Coursework Report

Jostein Dyrseth
40223535@live.napier.ac.uk
Edinburgh Napier University - Algorithms And Data Structures (SET09117)

Keywords – python, data-structures, checkers, game, AI

1 Introduction

Describing the problem and an overview of features:

1.1 Problem Domain

The problem domain in this task was to implement and integrate appropriate data structures for holding the different data in a checkers game. The overall structure needs to be well defined so that the objects can encapsulate it's logical, sensible and relevant data. What also are included in the problem domain is to understand the underlying effect of using and applying algorithms to the software so that given a specific goal, hopefully a good or an outstanding outcome will be returned. In this case by applying an algorithm to the game, the outcome we want is a series of moves in which the agent will be taking.

However, as well as all this it is also important to be considering the ways in which we as software developers can do all of this in a most efficient way as absolute possible. Not because it is necessarily important for this exact case, but to get hold of that knowledge so that it can be used in a larger scale, where efficiency is a lot more expensive.

1.2 Overview

The overview of the task is to apply our knowledge and understanding of the theory from the lectures as well as the provided- and online resources in relation to algorithms and data structures of how best to structure code to satisfy the data being held. We were to choose ourselves whichever language and technologies to use for this task. The game must be able to be played in between two humans, two computers, or between a human and computer. It should also include play history, replay-function and a simple AI-agent. The game is to be run from the command line. We could also add more functionality to the game to higher the grades as well as implementing an graphical user interface.

Python is the technology being used for this game in the command line.

1.3 Features

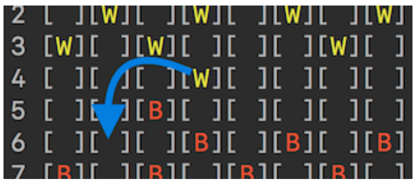
Briefly explaining the features of the game; firstly looking at the fact that the user can choose if he wants to play against a computer, another human or watch two computers play against each other (see how this is done in the design section) has been implemented and taken into account in the code, but the game never asks for this input yet. A move is the default functions or prompt that the user gives (if the user chooses to be a human). Then the prompt will ask you where to move to. After the given input is interpreted the move-calculation will be done and the next-turn method will be executed giving the opponent the next move.

Also, mentioning that whenever the board is being drawn all the pieces has different colors applied to them to easier distinguish which belongs to which player (See figure below).

A jump can be made, and must be made if a piece of opposite color are at the forward-right, or forward-left (diagonal) position from the pieces' perspective. The piece that has been jumped over will also disappear after this action is been made. Also when this type of jump has been made, it will still give that same player one more move which is how checkers should also work in real life.

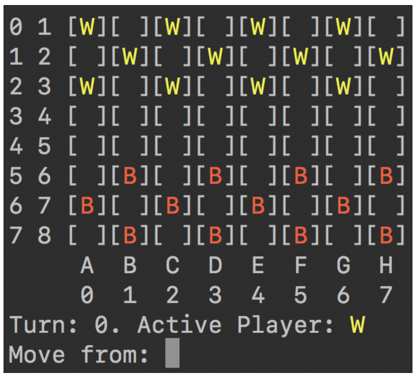
2 Design

Explaining how you designed architected your software paying particular attention to the algorithms and data structures used:



```
2 [ ][W][ ][W][ ][W][ ][W]
3 [W][ ][W][ ][ ][ ][W][ ]
4 [ ][ ][ ][W][ ][ ][ ][ ]
5 [ ][ ][B][ ][ ][ ][ ][ ]
6 [ ][ ][ ][B][ ][B][ ][B]
7 [B][ ][B][ ][B][ ][B][ ]
```

Figure 1: This is a mandatory jump for player white, since it is player white's turn.



```
0 1 [W][ ][W][ ][W][ ][W][ ]
1 2 [ ][W][ ][W][ ][W][ ][W]
2 3 [W][ ][W][ ][W][ ][W][ ]
3 4 [ ][ ][ ][ ][ ][ ][ ][ ]
4 5 [ ][ ][ ][ ][ ][ ][ ][ ]
5 6 [ ][B][ ][B][ ][B][ ][B]
6 7 [B][ ][B][ ][B][ ][B][ ]
7 8 [ ][B][ ][B][ ][B][ ][B]
     A  B  C  D  E  F  G  H
     0  1  2  3  4  5  6  7
Turn: 0. Active Player: W
Move from: 
```

Figure 2: The board-pieces has different colors to better distinguish the players from each other when a human is playing or spectating a game between two computers.

2.1 Game Object

The very first thing that happens when the game is being run is that the main function will initialize a game object stored as a game variable. The game takes in two arguments: player-kind (human, or computer) for player as well as for player two.

2.1.1 Constructor

The game constructor will set the turn count to 0, set the two players' color and create a list of initial pieces. The initial pieces are stored in two separate static variables within the Player class. One for player one, and another for player two. Then it will initialize the board-object, as well as setting the active player to player one.

2.2 Player Object

The players have a class that when being initialized will construct a player-object with the player-kind, color, a list of pieces and a list of initial pieces depending on what player type the player is. The player class also holds two enumerator-objects (See below in the Enum section).

The player class also has methods for printing the pieces it holds, return true or false depending on if it has pieces or not and appending a piece.

2.3 Board Object

A good way to store the board most likely is to make a list of lists (2D-Array). The board has a fixed size of 8 * 8. For every slot in the board there will be created a square object. Realizing the way of structuring the board object was important from the very beginning as this object might be the most central object of them all. The reason the board always has a fixed size is simply to make the moves easier. Also since we know that the size will always be 8 * 8 and never change we can do this, otherwise it would maybe be better to store each piece in a dynamic list rather.

```
board = [[Square(Point(x, y)) for y in range(BOARD_SIZE)] for x in range(BOARD_SIZE)]
```

This class also has a method to populate the board. For all the points in the player's static piece-list, place a piece in the board at that vary space.

2.4 Enums

Enums are useful to stor static data of the same genre. Like already explained, the player class holds two enums-objects. The first enum tells if it is a human, or a computer. This gives meaning, it is logical and is easily interpreted by someone else potentially using the code at some point, as well as the documentation provided in the code itself.

```
class PlayerKind(Enum):
HUMAN = 1
COMPUTER = 2
```

This is an efficient way of storing simple constant data that gives meaning to the class.

The PlayerColor-Enum uses the same data structure and syntax. WHITE: is "W" and BLACK: is "B". Again this is to simplify and clarify the meaning and semantics of the overall purpose this object has.

2.5 Square Object

A square object should represent the square or space on the actual board itself when playing checkers. The square holds a piece if and only if there is a piece at that given square, else it will just hold a None-value as a piece which really is just not defined at all. Or put in other words, returns literally nothing at all which is quite what we want in this case.

As mentioned the board holds empty squares in all of it's slots (8 * 8), including those that already have gotten pieces in them too obviously. This should be an equivalent representation of the physical object.

Interestingly this class also was made with a set-piece-method. This is just a setter method, but it doesn't only set the piece, it also sets the current-square to equal itself. The reason I choose to do reference each other is so that the pieces always knows what square it is in, but also so that the square knows what piece is in it.

2.6 Piece Object

So the piece will hold information about the player that owns the pieces itself, a list of it's valid moves, a list of potentially mandatory moves and the current square it is bound to which was just explained above.

The piece also has a string representation where if trying to print this in the debug version in the console, you will simply see it's position or x and y values.

We also populate the player-list with the add-piece method.

2.7 Point Object

The point object's purpose was to hold coordinates for x and y so that it is possible to know where each piece is at all times. However, this was being discarded as a false or bad way of implementing the overall structure, the reason being that the piece does not in theory need to know where it is. It only needs to know what square is holding it, rather.

2.8 Turn Object

The turn class's purpose is to hold information about the current turn. This also would make the undo redo functionality easier on a later stage. It should be possible to pop them off of the turn-list if an undo is done, and append them back on if the user is doing a redo.

The turn holds information about the current piece that should be moved, the destination-square, the board itself and additionally if the player should get another move (only if a jump has been made).

The turn class will then run the prompt user method, or calculate move depending on what the current kind of player it is. If player kind is a human then the prompting will happen. Here as previously and briefly described, the input from the user will be mapped to the location of that given piece's coordinates. This will then ask if that piece exists, and then set the current piece. If the piece was not found, then return an error to the user telling them that there is no piece at the give location.

3 Enhancements

Describing the features that you would add or improve if you had more time:

3.1 Undo(), Redo()

Given the overall well structure of the game, the fact that I did not have time to properly implement the undo and redo functionality was quite disappointing. I had a few errors the last couple of days, so I could not figure out what was wrong, hence making it difficult to implement just a few more things that I wanted in a very short time-window.

As briefly mentioned earlier the undo should just simply pop of one value from the turn-list which is being hold by the game-class. And the if the user does a redo, then simply re-add/append that very turn-object to the list again. This should in it's simplicity and in theory be quite straight forward.

3.1.1 Replay

For the replay as well, this is just a list of moves/turns that has been made in the correct sequence in the turn-list. Was there more time I simply would have re-drawn the board for each turn again in the list, and let the user press n (next) in the console to move foreward, and b (back) to go back.

3.2 AI Agent

The Algorithm I had to implement was there just no more time for. I should have made a simple one, but I just could not find the time to do it. The game was nicely laid up for it to been implemented, but unfortunately there is none at this version.

3.3 Debug Version

Making the debug version work is also something I wanted to implement.

4 Critical Evaluation

Explaining the features that you feel work well, or work poorly, and why you think this:

4.1 What I Feel Worked

I think that the move of the pieces works very well. The pieces are only allowed to move diagonally forward relative to the player. So most of the time only two places are valid moves for each piece. If there is a piece of the opposite color, then the player have to do a jump - nothing else works. I think that the validation is quite good and I spent a fair time dealing with that and the rules of what the players are allowed to do or not.

The game object initiates and stores each turn directly in the turn list, making the code quite compact and readable.

The structure of the board is good as well as the other classes. They are all holding the relevant data it is supposed to hold.

I also like the structure of the files. Instead of having just one file with all the code, I split the files up each containing it's own class, and then rather importing the parts of the modules that I needed at any given time.

4.2 What I Feel Worked Not So Well

The turn class should maybe have been split up to a move class as well. The validation should probably happen after the user gives the source-piece as well as the destination, instead of after both, hence making it more stable.

The validation function in the board class also got too long, so I would definitely have refactored that one too, making the code neater and simpler to read.

5 Personal Evaluation

Reflecting on what you learned, the challenges you faced, the methods you used to overcome challenges, and how you feel you performed:

I was facing a lot of challenges. First and foremost was the overall structure of the Python language. This was quite similar as other languages, especially C, but it still had some slight changes. I had to look online and in the documentation to gather some information of how to write the actual syntax.

I feel I both used a lot of time on the project, giving a lot of effort and basically trying my best to finish it. There were so little time in the end as mentioned, so the last few features were no time for anymore.

I learned a lot in this coursework. Not only how to better structure the code, but also some great syntactic features that python is providing in it's language where doing multiple loops in one line, creating immutable tuples, mapping values, storing data as enums, making subclasses, importing modules from my local library etc. All this made me like python a lot more and opened up a new door for me.