Übung 1

Table of Contents

1. Setup

```
Number of cores: 4
Memory size 16,0 GB
CPU type: Intel Core i7-8565U 1.80GHz
System: Windows 11
```

2. Theory

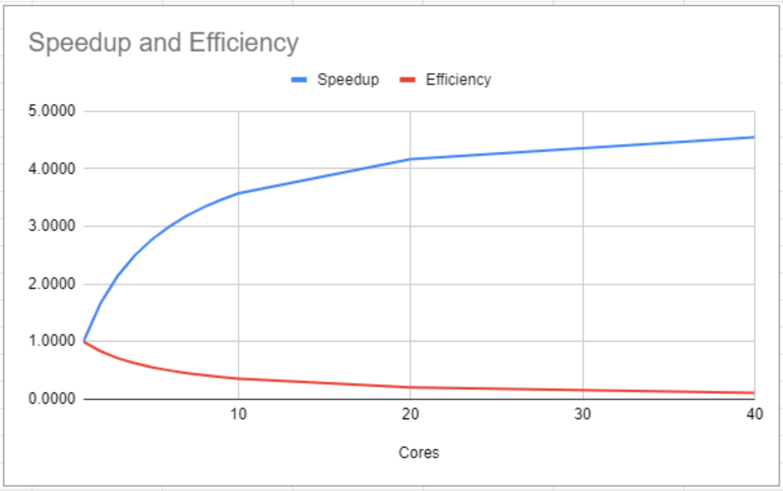
2.1. Calculate and plot speedup and efficiency

Speedup = 1 / (+ (1 - Sigma) / Cores)
Efficiency = Speedup / Cores

Seq: 0.2
Par: 0.8
Cores: n

σ	0.2	Sequential part
p	10	

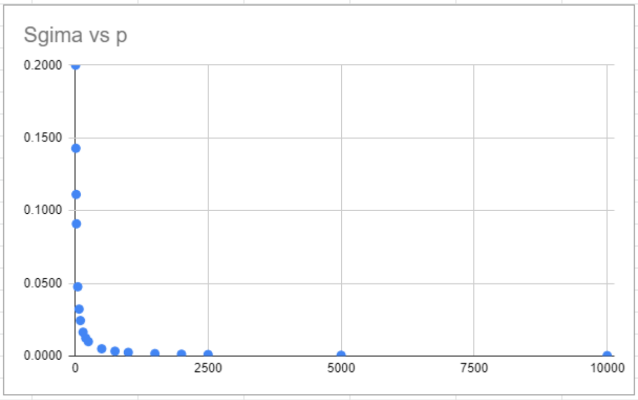
Cores	Speedup	Efficiency
1	1.0000	1.0000
2	1.6667	0.8333
3	2.1429	0.7143
4	2.5000	0.6250
5	2.7778	0.5556
6	3.0000	0.5000
7	3.1818	0.4545
8	3.3333	0.4167
9	3.4615	0.3846
10	3.5714	0.3571
20	4.1667	0.2083
40	4.5455	0.1136
80	4.7619	0.0595
100	4.8077	0.0481
1000	4.9801	0.0050
10000	4.9980	0.0005



The limit for speed up is 5.

2.2. Calculate and plot Sigma with increasing problem sizes

Problem size	Sequential	Parallel	σ	Cores	Speedup	Efficiency
10	2.00	8.00	0.2000	2	1.666666667	0.8333333333
15	3.00	18.00	0.1429	2	1.75	0.875
20	4.00	32.00	0.1111	3	2.454545455	0.8181818182
25	5.00	50.00	0.0909	3	2.538461538	0.8461538462
50	10.00	200.00	0.0476	6	4.846153846	0.8076923077
75	15.00	450.00	0.0323	8	6.526315789	0.8157894737
100	20.00	800.00	0.0244	11	8.843137255	0.8039215686
150	30.00	1,800.00	0.0164	16	12.84210526	0.8026315789
200	40.00	3,200.00	0.0123	21	16.84158416	0.801980198
250	50.00	5,000.00	0.0099	26	20.84126984	0.8015873016
500	100.00	20,000.00	0.0050	51	40.84063745	0.8007968127
750	150.00	45,000.00	0.0033	76	60.84042553	0.8005319149
1000	200.00	80,000.00	0.0025	101	80.84031936	0.8003992016
1500	300.00	180,000.00	0.0017	151	120.840213	0.8002663116
2000	400.00	320,000.00	0.0012	201	160.8401598	0.8001998002
2500	500.00	500,000.00	0.0010	251	200.8401279	0.8001598721
5000	1,000.00	2,000,000.00	0.0005	501	400.840064	0.800079968
10000	2,000.00	8,000,000.00	0.0002	1001	800.840032	0.800039992



With a problem size of 10000 the sequential part gets extremely small.

2.3. How many processors can be utilized?

p = 100: 11 cores are needed

p = 1000: 100 cores are needed

p = 10000: With 1000 cores an efficiency of 80% can be reached

Problem size	Sequential	Parallel	σ	Cores	Speedup	Efficiency
10	2.00	8.00	0.2000	2	1.666666667	0.8333333333
15	3.00	18.00	0.1429	2	1.75	0.875
20	4.00	32.00	0.1111	3	2.454545455	0.8181818182
25	5.00	50.00	0.0909	3	2.538461538	0.8461538462
50	10.00	200.00	0.0476	6	4.846153846	0.8076923077
75	15.00	450.00	0.0323	8	6.526315789	0.8157894737
100	20.00	800.00	0.0244	11	8.843137255	0.8039215686
150	30.00	1,800.00	0.0164	16	12.84210526	0.8026315789
200	40.00	3,200.00	0.0123	21	16.84158416	0.801980198
250	50.00	5,000.00	0.0099	26	20.84126984	0.8015873016
500	100.00	20,000.00	0.0050	51	40.84063745	0.8007968127
750	150.00	45,000.00	0.0033	76	60.84042553	0.8005319149
1000	200.00	80,000.00	0.0025	101	80.84031936	0.8003992016
1500	300.00	180,000.00	0.0017	151	120.840213	0.8002663116
2000	400.00	320,000.00	0.0012	201	160.8401598	0.8001998002
2500	500.00	500,000.00	0.0010	251	200.8401279	0.8001598721
5000	1,000.00	2,000,000.00	0.0005	501	400.840064	0.800079968
10000	2,000.00	8,000,000.00	0.0002	1001	800.840032	0.800039992

3. Wator

3.1. Review of the application

3.1.1. Design

The methods are too long and could often be simplified. Also some methods could be created in separate classes instead they are all in OriginalWatorWorld.

3.1.2. Efficiency

The program in case of efficiency has definitely potential for improvement. For example the use of two dimensional arrays or the creation of an position array each time the GetNeighbors function is called is not ideal. Also the shuffle method is not very well implemented.

3.1.3. Clarity

With the GetNeighbors function, the if else constructs are not very ideal to read and could be improved or separate methods could be created.

3.1.4. Readability

Readability is definitely not the best. For example the GetNeighbours function is too long with lots of code duplications.

3.2. Three improvements

3.2.1. Improvement 1

Changed point array of GetNeighbours to List. Also introduced it globally to reduce the work of the garbage collector. Otherwise each time a new array has to be created. The list gets reused every time.

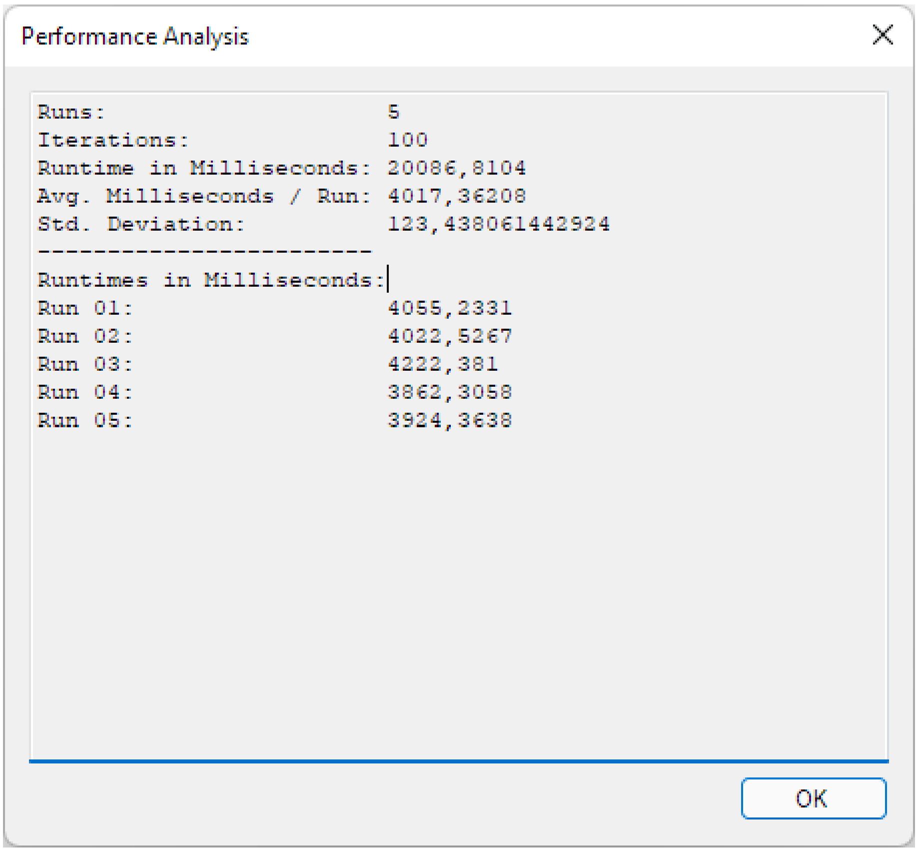


Figure 1. Original

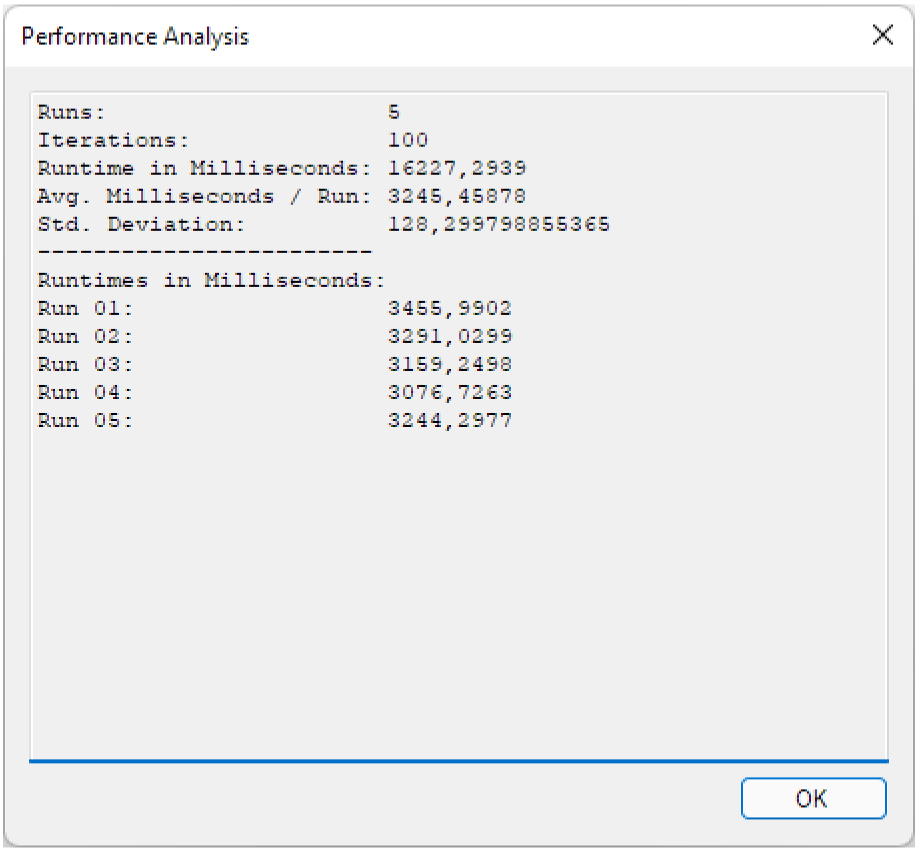
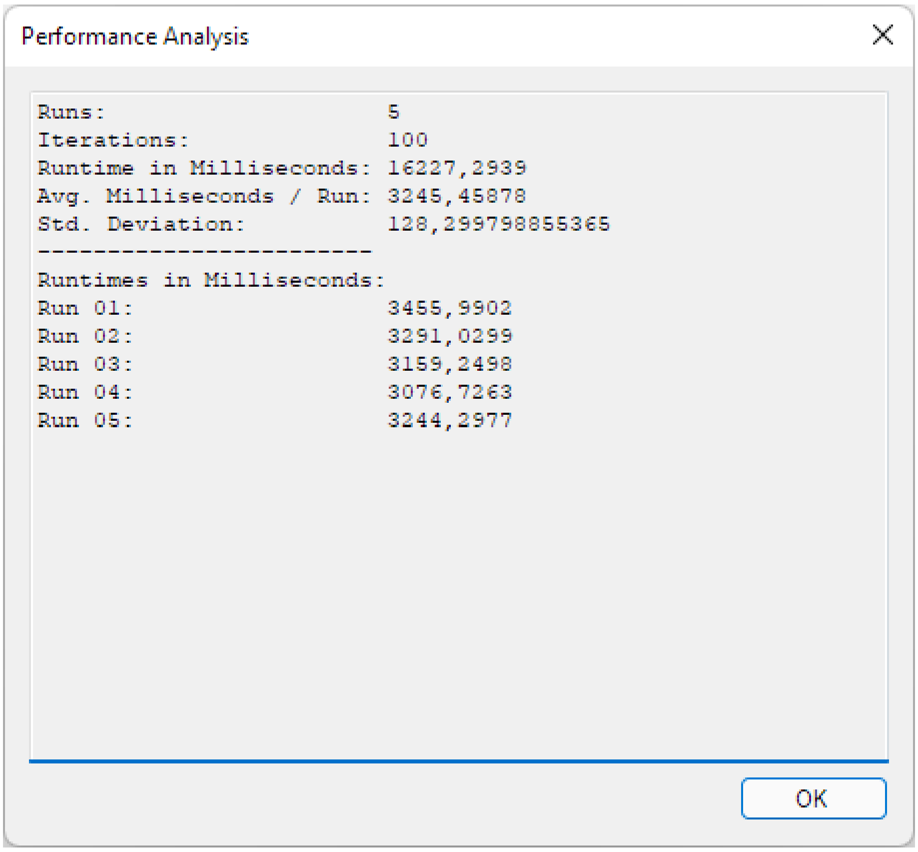


Figure 2. Improved

```
public IList<Point> GetNeighbours(Type type, Point position)
{
    points.Clear();
    int i, j;

    // look north
    i = position.X;
    j = (position.Y + Height - 1) % Height;
    if (type == null && Grid[i, j] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[i, j]))
    {
        if (Grid[i, j] != null && !Grid[i, j].Moved)
        {  // ignore animals moved in the current iteration
            points.Add(new Point(i, j));
        }
    }
    // ...
}
```

3.2.2. Improvement 2

Changed all two dimensional matrices do one dimensional. That means that the animal board and the the matrix for the random positioning are only simple arrays where the index get calculated with a special function. Improvements can be seen especially on the ExecuteStep and RandomizeMatrix functions.



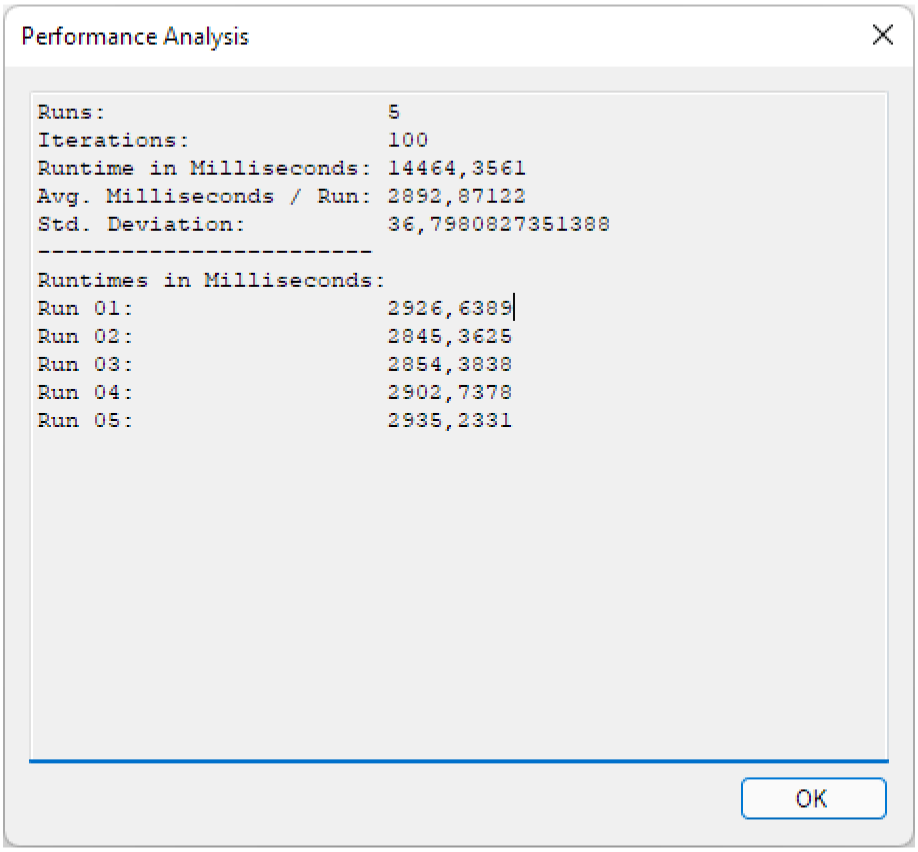


Figure 3. Prior

Listing 1. Improved

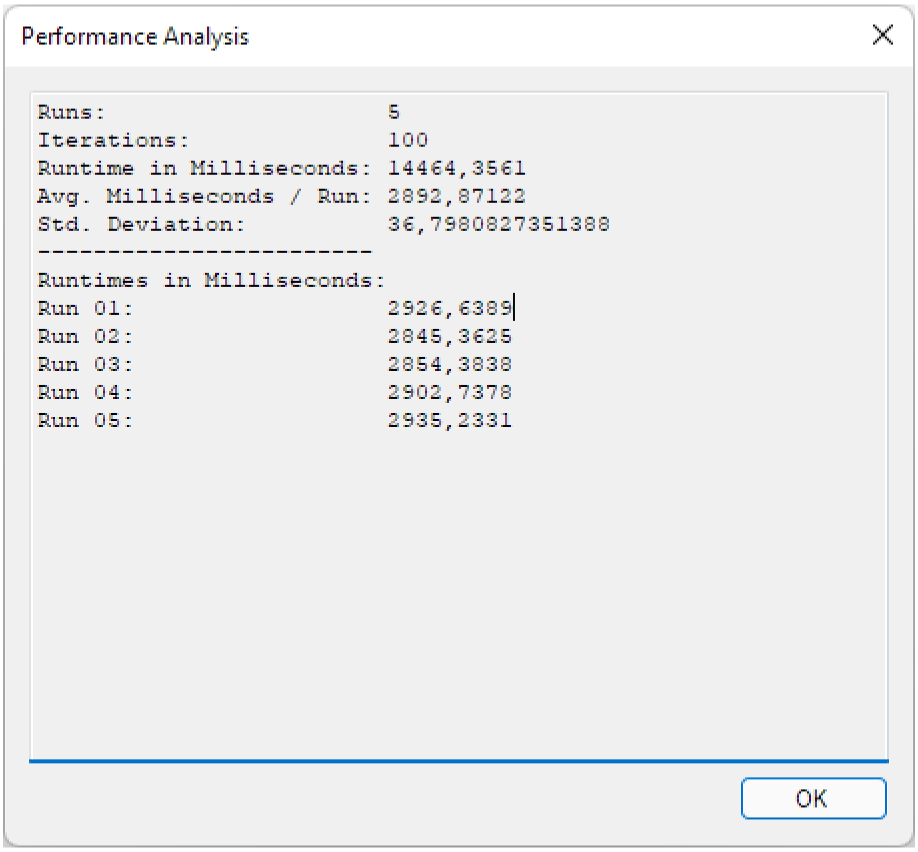
```
private int[] randomMatrix;
```

```
public int GetGridIndex(int row, int column)
{
    return row * Width + column;
}

// shuffle values of the matrix
private void RandomizeMatrix(int[] matrix)
{
    // perform Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
    int temp, selectedRow, selectedCol;

    int row = 0;
    int col = 0;
    for (int i = 0; i < Height * Width; i++)
    {
        temp = matrix[GetGridIndex(row, col)];

        // select random element from remaining elements
        // already processed elements must not be chosen a second time
        selectedRow = random.Next(row, Height);
        if (selectedRow == row) selectedCol = random.Next(col, Width);  // current row selected
-> select from remaining columns
        else selectedCol = random.Next(Width);  // new row selected -> select any column

        // swap
        matrix[GetGridIndex(row, col)] = matrix[GetGridIndex(selectedRow, selectedCol)];
        matrix[GetGridIndex(selectedRow, selectedCol)] = temp;

        // incremet col and row
        col++;
        if (col >= Width) { col = 0; row++; }
    }
}

public IList<Point> GetNeighbors(Type type, Point position)
{
    points.Clear();
    int i, j;

    // look north
    i = position.X;
    j = (position.Y + Height - 1) % Height;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
    {
        if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
        {  // ignore animals moved in the current iteration
            points.Add(new Point(i, j));
        }
    }
    // look east
    i = (position.X + 1) % Width;
    j = position.Y;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
```

```
    else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
    {
        if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
        {
            points.Add(new Point(i, j));
        }
    }
    // look south
    i = position.X;
    j = (position.Y + 1) % Height;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
    {
        if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
        {
            points.Add(new Point(i, j));
        }
    }
    // look west
    i = (position.X + Width - 1) % Width;
    j = position.Y;
    if (type == null && Grid[GetGridIndex(j, i)] == null)
    {
        points.Add(new Point(i, j));
    }
    else if (type != null && type.IsInstanceOfType(Grid[GetGridIndex(j, i)]))
    {
        if (Grid[GetGridIndex(j, i)] != null && !Grid[GetGridIndex(j, i)].Moved)
        {
            points.Add(new Point(i, j));
        }
    }

    return points;
}
```

3.2.3. Improvement 3

Used more performant version of the Knuth Shuffle. Also improved memory consumption of sharks by removing the second division.



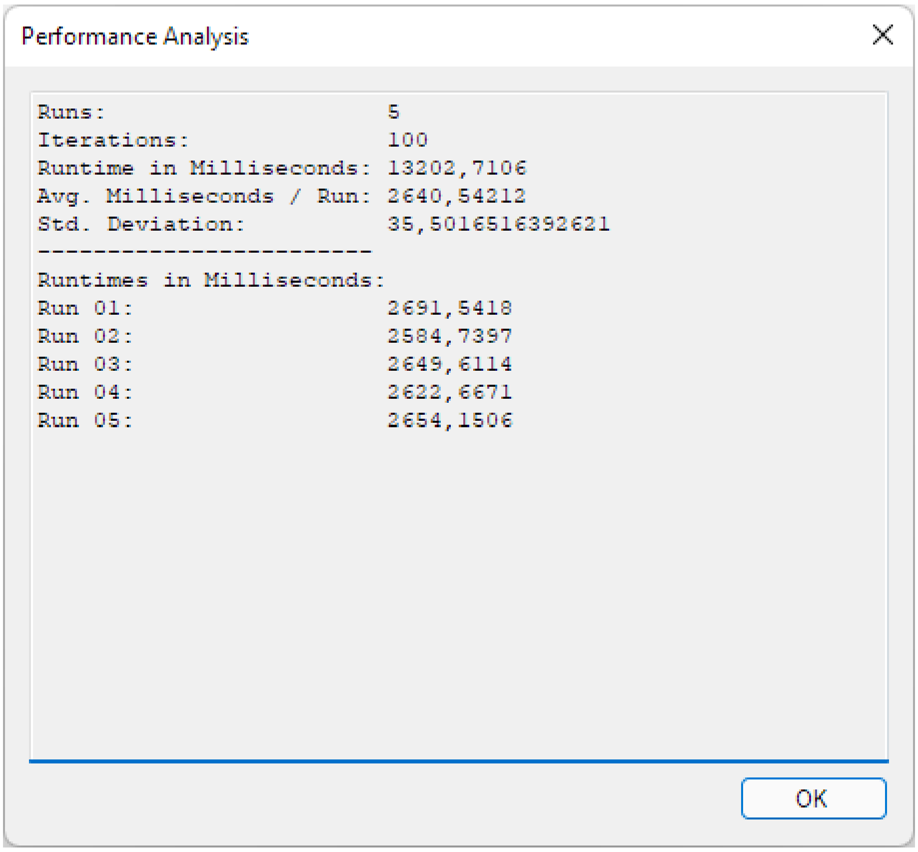


Figure 4. Prior

Listing 2. Improvement

```
private void RandomizeMatrix(int[] array)
{
    // perform Knuth shuffle (http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle)
    int size = array.Length;
    for (int i = 0; i < (size - 2); i++)
    {
        int result = random.Next(i, size);
        int temp = array[result];
        array[result] = array[i];
        array[i] = temp;
    }
}
public class Shark : Animal
{
    // spawning behaviour of sharks
    protected override void Spawn()
    {
        Point free = World.SelectNeighbor(null, Position);  // find a random empty neighboring
cell
        if (free.X != -1)
        {
            // empty neighboring cell found -> create new shark there and share energy between
parent and child shark
            Energy /= 2;
            new Shark(World, free, Energy);
        }
    }
}
```