Tic-Tac-Toe Coursework Report

Piotr Handkowski
09005378@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1 Introduction

The aim of this coursework was to implement a Tic-Tac-Toe console game in C language, using appropriate data structures and algorithms. The basic requirement was to develop a playable game with a representations for the board, players, pieces and positions. Additional requirements included an implementation of undo and redo feature as well as recording games in game history with the ability to replay them. With the basic features completed, further extensions to the game were allowed. To make the game more interesting a computer player has been designed to allow a single user to enjoy the game. Finally, a modification of the game was implemented, called Tic-Toc-Boom, where the player can select between 1 and 3 bombs that are randomly placed on the board. When a position with a bomb is selected, the bomb explodes and the player loses his/her turn. This extension makes the game more fun and allows for more strategies when compared with the classic game. Below you can see a logo of the game, displayed when the game is launched, that was created using Text to ASCII Art Generator [1].

Figure 1: **Tic-Tac-Toe** - Geme's Logo

Figure 2: **Tic-Tac-Toe** - Board with Positions

2 Design

The design process started with the choice of a data structure for the board. There were two candidates for this purpose. The first viable option was an array of arrays [3][3], with the second one being a one-dimensional array [9]. For a simple game with 9 positions, it seemed unnecessary to use the first approach, therefore the board has been implemented as a char array[9], with each cell representing a position on the board. When created, the board is initialised with a space character to represent an empty position, which is then replaced during a game with 'X' or 'O'. When the game is started a board with numbers is displayed, as can be seen in Figure 2, to familiarise the users with position numbers. When the first piece is put on the board the numbers disappear, which makes it clearer which positions are free.

When designing the game history two data structures were taken into consideration: an array and a linked list. The first structure seemed like a perfect solution with easy implementation and straightforward addition of new moves, as the last index is stored in the data structure. However, with a redo and undo functionality, it would be difficult to guess the size of the array, as in theory the players could undo and redo infinitely many moves. Therefore, the feature has been implemented using a linked list, which allows for a dynamic size increase each time a new move is added. The main disadvantage is that the structure is more complex. The moves added to the linked list were implemented as a struct called 'move', as it was necessary to keep the details of the position selected, the player that selected it and whether the position contained a bomb if the game played was played is Tic-Toc-Boom. To be able to store a full game history, more data about the game had to be kept. The players' names and the information about who won the game, therefore a 'game' struct was created, containing all those details and a list of moves. With the structure, a game could be saved and replayed at any time, however, a container for all saved games was also needed. As the user decides whether to save a game or not, it was not possible to use a fixed size array to store the games, hence a linked list was used for this purpose as well.

The next feature on the design list was the undo and redo functionality. It was initially designed as two stacks (mainStack and undoneMoves) implemented as linked lists joined

together in one structure, as it was thought that the number of undos and redos was not limited. During the testing phase of the structure it became apparent that the number of items in each of the stacks would not exceed 8. For this reason, the stacks have been redesigned to use arrays of size 8. This removed the unnecessary complexity keeping all the required functionality. Each time a move is made it is added to the mainStack, when a move is undone it is popped off this stack and pushed to the undoneMoves stack. Right after any move is undone it can be redone which pops it off the undoneMoves and pushes it to the mainStack. When a new move is made the undoneMoves stack is cleared and the user cannot redo any moves that were undone previously. Additionally, with every undo and redo a new move is added to the game history allowing them to be replayed along with the normal moves.

One of the most exciting parts of the design was the automated player. The algorithm that generates a move was designed to consist of 5 steps that are run in sequence until a good move is found. The first step checks if the board is empty, if this is the case any move is good, therefore a position is chosen at random. It could be argued that the best position is in the middle of the board, but it would be quite boring if the automated player made the same first move each time. The second step is the finisher. The algorithm looks for any combination of pieces on the board that can be finished to win the game. If the game cannot be won at this stage the algorithm performs the third step which is to block the opponent from winning if he/she has the correct configuration on the board. The fourth step is to find a free space next to the piece that is already on the board rather than placing it in a random position that does not contribute to winning the game. The last step is mainly designed for the Tic-Toc-Boom version where in an extreme case the bombs and the opponent can make your only piece to be cut off the rest of the empty tiles. The move is to simply select another position at random from the empty options. The algorithm makes the automated player quite a good opponent, especially for the Tic-Toc-Boom version where an element of luck is added to the equation.

The last extension that was designed, which was already mentioned a few times, is the Tic-Toc-Boom version of the game. When this game type is selected the user is asked to select the number of bombs (1-3), then the selected number of positions on the board are randomly selected and put in a separate array of size 3. Whenever a new move is made the array is checked to see if it contains the selected position. If it does then the bomb explodes and the player looses his/her turn. A simple graphic is displayed that was also created with Text to ASCII Generator [1].

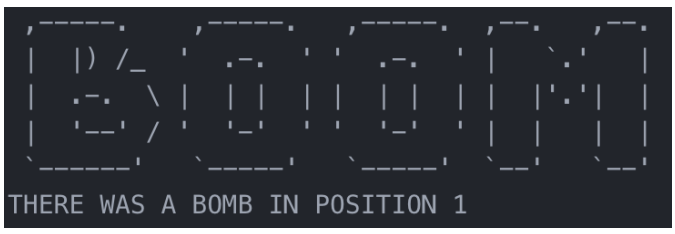

Figure 3: **Tic-Tac-Toe** - Bomb Exposion

The position then becomes safe and can be taken by the other user as part of his/her strategy.

After completing all the basic elements, they were ready to be put together to create a playable game. To allow the user to navigate through the game a menu was created asking the user to select one of the options, which can be seen in the below figure.

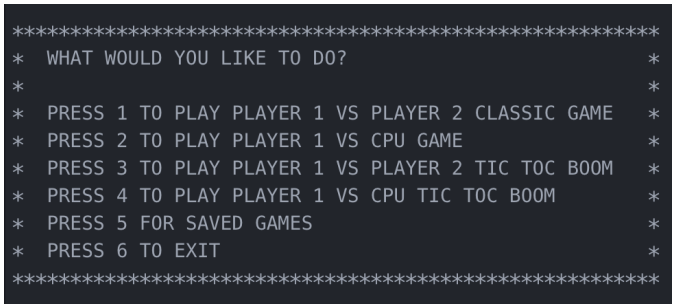

Figure 4: **Tic-Tac-Toe** - Menu

Once the type of the game is selected, the players are asked to enter their names. If player 1 plays against the automated player the name of the 2nd player will be Computer. In each game, the user that makes the first move is selected at random to avoid a situation where the same user starts if several games are played in a row. It is easy to track who needs to make a move at a particular time as a message is displayed with the user's name asking to select a position, which can be seen in figure 5. When playing a classic 2 players game the users can undo and redo their moves by pressing 10 and 11 respectively. The game can be undone right back to the initial state, while only the recently undone moves can be redone.

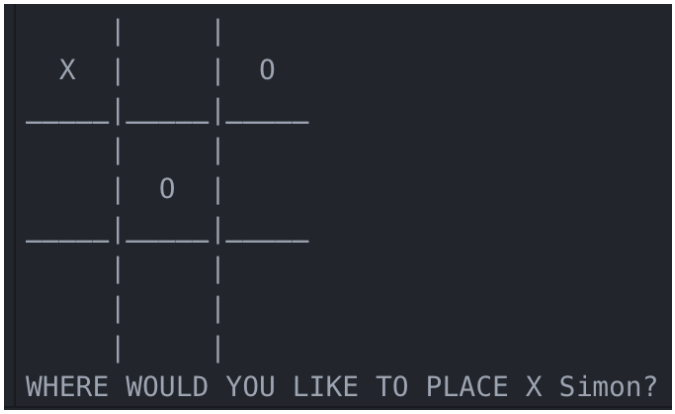

Figure 5: **Tic-Tac-Toe** - Useful Messages

When playing against the computer a 2-second pause was added to make it look like the computer is thinking what move to make next.

After finishing a single game the players are asked first whether they want to save the game and then if they want to play this type of game again. Additionally, the number of matches won by each user in this particular game type is displayed.

A list of all saved games can be accessed by selecting option 5 from the main menu. A numbered list will appear showing

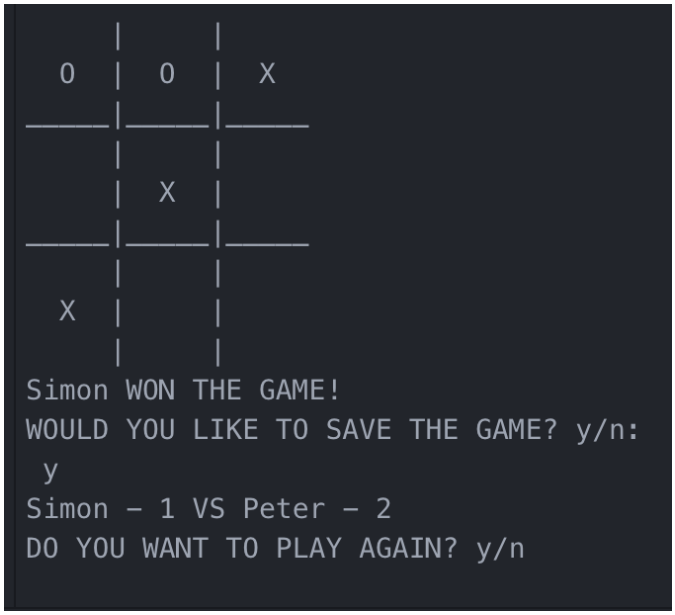

Figure 6: **Tic-Tac-Toe** - Useful Messages 2

the players names with asterisks characters around the name of the winner.

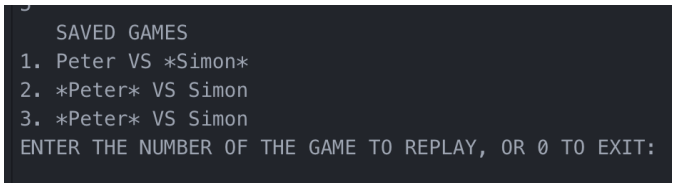

Figure 7: **Tic-Tac-Toe** - Saved Games

The game can be replayed by selecting the number of the game. Each move is then automatically replayed with 2-second pause between the moves. Additional information is displayed to inform who made the move and who won the game. An example of such a message can be seen in figure 8.

3 Enhancements

With more time available, a number of additional features and enhancements would be added. First of all, different board sizes would be implemented to make the game more exciting and keep the user entertained for longer. For this extension an array of arrays would be considered for the board implementation. Additionally, the persistence would be added to the saved games. Currently, the saved games are only available to be replayed during the same run of the application. With more time available, every time the game is added to the saved games list, it would also be saved in a csv file and when the game is started the saved games would be populated from that file.

When it comes to improvements, with an increased board size the automatic move generation would become too complicated with the current approach. Currently, with a 3X3

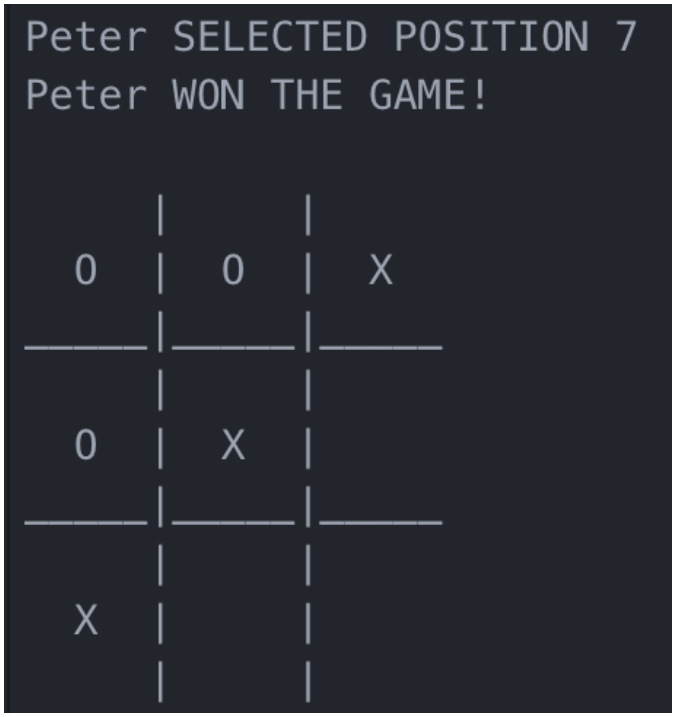

Figure 8: **Tic-Tac-Toe** - Automatic Game Replay

board size the number of 'good' moves is quite small, therefore it is easy to find one. It would be much harder to implement this kind of algorithms for a 5x5 board, where the number of combinations is significantly higher. Therefore, a good implementation of a min-max algorithm along with some pruning would be a good solution to this problem. Moreover, bigger boards would require a longer sequence of pieces to win the game, which would have to be taken into account by the algorithm as well.

Another small improvement would be made to the bomb feature. Currently, the positions of where the bombs were placed are stored in a separate array. This could be changed so that the bombs are placed directly on the board by changing the value of the position to 'b' to represent a bomb. The amendment would not make any changes to the user but would make the code implementation better.

4 Critical Evaluation

The features that are working very well and which I think have been well implemented are the undo/redo features. They are stacks which means that any of the undone/redone moves will be popped and pushed from the top of the stacks to which an index number is kept. This means that the operations will be made in O(n) time complexity. The next feature that meets the requirements of the users is the game history. As this is a flexible structure the players can make a large number of moves, using redo/undo feature, and all of them will be added to the history. The only downfall of the algorithm that adds a new move is that it needs to traverse through all already added moves to add a new one at the end. It can be seen in the below table and diagram that the addition time is quite small and constant up to 200 additions, but then grows

exponentially beyond this point.

Number of additions	10	50	100	200	500	1000
Average time 1	0.000036	0.00003	0.000056	0.000244	0.000529	0.002025
Average time 2	0.000018	0.00005	0.000117	0.000122	0.000741	0.002175
Average time 3	0.000027	0.000037	0.000065	0.000143	0.001152	0.002514
Average time 4	0.000039	0.000038	0.000152	0.000283	0.00089	0.002652
Average time 5	0.000043	0.000043	0.000063	0.000175	0.000772	0.002538
Average time 6	0.000027	0.000032	0.000098	0.000161	0.001049	0.002473
Total Average per addition	**0.000032**	**0.000038**	**0.000092**	**0.000118**	**0.0008555**	**0.002396**

Figure 9: **Tic-Tac-Toe** - Addition Times

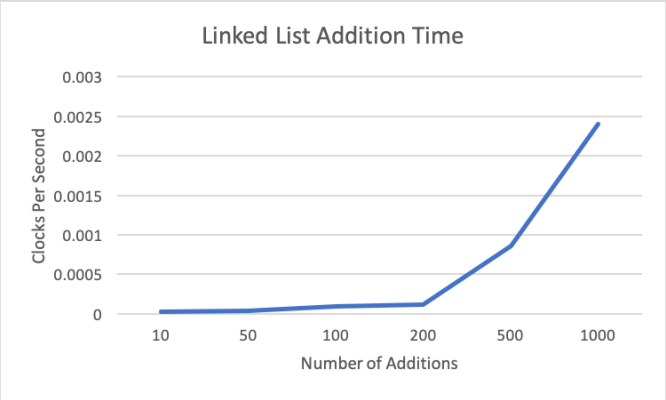

Figure 10: **Tic-Tac-Toe** - Addition Times Graph

For this particular 3x3 game this is not an issue as the number of moves, including undos/redos, will be too small to make any noticeable changes to performance. However, the problem could be solved by keeping a reference to the last item in the list, which would reduce the time complexity to O(1). This could be implemented as an enhancement to the game.

The next feature which on one hand works well and could be improved on the other is the automated player. The function always produces a good move but the approach that is used could be described by some as brute force. This is not entirely true as once a good move is found the function returns the move and does not go through the rest of the possible moves. Once there are 3 pieces on the board the function rarely goes beyond the 3rd step as it is mainly trying to block the opponent, or win the game by putting the last piece. Therefore, for the 3x3 board, the algorithm performs well. It would be difficult, however, to adapt it to bigger board sizes.

5 Personal Evaluation

This project was quite a challenge, as it required the application to be developed in C language. Only having experience in coding in languages like Java and C# the new concept of pointers, or rather a pointer to pointer, was quite a challenge to overcome. I have tried to find additional information on the subject, but most of the information available was just on simple pointers. With a lot of trials and fails, I finally managed to grasp the concept and now feel quite comfortable using it. Another concept that was new to me was memory allocation. This was not as difficult to understand as the pointers, but it did cause some problems when not used properly in the program. The new skill, learned with C language, help me better understand how the memory is managed behind the curtains, which was not possible with the languages that I had learned before. The next challenge that came from the use of the C language was the error messages given by the compiler. As the application was developed in a text file it was not possible to step through the code and check the values of variables as the code executes. To overcome the problem I learned to compile the program frequently to see if the newly added piece of code generates any errors, which helps with locating the bugs. I am quite happy with the new skills that the project allowed me to practice, as they will be helpful in learning the next language that I want to learn, which is C+. On the top of the skills related with the language, I got to practice the data structures that I learned in the first part of the module, especially arrays, linked lists and stacks. Overall the project was a positive experience and I am happy with the finished result of my hard work. The game that I have created is fully functional, including the additional requirements and I feel like the Tic-Toc-Boom extension is more fun then initially expected.

6 Conclusion

References

[1] Tekst to ASCII Art Generator, http://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20. Last accessed 25 March 2019