Yuhao Li: bxzr32
26 January 2021

System Programming Coursework

Reflective Report

1. Data structure

The board pointer is initialised in setup_board(). In this function, only the member winners is allocated with a piece of memory space, other members are not allocated with memory spaces nor assigned to initial values until the first time they are used. The detail structure of board_structure is designed as follow:

- **char ** data:** the 2-D array that stores the actual board token, including '.', 'x' and 'o', as a grid.
- **int row:** the row length of the grid determined in read_in_file() exclude the newline at the end of the file.
- **int col:** the column length of the grid determined in read_in_file() without counting '\n'.
- **int x_count, o_count, empty_cells:** the number of 'x', 'o' and '.' tokens from the input file, respectively. These values will be updated whenever the *board is changed.
- **struct winner * winners:** the **struct winner** array. The array can store at most 2 winner struct which indicates the connect4 game can have a maximum of 2 winners.
  - The **struct winner** is used to retrieve and to capitalise the winner path:
    - **char player:** the token of the winner.
    - **int direction:** the direction of the 4 consecutive tokens. 0 - row_up_down; 1 - col_left_right; 2 - main diagonal; 3 - minor diagonal; (same as matrix diagonal definitions).
    - **int row, col:** the position of the start token of the 4 consecutive tokens.
- **int winners_len:** the actual length of winners array.
- **char winner_state:** 'x'or 'o' indicates the corresponding player to win. 'd' indicates a draw. '.' indicates no player has won the game yet. The value will be updated when current_winner() is called.

2. Robustness

To describe the robustness of my implementation, a precondition is made which assumes the implementation is correct. The robustness indicates if my implementation can detect and possibly fix an unexpected input, for example, the input file stream and the input moves, from users.

In this game, the characters are classified into three types:

1. Illegal characters: any character that is not '.', 'x', 'o', '\n' and EOF. When occurs, the program will execute quit procedure.
2. Legal but ignored characters: '\n' and EOF. These tokens are used to determine several heuristic information such as total row number and the termination of a file stream, and will not be assigned to the grid.
3. Legal game tokens: '.', 'x' and 'o'. These tokens will be assigned to the grid and the total number of each of these tokens will be counted and stored in *board.

In this section, errors are classified into two types: fixable errors and fatal errors.

2.1 Fixable errors:

Errors that can be handled (**detected and fixed**)

- Invalid column and row inputs: In read_in_move() if the input column number or row number that is either a non numeric string or is a numeric string but out of bound. The program will let this player retry until it gives a legal move. During this period, no error is thrown and the grid remains unchanged.

The retry feature is achieved as follow:

1. Every time the play_move() function is called, it calls is_valid_move() to check is the given move is a valid move.
2. If the move is not a valid move, the move will not be performed and hence the board will not be updated.
3. The next_player() function infers the next player by the total numbers of 'x' tokens and 'o' tokens. As the board is not updated, the next_player() returns the same player as it returns last time which appears as a retry.

2.2 Fatal errors

Errors that can be detected but not fixable.

A quit procedure will be executed when a fatal error is occurred while the program has no ability to fix it:

1. Print a corresponding error message to stderr.
2. Exit the program with a non-zero exit code.

Fatal errors with their exit codes are defined as follow:

1. **Memory allocation failed (exit code 1):** the program will execute the quit procedure at anytime it fails to allocate a piece of memory space for a variable. This test is achieved by testing if malloc() returns a NULL pointer.
2. **Invalid input size (exit code 2):** the read_in_file() function will read the whole input file once and to determine the row length and column length needed for 2-D array memory space allocation. If the lengths are out of bounds, for instance, the column length is greater than 512, the program will execute the quit procedure.
3. **Invalid character (exit code 3):** After allocating the grid with a correct size of memory space, the read_in_file() function will assign characters to the correct position of the grid. If an illegal character is occurred from the file stream, the program will execute the quit procedure. (See term illegal character at the begin of this section)
4. **Token inconsistent (exit code 4):** the play_move() function will test if the sum of the memorised tokens (empty_cells + x_count + o_count) is equal to the total token number (row * col). If not, the program will execute the quit procedure. Indeed, if it is assumed that the implementation is correct, the number of tokens should be consistent at anytime. However, there is a possibility that some other softwares could modify the memory through some hack technologies. Therefore, this test is to verify the memory of the grid is not modified by the outer programs.
5. **Invalid file stream (exit code 5):** in both read_in_file() and write_out_file() functions, if the given file stream is null, the program will execute the quit procedure.

3. Memory efficiency

Several methods has been implemented to make the code more memory efficient.

1. Allocating the grid (the 2-D array) with an initial row and column (large) sizes might bring to a waste of memory space. In my implementation, the read_in_file() function will read the whole file once to determine the essential row length and column length needed. The file stream is then rewinded to the start of the file and will be read again for the token assignments.
2. Ideas from insertion sort has been absorbed to achieve the algorithm of 'fall by gravity'. This algorithm does not need to create a new 2-D array nor create a temporary column array. The idea is, for each column, it traverses each token from the bottom to the top of the column, and for each token, if it is an empty token, it keeps swapping this empty token with the token upon on it until a non-empty token is occurred.

4. Further Improvements

There are several improvements can be made to make the main.c and connect4.h more robust.

1. In main.c, it is suggested that null-check both infile and outfile before supply them to the functions.
2. In connect4.c, the name 'board_structure' can be replace by a macro definition, e.g., #define BOARD_STRUCT board_structure. And in connect4.h, #ifdef - #endif can be used to enclose the whole function definitions and struct definitions.