Edinburgh Napier UNIVERSITY

Checkers game coursework report

Alexis Canevali
40326915@live.napier.ac.uk
Edinburgh Napier University - Algorithms & Data structures (SET09117)

1 Introduction

This Coursework purpose is to implement a Checkers game (also named Draughts Game) using appropriate algorithms and data structures. To implement this game, we will be using Java language with an integrated Java UI called Java Swing. As there are different versions of checkers game from a country to another, We will implement the international Draughts rules. To find official rules, we refer to the official world draughts Federation (Fédération Mondiale du Jeu de Dames, http://fmjd.org). This website does not provide rules but it lists all national draughts federations, which could provide official rules. None regions of the UK have a website. We will then refer to the French draught federation who has a website with well explained rules, schemes and examples. You can find it out in french languageat: www.ffjd.fr/Web/index.php?page=reglesdujeu. The appendices[1] contains a recap of the rules traduced in english language.
The followings working features have been realised for this project: a gameboard representation, player representations, an artificial intelligence which is able to play, the game logic and rules, a graphical or textual interface, a game history with back/redo functionality, a game save/load option. This report will explain the choices made to implement these functionalities, the possible enhancement that could have been done if more time for this coursework was available, a critical evaluation and finally a personal evaluation.

2 Game design

2.1 Class diagram and object oriented conception

First, we conceived the structure of the game. We made an object model representation of the game[2]. To play a game, we need every object of the class Game. Indeed, this class references a gameboard composed of Checks, which can be the position of Pieces(whether Men or Kings). These pieces are owns by Players(who can whether be an AI or a Human).
Following section will explain in details how we implemented each class with appropriate data structures.

2.2 Main used data structures

2.2.1 Gameboard class

The first functionality to implement was the gameboard. With international draughts rules, The gameboard contains 10 lines and 10 columns (a total of 100 checks). The most efficient and practical way to do this is to represent it with a 2 integer arrays of size 10: gameboard = new int[10][10]. With this structure, we can access a check from its line number (for the first array) and column number (for the second one). For example: Check line5Column7 = gameboard[5][7]. This will also be useful to move piece because we will just need to take Piece's position and add or substract 1 from its current line and column to move it to its new position.

2.2.2 Checks class

The gameboard is composed of 100 Checks. A Check is an object which has attributes such as a line number, a column number, a reference to a piece object or null if there is nothing on it. Check is a composition association from the gameboard, so it has a reference to it.

2.2.3 Pieces abstract class (and Men/Kings classes implementation)

A piece is an abstract class. It can whether be a Man or a King. It has a position (a reference to a Check or null if the piece has been taken by the adversary). It also have a color (black/white). For men, they have a destination which is an integer (0 for black pieces : the piece must go from gameboard's top to bottom, 1 for white pieces : they must go from gameboard's bottom to top). This value is unnecessary for King because they can move in the direction they want. Kings and men have also specific capture and movement methods because these two entities do not respond to same rules.

[1] See the rules in the appendices page 5 to 8
[2] Class diagram in the appendices page 8

2.2.4 Player abstract class (and Humans/AI classes implementation)

Player is an abstract class because it can whether be a human or an AI. Each of them have an ArrayList linking to the pieces they own (it is the easiest way to stock and get a specific piece). Both human and computer have also a name (String).

2.2.5 Game class

The game class is the main class which regroup every necessary objects to play a game. It has 2 players, a gameboard and two LinkedList. The first LinkedList is to keep copies of the gameboard (to undo a move or more). The second one is to stock redo moves.

2.2.6 Graphical User Interface classes

The package UI contains every classes required for the Graphical User Interface. We will not explain in details the content of these classes as this functionality was not mandatory.

2.2.7 Other classes

The code also contains other classes such as Tree.java and DeepCopy.java The Tree class is a classical Node structure used to resprend n-ary trees. It's a generic class, the data stored can be the object or type we want. We needed it to calculate the longest capture possible for a given piece. The code is issued from the web[1]. We also implemented new methods such as getLongestTreePath() which return as ArrayList(s) the data of the longest branch(s) of a tree. The DeepCopy class is used to copy an object and all the referenced objects associated to it as new objects. It was also useful to calculate the longest capture possible by avoiding infinite looping on same piece capture. The code of this class is also issued by a website[2].

2.3 Main used Algorithms

We will cover in this section the most interesting and complex algorithms created for this coursework.

2.3.1 Getting possible moves (Method in class Man and King)

To choose the next piece destination, we call a method called getPossibleMoves() which returns a HashMap<ArrayList,Intger>. We needed this structure for following reasons :

- The Key is an ArrayList of Checks. This ArrayList is the "path" the piece is going to follow. If it just moves once (simple move or simple capture) the size of this ArrayList will be 1. If it is a multiple capture (called a riffle), size will be >1. As we examinate every options, the ArrayList is unique.
- We needed an integer value for the main loop which call this function. Indeed, it will help us classify the moves by type (0 for a simple move, 1 for a simple capture, 2 for a riffle).

This function works with the help of these other functions:

- getFrontMoves(): return the first diagonal checks if they are not occupied.
- getFrontCapture(Gameboard gameboard): Given a gameboard, we check if front diagonal check is occupied by an adversary piece and if the next diagonal check is free. If so, we make the piece disappear of the gameboard and we return it with this destination check (HashMap structure).
- getBackwardCapture(Gameboard gameboard): Given a gameboard, we check if back diagonal check is occupied by an adversary piece and if the next diagonal check is free. If so, we make the piece disappear of the gameboard and we return it with this destination check (HashMap structure).
- getRiffleMoves(): A recursive algorithm which build a n-ary Tree of Checks to calculate the longest capture possible. We're calling getFrontCapture and getBackwardCapture (passing a new DeepCopy of the gameboard at each call to avoid losing game data). As long as these methods do not return an empty HashMap, we do the same for the child nodes. An example is avalaible in the appendices[3]

We are calling these methods in the following order :

1. getRiffleMoves(). We transform the longest tree branch(s) into ArrayLists via a method located in the Tree class.
2. getFrontCapture / getBackwardCapture. We simply add the Check to an ArrayList (we don't need the gameboard copy).
3. getFrontMoves. We simply add the Check to an ArrayList.

We only return getRiffleMove() options in the HashMap of results if a multi-capture is possible. If not, we return in this HashMap of results all the possible front/back captures. If there are none, we return the HashMap of result with basic front moves or we return null if the piece can't move at all.

[3] See appendices page 8 and 9

2.3.2 Choosing a piece to move (Method in class Human)

The first action a player has to do to play is to choose the piece he wants to move. We simply get every pieces the player owns in an ArrayList (easiest to loop and get elements) and check if they have possible moves (calling for each piece the method getPossibleMoves()). It returns the HashMap<ArrayList<Check>,Integer>. We just need to get the Integer move type value to classify Pieces by move type (multiple piece capture, simple piece capture, simple move). If captures are possibles, other options aren't proposed to the player. If several pieces can do riffle capture, we will only return the piece(s) who can capture the most adversary pieces. Pieces are stocked with their HashMap of possible moves in a new HashMap<Piece, HashMap<ArrayList<Check>,Integer>>. This HashMap will be return to the main loop which will then show on the GUI the possible Pieces which can be moved.

2.3.3 Moving a piece (Method in class Man and King)

This function is used to move a piece from a current position to a new one (the position is a Check object). In this function we don't check if the move is legal, we simply execute it. We're also treating the case of a capture: if there is/are piece(s) in the diagonal between the piece current position and its new position, they are removed out of the gameboard (method die()). For men, we're also treating the case if a man arrive at the last row of the gameboard. If so, we create a new King object and delete this Man object.

2.3.4 Saving and loading a game (Method in class Game)

These methods are located in the game class. Save method is creating or overwriting in a file called "\data\gamesave.txt". We write in this file every object needed by the Game class and their attribute names + values. Each of them are delimited by character such as

- "*" Class definition (attributes name + value expected)
- "=" Class attributes name
- ">" Abstract class implementation
- "§" Abstract class implementation definition (attributes name + value expected)
- ":" Before ":" the attribute name, after the attribute value
- "|" Another attribute

The file is constructed following this structure. An extract of the game save file is available in the appendices[4]
The load function is called when the game is launched. It's opening the file (or throw an error if it doesn't exists).Then for each new object, it's reading his attributes names and values (and checking if syntax is correct, or printing the errors in the console). At the end, we're checking every attribute has been initialised and then we can start the main game loop.

2.3.5 Undo and Redo feature

The undo and redo feature is implemented and work using 2 LinkedList of the current game copies (1st linked list for the move to undo, 2nd linked list for the move to redo). At each round game, we add a DeepCopy of the game in the LinkedList of undo games. This deep copy uses Java.Serializable. Indeed, if we just add the current game in the linked list, it would be a reference to the current object and not a copy. The problem with Java.Serializable is it's a very heavy operation as it's copying every single object as new one (even referenced object). Our problem might be the copy of the Linked List because every time it copies them and all the game content as new objects. To avoid this problem, we empty the two Linked List before doing the DeepCopy. Indeed, if we included these LinkedList in the copy, this feature works well only when you play 6-7 times, then the computer becomes slow. The undo or redo feature is triggered when the user click on the related button. Onclick, we launch a method getPreviousGame() or getNextGame(). These methods check if the list is not null, and set the last game copy properties to the current game. We also remove this last element of the LinkedList that we just used and add it to the other LinkedList. We could also have use stack as it is as efficient as LinkedList for this application. However, the new properties can only be show to the user when the display is refreshed, so he musts click the button, play once to see the effect the undo or redo functionality.

2.3.6 AI

As our game logic makes mandatory the longest capture when it's possible, The AI is receiving the Piece it can move, and for each of them their possible moves. For each future possible moves, we're just checking if the piece isn't going to be captured at this next position. Depending on this condition, we're adding a HashMap <Piece, PossibleMove1 >element to:

- ArrayList<HashMap<ArrayList<Check>, Piece>>of good moves (Pieces won't die at that position)
- ArrayList<HashMap<ArrayList<Check>, Piece>>of bad moves (Pieces will be capture at that position)

Then we randomly choose an HashMap element in the good move or bad move one (if the first is empty) and execute for the given piece the associated move(s).

[4]See appendices page 9

3 Enhancement

If more time for this Coursework was available, following improvements could have been realised:

1. Being able to undo/redo the game at any time.
2. Improving AI by learning on how it can become more efficient (eg: adding new features such as making strategic plan).
3. Some parts of the code might be simplified and less data structures might be use.
4. Improvements possible on the complexity of the Algorithms/Data structures used to make the game smoother.
5. Re-structuring the code of the Graphical User Interface and better implement it with the game engine.

4 Critical evaluation

On my opinion, every essential feature to realise the game properly with command base interface is well realised. Indeed the gameboard representation, player representation, piece selection, piece possible moves, moving a piece, capturing an adversary piece is well realised. Easiest and most appropriate data structures were also use to represent games element, taking in count complexity on data structures operation (eg : Array is the most efficient way to access to a given check). Even if parts of the code are very complex and use costly recursive call and data structures (such as The getRiffleMove() method, HashMap to stock couples...) the execution time is still very quick and the user don't have to wait while playing a game.
On the other hand, we can admit the graphical interface isn't integrated properly because of my personal weak experience on the Java Swing GUI and the few times I had at the end of this coursework. Moreover this package can't wait for user action (eg: waiting for a click) which was very difficult to deal with. Some part might also be redundant and heavy (eg : get possible moves, creating new hashmap to check which of them must move in priority, looping on each HashMap element to compare their capture length...).

5 Personal evaluation

Realising this project made me face several challenges. The 1st of them was to list for a piece all its possible moves. Indeed it can be simple for a simple move or a single capture but it becomes more difficult when multiple captures are possible. To get all possible moves, the only solution was to get to the next check, analyse if other captures are possible without re-capturing the pieces we already have captured, without capturing our own pieces. This functionality took a lot of time because it was necessary to make copies of the gameboard every time a new capture is possible(and delete from this copy the pieces captured). The solution used with a n-ary tree representation of capture might not be the simplest and the most performing one but it works fast enough for the user.
Another challenge was to implement the AI. I never implemented one but as the rules oblige the player to capture the most adversary pieces, it was just a basic implementation with a single condition : not being captured at the next position. It can certainly be improved but a lot of more time and knowledge would have been necessary.
The Graphical Interface was also hard to implement because it needed changes in code compared to the command line code. Moreover my experience with Java Swing was very limited.
This coursework was a good way to see real life application of every structure seen in class and which of them is the most appropriate to use. Before, taking in count the efficiency and complexity of each algorithm and data structure used was not one of my programming habit. After having the problem with the games DeepCopy for the undo/redo feature, it helped me understand how important it can be. Furthermore, it also made me notice how every code structure/loop structure can change the smoothness of a program.
Eventually, I feel this Coursework allowed me to know and use new data structures in real context application. As most of the functionalities are realised and work properly I am satisfied by the work I provide for this coursework.

References

[1] Tree structure code from Stack Overflow forum, https://stackoverflow.com/questions/3522454/java-tree-data-structure

[2] Java Techniques Website, article about Deep Copies of Java objects, http://javatechniques.com/blog/faster-deep-copies-of-java-objects/

6 Appendices

6.1 Official rules of the international draughts\checkers game

These rules are provided by the FFJD (Fédération Française du Jeu de Dames), http://www.ffjd.fr/Web/index.php?page=reglesdujeu

6.1.1 Starting a new game

1. The international Draughts game is played on a checkerboard of 100 checks, alternatively of a clear then dark colour.
2. The game is being played on the dark checks.
3. Each player owns 20 pieces. They can choose between black or white pieces.

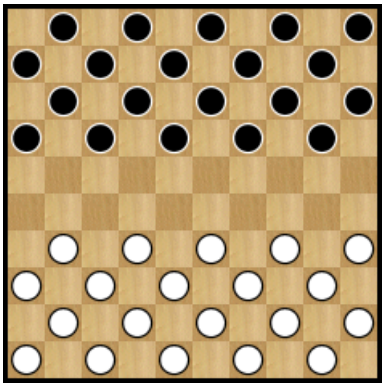

Figure 1: Initial position of the pieces on the checkerboard

6.1.2 Moving pieces

1. There are 2 types of pieces : the Men and the Kings.
2. 1st move is always played by white pieces. Then players one after the other.
3. A man is moving only forward, in diagonal, of one check on a free check of the next row.

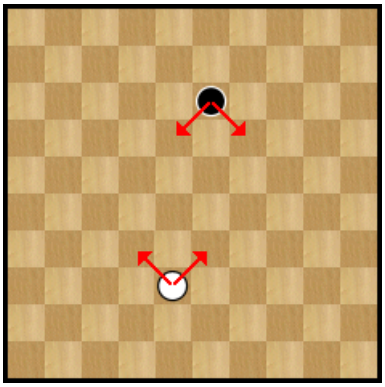

Figure 2: men moves

4. When a man arrive at the last row, it becomes a King. It is represented in our game with a red circle in the middle of the piece.

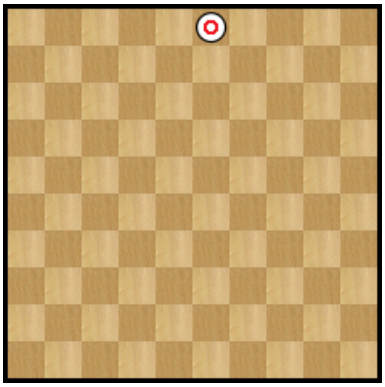

Figure 3: a white man which became a King

5. A king can move diagonally forward or backward as long as the checks are free.

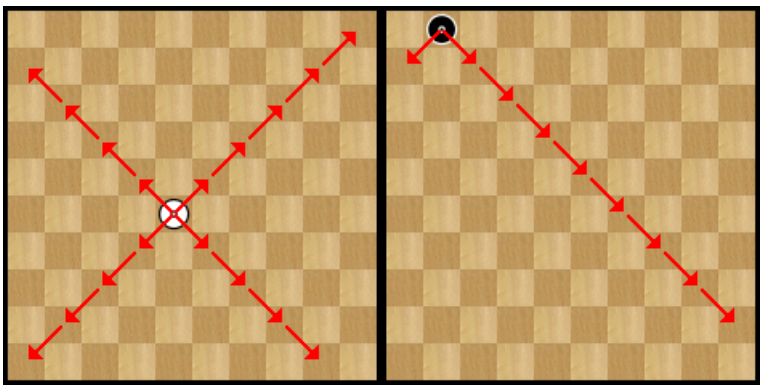

Figure 4: Kings possible moves

6.1.3 Capturing pieces

1. The capture of adversary pieces **is mandatory and must be executed forward and backward (for both men and kings).**
2. When a piece has an adversary piece diagonally close to it and that the next check is free, the piece must jump over the adversary piece and occupy the free check. The adversary piece is then remove of the checkboard. This operation is called a "capture by a piece".

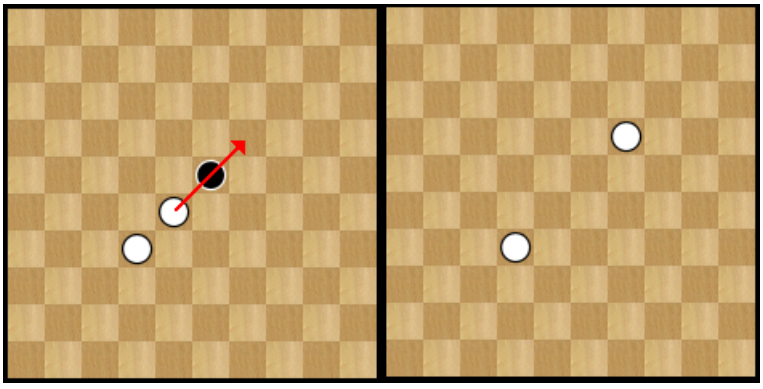

Figure 5: Capture by a piece

3. When an adversary piece is captured, and the piece can capture another one, it musts do it. As long as capture are possible, they must be executed. Captured adversary pieces are then removed of the checkboard. This operation is called a "riffle by a piece".

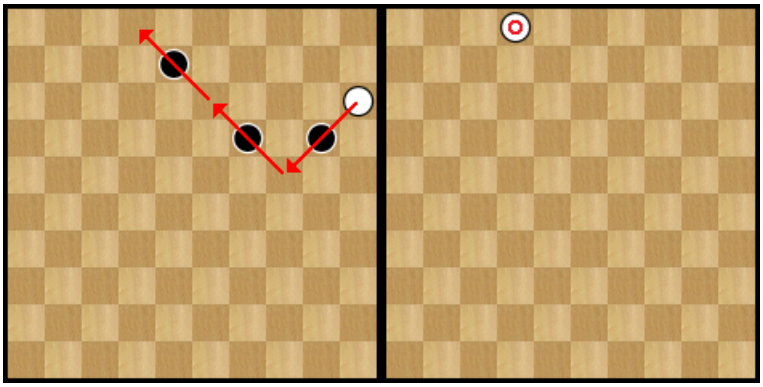

Figure 6: Riffle by a piece

4. Riffle are also possible for Kings. When an adversary piece is present on a King's diagonal, it musts be captured (if free check(s) are present after this adversary piece).

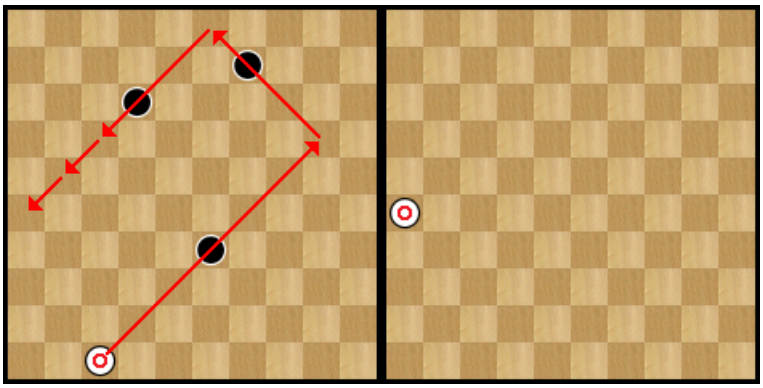

Figure 7: Riffle by a King

5. The capture of the greatest number of pieces is mandatory. In this case a King count as much as a man. It doesn't give any capture priority.

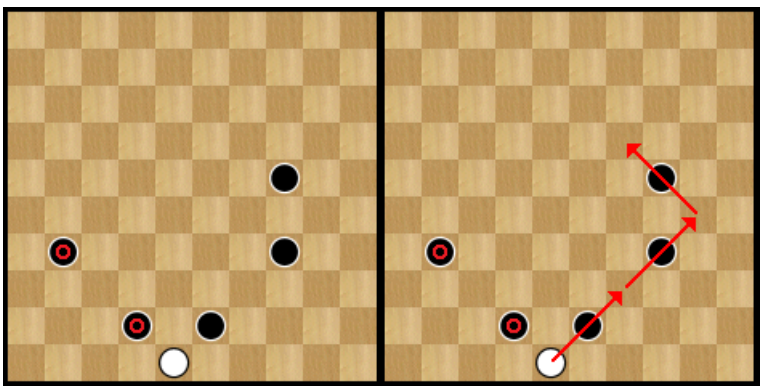

Figure 8: Capture priority

6.1.4 Irregularities

If an irregularity has been committed, the other play must decide if this irregularity must be rectified or kept. Here are few examples of irregularities :

- Player twice
- Playing with an adversary piece.
- Playing with a piece while a capture or a riffle is possible.
- Removing without any reasons adversary or own pieces of the gameboard.
- Capturing a greater or inferior number of pieces than the maximum possible.

- Stop the execution of a riffle.
- Removing pieces while executing the riffle.

6.1.5 Equal end of a game

A game is considered as equal when : (not implemented in this game)

- The same position is played for the 3rd time.
- For 25 game turns, there is no piece capture.
- If there are only 3 kings left, 2 king and 1 man or 1 king and 2 men, the game will be considered as equal after each player played 16 times.
- If there are 2 kings playing versus 2 kings or 1 king vs another king.
- If equality is decided by both players.
- None of the player manage to win the game.

6.1.6 End of a game

A player wins the game when :

- The other player has abandoned.
- The other player can't move any piece.
- The other player have no remaining piece on the checkboard.

6.2 Class diagram

During the conception phase of this game development, following class diagram has been conceived. Please note that all methods aren't present in this diagram.

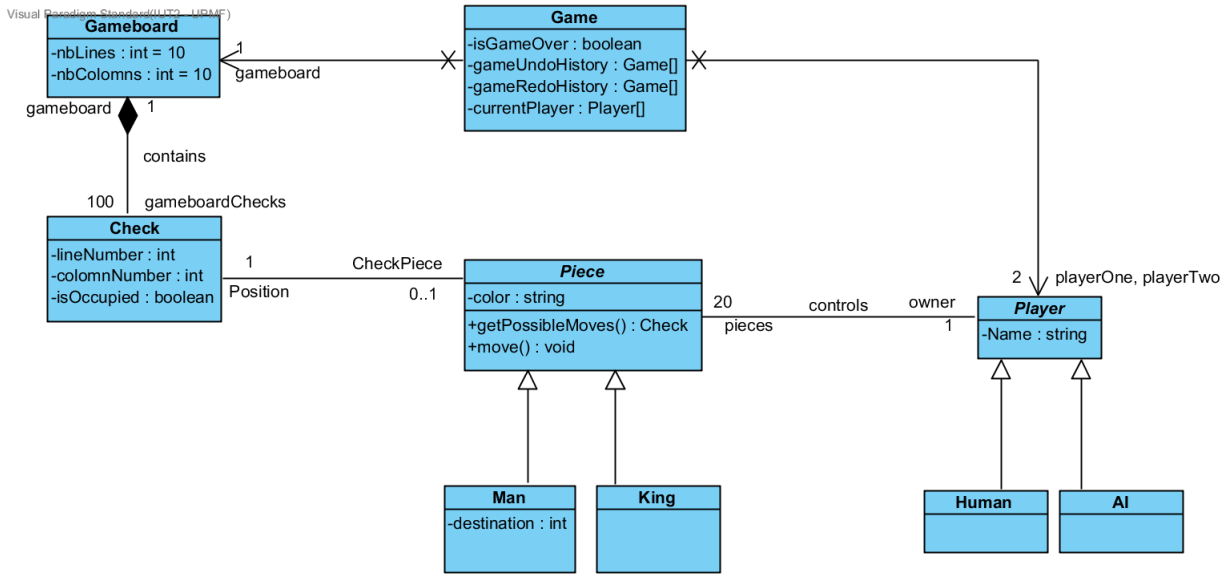

Figure 9: Class diagram of the international draughts games

6.3 getRiffleMove() method example

This example shows the tree build by the function getRiffleMove(). The root node is the actual piece position, children node are free check capture options. This algorithm returns this tree in the getPossibleMove() method. If the tree size is greater than 1, a riffle is possible (else it's just a simple capture). The getPossibleMove() transform the longest tree branch(s) to an ArrayList of Checks and via a method located in the Tree.java class.

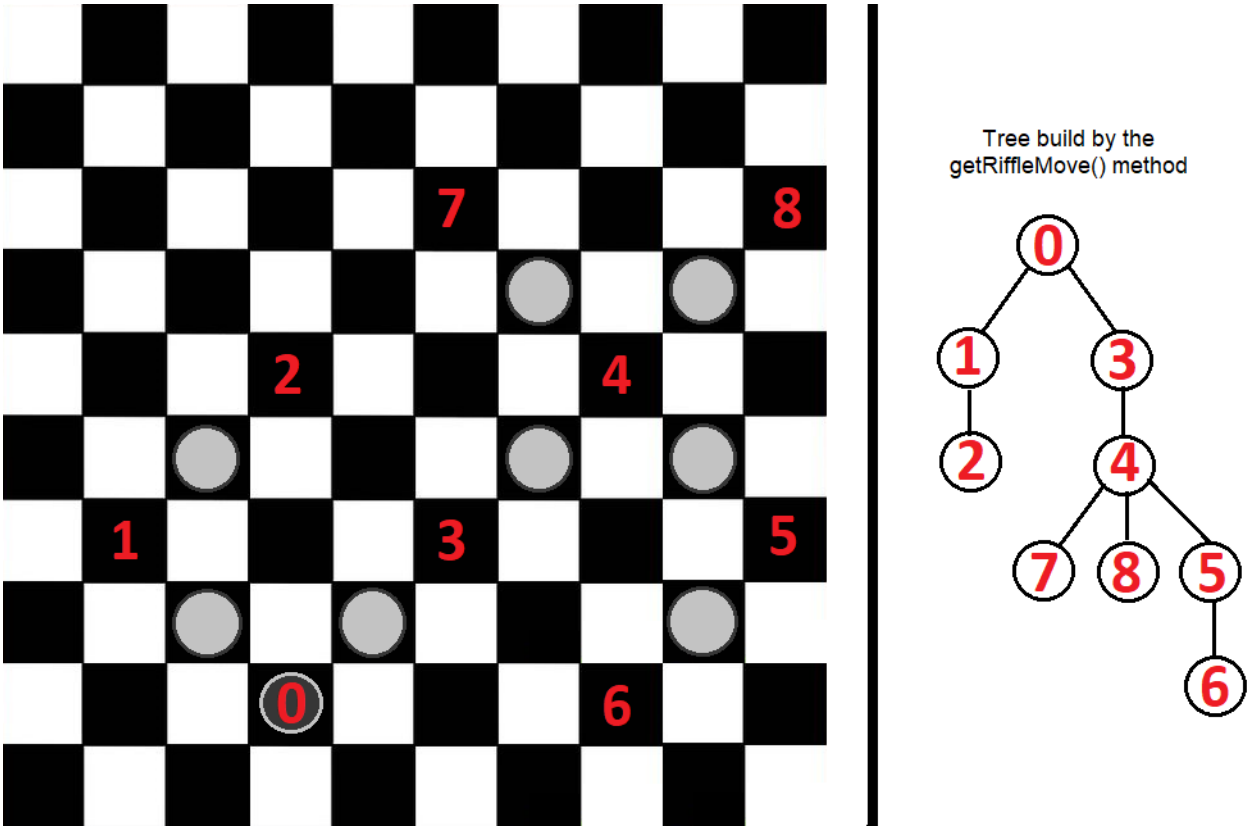

Figure 10: Tree build given an example scenario

6.4 Game save file extract

This is what the gamesave.txt file look like (Please note some lines for the checks has been omitted because they were similar to others).

PLAYERONE=PLAYER>HUMAN§NAME:Test
*PLAYERTWO=PLAYER>AI§NAME:Computer
*CURRENTPLAYER=PLAYER:PLAYERONE
*CHECK=LINENUMBER:0|COLOMNNUMBER:0|PIECE:null
*CHECK=LINENUMBER:0|COLOMNNUMBER:1|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:0|COLOMNNUMBER:2|PIECE:null
*CHECK=LINENUMBER:0|COLOMNNUMBER:3|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:0|COLOMNNUMBER:4|PIECE:null
*CHECK=LINENUMBER:0|COLOMNNUMBER:5|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:0|COLOMNNUMBER:6|PIECE:null
*CHECK=LINENUMBER:0|COLOMNNUMBER:7|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:0|COLOMNNUMBER:8|PIECE:null
*CHECK=LINENUMBER:0|COLOMNNUMBER:9|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:1|COLOMNNUMBER:0|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:1|COLOMNNUMBER:1|PIECE:null
*CHECK=LINENUMBER:1|COLOMNNUMBER:2|PIECE>MAN§OWNER:PLAYERONE|COLOR:black|DESTINATION:0
*CHECK=LINENUMBER:4|COLOMNNUMBER:3|PIECE>KING§OWNER:PLAYERTWO|COLOR:white
[...]