tic-tac-toe - Coursework Report

Alex McGill
40276245@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures

1 Introduction

The aim of this coursework was to create a text based tic-tac-toe game in C to demonstrate an understanding of the data structures and algorithms learnt throughout the module. The minimum requirement was a working player versus player tic-tac-toe game but this was expanded upon a lot to improve the experience with a variety of features. Some of the key features implemented include main menu options, persistent match history, the ability to replay back previous matches automatically, score tracking, overall player statistics such as win and loss percentages, undoing and redoing player moves, changing the size of the board, player versus player, player versus computer, computer versus computer, and a visually appealing board and game pieces.

The game takes advantage of data structures such as arrays, 2D arrays, and stacks to store information such as the state of the board, tracking moves and undone moves, and match results.

2 Design

The first thing that the user sees when they launch the game is the main menu which provides various options for different game modes and features as shown in Figure 1.

```
Tic Tac Toe by Alex McGill

MAIN MENU:
[1] Player vs Player
[2] Player vs Computer
[3] Computer vs Computer
[4] Change Board Size
[5] Match History
[6] Replay Match
[7] Exit

Select an option (1-7): _
```

Figure 1: **Main Menu**

For the game to work properly, the positions that each player chooses need to be stored in a data structure. There are quite a few approaches to this problem, all of which have their advantages and disadvantages. The first thing that needs to be considered is the data type of a position on the board. The players will see an 'X', a ' ' or an 'O' on the board so it would make sense to use an array of characters, where each position on the board is represented by the value at the corresponding position in the array. However, if the board contains characters it can make checking for a win more complex as it would require some kind of pattern matching. On small boards such as a 3x3 this may not be an issue but as the board scales up there are many more patterns to identify. An alternative approach is to use integers to represent the values on the board. For example, 'X' = 1, ' ' = 0 and 'O' = -1. This helps simplify checking the status of a win as groups of values can be summed and compared against the required total for a win.

The second thing that needs to be considered is the data structure that value of each position will be stored in. Although a one dimensional array seems less complicated, due to the layout of the board being two dimensional, I found that using a 2D array for the board works much better. Due to the nature of a 2D array, positions are naturally split into rows and columns, making it easy to iterate over specific rows or columns. Values are accessed using board[row][column], for example - board[0][0] accessing the value of the first position on a 3x3 and board[2][2] accessing the last. As shown in Figure 2, users still perceive these positions as integers so they are converted into a row and column to retrieve the value on the board that the position is referencing.

```
Tic Tac Toe (3x3) - 3 in a row to win
Commands: 'undo', 'redo'
SCORE: (X) 0 - 2 (O)
o----------o----------o----------o
| db    db |          |          |
| `8b  d8' |          |          |
|  `8bd8'  |          |          |
|  .dPYb.  |    02    |    03    |
| .8P  Y8. |          |          |
| YP    YP |          |          |
o----------o----------o----------o
|          | db    db |          |
|          | `8b  d8' |          |
|          |  `8bd8'  |          |
|    04    |  .dPYb.  |    06    |
|          | .8P  Y8. |          |
|          | YP    YP |          |
o----------o----------o----------o
|  .d88b.  | db    db |  .d88b.  |
| .8P  Y8. | `8b  d8' | .8P  Y8. |
| 88    88 |  `8bd8'  | 88    88 |
| 88    88 |  .dPYb.  | 88    88 |
| .8P  Y8. | .8P  Y8. | .8P  Y8. |
|  `Y88P'  | YP    YP |  `Y88P'  |
o----------o----------o----------o
Computer (0) is thinking..
```

Figure 2: **3x3 Board** - Player vs Computer

I originally implemented the board using a constant size 2D integer array as it was a quick way to get started working on the project. This setup would be fine if there was no option to change the size of the board but that was functionality that I definitely wanted to add so I ended up switching to

using a dynamic 2D integer array (a pointer to a pointer). This involved allocating memory for the entire board based on the user specified board size and then looping through each row in the board and allocating enough memory for the values in the row. Once the memory for the current row was allocated, each column in the row was initialised with a 0 to specify that it was an empty position on the board.

Once the ability to change the board size was added I realised while testing out other sizes that certain game configurations were quite flawed. For example, if the board is 4x4 and the win requirement is three in a row then player one can always win straight away but the win requirement was 4 on a 4x4 than they might as well be playing on a 3x3 because it is the exact same experience. After doing some testing I found that the best design choice was to limit the available board sizes to 3, 5, and 7. The win requirement for a 3x3 is three in a row and for 5x5 and 7x7 the win requirement is four in a row. This often leads to interesting matches, especially on the 7x7 where there are so many opportunities to win.

The game was designed to give players the opportunity to undo and redo moves within a game if they want to. I decided that a stack was the best data structure to store this information because of its 'last in first out' nature. An alternative data structure that could have been used was an array but as C has no array length functionality built in it would involve using another variable to keep track of the number of moves in each in order to access the last value or looping through them all until the last value is reached. With this in consideration I decided to implement two stacks, a move stack and a redo stack. When a move is made on the board it is added to the move stack. If the player decides to undo the move then it is popped from the top of the move stack and pushed to the redo stack. The redo stack is what allows players to redo moves that have been undone. If the player decides to redo the move it gets popped from the redo stack and pushed back onto the move stack. If the player then makes a different move it is added to the move stack as expected but the redo stack is reset. This is to ensure there are no conflicts between the current state of the game and the previous state. When playing against the computer both the undo and redo commands are called twice. This is to ensure that the player cannot keep altering the position of the computer.

There are many ways of checking if a player has won the game but as the board size increases it becomes clear that certain methods are much more efficient than others. The first method I implemented was an exhaustive search, where the game would loop through every row and column looking for a horizontal win, then a vertical win, then a diagonal win. It grouped positions together by the amount that was required to win and checked the sum of the values to see if they met the requirement. If so then it would return the winner, otherwise it checked the next group of values until either a winner was found or there were no groups left to check.

This exhaustive search was very inefficient and I felt there was a lot of room for improvement which is why I decided to come up with my own 'selective' search (unrelated to an object recognition selective search algorithm), which uses math and the latest position to select which groups of values are worth checking. This was based on the assumption that the game only needs to check if the current position has resulted in a win. If another move in a previous position resulted in a win then at one point it was the current position which would've ended the game already.

By passing the latest position to check status function (which calls functions for checking horizontal, vertical and diagonal wins), the algorithm can focus on the specific row / column / diagonal that is relevant to the position. It is fairly simple for horizontal and vertical checking but is much some additional logic on diagonals. For example, for the vertical check it simply starts at the top of the row and checks if the first group of values (0-3) meets the required amount to win. If it does then it returns the winner otherwise it checks the next group of values (1-4). The horizontal check works the same way but horizontally instead of vertically.

The diagonal win check is split into two checks - the diagonals starting at the left and going down to the right and the diagonals starting at the right and going down to the left. For right diagonals the first step is to find the top or right edge from the latest move position. It does this by adding 1 to the current column and subtracting 1 from the current row in a loop until it finds an edge as shown in Figure 3.

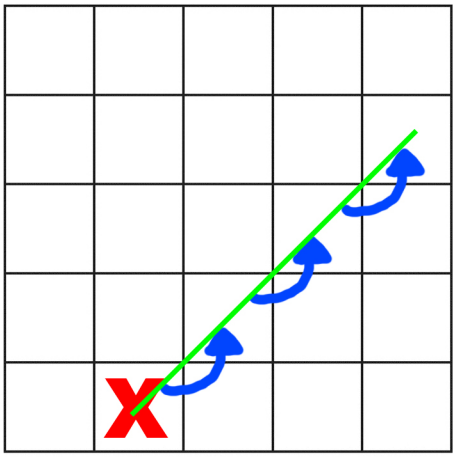


Figure 3: **Right Diagonal** - Finding the top / right edge

The row and column of the **new** location is then used to calculate how many times the loop for checking the diagonal of the location should iterate. The formula for this can be seen below where l(n) is the loop amount for the checking left diagonals and r(n) is the loop amount for checking right diagonals.

$$l(n) = required - column - row - 2 \quad (1)$$

$$r(n) = column - row - required + 2 \quad (2)$$

The loop amount values for left and right diagonals in each position on the board can be seen below on a 5x5 grid in Figure 4 to help visualise the formula output. The number represents how many times the algorithm should loop for that position. Any result less than 0 is set to 0 for clarity.

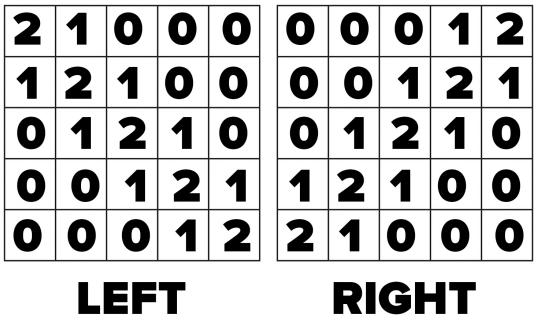


Figure 4: **Position Loop Amount** - Left and right diagonals

Using the loop amount for the new position allows the algorithm to loop through each group in that diagonal row the correct amount of times. For positions where it is impossible for a diagonal win (represented by a 0 on the grid in Figure 4), the loop count is less than 1 which means that no time is wasted attempting to check that position for the corresponding diagonal.

The result of a full check using this selective algorithm can be visualised in Figure 5 where it is clear which groups of values are searched for the marked position. As the marker is in a position where it is not possible for there to be any left diagonals, it is skipped over completely and only the other checks are present.

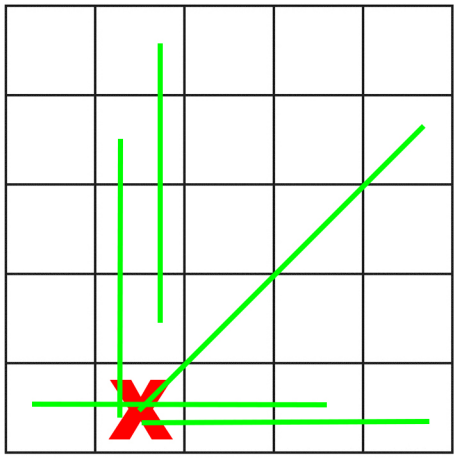


Figure 5: Win status check for current position

3 Enhancements

The majority of the time, especially on a 3x3 board, if both players take the time and actually think about which position to choose then the game will almost always result in a draw. To combat this, one addition to the game that I would make is having an option to set a time limit on making a move. For example, if players only had five seconds per move then there would be a lot more pressure to act quickly which would likely impact their decision making. If they fail to make a move within the time limit then the game would randomly assign them a position on the board.

Apart from variations in the size of the board, the game can become quite repetitive as it consists of the same basic moves every time. Different game modes such as both players having X's only and they score points by making your opponent place the winning move (you win by losing) would increase replayability and encourage new players who are bored of generic versions of tic-tac-toe to try the modified versions within the game.

One feature that would improve the usability of the game would be the ability to return to the main menu mid game. Currently, the user cannot return to the main menu until the game is finished unless they exit the game and reopen it. If a user started the wrong game mode by accident then they would have to play out the game before returning to their desired option.

Although the match history provides all the useful information from previous matches, it would be useful to have a filter to help narrow down the results. It wouldn't be that useful if there wasn't many games played but as the count increases, having the ability to only display games and statistics from a certain date or game mode would be definitely be beneficial. Currently the matches are sorted by oldest to newest but allowing the user to switch the ordering would make the game more customisable. Matches are saved to and loaded from a CSV file that is generated after the first match and is appended to after subsequent matches. The match history can be seen below in Figure 6.

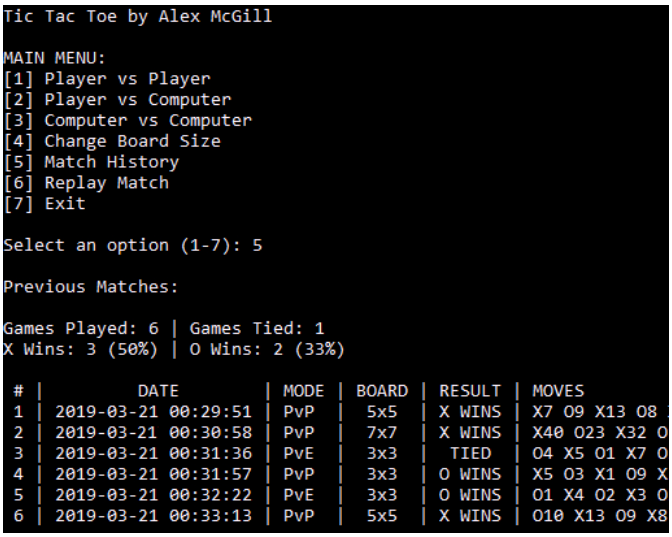


Figure 6: Match History

4 Critical Evaluation

There are a few small features in my implementation of tic-tac-toe which I think help make it stand out from other implementations. One of the optimisation's I made was to only check the status of the board for a win if there are enough moves on the board for a win to actually be possible. For example, there is no point in checking the board immediately after the first move considering even the smallest board requires three in a row where the minimum moves required equal five (three from the winner and two from the opponent).

I knew in theory that my win status check selective search algorithm would be quicker than an exhaustive search but it was hard to quantify what the time savings would actually be. I decided to run a benchmark for the algorithm at each board size first with an exhaustive search and then using the selective search. In order to test the time taken on every position individually on the board I disabled all of the check status function returns so that they would run all the way through and not stop if one of the positions happened matched the win criteria with other positions. As the time taken to complete the check is so small, I set each search to run 100,000 times within the benchmark to accentuate the time differences. Had the board size been bigger, there would be less need to run the search so many times but this was the best way to test on a limited size without interfering with the results. For each position the benchmark was run 20 times and then an average time was recorded into a CSV file. The results of these benchmarks are shown in Appendix A, B, and C.

As shown in the results, on the 3x3 board there is only a minor improvement in speed of the check. However, as the size of the board increases the time savings increase by a considerable amount. The charts also help visualise the algorithm as they correspond to the left and right diagonal loop amount values as shown earlier in Figure 4. In positions where the loop amount is slightly higher the runtime increases which explains why the line for the the selective search on each chart is symmetrical.

Despite the effectiveness of the selective search, there are still ways to improve it further. Currently the search checks the entire row / column / diagonal of the position in groups of the required amount which doesn't really make a difference on smaller boards but if the board was much larger then this would be an obvious drawback to an otherwise very effective algorithm. The solution to this would be to only search the groups of values within range of the required amount. For example, if the position is at the end of the a horizontal row on a board that has a win requirement of four then only check the sum of the last four values instead of checking each group of four in the entire row.

The first implementation of the computer player simply chose an available position on the board at random. Once this was working, the next logical step was to give it some intelligence to allow it to make informed decisions and have a good chance at winning the game. This was achieved by implementing the minimax algorithm[1]. By looking at the outcome of all possible moves on the board it scores them based on the final outcome and selects the best scoring move, aiming to minimise the opponents score and maximise its own score. To make the game slightly less predictable, the first move of the computer is still always random. Many implementations of the minimax algorithm for tic-tac-toe generate a new board each time the state is changed so that previous boards are not effected by any changes. This can often lead to issues with memory consumption so to overcome this I use the same exact board throughout but simply reset any changes that minimax makes once it is complete.

An essential part of minimax is checking the status of the board to see if there is a winner so naturally it is able to take advantage of the time benefits of using the selective search. However, when minimax doesn't use an exhaustive search to look at all positions on the board there is a small percentage of situations where it doesn't recognise the opponent is about to make a winning move. When I noticed this the first thing I did was switch it to use an exhaustive search but once it was working properly again I found that the game was actually more fun when there was a possibility to win. With an exhaustive search the best case scenario for the player is a draw. I realised that the best way to help ensure player retention is to let them win sometimes which is exactly what had accidentally implemented.

There was one problem with the minimax implementation which is that the runtime of the algorithm on boards larger than a 3x3 was too long. There was a few options available to resolve this issue, the first of which was a quick fix which involved setting all computer moves to random on boards larger than 3x3. This isn't ideal but it is better than not being able to play against the computer on larger sized boards at all. The actual solution would be to add alpha beta pruning to the minimax algorithm and set a relatively low maximum depth to quickly make a decision rather than searching all subtrees first.

5 Personal Evaluation

Throughout this entire coursework I have noticed a difference in my approach to solving problems. Typically for coursework projects like this if I am not entirely sure how to do something it doesn't take long before I search for solutions online, take inspiration from a few and adapt it to work for my context. However, in this coursework I found that I was coming up with my own solutions instead by writing and drawing out concepts on paper. It took quite a while to figure out all of the details of my selective search implementation but writing out the math on paper and using the results to identify patterns which would be used for loop iterators was very helpful. Although likely due to the nature of the board structure, I often found myself coming up with formulas to extract information from the board and use it to perform certain actions. Due to this I was able to overcome a lot of the challenges I faced with original solutions that I have yet to see anywhere else on the internet.

One of the challenges I faced was when redesigning the board to make it look nicer. Originally, the player markers on the board were just single characters on a line. This was fairly simple to implement and didn't take much critical thinking. When coming up with ideas for how I could make the board and game pieces look better, it became clear that the best way to do it was having larger characters that span across multiple lines. It doesn't seem like this would be challenging, but because lines are printed one at a time, the large characters need to be split into up into sections with each section being printed once per iteration. This can be seen below in Figure 7. The final implementation for this actually is quite intuitive, using a function which takes in the character ('X' or 'O') and the line number to retrieve and print that specific segment.

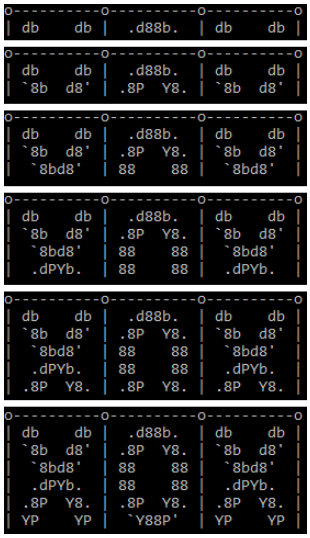


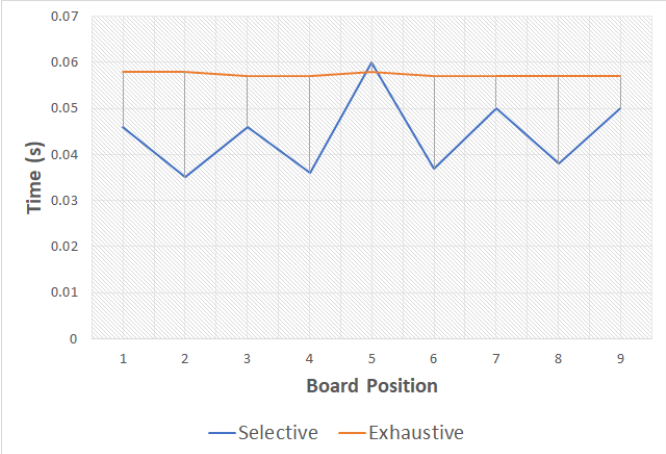
Figure 7: Game Pieces Printing

Originally the board position numbers were set to disappear after the first move so they were less distracting but it became clear that on larger boards they needed to be visible throughout the game as it was hard to tell which position was which. Redesigning the board resolved this issue as the game pieces are now much larger so the small position number doesn't attract any attention at all unless the player focuses on them specifically.

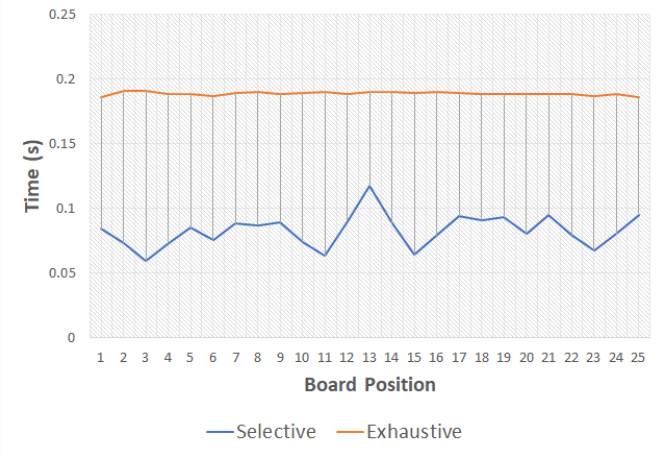
I think that I performed well on this coursework as I implemented all of the requirements and optional extras along with my own features but there is definitely still room for improvement. Currently the game is entirely written in a single C file consisting of over 1400 lines but breaking up key components into separate files would help improve the readability and maintainability of the code. Implementing alpha beta pruning on the computer player minimax algorithm would be one of the final tasks before releasing the game to make it feel complete and meet user expectations.

6 Appendices

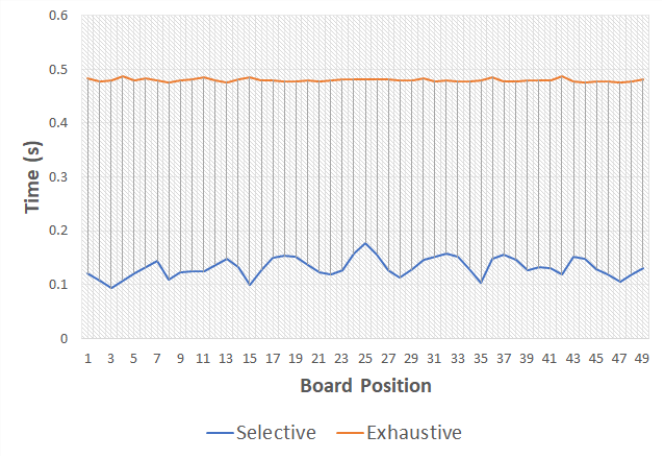
A 3x3 - Selective vs Exhaustive Search



B 5x5 - Selective vs Exhaustive Search



C 7x7 - Selective vs Exhaustive Search



References

[1] "Minimax." https://en.wikipedia.org/wiki/Minimax.