

Name Stefan Kert

Points _____

Effort in hours 5 h**1. Wator – Eat or be eaten ...****(8 Points)**

Wator is the Name of a small circular planet, far far away from our galaxy, where no man has ever gone before. On Wator there live two different kinds of species: *sharks* and *fish*. Both species live according to a very old set of rules which hasn't been changed for the last thousands of years.

For **fish** the rules are:

- at the beginning of all time there were f fish
- each fish has a constant energy E_f
- in each time step a fish moves randomly to one of its four adjacent cells (up, down, left or right), if and only if there is a free cell available
- if all adjacent cells are occupied, the fish doesn't move
- in each time step fish age by one time unit
- if a fish gets older than a specified limit B_f , the fish breeds (i.e., a new fish is born on a free adjacent cell, if such a cell is available)
- after the birth of a new fish the age of the parent fish is reduced by B_f

For **sharks** the rules are:

- at the beginning of all time there were s sharks, each with an initial energy of E_s
- in each time step a shark consumes one energy unit
- in each time step a shark eats a fish, if a fish is on one of its adjacent cells
- if a shark eats a fish, the energy of the shark increases by the energy value of the eaten fish
- if there is no fish adjacent to the shark, the shark moves like a fish to one of its neighbor cells
- if the energy of a shark gets 0, the shark dies
- if the energy of a shark gets larger than a specified limit B_s , the shark breeds and the energy of the parent shark is equally distributed among the parent and the child shark (i.e., a new shark is born on a free adjacent cell, if such a cell is available)

In the Moodle course you find a ready to use implementation of Wator. Make a critical review of the code and analyze its design, performance, readability, etc. **Write a short report** which outlines the results of your review.

To get a fair comparison of the application's performance, **analyze** a Wator world of 500 x 500 cells. How long takes a run of 100 iterations on average with deactivated graphical output? Execute several independent test runs and **document the results in a table** (also calculate the mean value and the standard deviation). Then **answer the following questions**: Where and what for is most of the runtime consumed? What can be done to improve performance? What are the most performance-critical aspects?

2. Wator – Optimization**(16 Points)**

Based on your analysis, change the application step by step to improve performance. Think of at least **three concrete improvements** and implement them. Document for each improvement how the runtime changes (in comparison to the prior and to the initial version) and calculate the speedup. Each single optimization should yield a speedup of at least 1.05 compared to the prior version. Test your improvements with the settings given in the previous task.

VPS5 - UE1

Wator

Stefan Kert

1. April 2016

1 Analyse

1.1 Design und Lesbarkeit

Grundsätzlich lässt sich auf den ersten Blick erkennen, dass die Struktur des angegebenen Programmes sehr gut gestaltet ist. Durch die geringe Anzahl der Klassen ist die Einarbeitungszeit sehr kurz. Die meisten Klassen sind dabei für die Optimierung irrelevant. Lediglich die Klassen *OriginalWatorWorld*, *Animal*, *Fish* und *Shark* sind für eine Performanzoptimierung relevant. Dabei befindet sich der Großteil der Funktionalität in der Klasse *OriginalWaterWorld*. Einige der in dieser Klasse enthaltenen Methoden sind sehr lange und teilweise auch sehr unübersichtlich gestaltet. Durch Extrahieren von Methoden und kleinen Refactorings würde die Lesbarkeit und Wartbarkeit dieser Methoden sehr stark verbessert werden. Ein weiteres Problem stellen die vielen Kommentare dar, die teilweise unnötige sind, oder durch ein Extrahieren in eigene Methoden und vergeben vernünftiger Namen entfernt werden könnten.

1.2 Performance

Auf den ersten Blick lassen sich bereits im Quellcode Stellen erkennen die für die Performanz sehr schlecht sind. Es werden viele Objekte und Daten kopiert wie z.B. die Struktur *Point*, welche an verschiedenen Stellen neu erzeugt, kopiert und wieder gelöscht wird. Nach einer genaueren Analyse mit den Visual Studio Diagnosetools lässt sich erkennen, dass die meiste Zeit in der *ExecuteStep* und in der *SelectNeighbour* Methode bzw. in den von diesen Methoden aufgerufenen Methoden verloren geht, was in erster Linie damit zusammenhängt, dass diese Methoden sehr häufig aufgerufen werden. Ein weiterer Aspekt der einen negativen Einfluss auf die Performanz hat ist die Verwendung eines zweidimensionalen Arrays, welches immer wieder gemischt und zufällig generiert wird. Weiters sind die oben erwähnten Strukturen (*Point*) sehr kritisch zu sehen, da diese immer wieder kopiert und freigegeben werden müssen.

2 Fragen

2.1 Aktuelle Performanz

In dieser Tabelle lässt sich die aktuelle Performanz ablesen. Die Abweichungen zwischen den einzelnen Durchläufen sind sehr gering. Alle Tests wurden mit den folgenden Einstellungen durchgeführt:

- FishBreedTime: 10
- InitialFishEnergy: 10
- InitialFishPopulation: 20000
- DisplayInterval: 1
- DisplayWorld: false
- Height: 500
- Iterations: 100
- Runs: 5
- Width: 500
- Workers: 1
- InitialSharkEnergy: 25
- InitialSharPopulation: 5000
- SharkBreedEnergy: 50

	Original
1. Durchlauf	6293,59892
2. Durchlauf	6141,90
3. Durchlauf	6013,10
4. Durchlauf	5992,79
5. Durchlauf	6038,73
Durchschnitt	6096,02
Standardabweichung	211,97

2.2 Wo wird die meiste Leistung verbraucht?

Durch die Visual Studio Diagnostic Tools ist es sehr einfach herauszufinden, wo die meiste Leistung verbraucht wird. Die Grafik 1 zeigt sehr gut wo die meiste Leistung verbraucht wird. Weiters zeigt sie, dass die meisten Instanzen (>90%) vom Typ Point sind und dass diese den meisten Speicher benötigen. Die speicherintensivste Funktion ist die `GetNeighbour` Funktion, was darauf zurückzuführen ist, dass hier sehr viele Point-Arrays kopiert, gelöscht und verschoben werden, sowie sehr viele Instanzen vom Typ Point erstellt werden.



Abbildung 1: RAM Auslastung

2.3 Welche Teile sind die hinsichtlich Performanz kritischen Teile?

Auch bei der Analyse der benötigten Zeit für die einzelnen Funktionen liefern die Visual Studio Diagnostic Tools eine sehr gute Unterstützung. An Hand der Grafik 2 lässt sich gut erkennen, dass vor allem die Funktionen zum zufälligen Generieren der Matrix, die Funktion `ExecuteStep` und die Funktion `GetNeighbours` sehr zeitintensiv sind. Dies liegt in erster Linie daran, dass diese Funktionen sehr häufig ausgeführt werden und darin sehr viele Schritte durchgeführt werden.

2.4 Was kann getan werden um die Performanz zu verbessern?

Wie bereits oben beschrieben, kann die Funktion `GetNeighbours` überarbeitet werden. Es wird in dieser Funktion sehr viel kopiert und neu erzeugt, dies könnte man durch umschreiben der Logik (wird später in einem Verbesserungsschritt gezeigt) verbessern. Eine weitere Verbesserung der Performanz könnte durch eine andere Implementierung des *Shuffle* Algorithmus für die Matrix erreicht werden. Da sich die Breite und Höhe der Matrix während der Simulation könnte immer das gleiche Array verwendet werden, welches dann zufällig sortiert werden kann. Dadurch würde es nicht mehr nötig sein



Abbildung 2: Zeitverbrauch der Methoden

immer ein neues Array zu erzeugen und dies würde neben einer besseren Performanz zu einem geringeren Speicherverbrauch führen.

3 Verbesserungen

In diesem Abschnitt sollten die oben bereits kurz angeschnittenen Verbesserungen gezeigt werden.

3.1 Version 1

In der ersten Version wurden kleinere Refactorings durchgeführt die zu einer besseren Lesbarkeit des Quelltextes beitragen sollten. Weiters wurde die *GetNeighbours* Funktion soweit verändert, dass per Zufall eine Richtung generiert wird, diese der Funktion übergeben wird und danach der gewünschte Punkt zurückgegeben wird.

```
private int GetPosition(Direction direction, int position) {
    int pos;
    switch (direction) {
        case Direction.Up:
            pos = position - Width;
            if (pos < 0) pos += _maxPosition;
            return pos;
        case Direction.Down:
            pos = position + Width;
            if (pos >= _maxPosition) pos -= _maxPosition;
            return pos;
        case Direction.Left:
            pos = position - 1;
            if ((pos + 1) % Width == 0) pos += Width;
            return pos;
        case Direction.Right:
            pos = position + 1;
            if (pos % Width == 0) pos -= Width;
            return pos;
        default:
            throw new ArgumentException("Directiontype not
                supported.", nameof(direction));
    }
}
```

```

    }
}

public int SelectFreeNeighbor(int position) {
    ShuffleArray(_directions);
    foreach (var direction in _directions) {
        var point = GetPosition(direction, position);
        if (Grid[point] == null)
            return point;
    }
    return -1;
}

```

Dadurch, dass die Position vorher ermittelt wird, müssen nicht immer alle Positionen generiert werden sondern es wird vorher schon bestimmt welche Richtung das ausgewählt wird. Insofern die zurückgegebene Richtung nicht der gewünschten entspricht, wird eine weitere Richtung geprüft. Weiters gelingt eine Performanzverbesserung durch Ersetzen der *Shuffle*-Methode durch den Fisher-Yates-Sortieralgorithmus wird ebenfalls eine deutliche Performanzverbesserung erreicht:

```

private void ShuffleArray<T>(T[] arr) {
    for (var i = arr.Length - 1; i > 0; i--) {
        var index = _random.Next(i);
        if (index == i)
            continue;
        var tmp = arr[index];
        arr[index] = arr[i];
        arr[i] = tmp;
    }
}

```

3.2 Version 2

Durch ein Ändern der Reihenfolge der Checks auf den Typ des Nachbarfeldes kann bereits vorzeitig entschieden werden, ob der Wert gültig ist oder nicht. Die Reihenfolge dieser Checks bringt einen Performanzvorteil.

```

public int SelectNeighborOfType(Type expectedType, int position)
{
    ShuffleArray(_directions);
    foreach (var direction in _directions)
    {
        var point = GetPosition(direction, position);
        var animal = Grid[point];
        if (animal == null)
        {
            continue;
        }
        if (animal.Moved)
            continue;
        if (animal.GetType() == expectedType)

```

```

        return point;
    }
    return -1;
}

```

Weiters wurde die Berechnung der Gesamtfelder in den Konstruktor ausgelagert, da diese nicht bei jedem Aufruf erfolgen muss. In einem weiteren Schritt wurde das Setzen des Moved Flag insofern umgebaut, dass der Wert der aktuellen Iteration gespeichert wird und immer dieser Verglichen wird, dadurch ist das Aufrufen der *Commit()* Methode am ende jedes Durchlaufs nicht mehr nötig.

```

public void ExecuteStep() {
    ShuffleArray(_randomMatrix);
    Iteration++;
    for (var i = 0; i < _maxPosition; i++) {
        var animal = Grid[_randomMatrix[i]];
        if (animal != null && !animal.Moved)
            animal.ExecuteStep();
    }
}

```

3.3 Version 3

Bei der Version 3 wurde das mehrdimensionale Array durch ein eindimensionales Array mit *int* Werten statt *Point* Werten ersetzt. Dies spart erstens Speicher und die Generierung des Arrays wird dadurch sehr stark vereinfacht, da keine doppelt verschachtelten Schleifen mehr für das Initialisieren bzw. Ändern des Arrays nötig sind.

```

public OptimizedWatorWorld(Settings settings) {
    CopySettings(settings);
    _maxPosition = Width * Height;
    _directions =
        Enum.GetValues(typeof(Direction)).Cast<Direction>().ToArray();
    _rgbValues = new byte[_maxPosition * 4];
    _random = new FastRandom();
    Grid = new Animal[_maxPosition];
    Iteration = 0;

    for (var i = 0; i < _maxPosition; i++) {
        var value = _random.Next(_maxPosition);
        if (value < InitialFishPopulation)
            Grid[i] = new Fish(this, i, _random.Next(0,
                FishBreedTime));
        else if (value < InitialFishPopulation +
            InitialSharkPopulation)
            Grid[i] = new Shark(this, i, _random.Next(0,
                SharkBreedEnergy));
        else
            Grid[i] = null;
    }
    _randomMatrix = Enumerable.Range(0, _maxPosition).ToArray();
}

```

Für Version 3 wurde weiters der Standard .NET Randomizer durch eine schnellere Implementierung eines Randomizers ersetzt (<http://www.codeproject.com/Articles/9187/A-fast-equivalent-for-System-Random>). Weiters wurde in der Klasse *Fish* und in der Klasse *Shark* der Check auf das Moved Flag entfernt, bei dem eine Exception geworfen wurde, da dieser Fall nicht auftreten kann und im Grunde genommen eher durch einen Unittest abgedeckt werden sollte. Weiters wird in der Funktion *SelectNeighbourOfType* ein freies Feld über ein Outputargument zurückgegeben falls kein Feld mit dem gewünschten Typ existiert, wodurch der zweite Aufruf der *SelectNeighbours* Funktion eingespart wird.

```
public int SelectNeighborOfType(Type expectedType, int position,
    out int freeField)
{
    ShuffleArray(_directions);
    freeField = -1;
    foreach (var direction in _directions)
    {
        var point = GetPosition(direction, position);
        var animal = Grid[point];
        if (animal == null)
        {
            freeField = point;
            continue;
        }
        if (animal.Moved)
            continue;
        if (animal.GetType() == expectedType)
            return point;
    }
    return -1;
}
```

In den Klassen *Shark* und *Fish* wurden Änderungen vorgenommen, sodass der Aufruf der Methoden *Spawn* nicht erfolgt, insofern im vorhinein schon sichergestellt ist, dass diese Methode zu keinem Ergebnis führen würde, da keine Felder frei sind und somit kein neuer Fisch oder Hai erzeugt werden kann.

```
public override void ExecuteStep() {
    Age++;
    Energy--;
    int freeField;
    var fish = World.SelectNeighborOfType(typeof(Fish),
        Position, out freeField);
    if (fish != -1) {
        Energy += World.Grid[fish].Energy;
        Move(fish);
    }
    else if (freeField != -1)
        Move(freeField);

    if (fish != -1 || freeField != -1) {
        if (Energy >= World.SharkBreedEnergy)
            Spawn();
    }
}
```



```

    }
    if (Energy <= 0)
        World.Grid[Position] = null;
}

```

4 Ergebnisse

Durch die beschriebenen Verbesserungen konnte für die gesamte Simulation ein gesamter Speedup von ca. **2,8** erreicht werden. Details können der folgenden Tabelle entnommen werden.

	Original	Version 1	Version 2	Version 3
1. Durchlauf	6293,60	2846,71	2536,74	2143,34
2. Durchlauf	6141,90	2836,63	2587,69	2183,72
3. Durchlauf	6013,10	2640,58	2571,44	2142,96
4. Durchlauf	5992,79	2663,15	2548,94	2142,13
5. Durchlauf	6038,73	2636,04	2558,21	2156,96
Durchschnitt	6096,02	2724,62	2560,60	2153,82
Standardabweichung	211,97	98,19	29,99	39,14
Speedup zur vorherigen Version		2,24	1,06	1,19
Speedup zum Original		2,24	2,38	2,83