Algorithms and Data Structures:
Noughts and Crosses implementation and analysis

Ross Cockburn 40331161

Introduction:

The problem faced in this task was to implement the game of Noughts and Crosses using the C programming language with a particular focus on the data structures and algorithms used in the solution. While a traditional game board is 3x3, this implementation allows boards of any size specified by the user, along with a corresponding visual representation of the game being played out on any particular size of board. After a marker has been placed, it is possible for this move to be reversed using the “undo” feature. Moves can be undone right up to an empty board and it is also possible to “redo” any of the undone moves back up to the original state of the game.

When a game is finished, the score is updated and a new game may be started. Alternatively, the user may choose to replay a previously played game. This is possible because at the termination of each match, the move history for that particular match is stored in the current execution session. Once any of the previous games are chosen to be replayed, a visual representation of the board updates automatically move by move every 1 second.

This implementation also allows the user to challenge an automated opponent. While this feature exists for any sized game board, it is primarily implemented for the 3x3 setting. The automated opponent feature for board sizes of 3x3 and 2x2 is implemented using the Minimax algorithm and is therefore very hard to beat given the nature of the brute force algorithm. For board sizes 4x4 and above the automated opponent consists of a valid move selected at random from all possible valid moves. This is not very effective and was only introduced to give the automated opponent feature full coverage across all game board sizes.

Design:
Game Board - The game board was implemented using a 2 dimensional array of integers. The decision to choose a 2 dimensional array over a 1 dimensional array is arbitrary given that for an array of any dimension, it ultimately is just a contiguous block of memory, which can be thought of as a 1d array. [2]

Alternatives to representing the board as an array of integers would be to use an array of chars. Where an empty square is a space character and then “x” and “o” for the player markers. This method would have been slightly more efficient in terms of space required, a 3x3 board of integers requires 36 bytes of memory whereas a 3x3 board would only require 10 bytes of memory in a one dimensional array of chars (9 for each square and 1 for the null character). In a two dimensional array of chars the null character is not required and so the space for a 3x3 is only 9 bytes.
Another alternative would be to store the board as an associative data structure with key:value pairs. Each square of the board would be a key with the contents of that square being a value. The advantage of this structure is that the data does not need to be stored in a contiguous block of memory. This would be useful if the device running the program was highly fragmented and low in memory.

Information such as who is playing then next turn and the score of the move (used in minimax algorithm) was often required whenever the current board was also being accessed, so it made sense to create a state structure which contained the 2d array of integers representing the board, an integer representing the player who is placing then next marker and an integer representing the score of the board.

Storing the data in this structure was useful when implementing my move history data structure. It was decided that the best way to implement this feature was to use a doubly linked list. A disadvantage of this data structure is that accessing a particular element in the linked list requires traversal from the head of the list. However since the program only uses the doubly linked list to access the previous node or the next node and never skipping nodes in the sequence, this disadvantage is mitigated. The advantage of using this structure is that it scales very easily for any size of board as adding a node to the list is simple as memory for each node is allocated separately as required. The linked list structure contained a pointer to a state structure, a pointer to the next node in the list and a pointer to the previous node in the list.

An alternative might have been to use a stack data structure for this feature since we only access the elements from the top of the stack to the bottom when using undo. This however would not have been as effective when implementing the redo feature because “pop” removes the move from the structure and those moves would have needed to be stored in another structure and pushed back onto the stack if redo was ever used.

The user may replay a previous game, this means storing each completed game in memory. Since the replay was to be an automatic sequence of moves, it was necessary to store the

move history linked list. An array seemed an appropriate choice, this was because any previous game could be selected to be viewed so an access time of O(1) would be useful. Newly completed games would be added to the end of the array only and therefore there was no need to consider the drawbacks of having to insert an element in the middle of an array, a problem that makes arrays less efficient.[3] The array was an array of pointers to move history structures.

For the 3x3 board automated opponent feature, (and 2x2 however it is redundant as player 1 always wins this game mode) the minimax algorithm was implemented. This algorithm was chosen because, in its basic form, it is reasonably simple to implement yet very effective. This is at the cost of efficiency which is why it is not available for boards of size 3x3 and up. Nevertheless, it seemed appropriate given the time restrictions faced in this task.

The Minimax algorithm took a state structure and player as argument and returned a state structure. The board contained in the state structure would be checked for possible moves. The move would then be tried the board would be updated and the state passed into a new execution of the Minimax algorithm along with the next player. This recursive loop is called until a leaf node is reached at which point the score of the state is set depending on if the user one, the computer one or it was a draw. The scores of nodes in previous depths of the recursion tree are updated. Eventually an optimal moved is arrived at and the state is returned which contains the board that contains the move that was optimal. Fig.1 visualises this process.

Enhancements:
The problem with Minimax search is that the number of game states it has to examine is exponential in the depth of the tree. By using alpha beta pruning, the amount of nodes that need to be evaluated can be reduced significantly. Using alpha beta pruning on the Minimax search tree the algorithm would have $O(b^{3m/4})$. If some form of move ordering heuristic such as prioritising winning moves, or blocking moves then the time complexity could be reduced to $O(b^{m/2})$ [1]

Adding a depth cut off to the Minimax search would have also helped, for example only looking 4 moves ahead instead of all the way to the terminal moves would have drastically cut down the number of states evaluated. This would have been at the expense of the algorithm's guaranteed optimality. [1] Using this technique in combination with alpha beta pruning would allow the Minimax version of the automated opponent feature to be used in boards of size 4x4 and above. Although some form of limit on size would still need to be imposed as there is no avoiding the exponential nature of the algorithm.

Currently the history of games played and the ability to replay them exists only in the current execution environment. If the user ends the current session and relaunches the executable then the data is lost. Writing the array of move histories to a csv file and loading it on launch would solve this issue and keep a persistent history of games played

In this implementation, player 1 will always move first, this is problematic as player one has almost double the opportunities to win as player 2. [4] Randomly deciding who plays first or letting the loser of the previous game play first would be a much fairer system.

Critical Evaluation:
The priority of this implementation was efficiency of time, with efficiency of space being a secondary priority. It was important to think about the scalability, particularly when the decision was made to implement the option for larger board sizes. Generally this meant trying to keep runtimes of any commonly used function to O(1), O(log(n)) or O(n) where n is the number of squares on the board. On the whole this was achieved successfully with the obvious exception of the Minimax algorithm.

Accessing a particular square of the board takes O(1) time, however it is often required to iterate over each value in the array such as when checking for a win/loss/draw state. Using 1 loop for row and 1 loop for column of the array, the run time is $O(\sqrt{n}) * O(\sqrt{n}) = O(n)$ where n is the number of squares on the board, this is shown in Fig.2.

An area where there is perhaps room for improvement is the undo and redo moves. It might be expected that these operations are O(1) time since accessing the previous value of a list does not depend on any size of input to be successful. The problem is that there is a single game state that is referenced and updated throughout the match, after each move is played the game state is updated and a copy is made on a new entry to the move history list. For this reason when a move is undone, the state must be restored and this for this to work a deep copy of the board must be made. For this reason an undo or redo operation is O(n), where n is the number of squares on the board as shown in Fig.3.

The implemented Minimax algorithm did not include any performance increasing techniques such as alpha beta pruning or depth limited searching. This means that the runtime of Minimax is $O(b^0 + b^1 + ... + b^m) = O(b^m)$ where *b* is the number of legal moves at each point and *m* is the maximum depth of the tree [1]. During testing, the Minimax runtime for a response for 4x4 boards and above proved to be unacceptable (over an hour). This is because both the maximum depth (*m*) and the number of legal moves (*b*) increases exponentially as the board size increases.

For this reason a “dumb” automated opponent for 4x4 boards and above was implemented and has a much better time complexity. Whenever it is the automated opponent’s turn to move, random moves will be generated until a legal one is discovered, this means that the more developed the game board is the more iterations it will take on average to discover a legal move. The opponent plays every other move and so the probability of discovering a legal move reduces as the game progresses. So on an n by n board with a total of N squares we have the probability of a legal move being discovered defined by $(N-1)/N$, $(N-3)/N$, ... , $1/N$ where the numerator is the number of legal spaces remaining. It can be seen that the worst case of this method is when the last move of the game is being played which will take on average N iterations to discover a legal move, giving us a runtime of O(N).

Personal Evaluation:
I have learned a great deal from this exercise, having not done the Programming Fundamentals course, C as a programming language was very new to me and it requires a slightly different way of thinking when approaching problems. I enjoyed considering the underlying structures of the implementation and how they can be implemented, this is something that is easy to take for granted in languages such as Python, Java and C#.

I initially found the the concept of pointers challenging, in order to gain a strong understanding I often find that it is best to gain many sources perspective on the topic. After reading the recommend reading and various online resources on pointers, as well as completing the course's lab exercises I now feel confident when dealing with pointers.

It was important I understood this concept well as the first design decision I made was to make the board size dynamic. I decided to implement this from the start because I predicted that it would be complicated to switch from a fixed 3x3 board at the end, when all the other requirements had been completed. Despite this making the initial learning curve steeper I felt this was a good decision.

Another difficulty faced was with memory allocation and trying to ensure there was no memory leaks. I was conscious of incorrectly handling memory from the start of this exercise however, it is often easy to overlook freeing memory, especially when allocating in a different part of the program. With such a small program that would not be executed for long periods of time a memory leak would not be an issue, although I decided that they should be taken seriously just out of good practice. When deciding how to free data objects I found it useful to consider that for each malloc there should be a free.

Overall I felt I performed well in this exercise, providing a playable game with many features. I have been looking to implement the Minimax algorithm for some time now and i'm glad I have finally managed to apply it to something. I feel I have a confident base understanding of the important concepts in C and could use the language effectively for problems I face in the future.

References:
[1] Stuart Russell & Peter Norvig - Artificial Intelligence A Modern Approach, Third Edition, pages: 165 - 169
[2] https://link.springer.com/content/pdf/10.1007%2F978-1-4471-1023-1_1.pdf pages:7-10
[3] https://link.springer.com/content/pdf/10.1007%2F978-1-4612-0075-8_2.pdf page: 59
[4] Thomas Bolon - How to never lose at Tic-Tac-Toe
[5] https://users.sussex.ac.uk/~christ/crs/kr-ist/lec05a.html

Appendices:

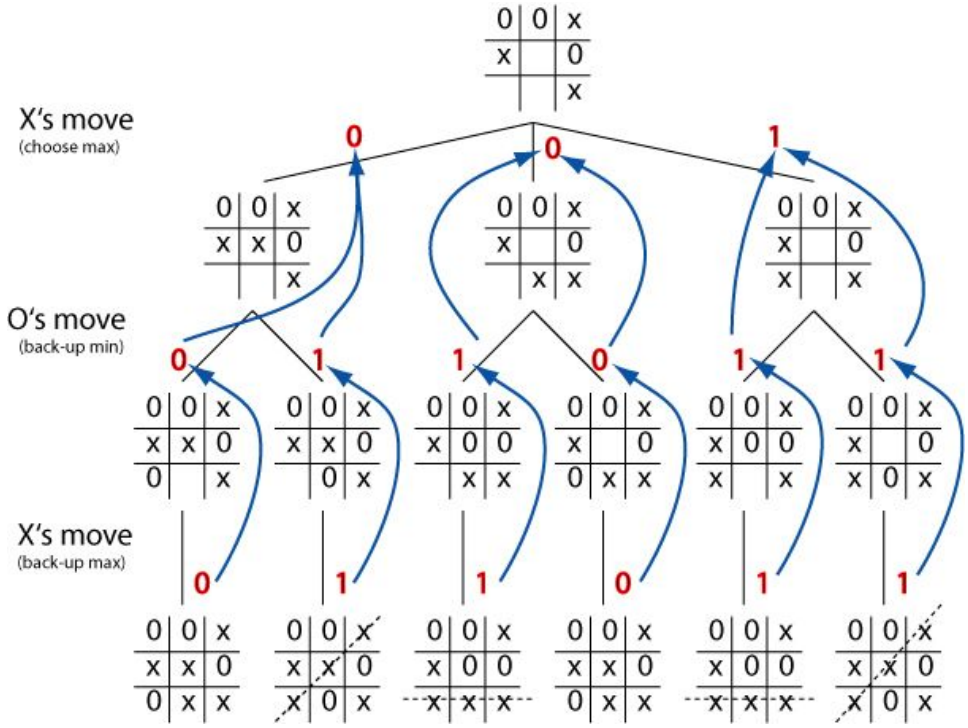

Fig.1 A visualisation of the Minimax algorithm for noughts and crosses [5]

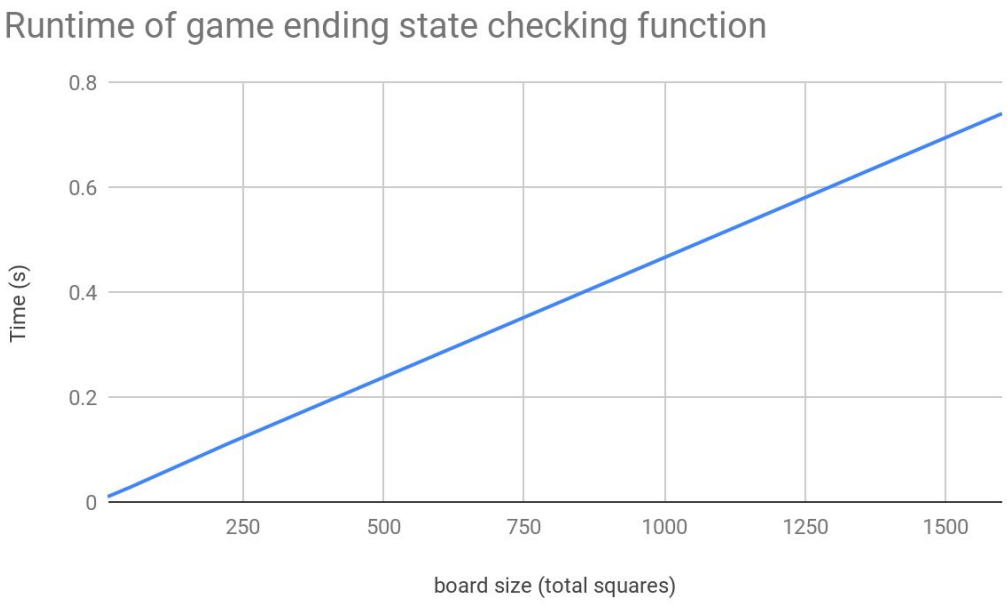

Fig.2 O(n) runtime of function checking for end-game state

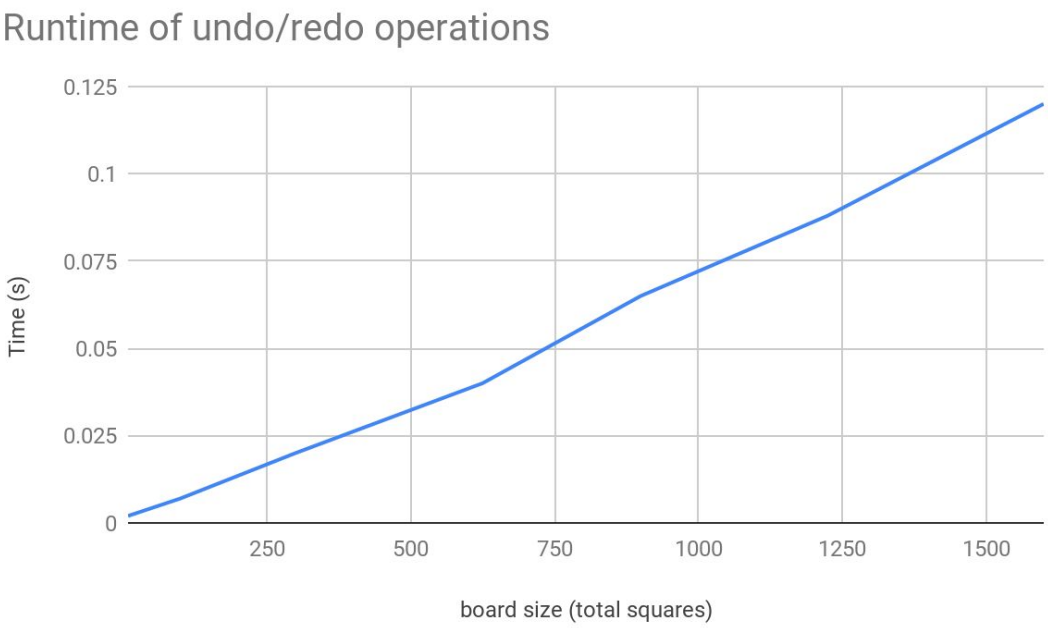

Fig.3 O(n) runtime of undo/redo operations