



ACCA5032: OOP AND ADVANCED SOFTWARE DEVELOPMENT (ALGORITHMS)

Table of Contents

Part A.....	2
1. [10 Marks].....	2
2. [10 Marks]	3
3. [20 Marks]	5
Part B.....	8
1. [20 marks]	8
2. [20 Marks]	11
Part C.....	12
1. [20 Marks]	12
Minimax Algorithm	12
Implementation of Minimax Algorithm for Connect 4 Game	14
References	17
Appendix A.....	18
1. Board.cs	18
2. Counter.cs	21
3. Game.cs.....	22
4. Gameplay.cs	22
5. Player.cs	31
6. Program.cs	31
7. Validation.cs.....	32

Part A

1. [10 Marks].

An array can be defined as a collection of similar types of items. An example is that a string array holds a collection of elements of type string, or that an integer array holds a collection of elements of type int. An important characteristic of arrays is that they have a fixed size which is defined upon initialization. Arrays can be multidimensional, with 1d and 2d arrays being the most widely used. Through using arrays other data structures can be implemented, for example 2D arrays can be used to represent matrices, and stacks, queues, trees, and linked lists are all also implemented from arrays.

A linked list is a dynamic arrangement [1] that logically links each node to another by storing pointers within the structure itself. Linked lists, unlike arrays, aren't in consecutive memory locations. There are many types of linked list, including singly linked lists, doubly (two way) linked lists, circular linked lists, and circular doubly linked lists. Unlike arrays, linked lists have no fixed size, and their size can vary throughout runtime.

An advantage that arrays have over linked lists is that the elements of the array can be accessed randomly by using the index, whereas direct access to an element is not possible in a linked list and traversal is more-time consuming when compared with an array, as nodes must be searched sequentially. Additionally, as arrays allocate memory in contiguous locations, there is no extra memory being allocated, avoiding memory overflow and memory shortage. However, this relies on a developer initialising an array to a suitable size. Linked lists on the other hand require more memory compared to an array as a pointer to the next element must also be stored, however there is more efficient memory utilization as the size of the linked list increases or decreases at run time and there is no need to pre-allocate memory.

A disadvantage of using arrays is as they are of a fixed size, once declared the size cannot be increased or decreased, which can cause out of bounds errors or memory wastage if not carefully considered. Comparatively, linked lists can allocate and deallocate memory at runtime as required.

Insertion and deletion of items are more difficult with arrays than linked lists as the elements are stored in consecutive locations, requiring the use of a temp variable to switch elements. Linked lists are less costly as only the next pointer in the previous node needs to be updated. However, removing values from the middle of the linked list, or inserting into the middle of the linked list incurs the cost of searching for the correct position first and therefore the cost becomes $O(n)$ similar to an array.

It is possible to reverse traverse an array due to the length of the array being known, however it is not possible in a singly linked list as there is no pointer to the previous node. However, a doubly linked list contains pointers to both the previous and next node and so it is possible, but this requires extra memory for the additional pointers and so should not be used unless reverse traversal is necessary.

Below is a table outlining the complexity comparison of linked lists and arrays:

Operation	Linked List Big O	Array Big O
Read	$O(n)$	$O(1)$
Add Value	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

From the table it can be said that arrays provide better read performance, but lists can give better manipulation (add, delete) performance. To decide which to use, specific user requirements must be considered, often varying between search performance and insertion/storage requirements. For example, if better search performance is more important, an array should be used, or if insertion and deletion is more important, linked lists would be a more suitable structure.

2. [10 Marks]

For the given requirements it would be preferable to use a linked list to store the list of names due to the unknown number of records to be stored per letter (A-Z). The overarching structure should be an array due to the known size (A-Z = 26 letters) and improved search performance, with each array element pointing to the head of the individual linked list. The diagram below outlines this.

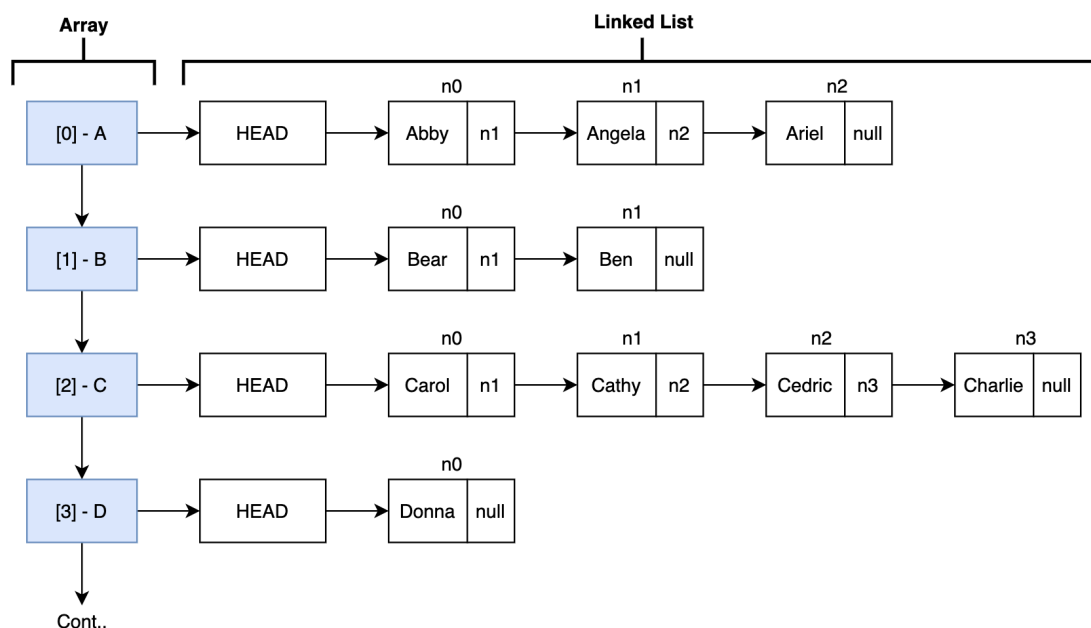


Figure 1. Simplified diagram of suggested structure

If each name was to be stored in alphabetical order, it would be easier to store all names beginning with letter A in a linked list, as when new names with the letter A are to be added, it would be less costly to insert them into a linked list than an array. However, the linked list will still need to be traversed to find the correct position to insert. If order of storage is not of importance, then records could be quickly appended to the end of the linked list.

If search performance was an important factor, then Linked Lists would not be preferable for searching due to the poor performance caused by the linear nature of traversal, however

the main requirement is to add thousands of records daily, and therefore linked list is more suitable.

Probable Performance for the Given Structure

To insert a value the overarching array could first be searched to find the correct position (letter of the alphabet), using a search such as binary search with a time complexity of $O(\log n)$, or, as each element is known, direct access could be used, e.g. $A = [0]$, $D = [3]$, $G = [6]$ with a constant time complexity of $O(1)$.

Insertion and deletion to/from a linked list would require a time complexity of $O(n)$, as you must potentially traverse the entire list before inserting/deleting a record, with a best case of $O(1)$ for the head node or simply making a change to a pointer. Insertion performance could be improved if there was no requirement for ordered storage, and would also result in a time complexity of $O(1)$ to append each item to the tail of the list.

To search for a value within this structure you would first need to search the overarching array with a time complexity of $O(1)$, $O(\log n)$ or $O(n)$ depending on the type of search used. You would then begin to traverse the linked list from the head node until the value is found, searching $O(n)$ names, where n may be very large given the requirement of inserting thousands of records daily.

Linked list search performance would not be as good as if the data was stored within an array, which could vary from a linear complexity of $O(n)$ for a linear search similarly to the linked list, or if the data was ordered a binary search algorithm could be used allowing for a time complexity of $O(\log n)$, with a best case complexity of $O(1)$ where the central index matches the desired value. However, to use an array within the given requirement, it is likely that the array size would have to be increased frequently, which would result in having to create a new array of increased size, and then copy each item of the old array into the new array, repeating for each time the array size increases drastically, which is less efficient than the flexible non-fixed size of the linked list.

If search performance is of great importance and an array was to be used, consideration must be made for if names are not evenly dispersed across the alphabet, such as groupings of many names in a single letter e.g. names starting with H as search performance would be less than optimal. In this case a quick divide and conquer search such as binary search would help to improve search performance.

3. [20 Marks]

A binary search tree for the list: 34, 6, 7, 18, 45, 12, 3, 26 would be constructed as shown in the diagram below.

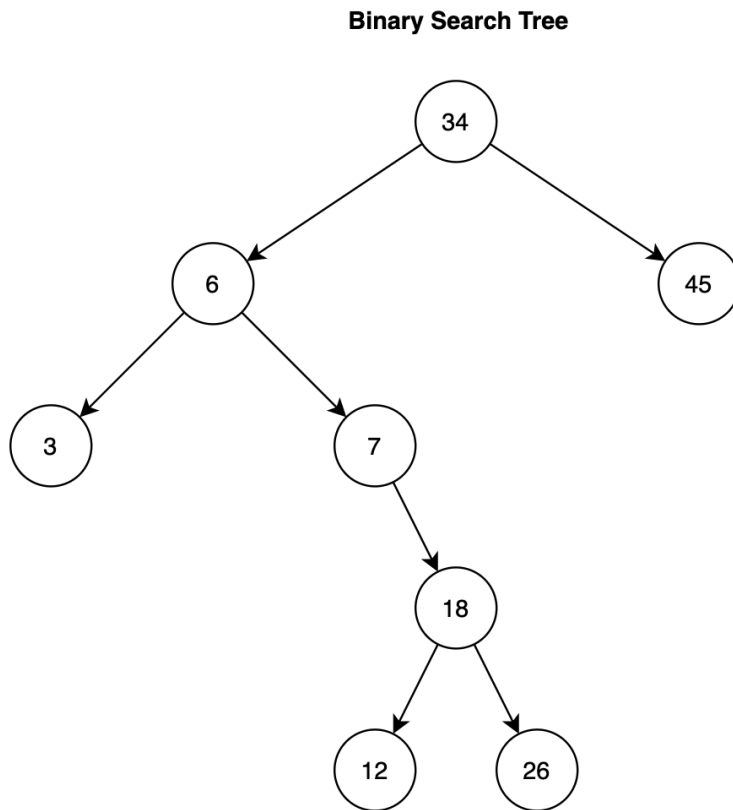


Figure 2. Binary Search Tree for values 34, 6, 7, 45, 12, 3, 26

A balanced binary tree is defined as a tree that the heights of the child subtrees of any node differ by one at most. The binary search tree above is visibly unbalanced, however, to confirm that the binary search tree is unbalanced the balance factor for each node should be calculated. To calculate the balance factor for each node, the following equation should be done:

$$\text{Balance Factor} = \text{Height}(\text{Right Subtree}) - \text{Height}(\text{Left Subtree})$$

If the balance factor for any node is outside of the range -1 to 1, the tree is unbalanced. The diagram below shows the balance factor for the current binary search tree.

Binary Search Tree with Balance Factors Shown

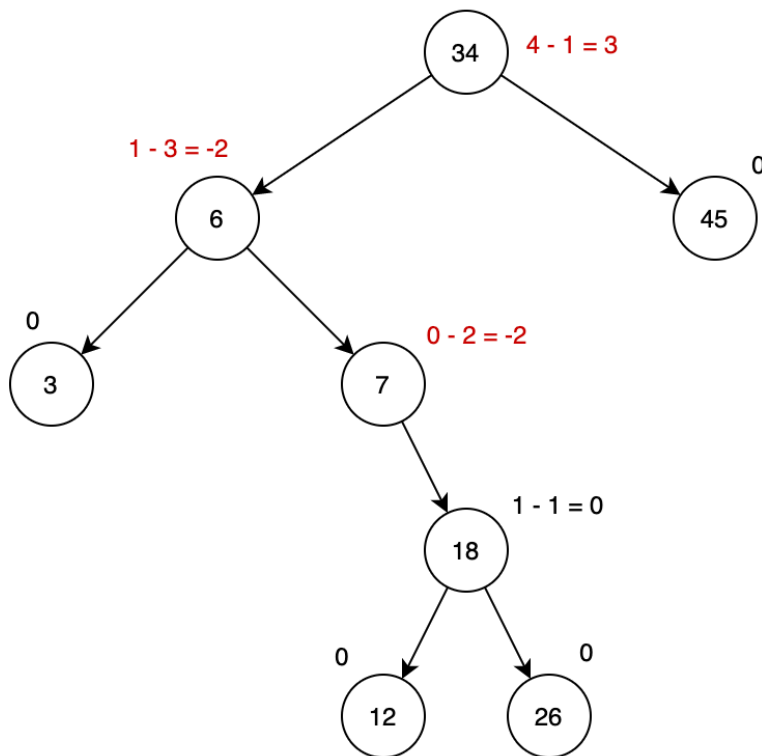


Figure 3. Binary Search Tree with Balance Factors Shown

As nodes (34), (6) and (7) all have balance factors outside of the allowed -1, 0 and 1, the tree can be said to be unbalanced. Maintaining a balanced binary tree is critical to ensuring good search performance. For example, to search for 26, it would require 5 comparisons (34, 6, 7, 18 and 26). When this example is compared with a balanced AVL tree for the same list, as shown below, this search takes only 3 comparisons (18, 34 and 26), reducing the search effort by 40%. This is a very small binary search tree, if you consider a large array, the extra number of searches in an unbalanced tree could have a considerable negative effect on the search performance.

Balanced AVL Tree

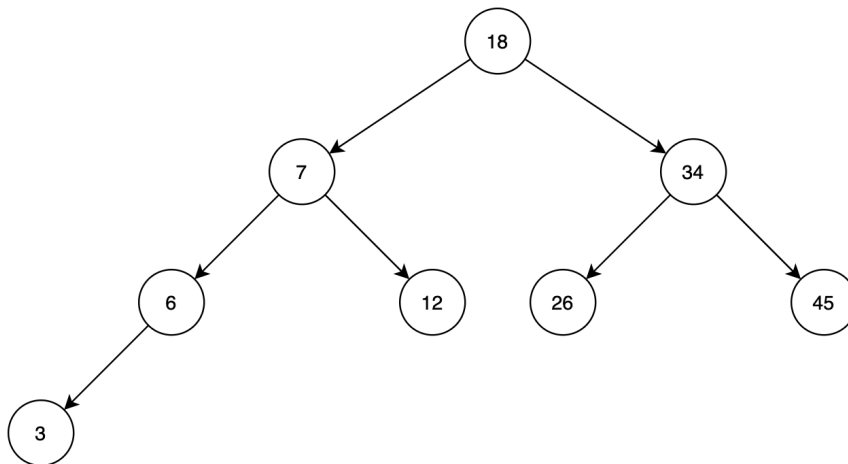


Figure 4. Given List as a Balanced AVL Tree

Similarly, if a binary search tree was constructed using an ordered list, this produces a worst-case tree where all nodes descend in a waterfall style, producing a search performance of $O(n)$, compared to a balanced binary tree's increased performance of $O(\log n)$.

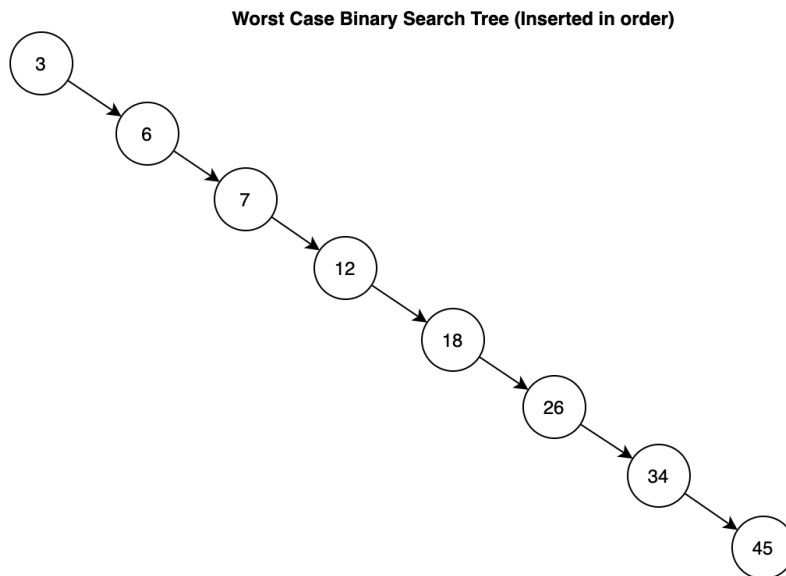


Figure 5. Worst Case Binary Search Tree

Since we cannot control the order in which values are added and the balance of the tree is critical to its search performance, then a self-balancing search tree is most suitable to use. An Adelson-Velsky and Landis (AVL) Tree is a self-balancing binary search tree that includes a balance factor with each node. When inserting new values into the tree, it rebalances itself by rotating the nodes until the balance factor of all nodes fall within the range of -1 to 1. This ensures that a balanced tree is maintained, with optimal search performance.

Part B

1. [20 marks]

Dijkstra's algorithm for finding the lowest cost resembles a breadth first search. However, if implemented with a priority queue, then the algorithm is reduced to $O(E \log V)$ time compared to $O(EV^2)$.

In order to demonstrate the trades between items (path between vertices) the following diagram and table has been produced. Note that the item IDs (A-F) have been used to simplify the diagrams.

Table 1. Directed Graphs showing each step of Dijkstra's Algorithm with Priority Queue - see table for each step

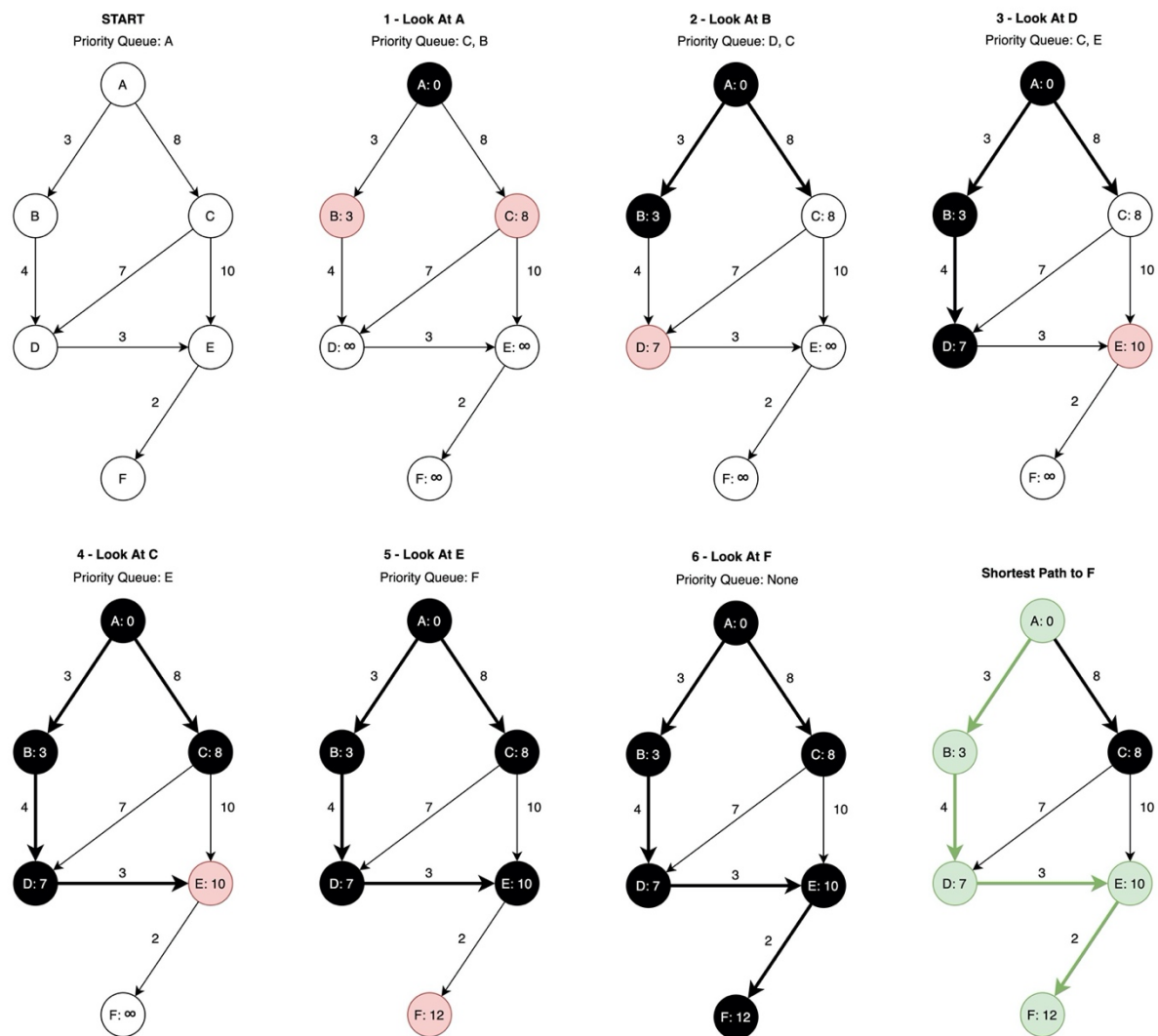


Table 2. Table showing each step of the algorithm (nodes discovered in red, visited, parents and costs)

	Vertex to Check	Vertex	A	B	C	D	E	F
Start	Start	Parent	Null	Null	Null	Null	Null	Null
		Cost	0	∞	∞	∞	∞	∞
1	A	Parent	Null	A	A	Null	Null	Null
		Cost	0	3	8	∞	∞	∞
2	B	Parent	Null	A	A	B	Null	Null
		Cost	0	3	8	7	∞	∞
3	D	Parent	Null	A	A	B	D	Null
		Cost	0	3	8	7	10	∞
4	C	Parent	Null	A	A	B	D	Null
		Cost	0	3	8	7	10	∞
5	E	Parent	Null	A	A	B	D	E
		Cost	0	3	8	7	10	12
6	F	Parent	Null	A	A	B	D	E
		Cost	0	3	8	7	10	12

Description of each step:

Start – The distance from the starting node A is set to 0 and all other nodes are set to infinity distance. A is added to the priority queue to be inspected next. The parent of the starting node is set to null.

1. Process starting node A
 - a. Inspect the neighbours of the current node A, starting with node C (not yet visited). Add C to the priority queue. Set the distance to the starting node to be the distance of the parent plus the edge cost (8).
 - b. Inspect the neighbours of the current node A, next is node B (not yet visited). Add B to the priority queue. Set the distance to the starting node to be the distance of the parent plus the edge cost (3).
 - c. Priority queue at this step: B, C
2. Process next node B (lowest distance in priority queue)
 - a. Inspect the neighbours of the current node B, next is node D (not visited). Add D to the priority queue. Set the distance to the starting node to be the distance of the parent plus the edge cost ($3 + 4 = 7$).
 - b. Priority queue at this step: D, C
3. Process next node D (lowest distance in priority queue)
 - a. Inspect the neighbours of the current node D, next is node E (not yet visited). Add E to the priority queue. Set the distance to the starting node to be the distance of the parent plus the edge cost ($7 + 3 = 10$).
 - b. Priority queue at this step: C, E
4. Process next node C (lowest distance in priority queue)
 - a. Inspect the neighbours of the current node C, next is node E (visited) – cost is greater than existing route ($10 + 8 = 18 > 10$) and so parent for E is left as D (cost 8).
 - b. Inspect the neighbours of the current node C, next is node D (visited) – cost is greater than existing route ($8 + 7 = 15 > 7$) and so parent for D is left as B (cost 7).
 - c. Priority queue at this step: E

5. Process next node E (only item in priority queue)
 - a. Inspect the neighbours of the current node E, next is node F (not yet visited). Add F to the priority queue. Set the distance to the starting node to be the distance of the parent plus the edge cost ($10 + 2 = 12$).
 - b. Priority queue at this step: F
6. Process next node F (only item in priority queue)
 - a. No neighbours for node F.
 - b. All nodes have been processed and the shortest route to F is found = A, B, D, E, F.

2. [20 Marks]

Dijkstra's algorithm runs in $O(EV^2)$ time, where E is the number of edges and V is the number of vertices. This can be improved by using a priority queue which results in $O(E \log V)$ time performance [2].

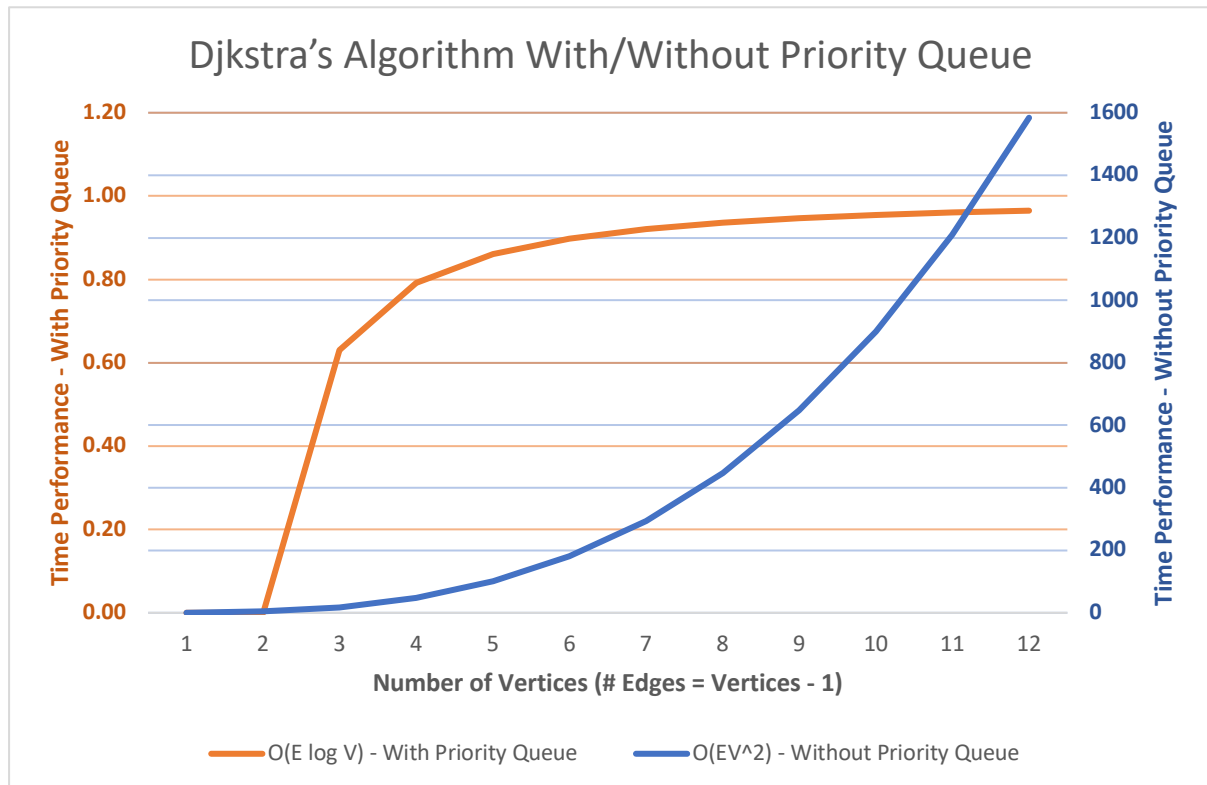


Figure 6. Big O Time Performance Comparison for Dijkstra's Algorithm With/Without Priority Queue

As shown in the above graph, the difference in performance between Dijkstra's algorithm with and without a priority queue differs greatly – note the scale on the left and right Y-axis.

As each vertices (and edge) is increased, the time increases dramatically for the $O(EV^2)$ time and only fractionally for the $O(E \log V)$ time.

For example, the increase in time performance between 1 vertex and 12 vertices without a priority queue is 1584 compared with only 0.9650 with a priority queue, a 199.756% difference [3]. In practice over a larger graph with hundreds or thousands of vertices and edges this would result in very large performance differences. The table below shows the time performance difference when searching 100 vertices (99 edges).

Type	Time Performance for 100 Vertices (99 Edges)
$O(E \log V)$ – With Priority Queue	0.997817597
$O(EV^2)$ – Without Priority Queue	990000
Percentage Difference	200% [3]

The use of a priority queue significantly increases the efficiency of the algorithm and therefore the time performance to find the shortest path is greatly improved.

Part C

1. [20 Marks]

Minimax Algorithm

The minimax algorithm is a type of backtracking algorithm used in decision making and game theory to find the optimal move for a player, that is often used in two player turn-based games [4]. The two players are called minimizer and maximizer, where the minimizer tries to get the lowest possible score, and maximizer tries to get the highest possible score.

Each possible board state has an associated value and if in a given state the maximizer has the 'upper hand' then the board score will be a positive value, with the opposite for the minimizer.

The simplified diagram below outlines this theory.

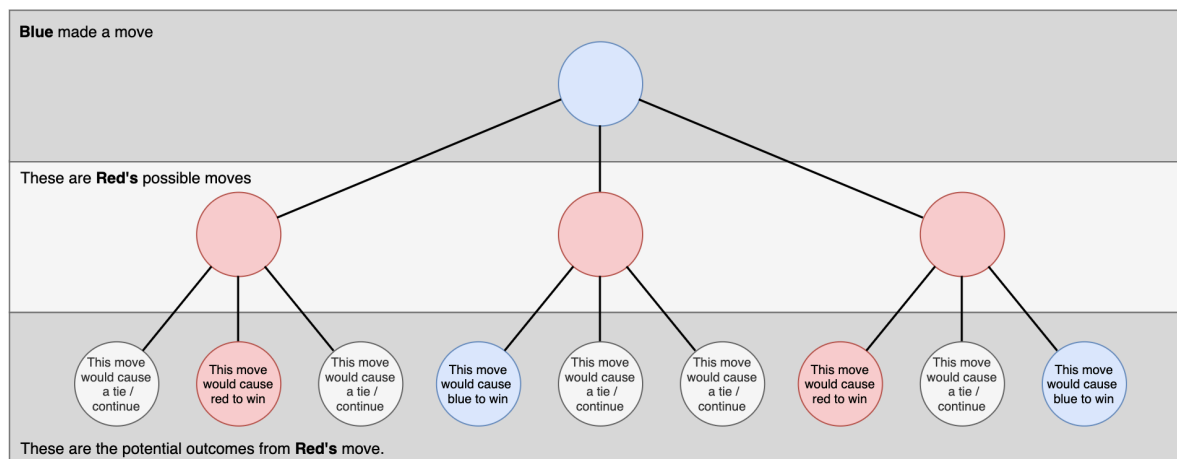


Figure 7. Simplified diagram explaining possible moves and outcomes

The pseudocode for a minimax evaluation function is as follows [5]:

```
function minimax(node, depth, maximizingPlayer):
    if depth == 0 or node is a terminal node then
        return static evaluation of node

    if MaximizingPlayer then // for Maximizer Player
        maxEvaluation = -infinity
        for each child of node do
            evaluation = minimax(child, depth-1, false)
        maxEvaluation = max(maxEvaluation, evaluation) //gives max of the values
        return maxEvaluation

    else // for Minimizer player
        minEvaluation = +infinity
        for each child of node do
            evaluation = minimax(child, depth-1, true)
        minEvaluation = min(minEvaluation, evaluation) //gives minimum of the values
        return minEvaluation
```

Note the use of recursion in the algorithm. The method will call itself with different depths until the leaf node has been reached.

To demonstrate how this would work for a connect 4 game, consider the following diagram showing player A's first move (blue), and the possible moves player B (minimax AI) could make. Note at this point all moves would result in a neutral score – continuation of the game.

Opponent A makes the first move (Column 1) - Showing Opponent B's Possible Moves

	1	2	3	4	5	6	7
1							
2							
3							
4							
5	2B						
6	1A	2B	2B	2B	2B	2B	2B

Figure 8. Possible First Moves in Connect 4

Once the game has progressed, there may be a potentially winning situation for the minimax AI player – such as below. The yellow squares outline all possible moves for Player B, with the one circled returning a positive score (best outcome) for the maximizer (supposing player B is the maximizer and player A is the minimizer).

Possible Win for Minimax Situation

	1	2	3	4	5	6	7
1							
2							
3							
4	8B?	8B?					
5	7A	3A	8B?	8B?			8B?
6	1A	2B	4B	6B	8B?	8B?	5A

Figure 9. Possible Win for Minimax Non-human Player (Maximizer) Connect4

Implementation of Minimax Algorithm for Connect 4 Game

The full working implementation of the minimax algorithm for the connect 4 game can be found in appendix A of this document.

The implementation makes use of some functions and classes of a C# connect 4 game code previously submitted for UWTSD Software Engineering Testing and Principles Module by the author, Siobhan Stephens 1800709.

The code has been manipulated to change the game board size to 7 columns x 6 rows, and to only require input from player 1 (human) where player 2 moves are added using the minimax AI algorithm. The **minimax evaluation function in Gameplay.cs** has also been added in order to demonstrate the working algorithm, called from **Game.cs**.

The implementation is fully functioning and can be played successfully – with player 1 as the human, requiring input for each selected column, and player 2 as the ‘computer’ – the minimax AI algorithm.

Below are some screenshots of the working implementation, demonstrating the user input and also the minimax algorithm’s moves (and win):

```
> Terminal - AdvSoftDev2_MinimaxConnect4

-----
Welcome to the Connect 4 Game!

Human = 0, Computer = X
-----

Human it's your turn!

  ..1  2  3  4  5  6  7
1 |---|---|---|---|---|---|
2 |---|---|---|---|---|---|
3 |---|---|---|---|---|---|
4 |---|---|---|---|---|---|
5 |---|---|---|---|---|---|
6 |---|---|---|---|---|---|
Please select a column (1-7).
█
```

Figure 10. Start of game

Human it's your turn!

	..1	2	3	4	5	6	7
1							
2							
3							
4							
5	X						
6	0	0			X		

Please select a column (1-7).

3

Human it's your turn!

	..1	2	3	4	5	6	7
1							
2							
3							
4							
5	X						
6	0	0	0	X	X		

Please select a column (1-7).

Figure 11. Note the minimax algorithm (X) blocks Player 1 (O) from making a winning move

Human it's your turn!

	1	2	3	4	5	6	7
1							
2							
3							
4	0				X		
5	X	0			X		
6	0	0	0	X	X		

Please select a column (1-7).

2

	1	2	3	4	5	6	7
1							
2							
3					X		
4	0	0			X		
5	X	0			X		
6	0	0	0	X	X		

You Lost!!

Do you want to play again? Enter 1 for yes or 0 for no.

Figure 12. Minimax AI player winning the game with 4 in a Row

References

- [1] "Advantages and Disadvantages of Linked List - GeeksforGeeks."
<https://www.geeksforgeeks.org/advantages-and-disadvantages-of-linked-list/>
(accessed Apr. 22, 2021).
- [2] "ACCA5032_ACCB5011_Sem2_CW_2020_21_OOP_ASD_Algorithms." .
- [3] "Percentage Difference." <https://www.mathsisfun.com/percentage-difference.html>
(accessed May 03, 2021).
- [4] "Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks."
<https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/> (accessed May 03, 2021).
- [5] "Artificial Intelligence | Mini-Max Algorithm - Javatpoint."
<https://www.javatpoint.com/mini-max-algorithm-in-ai> (accessed Apr. 24, 2021).

Appendix A

Connect 4 C# Game Code with Minimax Evaluation Function.

1. Board.cs

```
using System;
using System.Collections.Generic;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    /*
     * Base class for Connect4Board. Currently empty but could be reused as part of
     other board games.
     */
    public class Board
    {
        //Add default Board Properties Here eg:
        //public Player[,] PlayerPositions { get; set; }

        //Constructor method
        public Board()
        {
            // Add default board code here
        }

        virtual public void ResetBoard()
        {
            //Add default clear array code here
        }

        virtual public void DisplayBoard()
        {
            //add default DisplayBoard code here
            Console.WriteLine("Display Board Here");
        }
    }

    // -----

    /*
     * Connect4Board derived class (child) from Board
     * Sets the specifics for the Connect4Board.
     */
    public class Connect4Board : Board
    {
        //Property PlayerPositions to store the player's positions (2D array)
        public Player[,] PlayerPositions { get; set; }
        //public property to store number of rows on the board
        public int Rows { get; set; }
        //public property to store number of columns on the board
        public int Columns { get; set; }

        /*
         * Default Connect4Board constructor.
         * Sets PlayerPositions to be a 2D array of type Player, size 6 rows by 7
columns.
         */
        public Connect4Board()
```

```

    {
        //Set board size as 6 rows, 7 columns as per requirements
        Rows = 6;
        Columns = 7;
        PlayerPositions = new Player[Rows, Columns];
    }

    /*
     * Additional constructor used for unit testing. Sets PlayerPositions to
the array parameter.
    */
    public Connect4Board(Player[,] PlayerPositionsArr)
    {
        PlayerPositions = PlayerPositionsArr;
    }

    /*
     * StoreCounter method
     * Takes parameters columnIndex and player
     * columnIndex is the literal array index (column the user enters -1)
     * player is the counter to be stored at that position.
     * The method places the counter in the lowest possible position of the
column,
     * replicating the counter 'dropping' to the bottom of the column.
    */
    public bool StoreCounter(int columnIndex, Player player)
    {
        //start from the bottom, store the player in the lowest possible
position (counter drops to the bottom)
        for (int row = Rows - 1; row > -1; row--)
        {
            if (PlayerPositions[row, columnIndex] == null)
            {
                PlayerPositions[row, columnIndex] = player; //Store the counter
                return true; //if stored, return true to the calling function
            }
        }
        return false; //if column is full, return false (will ask player to pick
another column)
    }

    /*
     * Remove Top Counter method
     * Takes column int as a parameter to be used as an array position
     * Loops to take off the top counter in that column (if there is one)
     *
    */
    public bool RemoveTopCounter(int column)
    {
        int row = 0;
        //while row < max number of rows and board[row, column] is empty
        while (row < Rows && PlayerPositions[row, column] == null)
        {
            row++; //increase row
        }

        if (row == Rows) //if max is reached and all empty, return false
        {
            return false;
        }
        //If not empty, empty it
        PlayerPositions[row, column] = null;
        return true; //return true (counter has been removed)
    }

    /*
     * ResetBoard method

```

```

        * Clears the PlayerPositions 2D array.
        * Called when the game is started (and so also if the user chooses to play
again).
    */
    override public void ResetBoard()
    {
        Array.Clear(PlayerPositions, 0, PlayerPositions.Length);
    }

    /*
    * ConvertDisplay function
    * Takes int row and int column as parameters
    * Used by DisplayBoard to convert counter colours into X and Os.
    * Relies on the default constructor setting one of the player's colours to
Red.
    * In future ConsoleColor will be used.
    */
    private string ConvertDisplay(int row, int column)
    {
        if (PlayerPositions[row, column] != null)
        {
            if (PlayerPositions[row, column].PlayerCounter.Colour ==
ConsoleColor.Red)
            {
                return "O";
            }
            else
            {
                return "X";
            }
        }
        Console.BackgroundColor = ConsoleColor.White;
        return " "; //No player at this position, print a blank space
    }

    /*
    * DisplayBoard method
    * Prints a pretty graphical representation of the board state.
    * Calls ConvertDisplay to print the X and Os representing the user's
counters
    * (As shown on requirements doc).
    * This is specific to the board size = should be made more dynamic in
future
    */
    override public void DisplayBoard()
    {
        Console.WriteLine(@"
..1  2   3   4   5   6   7 " + @"
|---|---|---|---|---|---|---|
1 | " + ConvertDisplay(0, 0) + " | " + ConvertDisplay(0, 1) + " | " +
ConvertDisplay(0, 2) + " | " + ConvertDisplay(0, 3) + " | " + ConvertDisplay(0, 4)
+ " | " + ConvertDisplay(0, 5) + " | " + ConvertDisplay(0, 6) + @" |
|---|---|---|---|---|---|---|
2 | " + ConvertDisplay(1, 0) + " | " + ConvertDisplay(1, 1) + " | " +
ConvertDisplay(1, 2) + " | " + ConvertDisplay(1, 3) + " | " + ConvertDisplay(1, 4)
+ " | " + ConvertDisplay(1, 5) + " | " + ConvertDisplay(1, 6) + @" |
|---|---|---|---|---|---|---|
3 | " + ConvertDisplay(2, 0) + " | " + ConvertDisplay(2, 1) + " | " +
ConvertDisplay(2, 2) + " | " + ConvertDisplay(2, 3) + " | " + ConvertDisplay(2, 4)
+ " | " + ConvertDisplay(2, 5) + " | " + ConvertDisplay(2, 6) + @" |
|---|---|---|---|---|---|---|
4 | " + ConvertDisplay(3, 0) + " | " + ConvertDisplay(3, 1) + " | " +
ConvertDisplay(3, 2) + " | " + ConvertDisplay(3, 3) + " | " + ConvertDisplay(3, 4)
+ " | " + ConvertDisplay(3, 5) + " | " + ConvertDisplay(3, 6) + @" |
|---|---|---|---|---|---|---|

```

```

5 | " + ConvertDisplay(4, 0) + " | " + ConvertDisplay(4, 1) + " | " +
ConvertDisplay(4, 2) + " | " + ConvertDisplay(4, 3) + " | " + ConvertDisplay(4, 4)
+ " | " + ConvertDisplay(4, 5) + " | " + ConvertDisplay(4, 6) + "@" |
|---|---|---|---|---|---|---|
6 | " + ConvertDisplay(5, 0) + " | " + ConvertDisplay(5, 1) + " | " +
ConvertDisplay(5, 2) + " | " + ConvertDisplay(5, 3) + " | " + ConvertDisplay(5, 4)
+ " | " + ConvertDisplay(5, 5) + " | " + ConvertDisplay(5, 6) + "@" |
|---|---|---|---|---|---|---|");
    }
}
}

```

2. Counter.cs

```

using System;
using System.Text;
using System.Collections.Generic;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    /*
     * Counter Class
     * Used by Connect4Game only (derived class specifics could be used later)
     */
    public class Counter
    {
        /*
         * Counter constructor
         * Takes consoleColour as parameter
         * Sets property Colour to the parameter value.
         */
        public Counter(ConsoleColor consoleColor)
        {
            Colour = consoleColor;
        }
        //Property Colour
        public ConsoleColor Colour { get; set; }
    }
}

```

3. Game.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    public class Game
    {
        public List<Player> PlayerList { get; set; }

        private Connect4Board gameBoard;

        /*
         * Game constructor
         * Creates two players, Player 1 (Red counter) and Player 2 (Blue counter)
         * Assigns a List of type player to the PlayerList property
         * Adds the two players to the list
         * Creates a new instance of Connect4Board (default constructor)
         * Creates a new instance of Connect4Gameplay (default constructor)
         */
        public Game()
        {
            Player player1 = new Player(ConsoleColor.Red, "Human", 1);
            Player player2 = new Player(ConsoleColor.Blue, "Computer", 2);

            gameBoard = new Connect4Board();
            PlayerList = new List<Player>();

            PlayerList.Add(player1);
            PlayerList.Add(player2);
        }

        /*
         * PlayGame method
         * Handles the output and input for the connect 4 game.
         * Pseudocode can be found in the documentation.
         *
         * Code modified for Advanced Software Development Part C Question.
         */
        public void PlayGame()
        {
            var random = new Random();
            Console.Clear();//clear console output
            //declare and initialise required variables
            bool playAgain = true;
            bool endGame = false;
            bool validInput = false;
            string stringInput = "";
            int intInput = 0;
            Player winner = null;
            gameBoard.ResetBoard();//call reset board to begin

            //Loop through the 'welcome to game' while playAgain = true - the user
            //if they want to play again once the game has ended.
            while (playAgain == true)
            {
                Console.WriteLine("- - - - -");
            }
        }
    }
}
```

```

        Console.WriteLine("Welcome to the Connect 4 Game!");
        Console.WriteLine("\nHuman = 0, Computer = X\n");

        //Loop while endGame == false (game has not been won and game board
isn't full)
        while (endGame == false)
        {
            int initDepth = 6;//initial depth used in minimax algorithm
            var minimaxMoves = new List<Tuple<int, int>>();//used to store
minimax moves
            //Loop for each player (Will be player 1 and player 2, default
names set).
            foreach (Player player in PlayerList) {
                //if player 1 (human) ask for input
                if (player.ID == 1)
                {
                    Console.WriteLine("- - - - -");

                    //Print the player's name and "it's your turn".
                    Console.WriteLine("\n" + player.Name + " it's your
turn!");

                    /*
                     * Display the current board state
                     * On first iteration the board will be empty.
                     * After this, the board will show either an X or a 0
representing the players counters.
                     */
                    gameBoard.DisplayBoard();
                    //While NOT validInput
                    validInput = false;//reset variable
                    while (!validInput)
                    {
                        stringInput = ""; intInput = 0;//reset variables
                        //Ask user for
input of a number from 1 to max columns.
                        Console.WriteLine("Please select a column (1-" +
gameBoard.Columns + ").");
                        //Read user input into variable stringInput (type
string)
                        stringInput = Console.ReadLine();

                        //Call validation static method StringIsNumeric
                        //If the string is numeric (can be converted to an
int) continue
                        if (Validation.StringIsNumeric(stringInput))
                        {
                            try
                            {
                                //Convert to int wrapped in try, in case a
decimal number is entered which would throw exception.
                                intInput = Convert.ToInt32(stringInput);

                                //Call validation static method
                                //to be tested. Column should be 1 to max
                                //number of columns
                                if (Validation.IntWithinRange(intInput, 1,
gameBoard.Columns))
                                {
                                    if (!gameBoard.StoreCounter(intInput -
1, player))//if counter not stored successfully (col full)
                                    {
                                        Console.WriteLine("Sorry, that
column is full. Please enter another column (1-" + gameBoard.Columns + ").");

```

```

        }
        else //it is stored successfully
        {
            validInput = true;
        }
    }
    else //if the number entered wasn't between
1 and maxcols.
    {
        Console.WriteLine("Sorry, that wasn't
between 1 and 6, please enter a number between 1 and " + gameBoard.Columns + ".");
    }
}
catch (Exception)//Exception could be thrown if
decimal was entered
{
    Console.WriteLine("Error, please enter a
number between 1 and " + gameBoard.Columns + ".");
}
}
else //if the string cannot be converted to an int,
ask for input again
{
    Console.WriteLine("Error, please enter a
number between 1 and " + gameBoard.Columns + ".");
}
}

//Call CheckForGameWin function to see if there is 4 in
a row in any direction.
//Will return null if no winner, or a winner of type
Player if there is a winner
winner = Connect4Gameplay.CheckForGameWin(gameBoard,
PlayerList);

//if there is a winner, and the winner is the human
player (first in the list)
if (winner == PlayerList[0])
{
    //display the winning board, and print 'You win!'
    gameBoard.DisplayBoard();
    Console.WriteLine("You win!");
    endGame = true;//terminating loop variable
    break;
}

//if the board is full and there is no winner (checked
above)
if (Connect4Gameplay.BoardIsFull(gameBoard))
{
    gameBoard.DisplayBoard();
    Console.WriteLine("Tie!");//print tie if the board
is full

    endGame = true;//terminating loop variable
    break;
}
}

//-----

//if the player is the computer (AI) player
if (player.ID == 2)
{
    //Loop through each column
    for (int j = 0; j < gameBoard.Columns; j++)
    {
        //if the column is full, continue loop
        if (!gameBoard.StoreCounter(j, player))

```

```

        {
            continue;
        }
        //otherwise call the Minimax function with the
initial depth of 6 and store the maxEvaluation in the minimaxMoves list
        minimaxMoves.Add(Tuple.Create(j,
Connect4Gameplay.MiniMax(initDepth, gameBoard, false, PlayerList)));
        gameBoard.RemoveTopCounter(j); //remove the top
counter from that column
    }
    //get the max move score from the minimaxMoves list and
store
        int maximumMoveScore = minimaxMoves.Max(t => t.Item2);
        //get the best minimaxMoves for the best score in the
minimaxMoves list
        var bestMoves = minimaxMoves.Where(t => t.Item2 ==
maximumMoveScore).ToList();
        //Make that best move calculated by the minimax
function by storing the counter for the non-human player 2,
        //use the random function to choose the column from the
list of best moves (as there may be multiple so
        //avoid selecting first column all the time)
        gameBoard.StoreCounter(bestMoves[random.Next(0,
bestMoves.Count)].Item1, PlayerList[1]);

        //Check for 4 in a row in any direction, returns a
Player (winner) or null if no winner
        winner = Connect4Gameplay.CheckForGameWin(gameBoard,
PlayerList);

        //if there is a winner and it is player 2 (non-human)
        if (winner == PlayerList[1])
        {
            gameBoard.DisplayBoard();
            Console.WriteLine("You Lost!!"); //print you lost
            endGame = true; //terminating loop condition
            break;
        }
        //check if the game is full, if it is, print tie and
end the game
        if (Connect4Gameplay.BoardIsFull(gameBoard))
        {
            gameBoard.DisplayBoard();
            Console.WriteLine("Tie!");
            endGame = true;
            break;
        }
    }

    //-----

    }
}
//While NOT validInput
validInput = false; //reset variable
while (!validInput)
{
    stringInput = ""; intInput = 0; //reset variables
    //Ask user if they want to play again
    Console.WriteLine("Do you want to play again? Enter 1 for yes
or 0 for no.");
    //Read user input into stringInput variable
    stringInput = Console.ReadLine();

    //Call validation static method StringIsNumeric with the string
input.
    //If the string is numeric (can be converted to an int) continue
    if (Validation.StringIsNumeric(stringInput))

```

```

        {
            try
            {
                //Convert to int wrapped in try, in case a decimal
                number is entered.
                intInput = Convert.ToInt32(stringInput);
                //Call validation static method IntWithinRange with the
                integer value, and the range
                //to be tested. Value should be either 0 or 1.
                if (intInput == 0 || intInput == 1)
                {
                    validInput = true; //valid input, passed test
                }
            }
            catch (Exception) //Ask for new input
            {
                Console.WriteLine("Error, please enter 1 for yes, or 0
                for no.");
            }
        }
        else //Ask for new input
        {
            Console.WriteLine("Error, please enter 1 for yes, or 0 for
            no.");
        }
    }

    if (intInput == 0) //If user entered no (they do not want to play
    again)
    {
        //Set playAgain = false to exit loop
        playAgain = false;
        Console.WriteLine("Thanks for playing Connect4!");
    }
    else if (intInput == 1) //User wants to play again
    {
        playAgain = true; //stay within loop
        endGame = false; //reset endGame variable
        gameBoard.ResetBoard(); //Reset the game board
        (PlayerPositions)
    }
}
}
}

```

4. Gameplay.cs

```
using System;
using System.Collections.Generic;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    public class Gameplay
    {
        public Gameplay()
        {
        }

    }

    public class Connect4Gameplay : Gameplay
    {
        public Connect4Gameplay()
        {
        }

        /**
         * MiniMax Function
         * Uses recursion to call itself with different depths to find the min/max
values depending
         * on whether this is called for the maximizing or minimizing player.
         * Pseudocode can be found in the assignment document.
         */
        public static int MiniMax(int depth, Connect4Board gameBoard, bool
maximizingPlayer, List<Player> PlayerList)
        {
            // Terminating condition - eg the leaf node has been reached
            if (depth <= 0)
                return 0; //return static evaluationuation

            //Check for a winner using existing CheckForGameWin function (used in
Software Engineering Testing and Principles module)
            Player winner = CheckForGameWin(gameBoard, PlayerList);

            //If function returns a player (there is a winner)
            if(winner != null)
            {
                if (winner.ID == 2)//Maximizer
                {
                    return depth;//
                }
                else if (winner.ID == 1)//Minimizer
                {
                    return -depth;
                }
            }

            //If the gameboard is full, return a tie (already checked for winner
and there was none)
            if (BoardIsFull(gameBoard)){
                return 0;//if tie return 0 (end/tie)
            }

            int maxEvaluation = maximizingPlayer ? -1 : 1;//positive score for
maximizer, negative score for minimizer
        }
    }
}
```

```

        for (int i = 0; i < gameBoard.Columns; i++)//loop through gameboard
columns
    {
        //If it is the maximising player (non-human), try to store counter,
        else store counter for the 'human' player
        if (!gameBoard.StoreCounter(i, maximizingPlayer ? PlayerList[1] :
PlayerList[0]))
        {
            continue;
        }
        //Call minimax function (Recursion) with new depth (depth - 1) and
        opposite of current player
        int evaluation = MiniMax(depth - 1, gameBoard, !maximizingPlayer,
PlayerList);
        //If current move is maximizer, find the maximum attainable value,
        else (if current move is Minimizer),
        //find the minimum attainable value
        maxEvaluation = maximizingPlayer ? Math.Max(maxEvaluation,
evaluation) : Math.Min(maxEvaluation, evaluation);

        gameBoard.RemoveTopCounter(i);
    }
    //return max evaluation value
    return maxEvaluation;
}

/*
 * CheckForGameWin Function - as used in Software Engineering Principles
and Testing Module
 * Returns Player (either as null or as the winning player)
 * Takes the gameBoard (containing player positions) and the PlayerList as
parameters
 *
 * The function checks the four directions -
 * horizontal, vertical, ascending diagonal (right to left downwards) and
 * descending diagonal (right to left upwards for 4 in a row (connect 4
game win).
 *
 * If four in a row, the winner is returned to the calling function. If no
 * winner after all possibilities have been checked (66 possible moves), null
is returned.
 *
 * See documentation for full pseudocode and explanation of loops.
 */

public static Player CheckForGameWin(Connect4Board gameBoard, List<Player>
PlayerList)
{
    foreach (Player player in PlayerList)
    {
        //Throughout the following code:
        //i = row
        //j = col

        //Horizontal Row Check
        //See documentation for full explanation of code
        for (int i = 0; i < gameBoard.PlayerPositions.GetLength(0); i++)
        {
            for (int j = 0; j < gameBoard.PlayerPositions.GetLength(1) - 3;
j++)
            {
                if (gameBoard.PlayerPositions[i, j] == player &&
gameBoard.PlayerPositions[i, j + 1] == player && gameBoard.PlayerPositions[i, j +
2] == player && gameBoard.PlayerPositions[i, j + 3] == player)
                {
                    return player;
                }
            }
        }
    }
}

```

```

        }
    }
    //Vertical Check
    //See documentation for full explanation of code
    for (int j = 0; j < gameBoard.PlayerPositions.GetLength(1); j++)
    {
        for (int i = 0; i < gameBoard.PlayerPositions.GetLength(0) - 3;
i++)
        {
            if (gameBoard.PlayerPositions[i, j] == player &&
gameBoard.PlayerPositions[i + 1, j] == player && gameBoard.PlayerPositions[i + 2,
j] == player && gameBoard.PlayerPositions[i + 3, j] == player)
            {
                return player;
            }
        }
    }
    //Descending Diagonal Check (Right to left, downwards)
    //See documentation for full explanation of code
    for (int j = 3; j < gameBoard.PlayerPositions.GetLength(1); j++)
    {
        for (int i = 0; i < gameBoard.PlayerPositions.GetLength(0) - 3;
i++)
        {
            if (gameBoard.PlayerPositions[i, j] == player &&
gameBoard.PlayerPositions[i + 1, j - 1] == player && gameBoard.PlayerPositions[i +
2, j - 2] == player && gameBoard.PlayerPositions[i + 3, j - 3] == player)
            {
                return player;
            }
        }
    }
    //Ascending Diagonal Check (Right to left, upwards)
    //See documentation for full explanation of code
    for (int j = 3; j < gameBoard.PlayerPositions.GetLength(1); j++)
    {
        for (int i = 3; i < gameBoard.PlayerPositions.GetLength(0);
i++)
        {
            if (gameBoard.PlayerPositions[i, j] == player &&
gameBoard.PlayerPositions[i - 1, j - 1] == player && gameBoard.PlayerPositions[i -
2, j - 2] == player && gameBoard.PlayerPositions[i - 3, j - 3] == player)
                return player;
        }
    }
    }
    return null;//Return null - no winner
}

```

```

/*
 * BoardIsFull Function
 * Returns a boolean (true is end of game (board is full, no winner), false
if not)
 * Takes the gameBoard (containing player positions) as parameter
 *
 * The function checks the top row of the board (row 0) to check if all the
columns are full.
 */
public static bool BoardIsFull(Connect4Board gameBoard)
{
    for (int col = 0; col < gameBoard.PlayerPositions.GetLength(1); col++)
    {
        //top row index = 0
        //If top row, col empty then board is not full, return false
        if (gameBoard.PlayerPositions[0, col] == null)
        {
            return false;

```

```
        }
        //Later, more complex checking could be added to check for a no-
win game sooner than a full board,
        //Eg, when a win is not possible.
    }
    return true;
}
}
```

5. Player.cs

```
using System;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    /*
     * Player Class
     */
    public class Player
    {
        public Counter PlayerCounter { get; set; }
        public String Name { get; set; }
        public int ID { get; set; }
        /*
         * Player constructor
         * Takes colour and name as parameters
         * Sets counter to colour and name to name parameter
         */
        public Player(ConsoleColor colour, string name, int playerId)
        {
            PlayerCounter = new Counter(colour);
            Name = name;
            ID = playerId;
        }
    }
}
```

6. Program.cs

```
using System;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    /*
     * Solution has been implemented with C# naming conventions found here:
     * https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md
     */

    class Program
    {
        /*
         * Main method to start the game.
         * Create new instance of Game class and call PlayGame method.
         * The MVP has been implemented as per the user stories.
         * Note that the Check for Capture and Check for Destroy
         methods/functionality have
         * not been included in this game.
         */
        static void Main(string[] args)
        {
            Game game = new Game();
            game.PlayGame();
        }
    }
}
```

```
}
```

7. Validation.cs

```
using System;
using Microsoft.VisualBasic;

/* Siobhan Kelly Stephens - 1800709
 * Originally Submitted for Software Engineering Principles and Testing Assignment
 */

namespace Connect4Game
{
    public class Validation
    {
        /*
         * StringIsNumeric
         *
         * IsNumeric returns True if the data type of Expression is Boolean,
         * Byte, Decimal, Double, Integer, Long, SByte, Short, Single, UInteger,
         * ULong, or UShort. It also returns True if Expression is a Char,
         * String, or Object that can be successfully converted to a number.
         * Expression can contain non-numeric characters.
         * IsNumeric returns True if Expression is a string that contains a
         * valid hexadecimal or octal number. IsNumeric also returns True if
         * Expression contains a valid numeric expression that begins with a
         * + or - character or contains commas.
         *
         * [https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualbasic.information.isnumeric?view=net-5.0#Microsoft_VisualBasic_Information_IsNumeric_System_Object_]
         *
         * @param string value - value to be compared (gathered from user input)
         * @return boolean - true (valid integer) or false (invalid integer)
         */
        public static bool StringIsNumeric(string value)
        {
            //Test if the string is numeric using C# isNumeric
            return Information.IsNumeric(value);
        }

        /*
         * IntWithinRange
         * @param int value - value to be compared
         * @return boolean - true (valid integer within range) or false (invalid
integer not within range)
         */
        public static bool IntWithinRange(int value, int min, int max)
        {
            //If value is less than the minimum range, or greater than the maximum
range, return false.
            if(value < min || value > max)
            {
                return false;
            }
            //If passed the test, return true - it is valid.
            return true;
        }
    }
}
```