Edinburgh Napier UNIVERSITY

Coursework Report

Mandy Yip
40210041@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET09117)

Abstract

A report which discusses a python based Checker/English Draughts game designed to run on the command line.

Keywords – Fill, These, In, So, google, can, find, your, report

1 Introduction

Checkers/English Draughts is a strategic board game played by two players with 12 controllable pieces each on an 8x8 chequered board.

Players aim to win by removing the opponents pieces by moving their own pieces, represented by either "r/R" or "b/B", diagonally around the board and taking pieces when a valid jump is able to be made.

This report discusses a game written using Python, and can be played on the terminal/command prompt. The game allows players to play against another player or against an AI, with additional features such as undo/redo. The players can also choose to quit by typing "exit" so long as text can be inputted, signified by ">".

2 Design

The game primarily uses multiple functions which are called according to the user input. The functions are called in the last part of the code for cleaner code and for the user to easily call them via input prompts. Messages are printed as strings and most variables are global to be used in other functions.

This section of the report will discuss how the game was designed and what the functions do.

2.1 startup_rules() and rules()

A set of rules are printed when the game is launched, and shows a series of short messages which inform the players on how to play the game. The messages are individually printed using strings and are called at the beginning of the game, and when the user input is "rules" respectively.

2.2 print_grid()

The board was created using a 2d list called "b_grid", which holds the grid arranged in the standard Checkers/English Draughts layout and can be called in different functions when required.

print_grid() allows b_grid to take form as an 8x8 board using different ASCII characters printed out as strings and the function itself could be called to easily print out the board. The function used a while loop which incremented the integer variables "i" and "j", which also made the code cleaner to read.

2.3 ai_p2()

Players can choose to play against another player or against an AI that makes automated choices by typing either "1" or "2". The choice that the player makes is stored in a variable input called "game_mode".

A Boolean called "against_ai" is set to true if the player chooses to play against an AI, whereas if the player chooses to play against another player, the Boolean "against_p2" is set to true. Both variables can be used in other functions, and one variable is set to false, depending which one is set to true.

If the players types "rules", the game calls rules() and then loops back to ask the player to input their choice again until it can move on. If the player does not input a valid choice, the function is called again until a valid choice is entered.

2.4 update_player()

update_players() uses a integer variable called move_turn, which increments after any move is made. When the variable modulo two is equal to zero, then the current player is set to player two ("player_2/player_2K"), otherwise it is set to player one ("player_1/player_1K").The current player determines who can move piece during the current turn, as well as what pieces can be moved.

2.5 get_input()

The players select pieces by typing the co-ordinates on the grid in the format (x,y) via input prompt.

The user is prompted to type in the co-ordinates of the piece they desired to move in the format (x,y). If the user failed to provide a valid answer, the function's while loop would print an error message and ask the user to re-input a choice until the answer is either valid or "exit", which would terminate the game.

When the user's input is valid then the program exits the loop and splits the variable "move_from" where there is a comma. The split variable becomes two separate variables called "split_fromX" and "split_fromY", which contain the X and Y co-ordinates and are used in the next function. If the

user selected to play against the AI, the user input would be automated in the ai_select() function.

2.6 get_dir()

After the co-ordinates have been validated in get_input(), the player is asked to enter the direction that they would like to move the piece.

The player is given a visual aid to determine which direction to move the selected piece, and the look of the aid is determined by the current player. The players input a number between one and four (depending which direction they would like to move) and the input is stored into a variable called "move_from". If the input is not a number between one and four, an error message is shown and the function is called again.

The AI moves in the first direction where there is a blank space next to the selected piece, and prints a string which states the direction it chose.

2.7 make_move()

The moves are made when a user types "move", which calls the function make_move(). The function calls get_input() and get_input() within itself to allow the user to move their desired piece in the direction they would like.

The moves are made using the X and Y co-ordinates from get_input() and moved in the direction obtained in get_dir(). Pieces are moved either (-1, -1), (-1, +1), (+1, -1) or (+1, +1) from the original location depending on the input of the current player.

The function uses many nested if statements to check for conditions required to move pieces. If any normal pieces reaches the end of the opponents side of the grid, the piece is converted to a King piece. Pieces which are moved must be within the range of the grid and concur with the current player. If the piece is outwith the grid, an error message is shown

Originally, the functions get_input() and get_dir() were part of make_move(), which made the code difficult to loop back in the case of incorrect input. The functions were eventually split apart to create smaller, easier to handle chunks which helped to keep the code cleaner and more readable.

make_move() does not allow the players to take pieces as that is handled by the function mandatory_move(). The function only handles moving pieces to spaces where there are no pieces in the desired direction.

2.8 ai_select()

The function searches through the elements of b_grid for any AI pieces and appends them to a list called "ai_list". The function then uses a variable called "ai_selected" which selects a random item from ai_list and determines what the AI's choice would be. The loop is exited when there are no more AI piece found in the grid and make_move() is called along with update_player() after.

The lists "undo_grid" and "redo_grid" are set and copied before or after the move is made to allow the user to undo/redo their move.

2.9 mandatory_take()

mandatory_take() is called before the player can make a move and checks each element of the grid for pieces that can take another piece.

When a valid piece is found, one (or more) of four Boolean variables are set to true per iteration. The player is prompted to select a piece to take with the direction given. The For loop may find more than one valid piece, and the player can press enter to cycle through the multiple choices before selecting an option.

When the choice has been made and the pieces have moved, the next function is called to check for additional jumps that can be taken.

If the player has not chosen an option before the list ends, the function is called again until a choice is made. In addition, the user cannot exit from the loop until a choice has been made.

2.10 additional_takes()

The function allows player pieces to take any additional jumps after the initial jump was made in make_move. Structurally, additional_takes is very similar to the previous function and uses passed in variables, "temp_Y" and "temp_X".

When additional jumps are taken, a message will print to inform the user of what happened. If the piece can take more additional jumps after the second one, the function is called again after the pieces have been moved.

2.11 undo() and redo()

undo()/redo() allows players to undo or redo up to one turn and can only be used before or after a move has been made.

The functions uses undo_grid and redo_grid to save the state of b_grid at certain moments. undo_grid saves the state of the grid before a move is made, to allow the player to revert back to a state before the pieces were moved. redo_grid saves the state of the grid after the grid has been moved, to allow the player to move the pieces to the new positions.

The variable move_turn decreases by one if a move is undone, which also changes the current player at that turn. Similarly, if a move is redone then move_turn is incremented by one.

The functions cannot be used if the state of b_grid is equal to either undo_grid or redo_grid. Also, redo() cannot be used if the player has not previously called undo().

2.12 results()

The function loops through each element of the entire grid and searches for any playable pieces.

If any player one pieces are found, then a Boolean variable called "p1_in_grid" is set to true, otherwise it remains as false. Likewise, if any player two pieces are found then another Boolean variable called "p2_in_grid" is set to true, else it is remains as false.

If p1_in_grid is true and p2_in_grid is false, then a string is printed which states that player one has won, and vice versa for if p2_in_grid is true and p1_in_grid is false.

If both variables are set to true the the game continues and the function is called before players can take another action until one of the variables remains as false.

3 Enhancements

The game works and has many functioning features implemented to provide an enjoyable game. However, improvements and additional features could have been added with a longer time frame.

This section of the report will discuss the features that could have been added or improved.

The undo/redo features could be improved had they been able to undo/redo more than one action at a time. Two lists could be created to store b_grid's state before and after a move is made, which would mean that the grids would not be the same.

The AI could be improved if the user could choose the difficulty played against by selecting it when ai_p2() is called. The easier difficulty would be able to move and take valid pieces and provide a opponent to play against. The harder difficulty would be able to strategically move and take pieces with the intention to win, thus providing more skilled players with a challenge. Furthermore, the ai_p2() could have allowed players to watch a game between two AIs.

A history feature would have been a great addition to the game, allowing players to look back at moves made from the beginning to the current moment. Roughly, this could have been implemented by creating a list which stores b_grid's state after a move has been made. When the players inputs "history", a function would be called which prints the contents on the list, one by one. A replay feature could have also been added to allow players to watch back finished games in full.

Another feature that could be useful would be a cancel feature, to allow players to exit a loop if they wished to change their choice during a move, either to undo, check the board or rules. This could be done if the game was put into another function and called when the user input is "cancel".

4 Critical Evaluation

This section of the report will discuss what features of the game work well or work poorly and why.

The board and messages print in a well structured manner for players to easily understand and read. However, the program does not import "time", and does not use a sleep/timer function causing the strings print out instantly, which makes it difficult for the players to keep track regardless of the formatting.

The undo/redo features can be considered lacking, only able to undo/redo up to one move from the moment the player desires. The features would be more ideal if they were able to undo/redo more than once, as stated in the previous section.

The AI works well as a basic opponent, and can make choices without any user input. Nonetheless, the AI is considered weak as it cannot make strategic choices with an aim to win.

The ideas for allowing players to move pieces developed as the game was written. The original method was to ask players to enter two sets of co-ordinates; one for the initial location of the piece and the second for where the player would like to move the piece. The second and third idea was to number each piece, such as "r1", "r2", "b1" and "b2", and ask for the co-ordinates to move to or for the number in the direction to move to. The final and implemented idea was to ask the user for the co-ordinates and then ask for the direction to move in, which is rather foolproof compared the the other ideas.

5 Personal Evaluation

This section of the report will reflect on what was learned during the creation of the game and the challenges faced. In addition, it will also discuss how these challenges were overcome, as well as the performance during it.

The coursework presented a challenge as the game was written in a language which had not been used by the programmer. In addition, Checkers/English Draughts had not been played prior to the task received so the rules were difficult to adhere to, and the tests were difficult to perform due to lack a strategy. Through practice with the language and repeated tests, the coursework became easier to program.

Overall, the game functions well and has implemented multiple features stated on the requirement specification.