

Utilisation of Algorithms and Data Structures in the game Connect 4

Adam McKenzie

BSc (Hons) Software Development (GA) 2020

Matriculation ID: 40338725

Module: SET09417

1. Introduction

This report will cover the data structures and algorithms used in order to solve the storage and inner functions of the game Connect 4. In order for a game of connect 4 to work several details need to be recorded such as players moves in terms of both x and y coordinates, player identity, undoing and redoing moves that have already been made, as well as displaying the board game on screen through the command prompt interface.

Each of these features are necessary in order for the game to work however two main things are more important than all other features, specifically, what data structures are utilised in order to store all game data from each game played, but also the algorithms that mainly focus on continually check all moves throughout the game for win conditions.

The problem surrounding all of this comes down to time and memory allocation utilisation on computer hardware due to the limitations of computing power and bottlenecks in the architecture used whether Harvard architecture or Von Neumann architecture.

2. Design

In the design stage it was decided that a Heuristic approach was to be taken given the problem is a small sized application of the game connect 4, making use of logical and procedural method calls in order to allow the game to follow rules and flow in the correct manner.

In the solution covered in this report both stack and queue abstract data type structures were chosen for their ability to store and output data items in a particular order such as queues FIFO(first in first out) and stacks LIFO (last in first out), but also due to their nature of deleting records in the data structures that have been used, such as the pop functionality.

The data structure used in a Stack enables the ability to store data one on top of the other in turn allowing for a last in first out functionality, whereas the data structure used in queues allows for a first in first out flow. Both of these when used allow for functionality such as undoing and redoing player moves that have occurred already in the game being played.

When utilising the data structures mentioned above, in combination with the data structure of a 2-dimensional integer array, an X and Y coordinate grid can be made up enabling the ability to print and update the game board visually when looping through each of the coordinates, as well as storing each players move with an x and y coordinate themselves to check whether player one or two moves have a win outcome each time a move is made across the board. Given the requirements and due to the finite amount of data each game can hold, I decided it would be worthwhile using a basic linear search algorithm within the solution to check each item inside the stack for win conditions.

The data structures mentioned here are a perfect choice for each functionality the game requires when matched with the connect 4 game, offering a lower cost in terms of time complexity and memory size trade off due to the ease of access provided in the data structures to each bit of data stored within them whether on the top stack, or at the front of a queue. This reduces the need for searching through larger indexes of data which could be unordered and such meaning no sorting is required, whereas the structures used provide the ordering of the games data straight off the bat along with memory size management as each bit of data popped from the top of a stack or front of a queue are deleted from memory freeing up space when data is not being used.

At the end of each game, the entire played moves in a given game will be transferred to a list data structure of queue holding each move object for record keeping, and the original stacks and queues used in the current game itself are cleared and ready for use again in the next game if the user wishes to play another game.

Extra functionality for the game was considered in design such as the ability to record each game played in full, allowing the user on the main menu to either play another game, or choose a previously played game from a list and re-watch each move made.

Taking into account each game is made up of move objects stored in a stack, where functionality is required to re-watch each move from a previous game, the ordering of each move needs to be in reverse to that of the stack collection so all moves made of a previous game when stored need to be added to a queue in order to play them out in the correct order when watching a previous game.

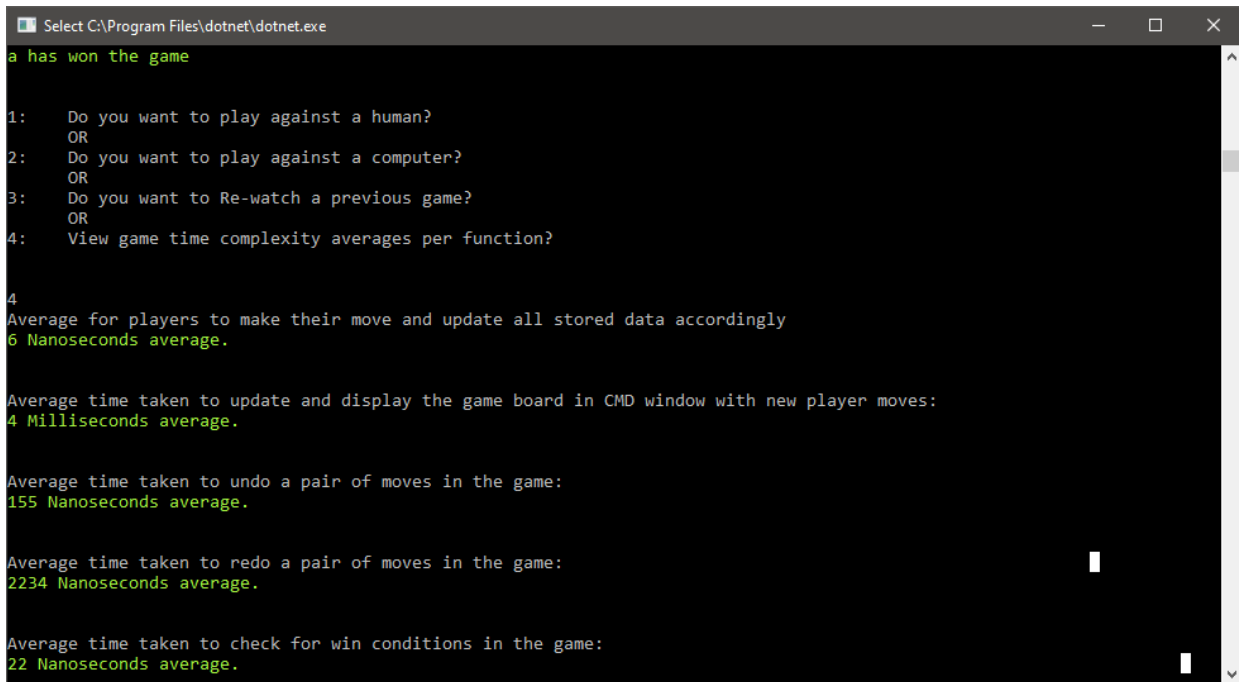
In order to store all games played an abstract data type of list was used in conjunction with queues, so that the list could contain each game played along with all its moves stored in a list of queues. Utilising a list means that no matter how many games are played, a list will always have room to add more to the list, unlike an array where a specific length of storage has to be defined. A list is also indexed and, in this case, makes it easier for the user to provide a number of the game they wish to watch and the list can return that object in a constant time period no matter which number/instance of game is requested.

As all of these data structures mentioned above will be defined as a class object, only one object will ever exist, again in turn lowering the memory allocation usage allowing for greater overall memory and time efficiencies within the game.

Functionality will also be built into the application to store time averages in milliseconds and nanoseconds for specific functions such as undo, redo, make a move and check for win conditions over the runtime of the application. This will be used to test and provide critical evaluation on the data structures used along with their algorithms.

3. Critical Evaluation

Due to the nature in which a stack and queue operate when used for handling of data in this solution, time complexity and memory allocation are utilisation well, allowing for logical and simple programmatic methods to be created to handle the flow of the game and its handling of data throughout.

A screenshot of a Windows command prompt window titled "Select C:\Program Files\dotnet\dotnet.exe". The window has a black background with green text. It displays the following content:

```
a has won the game

1:  Do you want to play against a human?
    OR
2:  Do you want to play against a computer?
    OR
3:  Do you want to Re-watch a previous game?
    OR
4:  View game time complexity averages per function?

4
Average for players to make their move and update all stored data accordingly
6 Nanoseconds average.

Average time taken to update and display the game board in CMD window with new player moves:
4 Milliseconds average.

Average time taken to undo a pair of moves in the game:
155 Nanoseconds average.

Average time taken to redo a pair of moves in the game:
2234 Nanoseconds average.

Average time taken to check for win conditions in the game:
22 Nanoseconds average.
```

Figure1. Screenshot of each functions average time in milliseconds and nanoseconds collected over 5 games using both milliseconds and nanoseconds.

Displayed below are the overall averages per each function in this game:

Average time in milliseconds to run the 'makeMove' method was **6 milliseconds**.

- This function takes in the users next move and updates the int array board of x and y coordinates, whilst also storing the record of the users move in the stack.

Average time in milliseconds to run the 'updateBoard' method was **4 milliseconds**.

- This function loops through both X and Y coordinates of the 2d integer array, visually displaying all users moves on the board via the command user interface

Average time in nanoseconds to run the 'undoMove' method was **155 nanoseconds**

- This function removes the last players moves from the stack and stores them in the undone moves stack of moves to redo later if need be.

Average time in nanoseconds to run the 'redoMove' method was **2234 nanoseconds**

- This function pops the last undone players moves to the current played moves stack, then removes that item from the undone moves stack.

Average time in nanoseconds to run the 'CheckWin' method was **22 nanoseconds**

- This function loops through every x and y coordinate on the board checking each move made in the stack to see if any of the two players have met the win condition which is 4 moves connecting one another in any direction.

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	Green Snap
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Figure2. Screenshot of Data structure Big O notation operations displaying each data structure and its performance in time and space complexity. (Omar Elgabry, 2016)

In this game the size of moves held in stacks can only reach a maximum of 42 items due to the win conditions in a 6 by 7 height playing board, otherwise it's a draw. From the extraction of figure 2 above we can see that stacks and queues have a linear time complexity of $O(1)$ which is a constant time operation, in consideration to this games data being small scale helps complement the use of these data structures in the implemented algorithms displayed in Figure1.

The largest constraint in terms of time complexity is when checking for win conditions in the game, which in this solution is a linear search through each move made in the game to see if any of the direction in which 4 moves by the same player align, resulting in a win. As mentioned, this is a linear logical algorithm as no sorting of data needs to occur, the algorithm loops through both X and Y axis of the board checking each move by each player for a match – in doing so makes use of the Stacks data type access functionality which consists of $O(N)$ meaning the complexity increases with each item held in this data structure. As mentioned before this can only go up to 42 items max with the game board being both 6x7 in size so the impact in this solution is not so bad, however this data structure and algorithm would not scale very well in larger applications. (Adamson 1996)

As shown in figure 1, the time taken for the undo and redo methods in this solution making use of queues and stacks is notably minuet down in the nano seconds due to the way stacks and queues time complexity handles the insertion and deletion of data in regards to Big-O as $O(1)$ constant time. Given the data being removed or replayed is no more than one or two player moves at a time, this showcases another good example as to why stacks and queue data types suit this application as discussed in the design section of this report. Had an array been used for example, the time complexity would scale with each iteration of the array to find the item that is to be removed or added to the game being played, leading to an outcome of greater time complexity to handle the same data and functionality within the game. (Adamson 1996)

References

Adamson, I., 1996. Data Structures And Algorithms. London: Springer.

<https://link.springer.com/book/10.1007/978-1-4471-1023-1>

Omar Elgabry, Medium. (2016). Data Structures — A Quick Comparison. [online] Available at:

<https://medium.com/omarelgabrys-blog/data-structures-a-quick-comparison-6689d725b3b0> [Accessed 2 Mar. 2020].

Appendix

Github Repo - https://github.com/Amckenzie93/Mckenzie_Adam_ads_connect4

Github SSH Clone URL - `git@github.com:Amckenzie93/Mckenzie_Adam_ads_connect4.git`