Exercise 1

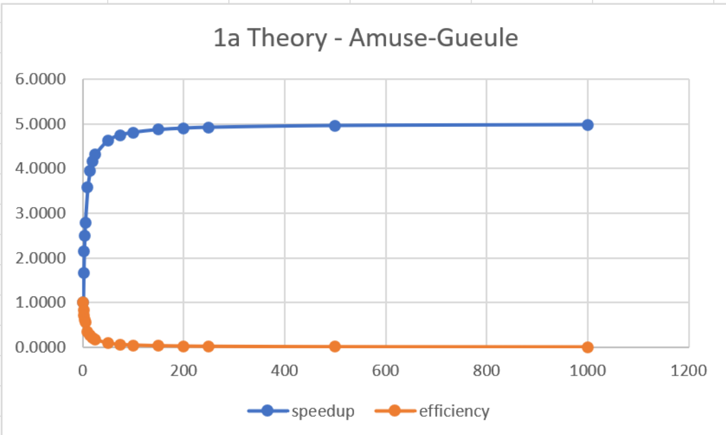
System Configuration

CPU	**Intel Core i7-7700K**
CPU Base Speed	4.20 GHz
CPU Boost Speed	4.50 GHz
Memory Size	16 GB
Memory Speed	3000 MHz
Hard Drive	1 TB M.2 SSD
Operating System	Windows 10 Pro
IDE	Jetbrains Rider V2021.2.2

Theory - Amuse-Gueule

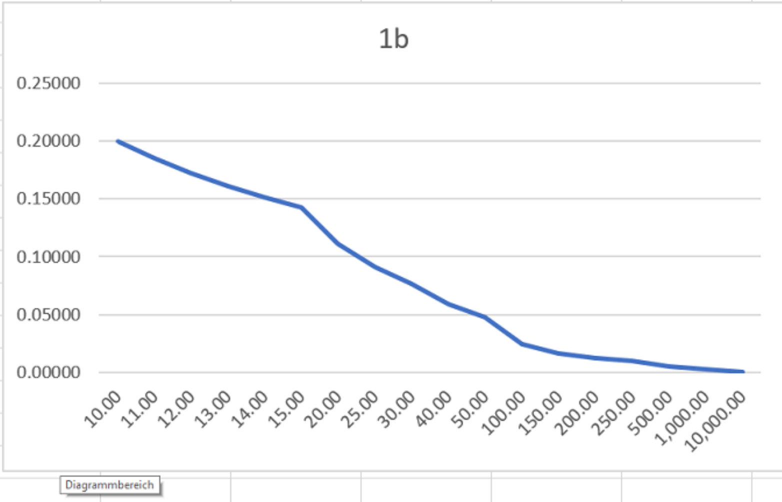
Speedup and Efficiency

p	sigma	n	speedup	efficiency
10	0.2	1	1.0000	1.0000
10	0.2	2	1.6667	0.8333
10	0.2	3	2.1429	0.7143
10	0.2	4	2.5000	0.6250
10	0.2	5	2.7778	0.5556
10	0.2	10	3.5714	0.3571
10	0.2	15	3.9474	0.2632
10	0.2	20	4.1667	0.2083
10	0.2	25	4.3103	0.1724
10	0.2	50	4.6296	0.0926
10	0.2	75	4.7468	0.0633
10	0.2	100	4.8077	0.0481
10	0.2	150	4.8701	0.0325
10	0.2	200	4.9020	0.0245
10	0.2	250	4.9213	0.0197
10	0.2	500	4.9603	0.0099
10	0.2	1000	4.9801	0.0050



Sigma with increasing problem size

p	seq	par	sigma
10.00	2.00	8.00	0.20000
11.00	2.20	9.68	0.18519
12.00	2.40	11.52	0.17241
13.00	2.60	13.52	0.16129
14.00	2.80	15.68	0.15152
15.00	3.00	18.00	0.14286
20.00	4.00	32.00	0.11111
25.00	5.00	50.00	0.09091
30.00	6.00	72.00	0.07692
40.00	8.00	128.00	0.05882
50.00	10.00	200.00	0.04762
100.00	20.00	800.00	0.02439
150.00	30.00	1,800.00	0.01639
200.00	40.00	3,200.00	0.01235
250.00	50.00	5,000.00	0.00990
500.00	100.00	20,000.00	0.00498
1,000.00	200.00	80,000.00	0.00249
10,000.00	2,000.00	8,000,000.00	0.00025



Efficiency with more processors

p	seq	par	sigma	n	speedup	efficiency
10.00	2.00	8.00	0.20000	1	1	1
11.00	2.20	9.68	0.18519	2	1.6875	0.84375
12.00	2.40	11.52	0.17241	2	1.70588235	0.85294118
13.00	2.60	13.52	0.16129	2	1.72222222	0.86111111
14.00	2.80	15.68	0.15152	2	1.73684211	0.86842105
15.00	3.00	18.00	0.14286	2	1.75	0.875
20.00	4.00	32.00	0.11111	3	2.45454545	0.81818182
25.00	5.00	50.00	0.09091	3	2.53846154	0.84615385
30.00	6.00	72.00	0.07692	4	3.25	0.8125
40.00	8.00	128.00	0.05882	5	4.04761905	0.80952381
50.00	10.00	200.00	0.04762	6	4.84615385	0.80769231
100.00	**20.00**	**800.00**	**0.02439**	**11**	**8.84313725**	**0.80392157**
150.00	30.00	1,800.00	0.01639	16	12.8421053	0.80263158
200.00	40.00	3,200.00	0.01235	21	16.8415842	0.8019802
250.00	50.00	5,000.00	0.00990	26	20.8412698	0.8015873
500.00	100.00	20,000.00	0.00498	51	40.8406375	0.80079681
1,000.00	**200.00**	**80,000.00**	**0.00249**	**101**	**80.8403194**	**0.8003992**
10,000.00	**2,000.00**	**8,000,000.00**	**0.00025**	**1001**	**800.840032**	**0.80003999**

Wator World

Review

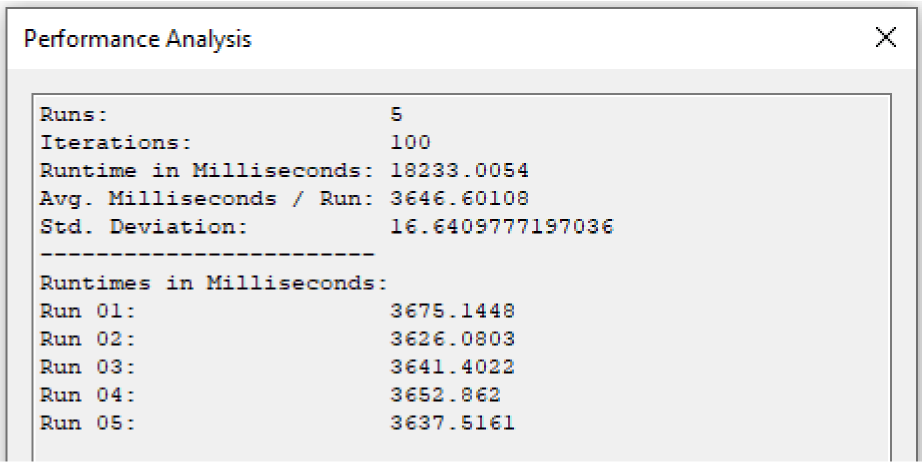
While WinForms is still supported by Microsoft, it is quite old technology and should not be used anymore. It also makes it harder to port the Application to other operating systems.

The Code suffers from issues with readability and efficiency in quite a few spots. The GetNeighbour-Method is one of the worst offenders for that. The method should extract some functionality (like looking at neighbouring cells) into another function.

General Settings should not be hardcoded, they could be imported from a settings file.

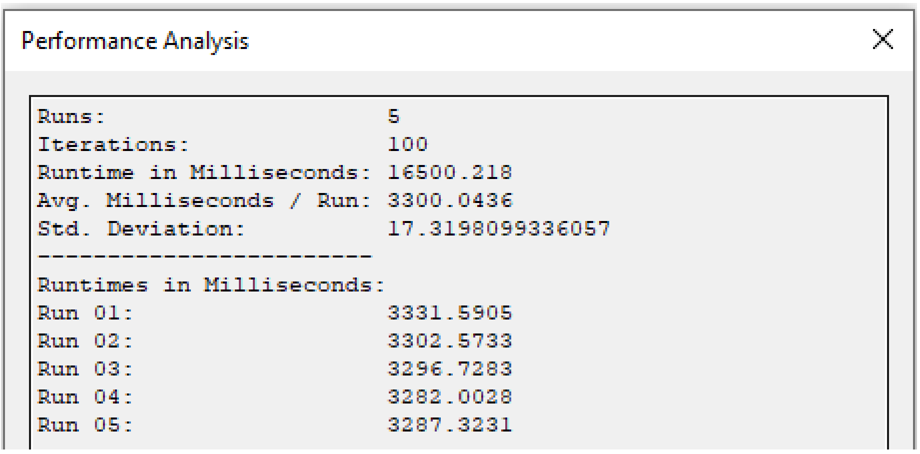
Improvements

Baseline Performance:



Optimization 1

Points required for the GetNeighbour-Method are allocated with every call. This has been reworked to only allocate the array once and reuse it for every call. **Speedup: 1.1**.



```
private Point[] neighbors = new Point[4];

// find all neighboring cells of the given position and type
public Point[] GetNeighbors(Type type, Point position) {
    //Point[] neighbors = new Point[4];
    int neighborIndex = 0;
    int i, j;

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

```
    } else if ((type != null) && (type.IsInstanceOfType(Grid[j * Width + i]))) {
        if ((Grid[j * Width + i] != null) && (!Grid[j * Width + i].Moved)) {
            neighbors[neighborIndex] = new Point(i, j);
            neighborIndex++;
        }
    }

    // create result array that only contains found cells
    Point[] result = new Point[neighborIndex];
    for (int x = 0; x < neighborIndex; x++)
        result[x] = neighbors[x];
    return result;
}
```

Optimization 2

Operations on a 2-Dimensional Array can be quite performance hungry, so both matrixes are replaced with an array. The index can be calculated using the width of the world. **Speedup: 1.06**.

Performance Analysis

```
Runs:                       5
Iterations:                 100
Runtime in Milliseconds:    15527.309
Avg. Milliseconds / Run:    3105.4618
Std. Deviation:             42.409367845947
------------------------
Runtimes in Milliseconds:
Run 01:                     3174.7234
Run 02:                     3070.7694
Run 03:                     3080.6307
Run 04:                     3066.475
Run 05:                     3134.7105
```

```
private int[] randomMatrix;
public Animal[] Grid { get; private set; }

Grid = new Animal[Width * Height];

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[row * Width + 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[row * Width + col] = matrix[selectedRow * Width + selectedCol];
            matrix[selectedRow * Width + selectedCol] = temp;

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

Optimization 3

The RandomizeMatrix method uses an inefficient version of the Knuth shuffle. This has been replaced with the more effiecent version which brings the asymptotic runtime complexity from O(n^2) to O(n). **Speedup: 1.1**.

Performance Analysis

```
Runs:                      5
Iterations:                100
Runtime in Milliseconds:   14097.3747
Avg. Milliseconds / Run:   2819.47494
Std. Deviation:            22.4575668697784
------------------------
Runtimes in Milliseconds:
Run 01:                    2856.6924
Run 02:                    2806.3503
Run 03:                    2799.5477
Run 04:                    2834.1036
Run 05:                    2800.6807
```

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

    for (int i = 0; i < Height * Width; i++) {
        int j = random.Next(i, Height * Width);
        temp = matrix[i];
        matrix[i] = matrix[j];
        matrix[j] = temp;
    }
}
```