* method, which will convert that letter into it's respective number, as shown in the table below.

Letter	Number
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7

Table 1: Convert X-Position

The Y-Input will then be validated to check that it is a number and that the number is in range. The same validation will be used to check the input for the move position. Using the validated Origin input, the co-ordinates will be passed into the *findChecker()* method within the **Model** class. This will perform a basic linear search to check if the input matches with a checker that is on the board. It will then return that checker so that it can be updated when the piece has been moved. The logic that this method follows for deciding if a move is valid is shown in the Flowchart provided in the appendices.

2.3.2 Undo/Redo

One of the requirements for this project is to have an Undo/Redo function that allows the User to Undo moves back to a point of their choosing. A user may undo a move from the last move played to the start of the game. A user may also redo a move if they begin to undo a move, but decide to cancel that.

For example, if 10 moves have been played, and the user decides to move back to the 5th move, they may undo moves back until that 5th move. However, if they decide that they would rather go back to the 7th move, they may redo the moves to the 7th move. Essentially the user is moving through the list of moves that has been played until they confirm which move they wish to move to. Once the undo has been confirmed, the user cannot then change their mind and redo a move.

For example; the user decides to move back to the 5th move, they then confirm this. The game then continues from the 5th move, which deletes all moves after the 5th move, so the user cannot change their mind and go forward again, to the 7th move. They will however, still be able to move backwards, from the 5th move, to the 3rd move.

2.3.3 AI

The AI used in this project is the beginnings of a Minimax algorithm, but due to time constraints it is a simpler version. The AI will work out all the possible moves that it can make at the time, and then will choose one of those moves at random. It does not provide the Player with a challenge, but for the purposes of this project, it demonstrates a working algorithm that acts as a computer player.

The concept of Minimax is an algorithm that assigns a score to each move based on its condition. It then uses that score to determine the best, more most valuable move. Like my AI, it goes through all the possible moves that can be made, and uses some logic to assign values, such as if the piece can become a King, or if that piece is going to come into danger. The algorithm would then pick the best move it can generate and play that move.[1]

3 Enhancements

3.1 Replay Feature

A feature that would have liked to have implemented would have been saving the Moves to a CSV file, loading them into the game and then stepping through each move to provide the user some feedback of their performance, or to use as a learning experience to develop strategies of playing. This would be simple to implement as most of the code would allow for this feature to be implemented, but due to time constraints this was a feature that I identified as not critical.

3.2 Complex AI

Another feature that I would have liked to have implemented would have been the more complex AI. I have looked at an example of a Minimax algorithm in Java[2] and attempted to try and implement this AI in the scope of my game, but I started to struggle understanding the code and the errors I was creating during the process, so I decided to take a step back and implement an easier version so the User can at least play against something.

4 Critical Evaluation

Overall I feel that this project was a success. I have been able to implement all the critical features so that two players can play against each other, or a player can play against an AI. If there were any changes I would have made it would have been to make the console output easier to read and follow what is happening on the board. It can be difficult at times, but it is certainly playable.

5 Personal Evaluation

The one issue I struggled the most with is definitely time constraints. Being a direct entrant from college, the way Edinburgh University do things is very different to what I was used to. I don't handle stress very well, so when it starts to

pile up I tend to just burn out more easily, but this semester has definitely made me realize the effort that I need to put in, so I can at least work on better techniques for handling the stress and dealing with time pressures. I am happy that I have managed to get all the work done, and in time for the deadline.

6 Code Listing

6.1 Board

Listing 1: 2.2.1 Board

```
int[][] board = new int[][]{
        {0, 1, 0, 1, 0, 1, 0, 1 },
        {1, 0, 1, 0, 1, 0, 1, 0 },
        {0, 1, 0, 1, 0, 1, 0, 1 },
        {0, 0, 0, 0, 0, 0, 0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0 },
        {2, 0, 2, 0, 2, 0, 2, 0 },
        {0, 2, 0, 2, 0, 2, 0, 2 },
        {2, 0, 2, 0, 2, 0, 2, 0 }
    };
```

6.2 Undo/Redo method

Listing 2: 2.2.4 Undo/Redo

```
    public void moveThroughList() {
        int choice = 0;
        int i = 0;
        int j = 0;

        System.out.println("1. " + "\t" + "undo ( Move backwards through list)");
        System.out.println("2. " + "\t" + "redo ( Move forwards through list");
        System.out.println("3. " + "\t" + "Exit Method");

        @SuppressWarnings("resource")
        Scanner input = new Scanner(System.in);
        Move test;



    do {
        try {
        choice = input.nextInt();
        //Undo Move
        if (choice == 1) {

            i = (model.moves.size() - 1);
        //  System.out.println("i is: " + i);
            if(i == 0) {
                System.out.println("There are no more moves to undo");
            }

            test = model.moves.get(i);
            undoBoard(test);
            model.copy.add(test);
            model.moves.remove(test);
             System.out.println("Move has been removed ");
        //  model.printList();

            //Redo Move
        } else if (choice == 2) {

            j = (model.copy.size() - 1);
            //System.out.println("J is: " + j );
            if(j == -1) {
                System.out.println("There are no moves to copy");

            } else {

            Move copy = model.copy.get(j);
            model.moves.add(copy);
            model.copy.remove(copy);
            i = (model.moves.size() - 1);
            //System.out.println("i is: " + i);
            test = model.moves.get(i);
            redoBoard(test);
            System.out.println("move added");
        //  model.printList();
            }
        }
        }catch(InputMismatchException e) {
            System.out.println("Error in input. Please try again");
        }
        //Exit method
    } while(choice != 3);


    }

        public void undoBoard(Move move) {

     int type = board.getBoard()[move.getyMove()][move.getxMove()];

     board.getBoard()[move.getyOrigin()][move.getxOrigin()] = type;
     board.getBoard()[move.getyMove()][move.getxMove()] = 0;
     model.undoChecker(move, model.findChecker(move.getxMove(), move.getyMove()));


    }

    public void redoBoard(Move move) {
        int type = board.getBoard()[move.getyOrigin()][move.getxOrigin()];
        int xValue = move.getxOrigin();
        int yValue = move.getyOrigin();
        board.getBoard()[move.getyOrigin()][move.getxOrigin()] = 0;
        board.getBoard()[move.getyMove()][move.getxMove()] = type;
        Checker checker = model.findChecker(xValue, yValue);
        model.redoChecker(move, checker);
    }
```

7 References

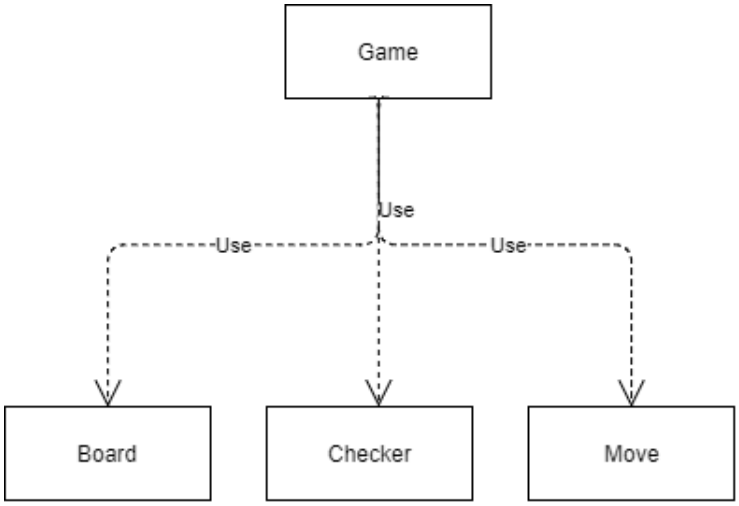
References

[1] Stanford College. *Strategies of Play*. 1998. URL: hhttps://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/index.html (visited on 11/10/2017).

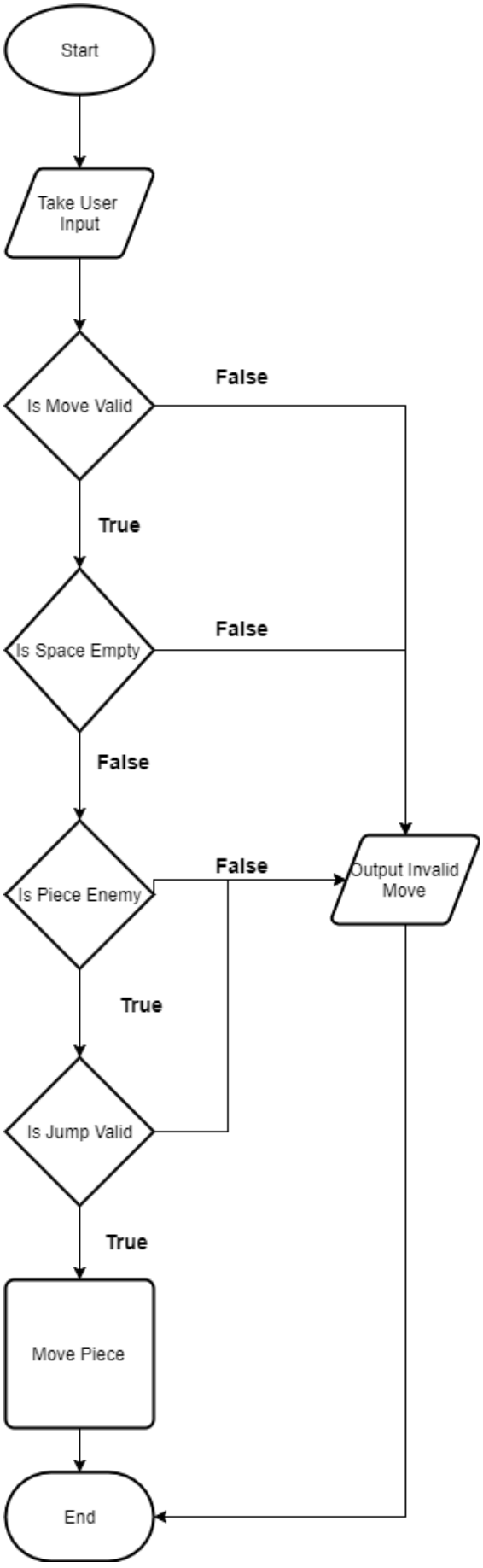
[2] Antoine Vianey. *Minimax example*. 2015. URL: https://www.programcreek.com/java-api-examples/index.php?source_dir=minimax4j-master/minimax4j/src/main/java/fr/avianey/minimax4j/Minimax.java (visited on 11/10/2017).

[3] Whatis. *What is view-model-controller*. 2011. URL: http://whatis.techtarget.com/definition/model-view-controller-MVC (visited on 11/10/2017).

8 Appendices



Class Diagram



flowchart for movePiece()

Logic