Edinburgh Napier UNIVERSITY

Coursework Report

Rafal Ozog
40217610@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Keywords – checkers, coursework, AI, multiplayer, game, algorithms, rafal, ozog, report, data, structures, Java, artificial, intelligence

1 Checkers - simple, but complicated game

The coursework aim was to create a text-interface game, presenting our understanding and proper using of appropriate algorithms and data structures. The game doesn't present a pretty design, however it was not the goal to make the interface good-looking. Antoine de Saint ExupÃľry said "what is essential is invisible to the eye". I think this statement refers to my project in some way. The most important things - internal code structure, well-organized alogorithms, properly chosen, implemented and used data structures, readable and easy to use interface, Artificial Intelligence - are hidden inside. Let me explain all of them in details.

1.1 American or English rules?

Before I started making first schemes I had to find out the game rules, which I was actually considering as 'client requirements' of this project. I realized that there are many 'Checkers' versions around the world - each of them is a little different from each other. I have decided to follow the rules of the classic version, called "English draughts". In this edition the pawns can move and attack only forwards, however there are also kings (pawns, which achieved the last row of the board), who can attack backwards as well. In compare to American version of 'Checkers', here the king cannot move further than onto an adjacent position. The player who lost all pawns or doesn't have any sensible move opportunity - loses the whole game.

1.2 Two game modes

When the user starts the game, the first thing he is asked to choose is the game mode. There are actually two of them: 'human-computer' and 'human-human'. The user can make a choice simply using the keyboard (according to displayed instruction, he just need to type a correct number and press 'Enter'). If the first option has been chosen, the user will have to decide how smart his artificial opponent should be...

1.3 3 difficulty levels

The game has 3 different difficulty levels - easy one, medium and hard. Each of them contains a little different

```
Welcome to 'Checkers V2 3D 4K'!
Before we'll start a game, please choose a game mode:
Press '1' to play against another human player,
 or press '2' to play against a computer player.
2
Please choose a difficulty level:
Please type 'easy', 'medium' or 'hard'.
```

Figure 1: **Game Main Menu**

internal logic, allowing AI player to act better or worse, depending on chosen option. It was one of the main challenges of this project - to decide how smart the computer player should be - and how to implement that Artificial Intelligence within the game. I would like to explain how these algorithms work in details - later on.

1.4 'Undo', 'redo' or maybe 'replay'?

The game allows user to shift back and forward. These functions ('undo' and 'redo') have been implemented using two different linked lists, which store past and present moves (see more details in section 3.2). To choose any of these functions user only need to type its name - in any moment of the game. When the user types 'replay', the program will display whole game from the beginning, move by move. After this the game can be continued.

2 Code Design - how the game looks like inside

Before I started choosing proper data structures and designing algorithms I needed - I had to decide how the Java code should be organized. It was much more important for the whole project than it seemed to be at the beginning. In the middle of the work I realized that the code is too messy. When I had implemented first AI algorithms, which allowed me to play against simple, computer player, I understood that the code was working properly, but it was not enough. I have concluded that the code has to be clear enough to be understood by other people, who would like to improve it or reorganize in the future. Another reason it is important for, is that I wouldn't be able to add more complicated AI and rule-checking algorithms in such messy code, being sure that everything works well together. It was the crucial moment of the whole work. Finally I've decided to create a totally

new Java project, which would be based on the code I already had prepared. My new project code has been split into following classes, containing different functionality sections...

2.1 Game class

This is the main class, which contains method 'GameMenu'. It is actually the main interface, which allows user to choose different game options (level, mode). When the settings are chosen, the method proceed the whole game using other methods, which perform moves, check if the game is finished etc. In this section there are also located some global Linked Lists, which store moves both for 'undo' and 'redo' functions (I will tell about them a little bit later).

2.2 Board class

The Board class contains a few static (as there is only one board for the whole game) methods: 'newBoard' (creates a board), 'putPawns' (puts pawns on the created board), 'printBoard' (displays the board with current pawns arrangement) and 'checkIfEnd' (checks if both players still have at least one pawn on the board).

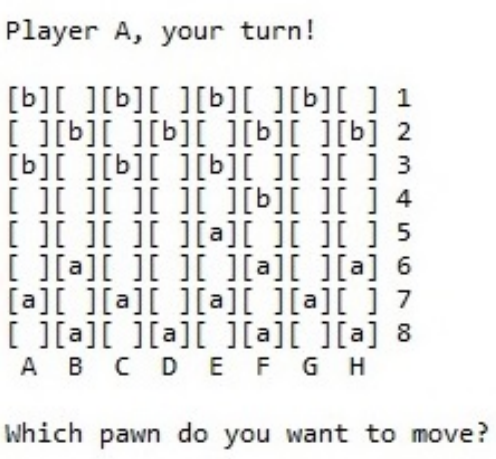

Figure 2: **Game board**

2.3 Movement class

In this section I have placed all functions related to pawn movement. The most important of these method is 'checkMoveValidity', which takes the details of the next move and checks if is it valid across a set of game rules (rule by rule). If it's not - then the user has to choose another one. There are also functions, which perform the move (one for Player A, one for Player B and one for Computer Player), 'undo'/'redo' and some additional ones used by them.

2.4 ComputerMove class

In terms of code design it was definitely the most challenging part of functionality. I have decided to create this class exclusively for AI purposes. One object of this class stands for one particular move, that will be performed by AI player (the object is passed to the Movement class). The function checks if the pawn has a single move opportunity and if it can jump over other pawns. If the jump (attack) is possible, then the proper algorithm compares all possible attacks and chooses the most powerful one (which details and coordinates are passed within the 'ComputerMove' object). I would like to explain how the used algorithms work and perform in the third section of this report.

3 Internal Design - Data Structures and Algorithms

3.1 Game Board

The board has been implemented simply using **two-dimensional array**. This solution allowed me to be sure, that the **fixed size** of the board will not be changed accidentaly (like it could happen with a list for example). Each array cell represents one particular field of the board. Different numerical values indicate different fields statuses (occupied by A player, occupied by B player or empty). This simple solution allowed me to check the current status of each field, print the board and apply validating/AI alogrithms quite easily and in readable way.

3.2 Moves Storage - 'undo', 'redo', 'replay'

The game allows user to use 'undo' and 'redo' functions. It is also possible to replay already recorded game (move by move; when the user types 'replay'). All these features have been implemented using two particular Linked Lists - 'movesStorage' and 'revokedMovesStorage'. Each of the lists can store two-dimensional arrays, which represent different board states (pawns arrangements). I have decided to use **Linked Lists** for this purpose, because of their **flexibility**. I didn't have to fix their size and could easily access each single element contained within them. The second feature was especially important for **'replay' function**, which has to read all past board states (so needs an access to all list elements). When the user calls **'undo'** function, the most recent game state is moved from 'movesStorage' to 'revokedMovesStorage' and the second recent game state is set as the current one. When the user types **'redo'** - the process is proceeded in the opposite direction. When the user decides to perform a totally new move, the 'revokedMovesStorage' is cleared and ready to be used again.
As the 'undo' and 'redo' functions operate only on the one end of the list (despite the 'clear' method) - it is a kind of **'stack'** (it is not in real, because it has been implemented using list, however I mean about a data structure abstract, useful because of designing purposes).

3.3 Validating Algorithm

One of the most important algorithms, which actually makes the 'checkers' game (instead of randomly moving pawns) is the algorithm that proceed the move vali-

dation. I have concluded that the structure of this function should be simple and readable enough to allow changing or adding game rules in the future. Finally, **the algorithm is basically a set of rules** - conditions that the correct user move has to meet. The conditions are checked in order. If any of them cannot be met, then the function terminates and returns 'false' value. In this situation the move cannot be performed and the user is asked to choose another one. The algorithm checks:
-if the user is going to move a correct pawn,
-if does the move have a correct direction,
-if the destination is not already occupied (and if is not out of the board),
-if the pawn is going to move forwards (unless it is a king),
-if the move distance is not longer than one unit (unless it is a correct jump).
The algorithm checks only human player moves (the AI player doesn't need it - it will be explained it in the next paragraph).
A big advantage of this algorithm is that **its performance doesn't depend on the game situation** and is directly proportional to pawns number (to board size) with the **complexity equal O(n)**.

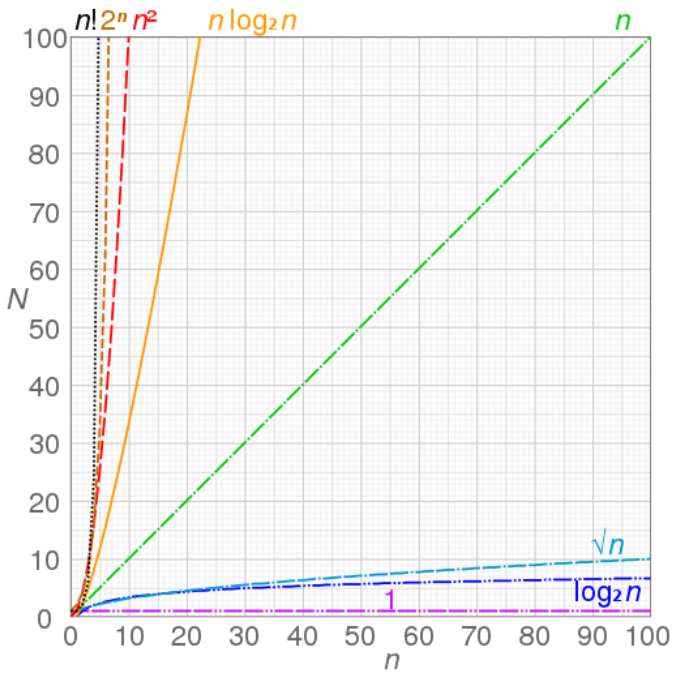

Figure 3: **- O(n) compared to other complexity types**

3.4 Artificial Intelligence - Computer Player

The most challenging part of this project (for me personally) was **the AI player implementation**. It consists of a few code modules, so I will start my explanation from the general algorithm dscription. The main aim of this code is **to find out the best possible move**, which will be performed by the Computer Player. The algorithm is going to check each single pawn and evaluate its 'potential'. In my project this term indicates an overall value of any single move. **The evaluation** consists of **three steps**:
1) Check if the pawn can perform an attack - if yes, then evaluate how many pawns can be maximally jumped over - one potentially attacked pawn adds 3 points to the 'potential' value.
2) Check if the pawn can move forward - if yes, then this opportunity equals 1 point (of the 'potential' value).
3) Check if the pawn can be attacked on its final position - if yes, one point will be subtracted. The pawn with the highest move 'potential' will be chosen by Computer player (to perform that particular move).

3.5 How to evaluate the attack potential?

The attack potential is evaluated by a separate function called **'checkAttackPotential'**. This is **a sub-algorithm** based on **my own idea**. The function creates two separate lists (Linked Lists) - one for left-side path, second for right-side one. Special 'if' statement checks if the attack is possible in any of these ways. If yes - then the function is called recursively for the next position (attack final position), which in turn creates another two lists and checks its left and right attack opportunities. If the position doesn't contain any attack possibility, then adds its own position into the empty list and returns it. Finally, the algorithm gets two lists - one with the longest possible attack path for the left side and the second for the right one. The longer one will be proceeded.
It is another situation, where **the data structure (attack path)** is implemented using a list, however the algorithm operates only on its one end, treating it as a **'stack'**. Data structures illustrated by abstracts can be useful in designing process (may help us to understand the internal logic of algorithms).

3.6 Three difficulty levels implemented within one algorithm

As I already mentioned, the game contain **three different difficulty levels** - all of them are preformed within the same AI algorithm. The solution is quite simple - the hard mode takes all three evaluation steps into account, the medium one doesn't care about opponent's attack and the easiest one - doesn't even check its own attack opportunities (considers only single moves).

4 Enhancements - what could be improved?

4.1 Graphic Interface

The most significant feature I would like to add to this project would be a beautiful, graphic interface. I am convinced that the first impression is even more important than internal logic of the program. If the game doesn't look pretty, user just switch it off and doesn't have an opportunity to get to know what's hidden inside.
I would like to implement a good-looking GUI main menu interface, graphically prepared board and pretty pawns. Another feature, which would also be desired is an animation of the move.

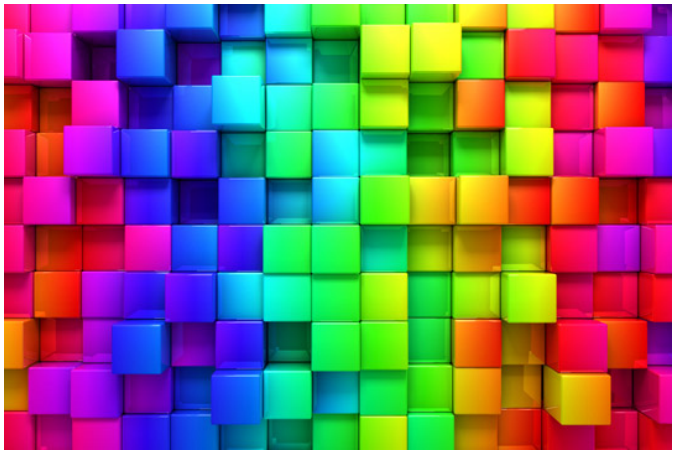

Figure 4: **Colors** - something that my current interface is missing

4.2 AI - the opponent move

Another feature I would like to implement would be some AI player improvement. In this moment the algorithm takes into account computer player's attack opportunities, single moves and in some way tries to avoid being jumped over. However, it doesn't consider its opponent possibilities. I would like to implement some algorithm (like Min-Max), which would allow the computer player not only to perform the currently best possible move, but which would also assume the best opponent's move performance and would take it into account.

5 Critical Evaluation - how do I judge myself?

5.1 What is good?

It is always difficult to grade our own work. In my own feeling, I am satisfied with the internal logic of the game. **AI player with 3 different difficulty levels** is a piece of code **based on my own algorithm idea** and makes me content (as few months ago I wouldn't even know how to start doing things like that).
The second feature I am happy with is **the rules validation**. I have been following the English draughts rules and have implemented almost all of them successfully. In my opinion it makes the game realistic and by that - really playable.
Last thing I would like to mention as being proud of is **a code quality**. As I already mentioned I have proceeded the whole code **refactorization** during the work - and I am convinced that after this stage, my code became much more readable and neat.

5.2 What could I do better?

I am not really happy with the fact that I didn't spend enough time to provide **sufficient testing** of all features combined together. I used to proceed only simple testing, usually just after some feature had been added. Unfortunately, it resulted in some hidden errors and bugs.

6 Personal Evaluation

I can honestly say that this project allowed me to improve my programming skillset.
I was already familiar with the majority of data structures I needed, however I became much more familiar with their different implementations and interface methods. I can say I have achieved some knowledge and understanding of useful, programming abstracts.
I had a great opportunity to practise alogrithm designing. As I mentioned before, some time ago I would not have any idea how to implement any AI within the program. There were two alogorithms in this project, which were for me especially challenging - move validation and the function looking for best possible move for the computer player.
What is more I have practised other IT skills during this work - Java programming, code quality keeping, GitHub using.
At the end I can summarize this project saying - **I had a good fun working on this - and I feel that I took some values from it.**

References

'colors' image source:
http://www.businesscomputingworld.co.uk/black-grey-red-or-blue-what-colour-says-about-your-business

'big O complexity' graph:
Wikipedia resources