Knights Tour Assignment

Assignment Two C#

Luka Necajev

ID=991475972

2020-6-30

Table Of Contents

AbsMethod

The abstract method is used to hold the values of the Intelligent and Unintelligent classes methods. The methods that can be overrided in other classes are Game, Move, CheckIfMovePossible, DsipalyBoard and Create Array

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

namespace Luka_Necajev_FormsA2
{
    abstract class AbsMethod
    {
        public abstract int[] Game(int games, int[] startPosition);
        public abstract void Move( int[] startPosition);
        public abstract int CheckIfMovePossible(int[] startPosition);

        public abstract void DisplayBoard();
```

CreateArray() Method

This method will display the board in the console.

```
        public abstract void CreateArray();


    }
}
```

Unintelligent Method pg 4-8

The Unintelligent Method is a child of the abstract AbsMethod class, it is a child because it will be incorporating multiple methods that are the same in both the intelligent and unintelligent methods. My strategy for the unintelligent method is that I will get the starting position and the number of times the game is going to be played from the user and pass it to my Game Method. With this information I will start playing the game using the "Game" method. This method acts like a buffer from the forms page to the other Intelligent methods. At the top of the class I am instantiating static variables that will be able to be accessed and store information that I want to pass to other methods. Static variables include the row and columns of the board (these can be changed if you want a different sized board), the holder variable is used to keep track of the increasing number each iteration each move, the moves array is an array of the board (it is as big as the rows and columns), I also have 2 1D arrays that will keep the possible moves on the X and Y coordinates. The Game method will start the program and it is like a main method. The Boolean gameOn is true keep playing the game, This method will call the move method which will call another method to check if there are any moves possible, if so move the Knight into a random possible index. and will keep doing this until it does not have any more spaces in its possible moves array, it will then exit out and write the results of the games to the unintelligent method file.

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

namespace Luka_Necajev_FormsA2
{
    class UnintelligentMethod:AbsMethod
    {
        static int x;
        static int y;
        private const int Row = 8;
        private const int Col = 8;
        // static int[] startPosition = new int[2];
        static int alreadyMoved = 1;
        static int holder = 1;
        //private static int MoveSymbol = 1;
        public static int[,] moves = new int[Row, Col];
        private static readonly int[] PossibleMovesX = new int[] { -1, 1, -2, 2, 1, -1,
2, -2 };
        private static readonly int[] PossibleMovesY = new int[] { 2, 2, 1, 1, -2, -2, -
1, -1 };
        //will kill the game.
        static bool gameOn = true;
        //Will move the peice once.
        static List<int> goodIndexes;
```

Unintelligent Game Method

This method will act like the main method in a program, it will get the values and call the createArray function to create the board, and then call the move class to move the knight until it cannot be moved anymore, after that this class will write the output to the file.

```
        //creates board,displays everything
        public override int[] Game(int games, int[] startPosition)
        {
            //getting x position
            //startPosition[0] = ;
            //getting y position.
            // startPosition[1] = 0;
            holder = 1;
            //# games
            //int games = 2;
            int playedGames = 0;
            //if you win
            // bool winner = false;

            gameOn = true;
            //creating a array to hold the values of each game
            String[] write = new string[games];
            int[] roundNumCounter = new int[games];
            moves[startPosition[0], startPosition[1]] = 1;
            while (playedGames < games)
            {
                CreateArray();
                moves[startPosition[0], startPosition[1]] = 1;
                //create the board.
                gameOn = true;



                //Console.WriteLine("starting position {0}, and {1}", startPosition[0],
startPosition[1]);
                //Console.WriteLine("hmove {0}, and vMove{1}", hMove, vMove);

                while (gameOn == true)
                {

                    Move(startPosition);
                    //this will add the scores to a holder to calculate the average
                    roundNumCounter[playedGames] = holder;
                    //this add items to be written to file. includes number of played
games to the file.
                    write[playedGames] = "Trial  :" + playedGames + "  was able to
successfully touch  " + holder + "  squares";
                }



                DisplayBoard();
                holder = 0;
                playedGames++;
```

```
            }
            try
            {
                System.IO.File.WriteAllLines("LukaNecajev_InteligentMethod.txt", write);
            }
            catch (Exception Ex)
            {
                Console.WriteLine("NOT WRITING");
            }
            return roundNumCounter;

        }
```

Unintelligent Move Method

This method will act like the main method in a program, it will get the starting position from the game method and check the possible moves, it will then call the “CheckIfMovePossible” function and will populate the possible moves arrays. Once that is done it will get a random index and set it to the move, and move the item.

```
        public override void Move(int[] startPosition)
        {
            //move while goodmoves moves is not ==0

            //this will check the best possible move for the knight
            int moveIndex = CheckIfMovePossible(startPosition);
            Random random = new Random();

            moveIndex = random.Next(0, goodIndexes.Count);
            if (!goodIndexes.Any())
            {
                gameOn = false;
                goodIndexes.Clear();
            }
            else
            {
                //if it meets the criteria  add one to the loop and make the move the
Move Symbol
                x = startPosition[0] + PossibleMovesX[goodIndexes[moveIndex]];
                y = startPosition[1] + PossibleMovesY[goodIndexes[moveIndex]];

                //make the starting position where the last one finished
                //Console.WriteLine("hmove {0}, and vMove{1}", hMove, vMove);
                if (((x < Row && x >= 0) && (y < Col && y >= 0)) &&
                        (moves[x, y] == 0))
                {
                    moves[x, y] = 1 + holder;
                    startPosition[0] = x;
                    startPosition[1] = y;
                    //DEBugging
                    //  Console.WriteLine("new start {0}, {1}", hMove, vMove);
                }
            }
            //adding to the move counter variable
```

```
            holder++;
        }
```

Unintelligent CheckIfMovePossible Method

This method will check if the move is possible, it will get the possible moves for the values for each possible move, it will check if it is possible and add the index to the possible moves index (goodMovesIndex)

```
        //for each possible move, it will check if it is possible and add the index to
the possible moves index (goodMovesIndex)
        public override int CheckIfMovePossible(int[] startMoves)
        {

            // will get the index of the moves
            List<int> goodMovesIndex = new List<int>();


            //int[] goodMoves = new int[8];

            //checks possible moves 8
            for (int i = 0; i < 8; i++)
            {
                int x = startMoves[0] + PossibleMovesX[i];
                int y = startMoves[1] + PossibleMovesY[i];

                //if its possible to move
                //if the moves are within the vertical borders (8 being top of the
screen/ 0 being bottom)
                //if the moves are within the horizontal borders (8 being right of the
screen/ 0 being the left)
                //if the move was made, it will have an X as we have already been there.
                if (((x < Row && x >= 0) && (y < Col && y >= 0)) &&
                    (moves[x, y] == 0))
                {

                    //if the move is possible add its index to the list.
                    goodMovesIndex.Add(i);


                    //DisplayGrid();
                    //Console.ReadKey();
                }

            }//end of checking for loop
            int moveIndex = 0;

            //getting the smalles value
            //if there are no moves in the index then kill the game
            if (!goodMovesIndex.Any())
            {
                gameOn = false;
                //clears the arrays
                goodMovesIndex.Clear();
```

```
            }
            //other wise get the smallest indexes
            else
            {
                goodIndexes = goodMovesIndex;
            }
            //make the smallest indexes = to the good moves index and pass the
            goodIndexes = goodMovesIndex;
            //return nothing due to the lists being static.
            return moveIndex;
        }
```

DisplayBoard() Method

This method will display the board in the console.

```
        //this method is overided and will display the board for the user.
        public override void DisplayBoard()
        {

            Console.Write(" ");
            //setting the letters at the top to be as long as the board
            for (int i = 65; i < Col + 65; i++)
                Console.Write(Convert.ToChar(i));

            Console.WriteLine();
            //creating the numbers on the side, and then the board using a value to
populate each one.
            for (int r = 0; r < Row; r++)
            {
                Console.Write(r + 1);
                for (int c = 0; c < Col; c++)
                {
                    //populating the board
                    Console.Write(" " + moves[r, c]);
                }
                Console.WriteLine();
            }
        }
```

CreateArray() Method

This method will display the board in the console.

```
//Overridded method Creating the initial array for the second class.
        public override void CreateArray()
        {
            for (int i = 0; i < Row; i++)
            {
                for (int c = 0; c < Col; c++)
                {
                    //adds a grid symbol (0) where we have moved at coords of row and
column
```

```
                    moves[i, c] = 0;
                }
            }
        }


    }
}
```

Intelligent Method pg 9-17

The Unintelligent Method is a child of the abstract AbsMethod class, it is a child because it will be incorporating multiple methods that are the same in both the intelligent and unintelligent methods. My strategy for the intelligent method is that I will get the starting position and the number of times the game is going to be played from the user and pass it to my Game Method. With this information I will start playing the game using the “Game” method.The game method will be working similar to the unintelligent method, and will call the move method. In the move method will check if the move is possible but will also get the possible moves and its index on the huristic board. With these arrays, I will search through the list of the good moves array and get the smallest value from the array and add it to a list in the program to keep the Index values of the smallest value/values. I then pass it back to the move method in which I will move to the smallest index or if there is more then one smallest index, randomize the move and pass the information back to the game function to print them out. In this function I also define the standard Deviation and Average functions and call them into the Forms application.

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

namespace Luka_Necajev_FormsA2
{
  class IntelligentMethod : AbsMethod
  {
    static int x;
    static int y;
    private const int Row = 8;
    private const int Col = 8;
    // static int[] startPosition = new int[2];

    static int hold = 1;
    //private static int MoveSymbol = 1;
    public static int[,] board = new int[Row, Col];
    private static readonly int[] PossibleMovesX = new int[] { -1, 1, -2, 2, 1, -1, 2, -2 };
    private static readonly int[] PossibleMovesY = new int[] { 2, 2, 1, 1, -2, -2, -1, -1 };
    //will kill the game.
    static bool gameOn = true;
    //Will move the peice once.
    static List<int> goodIndexes;
    public static int[,] huristucBoard = new int[Row, Col]
```

```
        {
         {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}
};
```

Intelligent Game Method

This method will act like the main method in a program, it will get the values and call the createArray function to create the board, and then call the move class to move the knight until it cannot be moved anymore, after that this class will write the output to the file.

```
//creates board,displays everything

    public override int[] Game(int games, int[] startPosition)
    {


      //getting x position
      //startPosition[0] = ;
      //getting y position.
      // startPosition[1] = 0;
      hold = 1;
      //# games
      //int games = 2;
      int playedGames = 0;
      //if you win
      // bool winner = false;
      //

      gameOn = true;
      //creating a array to hold the values of each game
      String[] write = new string[games];
      int[] roundNumCounter = new int[games];
      board[startPosition[0], startPosition[1]] = 1;
      while (playedGames < games)
      {
        CreateArray();
        board[startPosition[0], startPosition[1]] = 1;
        //create the board.
```

```
            gameOn = true;



            //Console.WriteLine("starting position {0}, and {1}", startPosition[0],
startPosition[1]);
            //Console.WriteLine("hmove {0}, and vMove{1}", hMove, vMove);

            while (gameOn == true)
            {

               Move(startPosition);
               //this will add the scores to a holder to calculate the average
               roundNumCounter[playedGames] = hold;
               //this add items to be written to file. includes number of played games to the
file.
               write[playedGames] = "Trial  " + playedGames + "  was able to successfully
touch  " + hold + "  squares";
            }



            DisplayBoard();
            hold = 0;
            playedGames++;


         }
         //writing to the file
         try
         {
            System.IO.File.WriteAllLines("LukaNecajev_InteligentMethod.txt", write);
         }
         catch (Exception )
         {
            Console.WriteLine("NOT WRITING");
         }
         return roundNumCounter;

      }
```

Intelligent Move Method

This method will act like the main method in a program, it will get the starting position from the game method and check the possible moves, it will then call the “CheckIfMovePossible” function and will populate the possible moves arrays. Once that is done it will get a random index and set it to the move, and move the item.

```
public override void Move(int[] startPosition)
    {
      //move while goodmoves moves is not ==0

      //this will check the best possible move for the knight
      int moveIndex = CheckIfMovePossible(startPosition);
      Random random = new Random();

      moveIndex = random.Next(0, goodIndexes.Count);
      if (!goodIndexes.Any())
      {
        gameOn = false;
        goodIndexes.Clear();
      }
      else
      {
        //if it meets the criteria  add one to the loop and make the move the Move
Symbol
        x = startPosition[0] + PossibleMovesX[goodIndexes[moveIndex]];
        y = startPosition[1] + PossibleMovesY[goodIndexes[moveIndex]];

        //make the starting position where the last one finished
        //Console.WriteLine("hmove {0}, and vMove{1}", hMove, vMove);
        if (((x < Row && x >= 0) && (y < Col && y >= 0)) &&
            (board[x, y] == 0))
        {
          board[x, y] = 1 + hold;
          startPosition[0] = x;
          startPosition[1] = y;
          //DEBugging
          //  Console.WriteLine("new start {0}, {1}", hMove, vMove);
        }
      }
      //adding to the move counter variable
      hold++;
    }
```

Intelligent Move Method

this method will check if the move is possible, it will get the possible moves for the values, calls the heuristic board method which will get the values of the heuristic board. For each possible move, it will check if it is possible and it will also check if it is the smallest number in the possible moves, and only if it is or it is an equal number to it then add the index to the possible moves index (goodMovesIndex) and send it back to the move function to be written to.

```
//this method will check if the move is possible, it will get the possible moves for the values, calls the huristic board method which will get the values of the huristic board.
//for each possible move, it will check if it is possible and add the index to the possible moves index (goodMovesIndex)

public override int CheckIfMovePossible(int[] startMoves)
{

    // will get the index of the moves
    List<int> goodMovesIndex = new List<int>();
    //will get the moves position in the huristic table
    List<int> goodMoves = new List<int>();

    //int[] goodMoves = new int[8];

    //checks possible moves 8
    for (int i = 0; i < 8; i++)
    {
        int x = startMoves[0] + PossibleMovesX[i];
        int y = startMoves[1] + PossibleMovesY[i];



        //if its possible to move
        //if the moves are within the vertical borders (8 being top of the screen/ 0 being bottom)
        //if the moves are within the horizontal borders (8 being right of the screen/ 0 being the left)
        //if the move was made, it will have an X as we have already been there.
        if (((x < Row && x >= 0) && (y < Col && y >= 0)) &&
            (board[x, y] == 0))
        {
            //if it passes the condition, find the possible moves locations on the huristic board
```

```
            //take this out and make the other for loop end before the smallest index is
called.
            /* for (int j = 0; j < 8; j++)
             {*/
            //making the moves = to one of the possible moves
            /*int x = startMoves[0] + PossibleMovesX[j];
            int y = startMoves[1] + PossibleMovesY[j];*/
            //getting a value out of the board for each possible move
            int checkItOut = CheckHuristicBoard(x, y);
            //if the returned values is not 0 add it to the list.
            if (CheckHuristicBoard(x, y) != 0)
            {
               //goodMoves[i] = checkItOut;
               goodMoves.Add(checkItOut);
               goodMovesIndex.Add(i);
            }

            //DisplayGrid();
            //Console.ReadKey();
         }



         //return the move index so we can move.

      }//end of checking for loop
      int moveIndex = 0;


      //if there are no moves in the index then kill the game
      if (!goodMovesIndex.Any())
      {
         gameOn = false;
         //clears the arrays
         goodMoves.Clear();
         goodMovesIndex.Clear();
      }
      //other wise get the smallest indexes
      else
      {
         goodIndexes = SmallestIndex(goodMoves, goodMovesIndex);
      }
      //make the smallest indexes = to the good moves index and pass the
```

```
            goodIndexes = goodMovesIndex;
            //return nothing due to the lists being static.
            return moveIndex;
        }
```

Intelligent Move Method

gets the smallest index and passes the indexes of all of the smallest items in the possible moves array. If the heuristic is smaller then the first element in the array make this one the smallest element and continue through the possible moves. Add the lowest index and loop throught and make sure if there is more then one that I add it to the index array as well. This will return the List of GoodIndexes and will populate the static Indexes List to be used to randomize the index.

```
        //gets the smallest index and passes the indexes of all of the smallest items in the
possible moves array.
        public static List<int> SmallestIndex(List<int> goodMoves, List<int>
goodMovesIndex)
        {

            List<int> returnedIndex = new List<int>();
            //gets the smallest number
            int small = goodMoves[0];

            for (int i = 0; i < goodMoves.Count; i++)
            {
                // Console.WriteLine(goodMoves[i]);

                //compare if small is greater than of any element of the array
                //assign that element in it.
                if (small < goodMoves[i])
                {

                    //if its smaller make small value
                    small = goodMoves[i];

                }


            }
            //adds the lowest count index possible
            for (int i = 0; i < goodMoves.Count; i++)
            {
                //loops through each item and checks if they are the same so I can randomly
choose the index in the array.
                if (small == goodMoves[i])
```

```
            {
                returnedIndex.Add(i);


            }
        }
        //returning the arrry of lowest indexes
        return returnedIndex;
    }
```

CheckHuristicBoard()

This method will take a possible move and check what value the item has on the huristic board and place the index and value Lists.

```
    public static int CheckHuristicBoard(int x, int y)
    {
        int LowestNumReached = 0;
        int caseswitch = 0;
        if (((x < Row && x >= 0) && (y < Col && y >= 0)) &&
              (board[x, y] == 0))
        {

            caseswitch = huristucBoard[x, y];
        }
        else
        {
            caseswitch = 0;
        }
        switch (caseswitch)
        {
            case 2:
                //  Console.WriteLine("its a 2");
                LowestNumReached = 2;

                break;
            case 3:
                //  Console.WriteLine("its a 3");
                LowestNumReached = 3;
                break;
            case 4:
                // Console.WriteLine("its a 4");
                LowestNumReached = 4;
                break;
            case 6:
```

```
                // Console.WriteLine("its a 6");
                LowestNumReached = 6;
                break;
            case 8:
                //  Console.WriteLine("its a 8");
                LowestNumReached = 8;
                break;
            default:

                break;
        }


        return LowestNumReached;
    }

    //this method is overided and will display the board for the user.
    public override void DisplayBoard()
    {

        Console.Write(" ");
        //setting the letters at the top to be as long as the board
        for (int i = 65; i < Col + 65; i++)
            Console.Write(Convert.ToChar(i));

        Console.WriteLine();
        //creating the numbers on the side, and then the board using a value to populate
each one.
        for (int r = 0; r < Row; r++)
        {
            Console.Write(r + 1);
            for (int c = 0; c < Col; c++)
            {
                //populating the board
                Console.Write(" " + board[r, c]);
            }
            Console.WriteLine();
        }
    }
```

CreateArray() Method

This method will display the board in the console.

```
    //Overridded method Creating the initial array for the second class.
```

```
public override void CreateArray()
{
   for (int i = 0; i < Row; i++)
   {
      for (int c = 0; c < Col; c++)
      {
         //adds a grid symbol (0) where we have moved at coords of row and column
         board[i, c] = 0;
      }
   }
}
```

Standard Deviation

This method will calculate the standard deviation of the methods outputs. And will return the standard deviation

```
//Creating a standard Deviation method which will return the standard deviation to the forms class
public static double StandardDeviation(int numberOfRepeats, int[] roundNumCounter)
{
   double stdDev, varience = 0;
   for (int i = 0; i < numberOfRepeats; i++)
   {
      //calculating the standard devience
      varience = varience + Math.Pow((roundNumCounter[i] - Average(numberOfRepeats, roundNumCounter)), 2);
   }


   stdDev = varience / (numberOfRepeats - 1);
   stdDev = Math.Sqrt(stdDev);
   return Math.Round(stdDev, 1);
}
```

Average

This method will calculate the average and will return the double average

```
//Creating a Average method which will return the averagem, and is used in the standard deviation method to be displayed in the forms class
public static double Average(int numberOfRepeats, int[] roundNumCounter)
{
   double average = 0;
   for (int i = 0; i < numberOfRepeats; i++)
   {
      average = average + roundNumCounter[i];
```

```
        }

        average = average / Convert.ToDouble(roundNumCounter.Length);

        return Math.Round(average, 1);
    }
}}
```

Form1 pg-18-22

The forms page will create the user interface and will run the smart and dumb methods. This class will create instances of each method, and pass the users starting positions and number of runs to the respective methods. This class will also take the information from the Game methods and will display the items in the respective text boxes, as well as call the Average and Standard Deviation methods and display them in labels on the screen. The Form will also display the last board once you run it.

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

namespace Luka_Necajev_FormsA2
{

    public partial class Form1 : Form
    {
        private const int Row = 8;
        private const int Col = 8;
        private static int[] startPosition = new int[2];


        public Form1()
        {
            InitializeComponent();

        }

        private void label1_Click(object sender, EventArgs e)
        {

        }

        private void label2_Click(object sender, EventArgs e)
        {

        }

        private void label3_Click(object sender, EventArgs e)
        {

        }

        private void label4_Click(object sender, EventArgs e)
        {

        }
```

```
        private void textBox1_TextChanged(object sender, EventArgs e)
        {

        }

        private void btnRunUnintelligent_Click(object sender, EventArgs e)
        {
            UnintelligentMethod dumb = new UnintelligentMethod();

            //holder for the best moves.
            int bestMoves = 0;
            //best round
            int bestRound = 0;
            //clears the board
            tableLayoutPanel1.Controls.Clear();
            //gets the x and y position from user
            startPosition[0] = Convert.ToInt32(tbStartPositionX.Text);
            startPosition[1] = Convert.ToInt32(tbStartPositionY.Text);
            //gets the run count or games from the user
            int runCount = Convert.ToInt32(tbHowMany.Text);
            //creating the total round move counter.
            int[] roundNumCounter = new int[runCount];
            //creating the Game and moving and completing the tour.
            roundNumCounter = dumb.Game(runCount, startPosition);


            //this will read the results from the file.
            using (StreamReader reader = new StreamReader("InteligentMethod.txt"))
            {
                while (!reader.EndOfStream)
                {
                    richTextBox1.AppendText(reader.ReadLine());
                }
            }
            //calculates the average
            double average = IntelligentMethod.Average(runCount, roundNumCounter);
            //if the deviation is

                double stndDevi = IntelligentMethod.StandardDeviation(runCount,
roundNumCounter);
                laDeviation.Text = Convert.ToString(stndDevi);


            for (int i = 0; i < roundNumCounter.Length; i++)
            {
                if (roundNumCounter[i] > bestMoves)
                {
                    bestRound = i;
                    bestMoves = roundNumCounter[i];
                }
            }

            //displaying the best number of moves achieved and the round it was achieved
at
            laNumberOfMoves.Text = Convert.ToString(bestMoves);
            laBestMoveCount.Text = Convert.ToString(bestRound);
```

```
            //displaying the average.
            laAverage.Text = Convert.ToString(average);


            //creates the board
            Label[] lbl = new Label[65];
            for (int r = 0; r < Row; r++)
            {
                for (int c = 0; c < Col; c++)
                {
                    lbl[c] = new Label();
                    lbl[c].Text = UnintelligentMethod.moves[r, c] + " ";
                    tableLayoutPanel1.Controls.Add(lbl[c]);
                }
            }



        }

        private void btnRunInteligent_Click(object sender, EventArgs e)
        {
            IntelligentMethod smart = new IntelligentMethod();
            //holder for the best moves.
            int bestMoves = 0;
            //best round
            int bestRound = 0;
            //clears the board
            tableLayoutPanel1.Controls.Clear();
            //gets the x and y position from user
            startPosition[0] = Convert.ToInt32(tbStartPositionX.Text);
            startPosition[1] = Convert.ToInt32(tbStartPositionY.Text);
            //gets the run count or games from the user
            int runCount = Convert.ToInt32(tbHowMany.Text);
            //creating the total round move counter.
            int[] roundNumCounter = new int[runCount];
            //creating the Game and moving and completing the tour.
            roundNumCounter = smart.Game(runCount, startPosition);


            //this will read the results from the file.
            using (StreamReader reader = new StreamReader("InteligentMethod.txt"))
            {
                while (!reader.EndOfStream)
                {
                    richTextBox1.AppendText(reader.ReadLine());
                }
            }
            //calculates the average
            double average = IntelligentMethod.Average(runCount, roundNumCounter);
            //calculates the deviation

                double stndDevi = IntelligentMethod.StandardDeviation(runCount,
roundNumCounter);
                laDeviation.Text = Convert.ToString(stndDevi);

            //gets the best moves from the results array
```

```
            for (int i = 0; i < roundNumCounter.Length; i++)
            {
                if (roundNumCounter[i] > bestMoves)
                {
                    bestRound = i;
                    bestMoves = roundNumCounter[i];
                }
            }

            //displaying the best number of moves achieved and the round it was achieved
at
            laNumberOfMoves.Text = Convert.ToString(bestMoves);
            laBestMoveCount.Text = Convert.ToString(bestRound);


            //displaying the average.
            laAverage.Text = Convert.ToString(average);


            //creates the board
            Label[] lbl = new Label[65];
            for (int r = 0; r < Row; r++)
            {
                for (int c = 0; c < Col; c++)
                {
                    lbl[c] = new Label();
                    lbl[c].Text = IntelligentMethod.board[r, c] + " ";
                    tableLayoutPanel1.Controls.Add(lbl[c]);
                }
            }




        }
        private void laBestMoveCount_Click(object sender, EventArgs e)
        {

        }

        private void laNumberOfMoves_Click(object sender, EventArgs e)
        {

        }

        private void laDeviation_Click(object sender, EventArgs e)
        {

        }
        private void laAverage_Click(object sender, EventArgs e)
        {

        }

        private void lvTrials_SelectedIndexChanged(object sender, EventArgs e)
        {
```

```
        }

        private void tableLayoutPanel1_Paint(object sender, PaintEventArgs e)
        {

        }

        private void label8_Click(object sender, EventArgs e)
        {

        }

        private void label5_Click(object sender, EventArgs e)
        {

        }

        private void tableLayoutPanel1_Paint_1(object sender, PaintEventArgs e)
        {

        }



        private void label9_Click(object sender, EventArgs e)
        {

        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void richTextBox1_TextChanged(object sender, EventArgs e)
        {

        }
    }
}
```