KNIGHT'S TOUR

PROG32356 – Assignment 1

Michal Zarnowski
991552312

Contents

Strategies for tour approaches

Before the tour can be taken, three different boards are initialized, and a Knight piece is initialized with his possible moves. The initialized boards include:

- A board which holds coordinates of each board landing spot in the format of [x, y]
- A board which holds the values of step number taken by the Knight
- A board which holds the accessibility values for the intelligent approach

All three of these boards are 8x8 to represent a real chess board.

Non-intelligent approach

Firstly, the algorithm will handle marking Knight's first spot by:

1. Place value of '1' on the board which holds the step number values on the coordinates chosen by the user
2. Replacing coordinates of current spot on the board which holds traversable coordinate values with unreachable coordinates for the Knight ensuring the spot can't be landed on again.

The algorithm then determines next possible move before taking it by taking the following steps:

3. Based on the Knight's current position, create arrays of coordinates of all board spaces reachable by the Knight.
4. Choose all the coordinate arrays which contain coordinates of still available board spaces (not already visited) and add them to a List which holds still available coordinate values.
5. Generate a random number ranging from 0 to the length of the List storing available move coordinates.
6. Provide the algorithm with the move chosen at random based on the previously generated random number.
7. If there are no moves left available, return a value to the algorithm which will be recognized as 'no spaces available'. If so, this is the end of the run, next move won't be made, and the results will be displayed.

Once the move is determined, it's time to take it:

8. Place the Knight on the new space acquired from point 4.
9. Find the coordinates of Knight's new spot and update the board holding the step number values with appropriate step number value.
10. Update the board which holds the coordinate values with an "unreachable value" so that the Knight can never land there again.
11. Return to point 1 to repeat the cycle.

Intelligent (heuristic) approach

Firstly, the algorithm will handle marking Knight's first spot by:

1. Place value of '1' on the board which holds the step number values on the coordinates chosen by the user
2. Replacing coordinates of current spot on the board which holds traversable coordinate values with unreachable coordinates for the Knight ensuring the spot can't be landed on again.

Secondly, the board holding accessibility values must be adjusted based on the first spot:

3. Get coordinates of all possible spots reachable by the Knight.
4. Based on those coordinates, adjust the values on the accessibility board, corresponding to the coordinate board, by subtracting one from the accessibility number if it's more than or equal to zero (it's still reachable). The number represents number of spots from which the spot can be reached.

The algorithm then determines next possible move before taking it by taking the following steps:

5. Based on the Knight's current position, create arrays of coordinates of all board spaces reachable by the Knight.
6. Choose all the coordinate arrays which contain coordinates of still available board spaces (not already visited) and add them to a List which holds still available coordinate values.
7. Get accessibility values of all reachable spots and place them inside a List.
8. Based on all accessibility values inside the List, find the smallest value and add its List index (position) to a new List. If multiple occurrences of the smallest value exist, add indexes of all of them to the new List.
9. Generate a random number ranging from 0 to the length of the List storing indexes of smallest accessible values and use the number to choose one of the indexes at random.
10. With the chosen random index, return the possible move from the List of all possible moves using that index (listOfPossibleMoves[randomlyChosenIndex]).
11. If there are no moves left available, return a value to the algorithm which will be recognized as 'no spaces available'. If so, this is the end of the run, next move won't be made, and the results will be displayed.

Once the move is determined, it's time to take it:

12. Place the Knight on the new space acquired from point 8.
13. Find the coordinates of Knight's new spot and update the board holding the step number values with appropriate step number value.
14. Update the board which holds the coordinate values with an "unreachable value" so that the Knight can never land there again.

Once the move is taken, the accessibility board must be adjusted. Repeat steps 3 and 4. If point 11 (end of run) wasn't reached, go back to step 1.

Program implementation

Data structures

My implementation consists of a variety of different data structures including:

- Single dimension integer array – used for the board which holds the accessibility value matrix. I decided to use a simple array because I knew the starting values for each array position, and I knew that the size of the array will never change.
- Multi-dimension integer array – holds the values for moves of a Knight chess piece for example [2,1] represents a Knight moving up 2 spaces and 1 across.
- Lists holding integer values – used throughout the whole implementation for functionality such as storing indexes of all possible moves. I used Lists instead of regular arrays because of the varying size nature of the structures. More indexes were being added to the list on the fly therefore an array of a fixed size would not suffice.
- List holing string values – used once in the Results class to hold the values of how many board spots were reached by the Knight on each run. The list is of type string and not integer as sometimes the knight will reach all the spots on the board and instead of displaying 64, I decided to go with the word “all”. Once again, I went with a List instead of an array due to varying size of the List.
- Lists holding arrays of integers – used on a few occasions, example being in the Results class to hold arrays containing traversed boards, more precisely, the values of the order the Knight landed on each spot
- Properties – used for most of all class properties for easy access of the variables without getter methods.

Implementation

My program is implemented using the Windows Forms module providing a GUI for the user to interact with. The GUI consists of radio buttons and number controls to enter the parameters of the Knight’s tour. The results are displayed using labels and text boxes populated automatically by the program once execution is complete.

The implementation is divided into several classes responsible for different functions:

- Form1 – serves the purpose of holding the main method and is the brain of the whole program. Its used to communicate with all other classes and handle the logic of the implementation. It also includes implementations of form’s control’s interaction methods.
- Form2 – second form accessible by the user by the click of the “help” button on form 1. This class holds the logic for displaying help information to the user upon its birth.
- ChessPiece – parent class utilizing OOP’s principle of inheritance. It consists of implementation of range check method to validate coordinates of the move taken by any chess piece.
- Knight – child class of the ChessPiece class used to model a Knight chess piece. Its properties include current x and y coordinates, as well as the possible moves allowed for the Knight.

- GameBoard – class utilizing another OOP principle called abstraction. Serves the purpose of a model for any type of game board requiring all classes using it to implement methods relative to all possible game boards.
- ChessBoard – class implementing the abstract class GameBoard. It overrides GameBoard's methods of showBoard to show coordinates of each spot and showTraversedBoard showing the board containing step numbers taken by the Knight. This class also implements its own method showing the accessibility matrix. This method is only relative to chess hence it's not included in the GameBoard class.
- Results – used to hold game results including number of spots reached on each run and the traversed board records of each run.

Source code

Form 1 (main)

```
using System;
using System.IO;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace KnightsTourForm
{
    6 references
    public partial class Form1 : Form
    {
        private static Random random = new Random();
        private static int moveCounter = 1, runCounter = 1, displayCounter;
        private static int startX, startY;
        private static Results results;
        2 references
        public Form1()
        {
            InitializeComponent();
        }

        1 reference
        private void Form1_Load(object sender, EventArgs e)
        {
            // Generate a Results object to store game results
            results = new Results();
        }

        1 reference
        private void startBtn_Click(object sender, EventArgs e)
        {
            // Get starting position from the user
            startX = Convert.ToInt32(rowCoordinate.Value) - 1;
            startY = Convert.ToInt32(colCoordinate.Value) - 1;

            // Get approach and attempts amount from the user
            int choice;
            if (nonIntelRadioBtn.Checked == true)
            {
                choice = 1;
            }
            else
            {
                choice = 2;
            }
            int attempts = Convert.ToInt32(attemptsEntry.Value);

            // Run program for the amount of times entered
            for (int i = 0; i < attempts; i++)
            {
```

```
            // Run approach based on choice
            switch (choice)
            {
                case 1:
                    runNonIntelligent();
                    break;
                case 2:
                    runIntelligent();
                    break;
                default:
                    Console.WriteLine("Invalid choice!");
                    break;
            }

            // Write run results to file
            writeResultsToFile(choice);

            // Adjust conuters for next run
            moveCounter = 1;
            runCounter++;
        }

        // Remove begin button
        Controls.Remove(startBtn);

        // Display results
```

```
            displayCounter = 1;
            runNoLbl.Text = "1";
            touchedLbl.Text = results.CompletedMovesResults[0];
            foreach (Control c in Controls)
            {
                if (c is TextBox)
                {
                    int location = Convert.ToInt32(c.Name.Substring(c.Name.Length - 2));
                    c.Text = results.GameResults[0][location].ToString();
                }
            }

            // Enable buttons to cycle through tours
            nextTourBtn.Visible = true;
            prevTourBtn.Visible = true;

            // Display averge and deviation
            int avg = getAverageRuns();
            int completedMoves = results.CompletedMovesResults[0] == "all" ?
                64 : Convert.ToInt32(results.CompletedMovesResults[0]);
            avgLbl.Text = "Average achieved steps: " + avg.ToString();
            deviationLbl.Text = "Deviation from average: " + (completedMoves - avg).ToString();
        }

        1 reference
        static void runNonIntelligent()
        {
            Console.WriteLine("Run started: " + runCounter);
            ChessBoard board = new ChessBoard();
            Knight knight = new Knight(startX, startY);

            // Handle starting position
            handleStartingPosition(board, knight);
```

```
            while (true)
            {
                // Determine next move
                int[] nextMove = determineNextMove(board, knight, 1);

                // If returned move is out of bounds, finish the run
                if (nextMove[0] == 9)
                {
                    break;
                }

                // Make the move
                makeMove(board, knight, nextMove);
            }
            board.showTraversedBoard();

            // Save game results
            saveGameResult(board);

        }
```

```
1 reference
static void runIntelligent()
{
    Console.WriteLine("Run started: " + runCounter);
    ChessBoard board = new ChessBoard();
    Knight knight = new Knight(startX, startY);

    // Handle starting position
    handleStartingPosition(board, knight);

    // Adjust accessibilty matrix based on starting position
    adjustAccessibilityMatrix(board, knight);

    while (true)
    {
        // Determine next move
        int[] nextMove = determineNextMove(board, knight, 2);

        // If returned move is out of bounds, finish the run
        if (nextMove[0] == 9)
        {
            break;
        }

        // Make the move
        makeMove(board, knight, nextMove);

        // Adjust accessibilty matrix based on new position
        adjustAccessibilityMatrix(board, knight);
    }
    board.showTraversedBoard();

    // Save game results
    saveGameResult(board);

}

2 references
static void handleStartingPosition(ChessBoard board, Knight knight)
{
    // Mark starting position on the board
    int[] startPosition = { knight.X, knight.Y };
    int indexOfStartPosition = board.Positions.FindIndex(startPosition.SequenceEqual);
    board.TraversedPositions[indexOfStartPosition] = moveCounter++;

    // Mark starting position on the board as used
    board.Positions[indexOfStartPosition] = new int[] { 99, 99 };
}
```

```
2 references
static int[] determineNextMove(ChessBoard board, Knight knight, int runMethod)
{
    // Create a list to store all possible next moves
    List<int[]> possibleMoves = new List<int[]>();

    // Loop through Knight's all possible moves
    for (int i = 0; i < knight.Moves.Length / 2; i++)
    {
        // Determine next possible move based on current position
        int nextPossibleX = knight.X + knight.Moves[i, 0];
        int nextPossibleY = knight.Y + knight.Moves[i, 1];
        int[] nextPossibleMove = { nextPossibleX, nextPossibleY };

        // Add next possible move to List of possible moves if it's available on the board
        if (board.Positions.Any(a => a.SequenceEqual(nextPossibleMove)))
        {
            possibleMoves.Add(nextPossibleMove);
        }
    }
```

```
            // Return next move based on game approach (non-intelligent/intelligent)
            switch (runMethod)
            {
                case 1:
                    // Generate random number based on number of possible moves
                    int randomIndex = random.Next(0, possibleMoves.Count);

                    // If any moves are possible, return a random one from the list of possible moces
                    if (possibleMoves.Count > 0)
                    {
                        return possibleMoves[randomIndex];
                    }
                    break;
                case 2:
                    // If any moves possible
                    if (possibleMoves.Count > 0)
                    {
                        // GET ACCESSIBILTY VALUES OF ALL BOARD POSITIONS REACHABLE BY THE KNIGHT
                        List<int> accessValsBasedOnIndexes = new List<int>();
                        int smallestIndex;

                        // Loop through possible moves
                        for (int i = 0; i < possibleMoves.Count; i++)
                        {
                            // Get relative board index of the possible move
                            int indexOfPossibleMove = board.Positions.FindIndex(possibleMoves[i].SequenceEqual);
                            // Add accessibilty matrix value of that board position into the list
                            accessValsBasedOnIndexes.Add(board.PositionsAccessibilty[indexOfPossibleMove]);
                        }

                        // DETERMINE ACCESSIBLE BOARD POSITION BASED ON SMALLEST ACCESSIBILTY VALUE
                        smallestIndex = accessValsBasedOnIndexes[0];

                        // Loop through all acquired accessibilty values to find the smallest value
                        for (int i = 0; i < accessValsBasedOnIndexes.Count; i++)
                        {
                            // If accessibilty value is smaller than currently assigned one,
                            // change it to current accessibilty value
                            if (accessValsBasedOnIndexes[i] < smallestIndex)
                            {
                                smallestIndex = accessValsBasedOnIndexes[i];
                            }
                        }

                        // Loop through all acquired accessibilty values to find all values equal to the
                        // smallest accessibilty value and add their List indexes to the list
                        List<int> indexesOfSmallestValues = new List<int>();
                        for (int i = 0; i < accessValsBasedOnIndexes.Count; i++)
                        {
                            if (accessValsBasedOnIndexes[i] == smallestIndex)
                            {
                                indexesOfSmallestValues.Add(i);
                            }
                        }
```

```
                        // Get randomly selected index of one of the smallest accessibility values
                        int randomVal = random.Next(0, indexesOfSmallestValues.Count);
                        int selectedIndex = indexesOfSmallestValues[randomVal];

                        // Return the move with the randomly selected smallest accessibilty value
                        return possibleMoves[selectedIndex];
                    }
                    break;
                default:
                    break;
            }

            // If no moves were possible, return and out of bounds move
            return new int[] { 9, 9 };
        }
```

```
2 references
static void makeMove(ChessBoard board, Knight knight, int[] move)
{
    // Place knight on new position
    knight.X = move[0];
    knight.Y = move[1];

    // Mark position with move counter
    int indexOfMovePosition = board.Positions.FindIndex(move.SequenceEqual);
    board.TraversedPositions[indexOfMovePosition] = moveCounter++;

    // Remove position from board positions
    board.Positions[indexOfMovePosition] = new int[] { 99, 99 };
}

1 reference
static void writeResultsToFile(int option)
{
    // Determine file name based on run approach
    string fileName = "";
    if (option == 1)
    {
        fileName += "michalzarnowskiNonIntelligentMethod.txt";
    }
    else if (option == 2)
    {
        fileName += "michalzarnowskiHeuristicMethod.txt";
    }

    // Compose path to file
    var path = System.AppContext.BaseDirectory;
    var filePath = path + fileName;

    // Determine amount of completed moves by the Knight
    String completedMoves = (moveCounter - 1).ToString();
    if (moveCounter - 1 == 64)
    {
        completedMoves = "all";
    }

    // Compose the line to save in the text file
    string printLine = "Trial " + runCounter + ": The Knight was able to successfully touch " + completedMoves + " squares.";

    // Save line to file
    TextWriter writer = new StreamWriter(filePath, true);
    writer.WriteLine(printLine);
    writer.Close();

    // Save game results
    results.addCompletedMovesResult(completedMoves);
}
```

```
2 references
static void adjustAccessibilityMatrix(ChessBoard board, Knight knight)
{
    int[] currentPosition = { knight.X, knight.Y };

    // Loop through Knight's all possible moves
    for (int i = 0; i < knight.Moves.Length / 2; i++)
    {
        // Get next possible move in array format
        int nextPossibleX = knight.X + knight.Moves[i, 0];
        int nextPossibleY = knight.Y + knight.Moves[i, 1];
        int[] nextPossibleMove = { nextPossibleX, nextPossibleY };

        // Find the index of the next move in board's position matrix
        int indexOfPossibleMovePosition = board.Positions.FindIndex(nextPossibleMove.SequenceEqual);

        // If next move possible (not -1), adjust its accessibilty value by subtracting 1
        if (indexOfPossibleMovePosition >= 0)
        {
            board.PositionsAccessibilty[indexOfPossibleMovePosition] = board.PositionsAccessibilty[indexOfPossibleMovePosition] - 1;
        }

    }
}
```

```
2 references
static void saveGameResult(ChessBoard board)
{
    int[] resultsArr = board.TraversedPositions.ToArray();
    results.addGameResult(resultsArr);

}

1 reference
private void runNoLbl_Click(object sender, EventArgs e)
{

}

1 reference
private void textBox2_TextChanged(object sender, EventArgs e)
{

}

1 reference
private void button1_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.ShowDialog();
}
```

```
private void prevTourBtn_Click(object sender, EventArgs e)
{
    if (displayCounter > 1)
    {
        // Display results based on adjustd display counter
        displayCounter--;
        displayResults();
    }
}

1 reference
private void nextTourBtn_Click(object sender, EventArgs e)
{
    if (results.AmountOfStoredResults > displayCounter)
    {
        // Display results based on adjustd display counter
        displayCounter++;
        displayResults();
    }
}

2 references
void displayResults()
{
    runNoLbl.Text = displayCounter.ToString();
    touchedLbl.Text = results.CompletedMovesResults[displayCounter - 1];
    foreach (Control c in Controls)
    {
        if (c is TextBox)
        {
            int location = Convert.ToInt32(c.Name.Substring(c.Name.Length - 2));
            c.Text = results.GameResults[displayCounter - 1][location].ToString();
        }
    }

    // Display deviation from average
    int completedMoves = results.CompletedMovesResults[displayCounter - 1] == "all" ?
    64 : Convert.ToInt32(results.CompletedMovesResults[displayCounter - 1]);
    deviationLbl.Text = "Deviation from average: " + (completedMoves - getAverageRuns()).ToString();
}
```

```
        static int getAverageRuns()
        {
            int sumOfMoves = 0;
            for(int i = 0; i < results.CompletedMovesResults.Count; i++)
            {
                if (results.CompletedMovesResults[i] == "all")
                {
                    sumOfMoves += 64;
                }
                else
                {
                    sumOfMoves += Convert.ToInt32(results.CompletedMovesResults[i]);
                }
            }

            return sumOfMoves / results.CompletedMovesResults.Count;
        }


        1 reference
        private void label3_Click(object sender, EventArgs e)
        {

        }

        1 reference
        private void label4_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Form 2 (help display)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace KnightsTourForm
{
    5 references
    public partial class Form2 : Form
    {
        1 reference
        public Form2()
        {
            InitializeComponent();
        }

        1 reference
        private void Form2_Load(object sender, EventArgs e)
        {
            backgroundInfoTxt.Text = "A knight's tour is a sequence of moves of a knight on a chessboard" +
                " such that the knight visits every square exactly once. If the knight ends on a square that" +
                " is one knight's move from the beginning square (so that it could tour the board again" +
                " immediately, following the same path), the tour is closed; otherwise, it is open.";

            playInfoTxt1.Text = "1) Select game approach. Non-intelligent will choose Knight's move at random," +
                " intelligent will choose next move based on accessibilty heuristic (higher chance of success).";
            playInfoTxt2.Text = "2) Select how many times you'd like the Knight to attempt the tour.";
            playInfoTxt3.Text = "3) Select the starting board position for the Knight by selecting row and column.";
            playInfoTxt4.Text = "4) Hit Begin tour!";
            playInfoTxt5.Text = "5) Results of the run will be displayed below showing the order of each move." +
                "Zeros indicate unreached positions.";
        }
    }
}
```

GameBoard (abstract)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KnightsTourForm
{
    1 reference
    abstract class GameBoard
    {
        1 reference
        public abstract void showBoard();
        3 references
        public abstract void showTraversedBoard();
    }
}
```

ChessBoard (implementing GameBoard)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KnightsTourForm
{
    10 references
    class ChessBoard : GameBoard
    {
        private int width = 8;
        private int height = 8;
        private List<int[]> positions;
        private List<int> traversedPositions;
        private int[] positionsAccessibilty = { 2, 3, 4, 4, 4, 4, 3, 2,
                                                3, 4, 6, 6, 6, 6, 4, 3,
                                                4, 6, 8, 8, 8, 8, 6, 4,
                                                4, 6, 8, 8, 8, 8, 6, 4,
                                                4, 6, 8, 8, 8, 8, 6, 4,
                                                4, 6, 8, 8, 8, 8, 6, 4,
                                                3, 4, 6, 6, 6, 6, 4, 3,
                                                2, 3, 4, 4, 4, 4, 3, 2,};

        7 references
        public List<int[]> Positions { get { return positions; } }
        3 references
        public List<int> TraversedPositions { get { return traversedPositions; } }
        3 references
        public int[] PositionsAccessibilty { get { return positionsAccessibilty; } }

        2 references
        public ChessBoard()
        {
            // Initialize positions List with board coordinates
            // and traversdedPositions List with empty spots
            positions = new List<int[]>();
            traversedPositions = new List<int>();

            for (int i = 0; i < height; i++)
            {
                for (int j = 0; j < width; j++)
                {
                    int[] tempArray = { i, j };
                    positions.Add(tempArray);

                    traversedPositions.Add(0);
                }
            }
        }

```

```
47          public override void showBoard()
48          {
49              int counter = 0, listLen = positions.Count;
50
51              while (counter < listLen)
52              {
53                  Console.Write("[{0} {1}] ", positions[counter][0].ToString(), positions[counter][1].ToString());
54                  if (counter != 0 && (counter + 1) % 8 == 0)
55                      Console.WriteLine();
56                  counter++;
57              }
58          }
59
            3 references
60          public override void showTraversedBoard()
61          {
62              int counter = 0, listLen = traversedPositions.Count;
63
64              while (counter < listLen)
65              {
66                  Console.Write("[{0}] ", traversedPositions[counter].ToString());
67
68                  // Spaces formatting
69                  if (traversedPositions[counter] / 10 < 1)
70                  {
71                      Console.Write(" ");
72                  }
73
74                  // New line formatting
75                  if (counter != 0 && (counter + 1) % 8 == 0)
76                      Console.WriteLine();
77
78                  counter++;
79              }
80          }
81
            0 references
82          public void showAccessibiltyMatrix()
83          {
84              int counter = 0, listLen = positionsAccessibilty.Length;
85
86              while (counter < listLen)
87              {
88                  Console.Write("[{0}] ", positionsAccessibilty[counter].ToString());
89
90                  // New line formatting
91                  if (counter != 0 && (counter + 1) % 8 == 0)
92                      Console.WriteLine();
93
94                  counter++;
95              }
96          }
97
98      }
99  }
```

ChessPiece (parent class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KnightsTourForm
{
    1 reference
    class ChessPiece
    {
        4 references
        public int rangeCheck(int val)
        {
            if (val < 0 || val > 7)
            {
                throw new ArgumentOutOfRangeException(val.ToString());
            }
            return val;
        }
    }
}
```

Knight (child class)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KnightsTourForm
{
    9 references
    class Knight : ChessPiece
    {
        private int x;
        private int y;
        private int[,] moves = { { 2, -1 }, { 1, -2 }, { -1, -2 }, { -2, -1 }, { -2, 1 }, { -1, 2 }, { 1, 2 }, { 2, 1 } };

        5 references
        public int X { get { return x; } set { x = rangeCheck(value); } }
        5 references
        public int Y { get { return y; } set { y = rangeCheck(value); } }
        6 references
        public int[,] Moves { get { return moves; } }

        2 references
        public Knight(int x, int y)
        {
            this.x = rangeCheck(x);
            this.y = rangeCheck(y);
        }
    }
}
```

Results

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace KnightsTourForm
{
    3 references
    class Results
    {
        private int amountOfStoredResutls;
        private List<int[]> gameResults;
        private List<string> completedMovesResults;
        1 reference
        public int AmountOfStoredResults { get { return amountOfStoredResutls; } }
        3 references
        public List<int[]> GameResults { get { return gameResults; } }
        3 references
        public List<string> CompletedMovesResults { get { return completedMovesResults; } }

        1 reference
        public Results()
        {
            amountOfStoredResutls = 0;
            gameResults = new List<int[]>();
            completedMovesResults = new List<string>();
        }

        1 reference
        public void addGameResult(int[] gameResult)
        {
            // Increase storage counter
            amountOfStoredResutls++;

            // Add supplied game result to stored results
            gameResults.Add(gameResult);
        }

        1 reference
        public void addCompletedMovesResult(string result)
        {
            completedMovesResults.Add(result);
        }
    }
}
```