A Critical Evaluation of the Algorithms and Data Structures Used To Build A Connect Four Game

By Monica Richardson,
Matriculation No. 40338729,
SET09417 Algorithms & Data Structures

1.0 Introduction

The aim of this project was to create a simple, command-line only version of the game "Connect Four", with particular focus on utilising the most appropriate and efficient data structures and algorithms to achieve best performance. The game produced offers the following features:

- **Recorded Game History**: All games played within the run-time session are recorded and users can replay earlier games.
- **Undo/Redo Functionality**: Users can undo moves made, right back to the initial game state. Users can also redo moves which have been un-done.
- **Custom Board Dimensions**: Users can specify the height and width of the board they wish to play. Any height and width between 4 and 26 can be chosen.
- **AI Player**: Users can choose to play against an AI player, which is programmed to choose "blocking", "selfish" or "random" moves based on factors including how close its opponent is to winning, and the status of slots surrounding its own previous moves.

2.0 Design

The solution is made up of 6 classes; "Program", "Display", "Game", "Player", "Board" and "Slot". For a full class diagram see Appendix A (p.5). The Program class houses the Main() method from which the application is run. The Display class contains all methods necessary for user interaction – such as methods which display menus, retrieve user selection, display the board throughout gameplay etc. Core game functionality is housed in the remaining 4 classes.

2.1 Storing Board Slots

When choosing a data structure to store slots within the Board class, I knew it was important that I was able to quickly retrieve information about a slot, using the coordinates of that slot (e.g. "A1"), without having to iterate every slot in the board. For this reason, I knew I needed a data structure which stored key-value pairs, with an access time complexity of O(1) – this narrowed the decision down to either a Dictionary or a Hashtable. When researching the data retrieval speed of these two data structures, I learned that data retrieval from Hashtables involves "boxing" and "unboxing" when accessing the hash code of each key within the Hashtable. This process of boxing and unboxing makes data retrieval from a Hashtable slower compared to that from a Dictionary. For this reason I chose to store the slots of a Board object in a Dictionary, where the key in each key-value pair is the slot coordinate as a string, and the value is the Slot object itself.

2.2 Storing Game Moves

A Stack data structure was chosen for the storage of moves within the Game class, because it was important that moves were stored in the order that they were played, and that the last move made could be easily accessed for undo and redo logic. When displaying each move on the board, it was also important that information about who made the move, as well as the location of the move was easily accessed. Therefore I chose to store each move in the

Stack as a key-value pair, where the key is the Slot object, and the value is the Player object. This meant, when implementing undo and redo functionality, I created a second stack for un-done moves, allowing me to "pop" and "push" moves from one stack to another whilst maintaining the order in which the moves were originally played.

2.3 Storing Players & Board Axis

Players in the Game class are stored in an array of Player objects, as there are guaranteed to be exactly two players per game. Therefore, since the data being stored was of a fixed-size, and arrays allow for easy and efficient access to each item via indexing, an array was deemed the most appropriate data structure for this purpose. This is also why arrays were chosen to store each item in the X and Y axis of the Board. Once the height and width of the board is established by the user, these values remain constant throughout each Game instance, and easy access via indexing is ideal when printing the board to the console.

2.4 The Win Condition Algorithm

The Winner() algorithm in the Game class, aims to sort and limit the quantity of slots which need to be checked after each move to quickly and efficiently establish whether or not a win condition has been met. First, the algorithm calls the GetSurroundingSlots() method to establish only the slots surrounding the last move where a win condition could take place. This means storing the coordinates of 3 slots either side of the last move, in all directions – 25 slot coordinates in total. For a diagram showing the coordinates this part of the algorithm stores, see Appendix B (p.6). At first, I stored these coordinates in 4 arrays – two for each diagonal set of slots, one for horizontal slots and one for vertical slots. However, I quickly encountered a problem in scenarios where there were less than 3 slots either side of the last move, in any direction, as this meant storing null values in the array, ultimately wasting algorithm run-time by iterating null values. Ideally, if the value was null, I preferred not to store it to improve efficiency. However, it was still important that items were stored in the order they needed to be checked in, and being able to specify the index of an item was crucial. For this reason, I chose to store the coordinates of the surrounding slots to 4 Sorted Lists, as this allowed me to use the GetByIndex() method to retrieve values, have them automatically sorted in order according to key, and allowed me to store only valid coordinates of surrounding slots as the value of each key-value pair – meaning each Sorted List did not always contain 7 items.

Once establishing the 4 Sorted Lists of surrounding slot coordinates, the FourInARow() method checks at least the first four values in each Sorted List. If the first four values are null or empty, the method moves on to the next Sorted List. If, however, a slot in the Sorted List is found to have been populated by the current player, the next 3 consecutive slots are checked. If an empty slot, or a slot not populated by the current player is encountered, the method moves on to the next Sorted List. If 4 consecutive slots are found to have been populated by the current player, the method returns a boolean value of "true" as the win condition has been met.

2.5 The AI Decision Making Algorithm

The AIMakeMove() decision making algorithm in the Player class makes use of the GetSurroundingSlots() method discussed in section 2.4 to store all the slots surrounding the last move made by the AI's opponent. The algorithm also uses the AIEstablishBestMove() method to break the AI's decision down into three possible choices – a "blocking" move, "selfish" move or "random" move.

First, the algorithm identifies each slot it could possibly fill in the next move – i.e. the next available slot in each column. Then the AI plays out each of these scenarios as if it were its opponent, running the Win Condition algorithm each time to establish whether the opponent could win in their next move. If the AI identifies a slot which

could allow the opponent to win in their next move, the AI will prioritise populating this slot first – dubbed in this case as a "blocking" move.

If the AI cannot identify a blocking move, it will instead use the GetSurroundingSlots() method to identify all the slots surrounding each of its own past moves. It will then check to see if any of these slots are slots which could be filled in the next move. The AI will populate the first empty slot available within the winning-radius of one of its own previous moves – dubbed in this case as a "selfish" move.

If the AI cannot identify a blocking move or a selfish move, it will pick one of the next available slots at random – dubbed in this case as a "random" move.

3.0 Critical Evaluation

3.1 The Win Condition Algorithm

The Win Condition algorithm is successful in ensuring that only a limited number of slots need to be checked each time a player makes a new move, regardless of the size of the board.

Initially, to evaluate the performance of the Win Condition algorithm I created an empty 7x9 board like the one shown in Appendix B (p.6), and populated slot 'F4'. I then populated the first 3 slots in the first collection of slots checked (C1, D2, E3), creating a win scenario, and timed how long the Win Condition algorithm took to identify this as a win. I then moved the location of the win-scenario 1 slot along (D2, E3, F4, G5), and proceeded to time how long it took for the Win Condition to identify each win, testing in the order in which the Win Condition would normally iterate the slots. What I hoped this would demonstrate was the increasing length of time taken to identify a Win Condition depending on its location within the iterated slots, and ultimately the worst case run-time when searching for a win condition in the last position of the last set of slots (F4, F3, F2, F1). I expected the outcome to look something like an O(n) graph curve, with the run-time increasing as the win condition occurred closer to the end of the iterated slots. However, the average outcome of this test after 10 test runs was not what I expected – see Appendix C (p.7) for full test data, Appendix D (p.8) for resultant graphs and Appendix E (p.9) for the code used to retrieve the data recorded.

One of the anomalies I noticed in this test data, is that a win located in the first position of the first Sorted List to be iterated takes consistently longer to be identified than a win located in any other position. I assumed this would be the quickest win to identify. When looking at the rest of the data, there appears to be an overall downward trend, with consistent increases in run-time when searching for a win located in the middle of the Sorted List being checked. Again, this contradicts my assumption that the run-time would increase as the win scenario moved to the latter part of the Sorted Lists being checked.

Since I was unable to draw any definitive conclusions about the Win Condition algorithm's overall performance from this test, I decided to test the algorithm in a more abstract way. When broken down, the Win Condition algorithm is an algorithm which creates and iterates a set of Sorted Lists. Therefore, to evaluate the performance of the Win Condition algorithm, I calculated the approximate growth rate and time complexity of iterating items in a Sorted List. I did this by writing a short function which recorded the time taken to insert one item to a Sorted List and iterate it. The function continues to add items to the Sorted List, recording the time taken to iterate the Sorted List each time it increases. I used this data to plot the time taken to iterate the Sorted List against the number of items in the Sorted List. For the resultant graph, along with the code used to retrieve this data, see Appendix F (p.10). While the resultant graph curve indicates a Big O time complexity of O(n), the number of Sorted List items being iterated will only ever consist of (at most) the 25 coordinate radius shown on the diagram in Appendix B (p.6). Therefore it can be

concluded that the overall time complexity of the Win Condition algorithm is relatively low, and works well for a solution of this size and complexity.

3.2 The AI Decision Making Algorithm

Unlike the Win Condition algorithm, the time complexity of the AI Decision Making algorithm discussed in section 2.5 does increase as the size of the board increases; the more available columns, the more potential moves, and the more scenarios the AI needs to play out in order to make a decision. This makes the design of this algorithm much less appropriate for use in larger or more complex solutions.

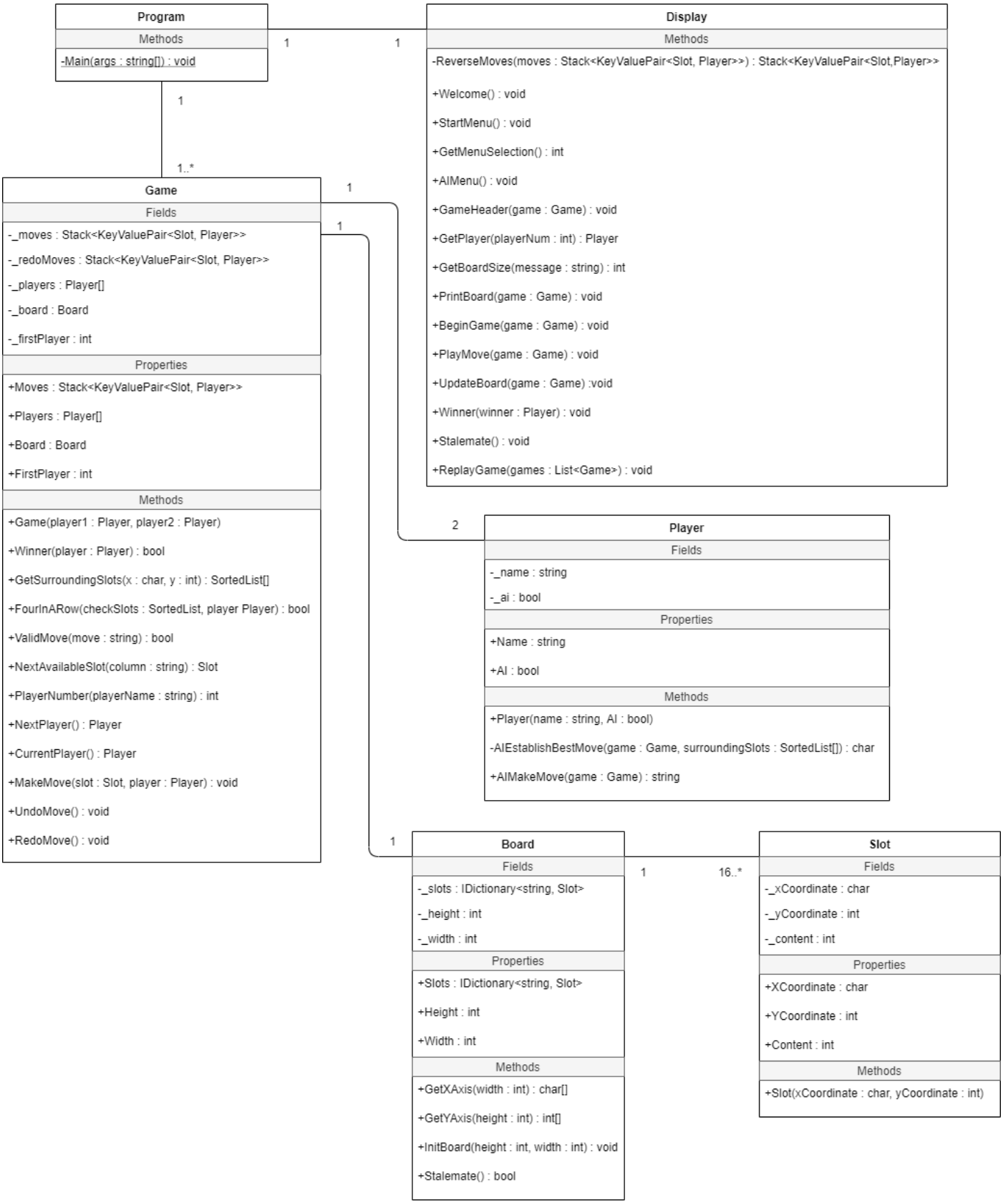
In terms of decision making ability, the AI's weakness in its current state is its inability to see beyond the next move. There are scenarios where the AI will select a move without realising that this move directly leverages the opponent to win in the following move. If I were to spend time developing the AI further, I would investigate ways to enable the AI to play out scenarios more than 1 move in advance, whilst somehow reducing run-time complexity.

4.0 Conclusion

I believe appropriate and memory-efficient data structures were utilised throughout the development of this game. While experimental performance tests conducted on the Win Condition algorithm did not produce the expected results, I am confident that the algorithm quickly and accurately identifies each win whilst effectively narrowing the search down to only relevant slots. If I were to invest more time on the development of this game, I would seek to break the Win Condition algorithm down further, to allow for more detailed analysis and testing. Hopefully this would help explain some of the anomalies encountered during experimental testing. I would also seek to narrow the AI's process of playing out various game scenarios, perhaps only playing out scenarios which the AI deems 'relevant' to the latest move. This would reduce the AI Decision Making algorithm's overall run-time complexity, making it a scalable AI solution for larger and more complex solutions. Overall I am pleased with the performance of the Connect Four game.

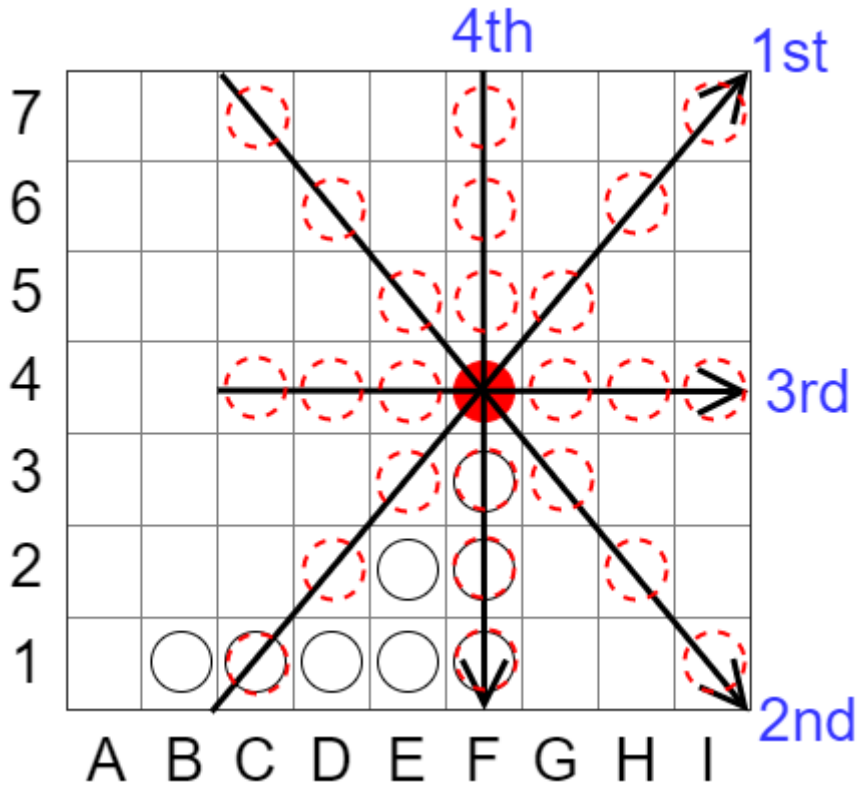
Appendix A

Connect Four Class Diagram



Appendix B

Diagram showing the 4 sets of slot coordinates which are stored by the Win Condition algorithm. The algorithm iterates these slots checking for 4 consecutive slots populated by a single player. The diagram shows the order and direction in which the slot coordinates are checked.



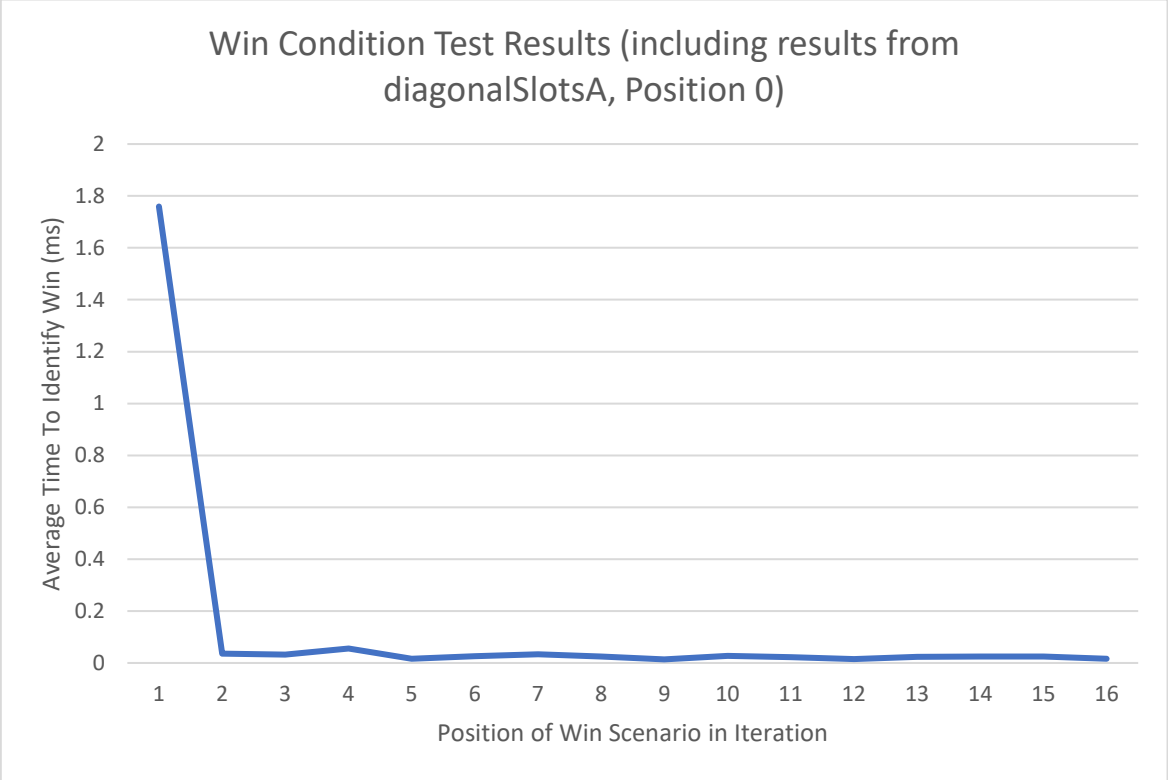
Appendix C

Resultant data from initial Win Condition time harness experiment, showing the elapsed time taken in milliseconds to identify a win condition at each consecutive position in the 4 Sorted Lists – Diagonal 1, Diagonal 2, Horizontal and Vertical – across 10 test runs, as well as the average elapsed time.

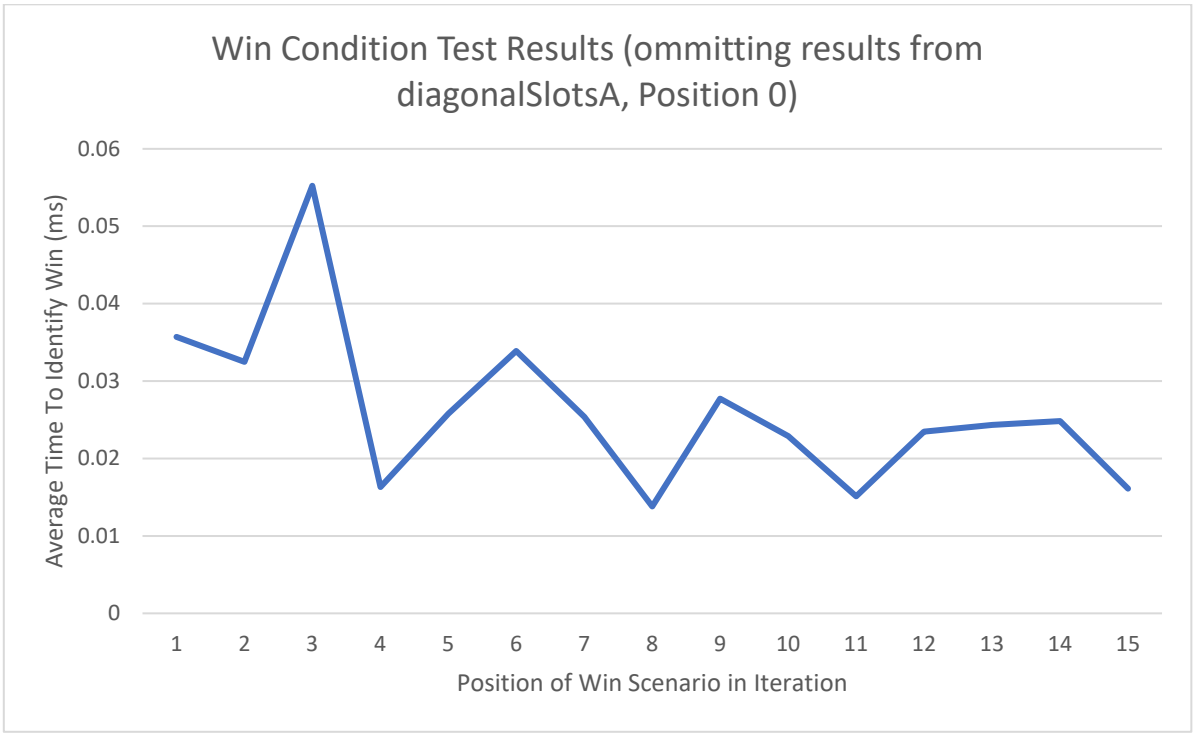
	Time Elapsed (ms)										
Win Scenario Location	Round 1	Round 2	Round 3	Round 4	Round 5	Round 6	Round 7	Round 8	Round 9	Round 10	Average
diagonalSlotsA, Position 0	1.3753	1.4677	2.628	1.3161	1.4525	1.2152	3.2362	1.8922	1.5407	1.4685	1.75924
diagonalSlotsA, Position 1	0.0337	0.0295	0.0288	0.0188	0.0318	0.1129	0.0213	0.0188	0.0256	0.0359	0.03571
diagonalSlotsA, Position 2	0.0201	0.039	0.0217	0.0265	0.0365	0.0418	0.0322	0.0231	0.0193	0.0645	0.03247
diagonalSlotsA, Position 3	0.0337	0.0825	0.0516	0.0429	0.0422	0.0868	0.0315	0.0365	0.0311	0.1135	0.05523
diagonalSlotsB, Position 0	0.0107	0.0167	0.0155	0.0176	0.0232	0.0293	0.0124	0.0109	0.011	0.0159	0.01632
diagonalSlotsB, Position 1	0.018	0.0254	0.0206	0.0289	0.0364	0.0363	0.0269	0.0184	0.0238	0.023	0.02577
diagonalSlotsB, Position 2	0.0263	0.0368	0.0366	0.044	0.029	0.0285	0.0286	0.0256	0.0494	0.0343	0.03391
diagonalSlotsB, Position 3	0.0173	0.032	0.0322	0.0199	0.0349	0.0184	0.021	0.018	0.0302	0.0297	0.02536
horizontalSlots, Position 0	0.0115	0.0151	0.0147	0.0139	0.0132	0.0114	0.0127	0.011	0.0141	0.0205	0.01381
horizontalSlots, Position 1	0.0299	0.0193	0.0209	0.0373	0.0211	0.0282	0.034	0.0187	0.0372	0.0307	0.02773
horizontalSlots, Position 2	0.0239	0.0182	0.0291	0.0177	0.031	0.0183	0.0214	0.0276	0.0218	0.0199	0.02289
horizontalSlots, Position 3	0.0168	0.0115	0.0163	0.011	0.0132	0.0114	0.0132	0.0129	0.0255	0.0193	0.01511
verticalSlots, Position 0	0.03	0.0321	0.0211	0.018	0.0193	0.0189	0.0215	0.02	0.0267	0.027	0.02346
verticalSlots, Position 1	0.0291	0.037	0.0211	0.0179	0.0186	0.0188	0.0214	0.019	0.0332	0.0273	0.02434
verticalSlots, Position 2	0.0245	0.0321	0.0199	0.0177	0.0377	0.0189	0.022	0.0249	0.0262	0.0246	0.02485
verticalSlots, Position 3	0.0196	0.0186	0.0116	0.0113	0.0184	0.0118	0.0135	0.0199	0.0237	0.0127	0.01611

Appendix D

Line graph derived from data in Appendix C, showing the average time taken to identify a win in milliseconds against the position of the win scenario in the iteration, including the average time taken to find a win in the first position of the iteration, which is noticeably higher than other recorded values.



Line graph derived from data in Appendix C, showing the average time taken to identify a win in milliseconds against the position of the win scenario in the iteration, omitting the average time taken to find a win in the first position of the iteration.



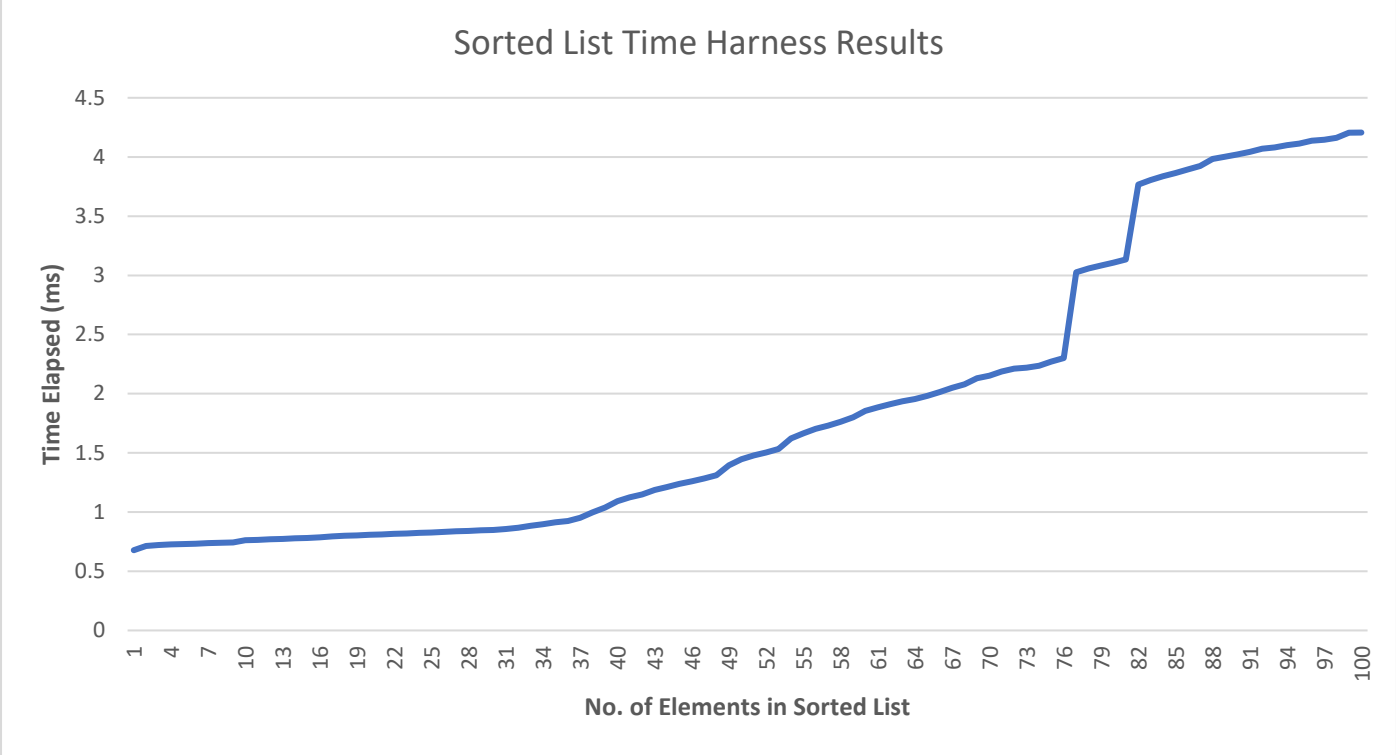
Appendix E

Code written to retrieve the data recorded in Appendix C.

```
static void Main(string[] args)
{

    // Instantiate players
    Player player1 = new Player("Player 1", false);
    Player player2 = new Player("Player 2", false);

    // Instantiate game
    Game game = new Game(player1, player2);

    // Create empty 7x9 board
    game.Board.InitBoard(7, 9);

    // Populate F4
    game.Board.Slots["F4"].Content = 1;

    // Get surrounding slot info
    SortedList[] surroundingSlots = game.GetSurroundingSlots('F', 4);

    foreach (SortedList slotGroup in surroundingSlots)
    {
        for (int i = 0; i < 4; i++)
        {
            // Create a win scenario in the set, moving the win-scenario, 1 place along each time
            game.Board.Slots[slotGroup.GetByIndex(i).ToString()].Content = 1;
            game.Board.Slots[slotGroup.GetByIndex(i + 1).ToString()].Content = 1;
            game.Board.Slots[slotGroup.GetByIndex(i + 2).ToString()].Content = 1;
            game.Board.Slots[slotGroup.GetByIndex(i + 3).ToString()].Content = 1;

            // Start timer
            var watch = new System.Diagnostics.Stopwatch();
            watch.Start();

            // Run win condition algorithm
            bool win = game.Winner(player1, 'F', 4);

            // Stop timer
            watch.Stop();
            Console.WriteLine("{0}", watch.Elapsed.TotalMilliseconds);

            // Reset slots
            game.Board.Slots[slotGroup.GetByIndex(i).ToString()].Content = 0;
            game.Board.Slots[slotGroup.GetByIndex(i + 1).ToString()].Content = 0;
            game.Board.Slots[slotGroup.GetByIndex(i + 2).ToString()].Content = 0;
            game.Board.Slots[slotGroup.GetByIndex(i + 3).ToString()].Content = 0;
        }
    }

}
```

Appendix F

Results of Sorted List Time Harness experiment, showing the length of time taken to iterate a Sorted List against the size of the Sorted List.



Code used to retrieve the data displayed above:

```
static void Main(string[] args)
{
    int count = 1;

    // Begin timer
    var watch = new System.Diagnostics.Stopwatch();
    watch.Start();

    // Create Sorted List
    SortedList test1 = new SortedList();

    do
    {
        // Add an item to Sorted List
        test1.Add(count.ToString(), "A" + count.ToString());

        // Iterate Sorted List and read each item value
        for (int i = 0; i < test1.Count; i++)
        {
            string readValue = test1.GetByIndex(i).ToString();
        }
        watch.Stop();
        Console.WriteLine(count + " items added and read. Elapsed Time: {0}",
        watch.Elapsed.TotalMilliseconds);

        count++;
        watch.Start();

    } while (count < 100);
    watch.Stop();
}
```