

Geburtstagskuchen

Inhalt

1	Lösungsidee.....	1
1.1	Zeichnen einer Herzform.....	1
1.2	Teilaufgabe 1	2
1.3	Teilaufgabe 2	3
2	Umsetzung	4
3	Laufzeit.....	6
4	Erweiterung: Viele tolle Formen.....	6
5	Erweiterung: Farbige Kerzen	6
6	Beispiele.....	7
7	Quellcode.....	11

1 Lösungsidee

1.1 Zeichnen einer Herzform

Es wird hier von einem informatischen Koordinatensystem ausgegangen. Das heißt, dass der Ursprung oben links liegt und von dort aus die positive x-Achsenrichtung nach rechts und die positive y-Achsenrichtung nach unten zeigt. Als Drehsinn wird kommt der mathematische Drehsinn zum Einsatz, das heißt Winkel werden linksdrehend angegeben.

Die Kontur eines Herzens besteht aus zwei Kreissektoren und zwei Linien. Dabei stehen zwei Parameter zur Verfügung, mit denen sich die Form eines Herzens anpassen lässt: Der Radius der beiden Kreissektoren r und ihr Winkel α , genauer gesagt der Winkel zwischen der x-Achse und der Linie zwischen dem Kreismittelpunkt und dem Punkt, an dem der Kreisbogen in eine Linie übergeht.

Um aus diesen Parametern ein Herz zu formen müssen erst 4 Punkte berechnet werden:

- Der Mittelpunkt M des linken Kreisbogens liegt bei $(r|r)$. Um die Position eines beliebigen Punkts auf diesem Kreis zu bestimmen, wählt man Polarkoordinaten mit einem beliebigen Winkel und dem Radius des Kreises, wandelt sie in kartesische Koordinaten um und addiert zu diesen den Ortsvektor von M . Der hier gesuchte Winkel ist $180^\circ + \alpha$, weshalb die kartesischen Koordinaten des Berührungpunktes zwischen linkem Kreis und linker Linie (der Punkt P_L) vereinfacht $(r - \cos(\alpha) \times r | r + \sin(\alpha) \times r)$ sind.
- Die beiden Kreissektoren berühren sich am Punkt $P_O = (2 \times r | r)$.
- Der Berührungpunkt zwischen rechtem Kreis und rechter Linie (P_R) kann fast vollständig analog zum linken Berührungpunkt ermittelt werden. Es muss nur beachtet werden, dass der Mittelpunkt des rechten Kreises bei $(3 \times r | r)$ liegt und der Winkel $-\alpha$ ist. Folglich ergeben sich die Koordinaten $(3 \times r + \cos(\alpha) \times r | r + \sin(\alpha) \times r)$.
- Um den Schnittpunkt zwischen den beiden Linien (P_u) zu berechnen, wird zuerst der Richtungsvektor \vec{v} einer Gerade mit einem Winkel zur x-Achse von $180^\circ + \alpha$ ermittelt. Dazu kommen wieder Polarkoordinaten zum Einsatz, deren Ortsvektor \vec{v} entspricht. Vereinfacht

ergibt sich $\vec{v} = \begin{pmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{pmatrix}$. Der Aufpunkt der Gerade ist dabei P_R . Um P_U zu bestimmen, wird jetzt nur noch der Schnittpunkt der Gerade mit der Gerade $x = 2 \times \vec{v}$ berechnet.

Hat man die Herzform berechnet, muss man noch herausfinden können, ob ein bestimmter Punkt sich in mindestens einer dieser Formen und damit innerhalb des Herzens befindet:

- Ein Kreis mit dem Radius r und dem Mittelpunkt $(r|r)$
- Ein Kreis mit dem Radius r und dem Mittelpunkt $(3 \times r|r)$
- Ein Dreieck zwischen den Punkten P_L, P_O, P_R
- Ein Dreieck zwischen den Punkten P_L, P_U, P_R

Ich habe mich allerdings dazu entschieden, den Schnittpunkt mit polygonaler Approximation zu bestimmen, da diese implementierungstechnisch weniger aufwändig ist, weil diese Methode direkt von der verwendeten Zeichen-API angeboten wird. Dabei wird zuerst ein Polygon geformt, das eine Abschätzung der Herzform darstellt und dann wird geprüft, ob ein bestimmter Punkt sich in diesem Polygon befindet. Das führt zwar zu einer geringeren Genauigkeit, aber da Kerzen nur an ganzzahligen Koordinaten platziert werden dürfen, ist das nicht problematisch. Auch dauert polygonale Approximation länger, da aber das Überprüfen, ob ein Punkte im Herz ist nur für einen kleinen Teil der Laufzeit verantwortlich ist, ist das ebenfalls vernachlässigbar.

Die hier beschriebene Kollisionsprüfung beinhaltet keinen erzwungenen Mindestabstand zum Kuchenrand. Deshalb kann man Kerzen auch direkt am Rand platzieren. Da aber theoretisch nichts gegen eine solche Platzierung spricht und es trivial wäre, die Kollision dementsprechend abzuändern, habe ich mich dafür entschieden, keinen Mindestabstand zu erzwingen.

1.2 Teilaufgabe 1

Um zu entscheiden, wie gut eine Platzierung von Kerzen C ist, werden zuerst die Abstände jeder Kerze $c_i \in C$ zu ihrem nächsten Nachbarn ($d_s(c_i)$) ermittelt. Dabei gilt

$$d_s(c_i) = \min(\{dist(c_i, c_j) | c_j \in C \setminus \{c_i\}\})$$

, wobei $dist(c_i, c_j)$ die euklidische Distanz zwischen den Kerzen c_i und c_j angibt.

Als einfach Bewertungsfunktion kann man jetzt das arithmetische Mittel aller kürzesten Distanzen nehmen. Das ist gegeben als

$$A(C) = |C|^{-1} \sum_{c_i \in C} d_s(c_i).$$

Je größer $A(C)$ ist, desto besser ist die Platzierung der Kerzen, denn die Maximierung des arithmetischen Mittels sorgt dafür, dass die einzelnen Kerzen so weit wie möglich voneinander entfernt sind, was nur der Fall sein kann, wenn die Kerzen gleichmäßig verteilt sind. Diese Bewertungsfunktion liefert zwar akzeptable Ergebnisse, es wird aber nicht berücksichtigt, wie stark die einzelnen Distanzen vom Durchschnitt voneinander abweichen. Dadurch kann es passieren, dass eine Verteilung, die von dieser Funktion hoch bewertet wird, aufgrund der unterschiedlichen Abstände zwischen den Kerzen etwas strukturlos und chaotisch aussieht. Deshalb wird zusätzlich noch die Abweichung der einzelnen Distanzen von Durchschnitt berechnet. Dabei kommt aus Performancegründen die mittlere absolute Abweichung, also

$$V(C) = |C|^{-1} \sum_{c_i \in C} |A(C) - d_s(c_i)|$$

anstatt der üblichen Standardabweichung zum Einsatz.

Jetzt müssen die beiden Werte noch zu einer Bewertungsfunktion $E(C)$ kombiniert werden. Eine einfache Möglichkeit wäre es, $A(C)$ durch $V(C)$ zu teilen. Wenn aber die Abweichung sehr klein wird, dann würde diese Bewertungsfunktion stark ansteigen, weshalb das Minimieren der Abweichung weitaus mehr Einfluss auf die Bewertung hat als das Maximieren des arithmetischen Mittels. Deshalb habe ich mich für die Formel

$$E(C) = A(C) - V(C)$$

entschieden. Der Vorteil dieser Formel ist, dass sie die Abweichung zwar einbezieht, der Durchschnitt aber eine weitaus größere Auswirkung auf die letztendliche Bewertung hat. Dadurch wird sichergestellt, dass die Kerzen gleichmäßig verteilt sind, es aber auch eine Auswirkung auf die Bewertung hat, wie „geordnet“ diese Verteilung aussieht.

Durch eine ausschließliche Betrachtung der Distanz zum nächsten Nachbarn ergibt sich aber auch ein Nachteil: Kleinere „Lücken“ in der Platzierung wirken sich kaum negativ auf die Platzierung aus, da die Kerzen, die die Lücke umgeben, näher zu einer Randkerze sind als zu einer Kerze auf der gegenüberliegenden Seite der „Lücke“. Das kann im nächsten Teil der Aufgabe dazu führen, dass sich zwar eine recht regelmäßige Platzierung bildet, bei der die meisten Lücken viereckig ausschauen, aber manche Lücken eine fünfeckige oder sechseckige Form haben.

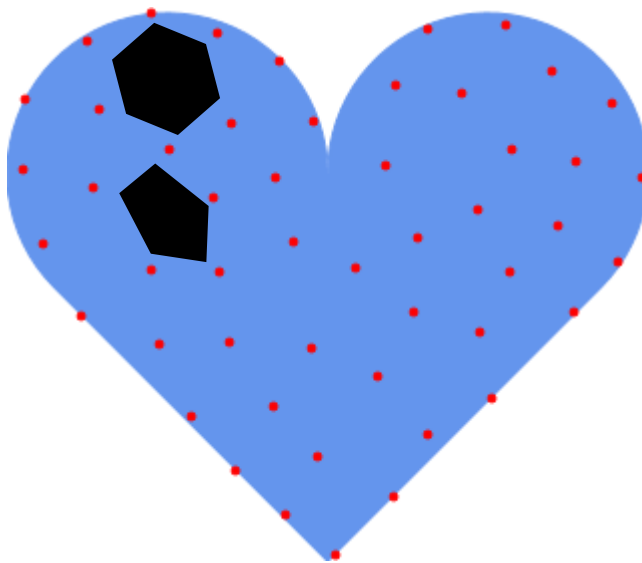


Abbildung 1 Beispiel für „Lücken“ in der Platzierung

1.3 Teilaufgabe 2

Um eine gleichmäßige Verteilung von Kerzen zu erzeugen, kommt ein evolutionärer Algorithmus zum Einsatz. Das heißt, es wird mit einer zufälligen Platzierung von Kerzen, der ersten Generation, begonnen und für jede neue Generation wird dann eine zufällig ausgewählte Kerze zufällig verschoben (Mutation). Wenn die so entstandene Platzierung besser, also die oben beschriebene Bewertungsfunktion für diese Platzierung größer ist als für die vorherige, dann wird die neue Platzierung behalten, sonst wird eine neue Mutation versucht.

Damit die Güte des Ergebnisses nicht zu sehr vom „Glück“ beim Erzeugen der ersten Generation abhängt, wird der Algorithmus parallelisiert ausgeführt. Es wird nicht nur eine erste Generation erstellt, sondern n und von diesen n Generationen werden dann parallelisiert Mutationen erzeugt. Eine Form der Selektion findet alle 10.000 Generationen statt: Sämtliche Individuen der neuesten

Generation, die mit unter 90% der besten Lösungen evaluiert wurden, werden durch das aktuell beste Individuum ersetzt. Das soll verhindern, dass eine Evolutionsreihe an einem Punkt angelangt, an dem es kaum Optimierungspotential gibt und dadurch Rechenzeit von vielversprechenderen Evolutionsreihen abgezogen wird. Da eine Mutation immer nur eine Kerze betrifft, kann es auch durchaus vorkommen, dass eine Evolutionsreihe in einer „Sackgasse“ hängenbleibt, weil eine signifikante Besserung nur möglich wäre, wenn zwei oder mehr Kerzen gleichzeitig bewegt werden.

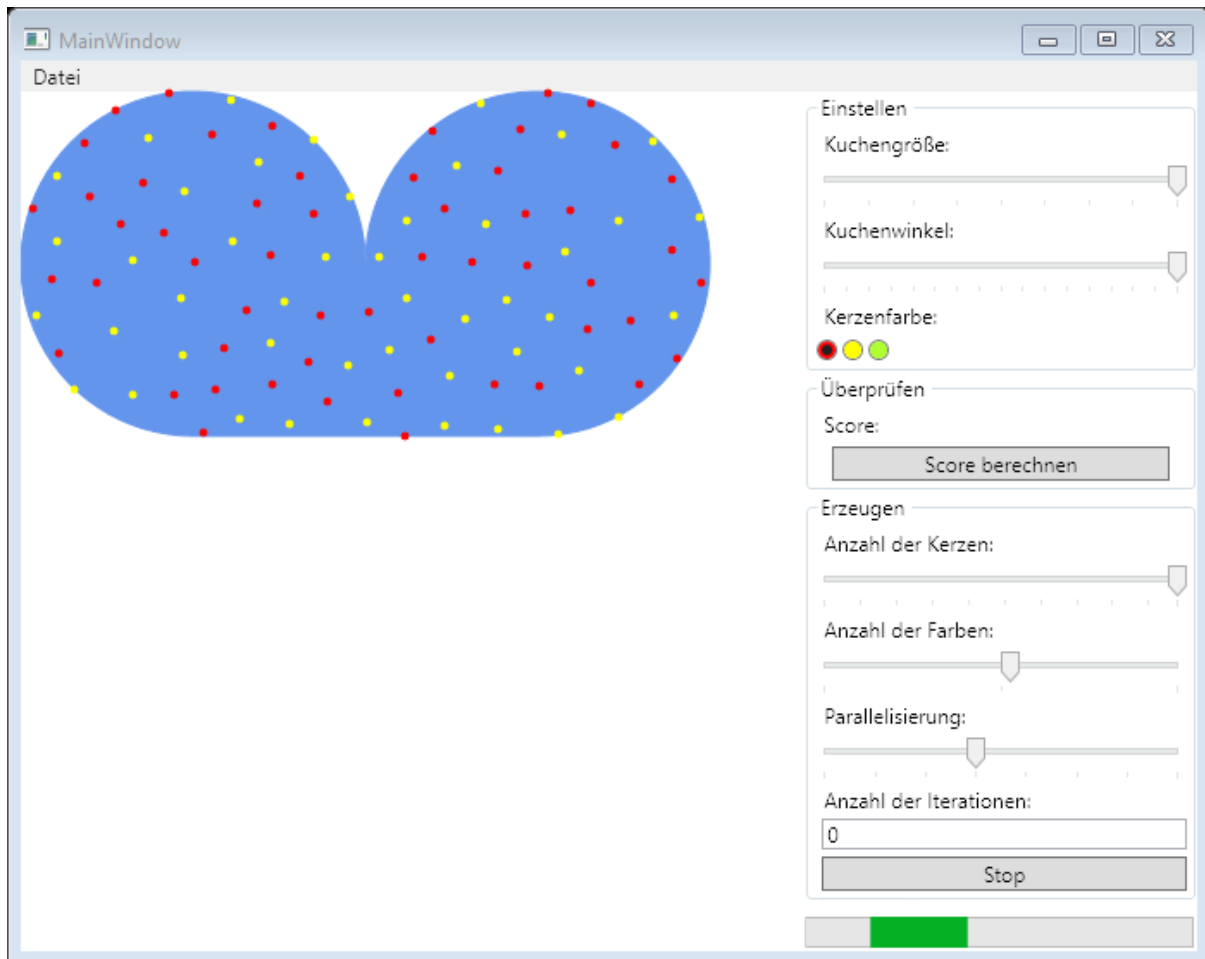
Eine weitere Verbesserung ist eine Heuristik, die den Grad der Veränderung pro Mutation im Laufe der Zeit verringert. Dabei wird ein Cooldown c nach der Formel $c = \min([Anzahl\ der\ Generationen/10000], 20)$ berechnet. Eine Verschiebung einer Kerze geschieht während der Mutation in x-Richtung um maximal $\frac{r \times 2}{c}$ und in y-Richtung um maximal $\frac{P_U \cdot Y}{2 \times c}$. Dadurch wird es im bei späteren Generationen wahrscheinlicher, dass kleinere Änderungen an den Kerzenpositionen vorgenommen werden. Da sich zu dieser Zeit schon eine einigermaßen gleichmäßige Verteilung eingestellt hat, kann davon ausgegangen werden, dass die Kerzen sich in der Nähe ihrer optimalen Position befinden, sodass größere Änderungen nur mit einer geringen Wahrscheinlichkeit zu einer Verbesserung der Platzierung führen. Diese Verbesserung führt dazu, dass die Zeit, die benötigt wird, um eine bessere Position für die Kerze zu finden im Durchschnitt geringer ist.

Auch die Auswahl der Kerze, die verschoben wird, ist wichtig für die Qualität der Ergebnisplatzierung und vor allem auch die Anzahl der Generationen die benötigt werden, um eine bestimmte Qualität der Platzierung zu erreichen. Deshalb kommt eine Heuristik zum Einsatz, die die zu verschiebende Kerze auswählt. Mit einer Wahrscheinlichkeit von 25% wird die Kerze genommen, die die geringste Entfernung zu ihrem nächsten Nachbarn hat und zu 75% wird eine zufällige Kerze gewählt.

Eine weitere Heuristik sorgt dafür, dass eine neue Kerze gewählt wird, falls es für 1000 Generationen fehlschlägt die aktuelle Kerze zu optimieren. Das kann passieren, wenn eine Kerze sich schon an einer guten Position befindet und die Wahrscheinlichkeit eine bessere Position zu finden sehr gering ist. Sollte das der Fall sein, dann wird eine neue, zufällige Kerze ausgewählt.

2 Umsetzung

Das Programm wurde in C# umgesetzt. Als GUI kommt WPF zum Einsatz. In der GUI kann der Herzwinkel und die Herzgröße eingestellt werden. Für den evolutionären Algorithmus kann die Anzahl der Kerzen und der Grad der Parallelisierung angepasst werden. Es wird dabei nicht empfohlen, den Grad der Parallelisierung zu verändern, da dieser standardmäßig auf die Anzahl der logischen Kerne angepasst wird und eine Erhöhung darüber hinaus zu einer Verschlechterung der Performance führt. Änderungen, die an den Parametern des evolutionären Algorithmus vorgenommen werden, werden außerdem nur verarbeitet, wenn der Algorithmus ohne bereits existierende Kerzen gestartet wird. Die aktuelle Platzierung der Kerzen lässt sich zurücksetzen, indem man Herzgröße oder Herzwinkel verändert. Der evolutionäre Algorithmus kann jederzeit unterbrochen und wieder fortgesetzt werden. Wenn eine Iterationszahl von 0 eingegeben wird, dann läuft der Algorithmus solange, bis er gestoppt wird, sonst stoppt er automatisch nach der angegebenen Zahl an Iterationen.



Kerzen können durch Klicken auf das Herz platziert werden. Um eine Liste von Kerzen einzulesen, wie es in der Aufgabenstellung gefordert ist, kann die Öffnen-Funktion verwendet werden. Diese erwartet eine Json-Datei, die wie folgt aufgebaut ist:

```
{
  "Candles":
  [
    {"X":21,"Y":8,"Color":0},
    {"X":143,"Y":9,"Color":0},
    {"X":80,"Y":131,"Color":0}
  ],
  "Size":40,
  "Angle":45.4752464
}
```

Die einzelnen Parameter des Dateiformates sollten selbsterklärend sein.

Die Renderinglogik für den Kuchen und die Bewertungsfunktion sind in der Klasse `Cake` implementiert. Die Klasse nimmt im Konstruktor r und α entgegen und berechnet dann in der Methode `CalculateShape()` die nötigen Punkte und Formen für das Rendern des Kuchens. Mit einem Aufruf von `Render()` werden die mit `CalculateShape()` erzeugten Formen dann auf der GUI dargestellt. In der Methode `CalculateScore()` ist die Bewertungsfunktion implementiert, die den Score des Kuchens als float zurückgibt.

Kerzen werden intern als Instanzen des Structs `Candle` dargestellt. Eine Kerze kennt dabei ihre x- und y-Position und eine Farb-ID. Der Farb-ID wird dann von `Cake` beim Rendern eine bestimmte Farbe zugewiesen. Die Position einer Kerze wird durch zwei `int` dargestellt, wodurch sichergestellt wird, dass eine Kerze sich nur an einer ganzzahligen Position befindet.

In der Klasse `CakeGenerator` ist der evolutionäre Algorithmus implementiert. Als Konstruktorargumente fordert die Klasse entweder einen bestehenden Kuchen oder die nötigen Parameter zum Erstellen eines neuen. Zusätzlich muss der Grad der Parallelisierung angegeben werden. Durch einen Aufruf von `Optimize()` kann man den evolutionären Algorithmus starten und durch ein `CancellationToken`, das an `Optimize()` übergeben wird kann man die Ausführung auch jederzeit unterbrechen. `Optimize()` überprüft zuerst, ob es sich um den Erstaufbau handelt. Wenn ja, dann werden mehrere zufällige Kuchen erzeugt, und zwar so viele wie der Parallelisierungsgrad vorgibt. Sonst werden alle Kuchen, deren Score nicht mindestens 90% des Scores des besten Kuchens ist, durch den besten Kuchen ersetzt. Dann werden Threads (so viele wie der Parallelisierungsgrad vorgibt) erstellt und jeder Thread führt `OptimizeInternal()` aus, wo der eigentliche evolutionäre Algorithmus implementiert ist.

Besonders zu erwähnen ist noch der Mechanismus, der den besten gefundenen Kuchen speichert. Da gleichzeitig mehrere Kuchen gefunden werden können, die besser als der aktuell gespeicherte beste Kuchen sind, muss dieser Zugriff synchronisiert werden. Damit aber nicht gewartet werden muss, um auf den besten Kuchen zuzugreifen, erfolgt das Aktualisieren des Kuchens asynchron durch einen Aufruf der Methode `UpdateCake()`.

3 Laufzeit

`CalculateScore()` betrachtet jedes Paar von Kerzen und berechnet die Distanz zwischen ihnen. Dazu sind $n^2 - n$ Betrachtungen nötig, wobei n für die Anzahl der Kerzen steht. Die Berechnung der Distanz und die Zuweisung dieser bestehen lediglich aus Operationen die in $\mathcal{O}(1)$ ablaufen. Danach erfolgt noch die Berechnung des Durchschnitts und der mittleren absoluten Abweichung die jeweils in $\mathcal{O}(n)$ ablaufen. So ergibt sich eine komplette Laufzeit für das Berechnen der Bewertungsfunktion von $\mathcal{O}(n^2)$.

Die Laufzeit des evolutionären Algorithmus ist logischerweise abhängig von der Anzahl der Iteration, im Folgenden als i bezeichnet. Für jede Iteration werden mehrere Zufallsgrößen berechnet, was je in $\mathcal{O}(1)$ geschieht. Dann wird (ebenfalls in konstanter Zeit) eine Kerze verschoben. Die Überprüfung, ob eine Kerze innerhalb des Herzens liegt ist lauffzeitmäßig nicht abhängig von i und n , weshalb sie auch als konstant angenommen werden kann. Sollte die neue Position einer Kerze gültig sein, so wird die Bewertungsfunktion ausgeführt, was eine Laufzeit von $\mathcal{O}(n^2)$ hat. Deshalb liegt die Worst-Case-Laufzeit des evolutionären Algorithmus bei $\mathcal{O}(i \times n^2)$.

4 Erweiterung: Viele tolle Formen

Dank der Möglichkeit den Winkel des Herzens zu verstellen, ist das Programm viel universeller einsetzbar. So sind zum Beispiel auch beliebige Kuchen in Eisform (ungefähr ab $\alpha < 20^\circ$), Mausehrenform ($\alpha \approx 80^\circ$), umgedrehter Sonnenbrillenform und Oberseite-eines-alten-Filmprojektors-Form ($\alpha = 90^\circ$) möglich.

5 Erweiterung: Farbige Kerzen

Eine weitere Erweiterung ist das Einführen von farbigen Kerzen. Ein wesentlicher Teil der Erweiterung ist das Finden einer neuen Bewertungsfunktion. Um eine Platzierung C zu evaluieren,

wird zuerst die bereits bekannte Funktion auf alle Kerzen einer Farbe angewendet. Das arithmetische Mittel dieses Wertes für alle Farben bezeichnen wir als $E_{colored}(C)$. Dadurch, dass versucht wird, $E_{colored}(C)$ zu optimieren wird versucht, alle Kerzen einer Farbe möglichst gleichmäßig aber dennoch möglichst weit voneinander entfernt zu platzieren. Außerdem wird weiterhin $E_{all}(C)$ berechnet, damit sichergestellt wird, dass auch die Abstände zwischen den Kerzen unterschiedlicher Farbe möglichst gleichmäßig sind. Zusätzlich soll noch sichergestellt werden, dass Kerzen gleicher Farbe nicht direkt nebeneinanderliegen. Dazu wird gezählt, wie oft der nächste Nachbar einer Kerze die gleiche Farbe hat. Diese Anzahl wird als $S(C)$ definiert.

Die endgültige Gewichtung der einzelnen Werte in der Bewertungsfunktion wurde experimentell ermittelt und wird wie folgt festgelegt:

$$E(C) = 4 \times E_{all}(C) + 2 \times E_{colored}(C) - S(C)$$



Während der eigentliche Algorithmus problemlos mit beliebig viele Farben zurechtkommt, ist das für die GUI und die Renderinglogik nicht der Fall (was größtenteils an der Wahl von ungeeigneten Zeichen-APIs beim ursprünglichen Implementieren liegt). Deshalb ist die Anzahl der Farben durch die GUI auf 3 begrenzt.

Die Laufzeit der neuen Bewertungsfunktion beträgt im Worst-Case $\mathcal{O}((n^2 - n) \times 2 + n) = \mathcal{O}(n^2)$, da dieser Fall genau dann eintritt, wenn es nur eine Farbe gibt ($a^2 + b^2 < (a + b)^2$). Dadurch hat der evolutionäre Algorithmus eine unveränderte Worst-Case-Laufzeit von $\mathcal{O}(i \times n^2)$.

Da die neue Bewertungsfunktion nicht direkt proportional zur alten Bewertungsfunktion ist, liegen in der Einsendung zwei Versionen des Programms bei. Da der Sourcecode der Erweiterung aber trivialerweise zum ursprünglichen Sourcecode umgeformt werden kann, enthält die Dokumentation nur die erweiterte Version.

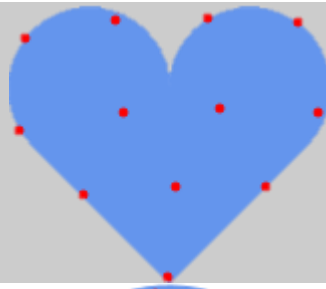
6 Beispiele

Die Dateien sind größtenteils nach einem konsistenten Schema benannt. Zuerst kommt die Anzahl der Kerzen gefolgt von einem c, dann der Radius des Herzen gefolgt von einem r und dann die Anzahl der Iterationen gefolgt von einem i. Sollte anstatt der Anzahl der Iterationen „LongEvo“ stehen heißt das, dass das Programm einfach eine Weile gelaufen ist. Alle Beispiele, bei denen die Erzeugung nicht angegeben ist, wurden von Hand erstellt.

Dateiname	Bild	Score
3cBorder.json		120,4
12c20r10000i.json		40,4

12c20r100000i.json

45,0



20c80r100000i.json

60,1



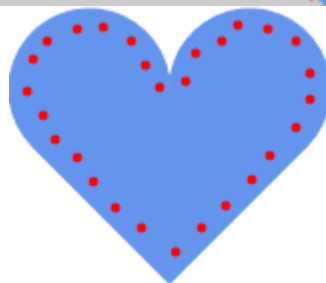
20c80rLongEvo.json

64,6



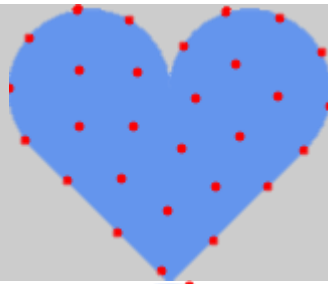
28c40rBorder.json

12,7



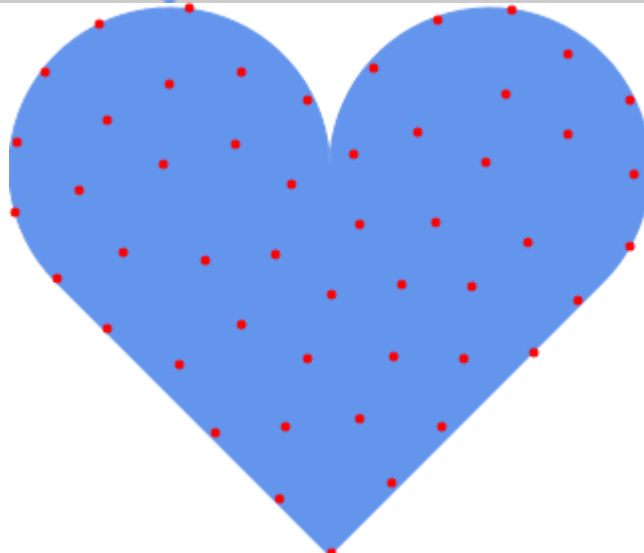
28c40r1000000i.json

26,0



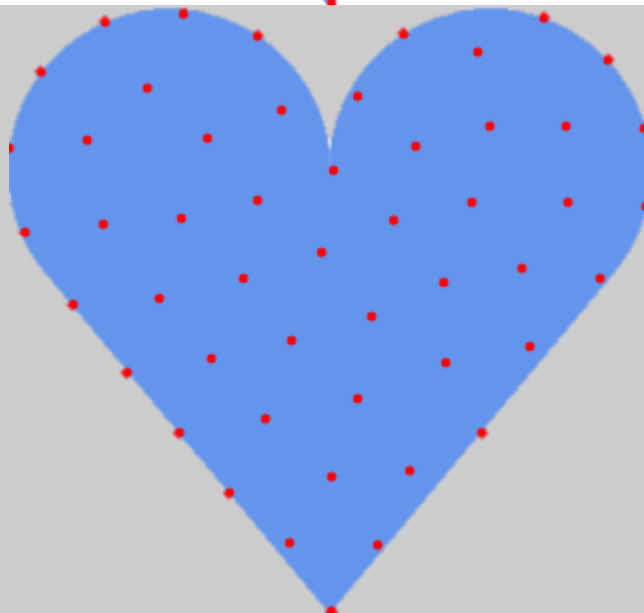
50c80r1000000i.json

34,7



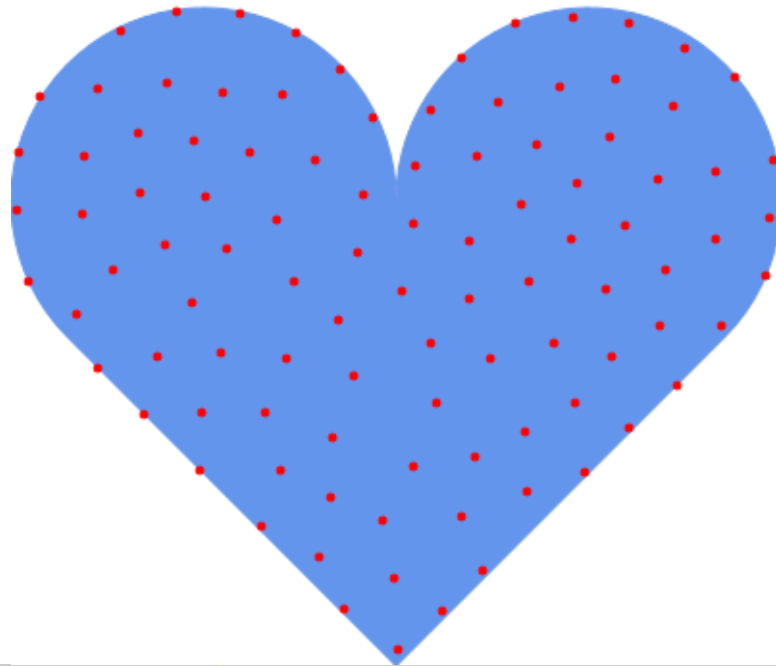
50c80rLongEvo.json

38,5



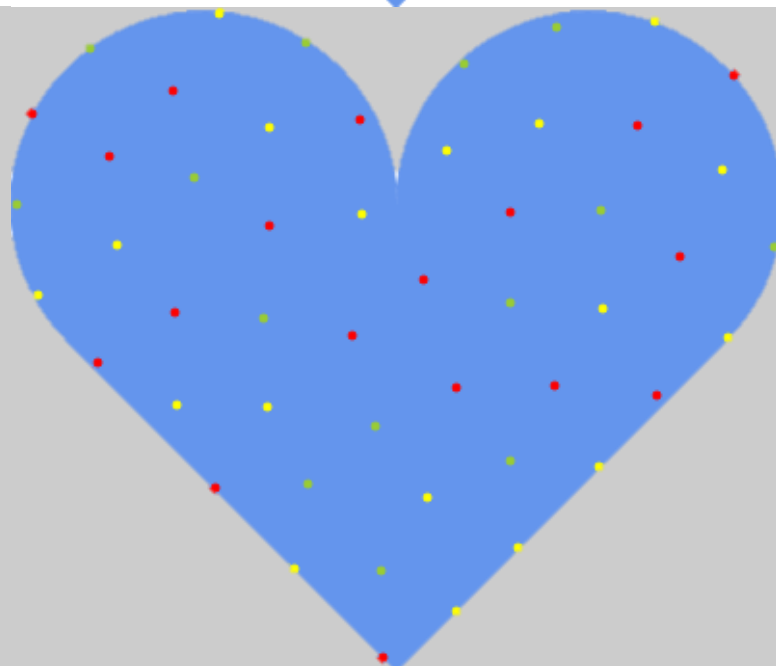
100c100rLongEvo.json

29,6



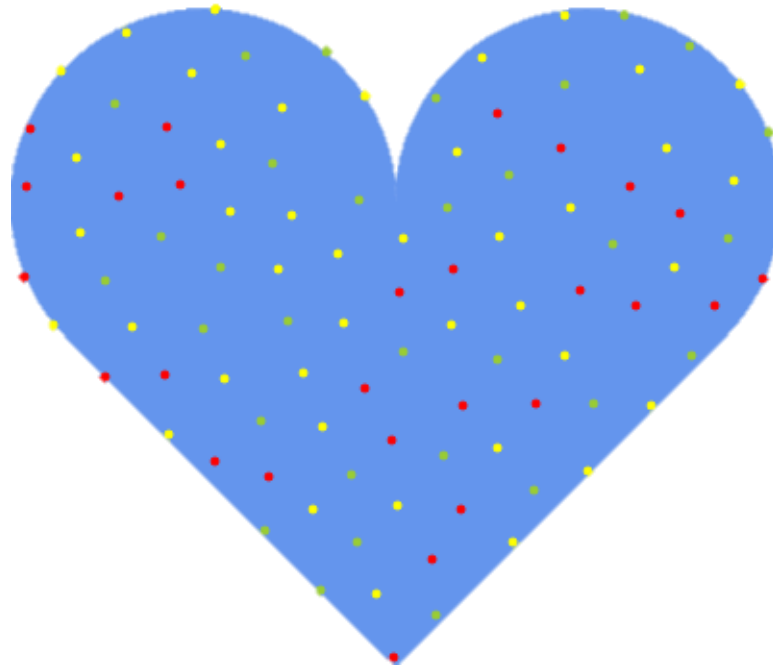
50c100rLongEvoC.json

159,9



100c100rLongEvoC.json

164,4



7 Quellcode

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Aufgabe_1___Geburtstagskuchen___GUI_
{
    struct Candle
    {
        public int X;
        public int Y;
        public int Color;

        public Candle(int x, int y, int color)
        {
            X = x;
            Y = y;
            Color = color;
        }
    }

    class Cake
    {
        public List<Candle> Candles;
        public readonly int Size;
        public readonly float Angle;

        [Newtonsoft.Json.JsonIgnore]
        public readonly Rect Bounds;
        [Newtonsoft.Json.JsonIgnore]
        public int NearestCandle = -1;
    }
}
```

```

        private Path redCandlePath, yellowCandlePath, greenCandlePath,
heartPath;
        private StreamGeometry heart;
        private Point startPoint, circleIntersectionPoint, circleEndPoint,
trianglePoint;

        private float minValue;
        private List<float> distanceToClosestNeighbor;

        public Cake(int size, float angle)
        {
            Candles = new List<Candle>();
            Size = size;
            Angle = angle;
            Application.Current.Dispatcher.Invoke(CalculateShape);
            Bounds = new Rect(0, 0, 4 * Size, trianglePoint.Y);
        }

        private void CalculateShape()
        {
            //Vorberechnen für bessere Performance
            float angle = Angle * (float)(Math.PI / 180);
            float sinAngle = (float)Math.Sin(angle);
            float cosAngle = (float)Math.Cos(angle);

            //P_L
            startPoint = new Point(Size - cosAngle * Size, Size + sinAngle *
Size);

            //P_O
            circleIntersectionPoint = new Point(2 * Size, Size);
            //P_R
            circleEndPoint = new Point(Size * 3 + cosAngle * Size, Size +
sinAngle * Size);

            //P_U
            Vector angledVector = new Vector(-sinAngle, cosAngle);
            double lineLength = (2 * Size - (Size * 3 + cosAngle * Size)) /
angledVector.X;
            trianglePoint = Point.Add(circleEndPoint,
Vector.Multiply(angledVector, lineLength));

            //Form aus Punkten erstellen
            heart = new StreamGeometry();
            using (StreamGeometryContext ctx = heart.Open())
            {
                ctx.BeginFigure(startPoint, true, true);
                ctx.ArcTo(circleIntersectionPoint, new Size(Size, Size),
Math.PI + angle, true, SweepDirection.Clockwise, true, false);
                ctx.ArcTo(circleEndPoint, new Size(Size, Size), Math.PI +
angle, true, SweepDirection.Clockwise, true, false);
                ctx.LineTo(trianglePoint, true, false);
            }
            heart.Freeze();
        }

        public bool AddCandle(int x, int y, int color, bool renderCandle = true)
        {
            if (Contains(x, y))
            {
                Candles.Add(new Candle(x, y, color));
                if (renderCandle) //Spart Renderingoverhead für
CakeGenerator

```

```

        {
            GeometryGroup candleGroup = null;
            switch (color)
            {
                case 0: //Rot
                    candleGroup =
(GeometryGroup)redCandlePath.Data;
                    break;
                case 1: //Gelb
                    candleGroup =
(GeometryGroup)yellowCandlePath.Data;
                    break;
                case 2: //Grün
                    candleGroup =
(GeometryGroup)greenCandlePath.Data;
                    break;
            }
            candleGroup.Children.Add(new EllipseGeometry(new
Point(x, y), 2, 2));
        }
        return true;
    }
    return false;
}

public bool Contains(int x, int y)
{
    return heart.FillContains(new Point(x, y), 1,
ToleranceType.Absolute);
}

public void Render(ref Canvas target)
{
    //Wenn das Herz bereits gerendert wurde, reicht es nur die Kerzen
neu zu zeichnen
    if (target.Children.Count == 2 && target.Children[1] is Path &&
        (target.Children[1] == redCandlePath || target.Children[1]
== yellowCandlePath || target.Children[1] == greenCandlePath))
    {
        RedrawCandles();
        return;
    }

    target.Children.Clear();

    //Herz rendern
    heartPath = new Path();
    heartPath.Data = heart;
    heartPath.Stroke = new SolidColorBrush(Colors.CornflowerBlue);
    heartPath.StrokeThickness = 1;
    heartPath.Fill = new SolidColorBrush(Colors.CornflowerBlue);
    target.Children.Add(heartPath);

    //Kerzen rendern
    redCandlePath = new Path();
    redCandlePath.Stroke = redCandlePath.Fill = new
SolidColorBrush(Colors.Red);
    yellowCandlePath = new Path();
    yellowCandlePath.Stroke = yellowCandlePath.Fill = new
SolidColorBrush(Colors.Yellow);
    greenCandlePath = new Path();

```

```

        greenCandlePath.Stroke = greenCandlePath.Fill = new
SolidColorBrush(Colors.YellowGreen);

        RedrawCandles();
        target.Children.Add(redCandlePath);
        target.Children.Add(yellowCandlePath);
        target.Children.Add(greenCandlePath);
    }

    private void RedrawCandles()
    {
        GeometryGroup redCandleGroup = new GeometryGroup(),
yellowCandleGroup = new GeometryGroup(), greenCandleGroup = new GeometryGroup();
        foreach (var candle in Candles)
        {
            switch (candle.Color)
            {
                case 0: //Rot
                    redCandleGroup.Children.Add(new
EllipseGeometry(new Point(candle.X, candle.Y), 2, 2));
                    break;
                case 1: //Gelb
                    yellowCandleGroup.Children.Add(new
EllipseGeometry(new Point(candle.X, candle.Y), 2, 2));
                    break;
                case 2: //Grün
                    greenCandleGroup.Children.Add(new
EllipseGeometry(new Point(candle.X, candle.Y), 2, 2));
                    break;
            }
        }
        redCandlePath.Data = redCandleGroup;
        yellowCandlePath.Data = yellowCandleGroup;
        greenCandlePath.Data = greenCandleGroup;
    }

    public float CalculateScore()
    {
        distanceToClosestNeighbor = new List<float>();
        for (int i = 0; i < Candles.Count; i++)
        {
            distanceToClosestNeighbor.Add(float.PositiveInfinity);
        }

        //Indexe der Kerzen nach Kerzenfarbe gruppieren
        List<List<int>> colorGroupings = new List<List<int>>();
        for (int i = 0; i < Candles.Count; i++)
        {
            var candle = Candles[i];
            if (candle.Color >= colorGroupings.Count) //Es existiert
noch keine Gruppierung mit dieser Farbe
            {
                //Solange Gruppierungen erstellen, bis die
entsprechende Gruppierung existiert
                for (int j = colorGroupings.Count; j <=
candle.Color; j++)
                {
                    colorGroupings.Add(new List<int>());
                }
            }

            colorGroupings[candle.Color].Add(i);
        }
    }

```

```

    }

    //Score für jede Farbe ermitteln
    List<float> scores = new List<float>();
    foreach (var color in colorGroupings)
    {
        scores.Add(CalculateScoreForColor(color));
    }

    //Distanzen zwischen allen Kerzen ermitteln
    var distanceToClosestNeighbor2 = new List<float>(Candles.Count);
    for (int i = 0; i < Candles.Count; i++)
    {
        distanceToClosestNeighbor2.Add(float.PositiveInfinity);
    }

    for (int i = 0; i < Candles.Count; i++)
    {
        for (int j = 0; j < Candles.Count; j++)
        {
            if (i == j)
                continue;
            float dist = (float)Math.Sqrt(Math.Pow(Candles[i].X
- Candles[j].X, 2) + Math.Pow(Candles[i].Y - Candles[j].Y, 2));
            if (dist < distanceToClosestNeighbor2[i])
            {
                distanceToClosestNeighbor2[i] = dist;
            }
        }
    }

    //Kerze mit dem nächsten Nachbarn ermitteln
    minValue = float.PositiveInfinity;
    for (int i = 0; i < distanceToClosestNeighbor2.Count; i++)
    {
        if (distanceToClosestNeighbor2[i] < minValue)
        {
            minValue = distanceToClosestNeighbor2[i];
            NearestCandle = i;
        }
    }

    float averageColor = scores.Average();
    float averageAll = distanceToClosestNeighbor2.Average();

    float deviation = 0;
    distanceToClosestNeighbor2.ForEach(d => deviation +=
Math.Abs(averageAll - d));
    deviation /= distanceToClosestNeighbor2.Count;

    //Anzahl der Kerzen, deren nächster Nachbar die gleiche Farbe hat
    ermitteln

    int colorMisdistribution = 0;
    for(int i = 0; i < Candles.Count; i++)
    {
        if (Math.Abs(distanceToClosestNeighbor2[i] -
distanceToClosestNeighbor[i]) < 0.0001f)
            colorMisdistribution++;
    }

    return 2 * averageColor + 4 * averageAll - 4 * deviation -
colorMisdistribution;

```

```

    }

    private float CalculateScoreForColor(List<int> colorerCandles)
    {
        //Berechne Distanzen zwischen allen Paaren in coloredCandles
        for (int i = 0; i < colorerCandles.Count; i++)
        {
            for (int j = 0; j < colorerCandles.Count; j++)
            {
                if (i == j)
                    continue;
                float dist =
                    (float)Math.Sqrt(Math.Pow(Candles[colorerCandles[i]].X - Candles[colorerCandles[j]].X,
                    2)
                    + Math.Pow(Candles[colorerCandles[i]].Y -
                    Candles[colorerCandles[j]].Y, 2));
                if (dist <
                    distanceToClosestNeighbor[colorerCandles[i]])
                {
                    distanceToClosestNeighbor[colorerCandles[i]]
                    = dist;
                }
            }
        }

        float average = (float)colorerCandles.Average();

        float deviation = 0;
        colorerCandles.ForEach(c => deviation += Math.Abs(average -
        distanceToClosestNeighbor[c]));
        deviation /= colorerCandles.Count;

        return average - deviation;
    }

    public Cake Clone() //Mache mehr Kuchen
    {
        var cake = new Cake(Size, Angle);
        cake.Candles.Capacity = Candles.Count;
        foreach (var candle in Candles)
            cake.Candles.Add(candle);
        return cake;
    }
}

class CakeGenerator
{
    public readonly int NumberOfCandles, DegreeOfParallelization, Colors;
    private float bestScore = float.NegativeInfinity;
    private Cake cake;
    private int globalIterations;

    //Der Zugriff auf den Kuchen ist threadsafe
    public Cake Cake
    {
        get
        {
            lock (cake)
            {
                return cake;
            }
        }
    }
}

```



```

    }

    public CakeGenerator(int numberOfCandles, int degreeOfParallelization,
int size, float angle, int colors)
    {
        //Spannende Zuweisungen
        NumberOfCandles = numberOfCandles;
        DegreeOfParallelization = degreeOfParallelization;
        Colors = colors;
        cake = new Cake(size, angle);

        threads = new Thread[DegreeOfParallelization];
        internalCakes = new Cake[DegreeOfParallelization];

        for (int i = 0; i < DegreeOfParallelization; i++)
            internalCakes[i] = cake.Clone();
    }

    public CakeGenerator(Cake cake, int degreeOfParallelization)
    {
        this.cake = cake;
        DegreeOfParallelization = degreeOfParallelization;
        NumberOfCandles = cake.Candles.Count;

        threads = new Thread[DegreeOfParallelization];
        internalCakes = new Cake[DegreeOfParallelization];

        for (int i = 0; i < DegreeOfParallelization; i++)
            internalCakes[i] = cake.Clone();

        //Workaround um zu verhindern das der Kuchen automatisch
überschrieben wird
        globalIterations = 1;
    }

    private Thread[] threads;
    private Cake[] internalCakes;
    public async void Optimize(int iterations, CancellationToken
cancellationToken, Action endedCallback)
    {
        //0 heißt "endlos" wiederholen
        if (iterations == 0)
            iterations = int.MaxValue;

        //Es wurden noch keine Kuchen erstellt
        if (globalIterations == 0)
        {
            Random random = new Random();
            int[] candleColors = new int[Colors];

            //Erstelle einen zufälligen Kuchen für jeden Thread
            for (int thread = 0; thread < DegreeOfParallelization;
thread++)
            {
                var cake = internalCakes[thread];
                int currentColor = -1, nextSwitch = 0;
                for (int i = 0; i < NumberOfCandles; i++)
                {
                    int x, y, color = 0;
                    if (thread == 0) //Wähle zufällige
Kerzenfarbe und zähle mit
                {

```

```

        color = random.Next(Colors);
        candleColors[color]++;
    }
    else //Wähle solange die gleiche Farbe bis
gleichviele Kerzen diese Farbe haben wie auf dem Kuchen von Thread 0
    {
        if (nextSwitch == i && currentColor <
Colors)
        {
            currentColor++;
            nextSwitch +=
candleColors[currentColor];
        }
        color = currentColor;
    }

    //Ereue solange zufällige Positionen bis
eine gültig ist
    do
    {
        x = random.Next((int)cake.Bounds.X,
(int)cake.Bounds.Width);
        y = random.Next((int)cake.Bounds.Y,
(int)cake.Bounds.Height);
    }
    while (!cake.Contains(x, y));
    cake.Candles.Add(new Candle(x, y, color));
}
}
//Es gibt schon Kuchen -> Selektion
else
{
    for (int i = 0; i < DegreeOfParallelization; i++)
        if (bestScore * 0.9 >
internalCakes[i].CalculateScore())
            internalCakes[i] = Cake.Clone();
}

//Diese Methode ist asynchron -> Tue irgendwas asynchrones damit
der Compiler nicht meckert
await Task.Run(() =>
{
    //Erzeuge und starte die Threads
    for (int i = 0; i < DegreeOfParallelization; i++)
    {
        threads[i] = new Thread(OptimizeInternal);
        threads[i].Name = "Evolution worker " + i;
        threads[i].Priority = ThreadPriority.BelowNormal;
//Sonst laggt die GUI zu sehr
        threads[i].Start(new Tuple<int, int,
CancellationTokens>(i, iterations, cancellationTokens));
    }
    //Warte darauf, dass alle Threads durchlaufen
    for (int i = 0; i < DegreeOfParallelization; i++)
    {
        threads[i].Join();
    }
});

//Führe das Callback auf dem GUI-Thread aus
Application.Current.Dispatcher.Invoke(endedCallback);

```

```

    }

    private void OptimizeInternal(object args)
    {
        //Argumente in sinnvolle Typen zurückverwandeln
        var argsTupel = (Tuple<int, int, CancellationToken>)args;
        int threadId = argsTupel.Item1;
        int iterations = argsTupel.Item2;
        var cancellationToken = argsTupel.Item3;

        var random = new Random();
        var cake = internalCakes[threadId];
        float lastScore = cake.CalculateScore();
        for (int i = 0; i < iterations; i++)
        {
            //Es muss nicht alles n-fach gezählt werden
            if (threadId == 0)
                globalIterations++;

            int randomCandle = random.NextDouble() >= 0.25 ?
cake.NearestCandle : random.Next(NumberOfCandles);
            int newX, newY;
            int oldX = cake.Candles[randomCandle].X, oldY =
cake.Candles[randomCandle].Y;
            int currentTries = 0;
            while (true)
            {
                i++;
                float cooldown =
(float)Math.Min(Math.Ceiling(globalIterations / 10000d), 20);

                //Evolutionen die nicht erfolgsversprechend sind
                zerstören
                if (i != 0 && i % 10000 == 0) //Zeit für Selektion
                {
                    if (bestScore * 0.9 > lastScore)
                    {
                        internalCakes[threadId] = cake =
Cake.Clone();
                        lastScore = bestScore;
                    }
                }

                currentTries++;
                //Wenn eine Kerze nicht mehr weiter optimiert werden
                kann, dann wird eine andere genommen
                if (currentTries == 1000)
                {
                    currentTries = 0;
                    //Aus stucks in einer Liste wird by value
                    zugegriffen -> Hacky workaround
                    var tmp = cake.Candles[randomCandle];
                    tmp.X = oldX; tmp.Y = oldY;
                    cake.Candles[randomCandle] = tmp;
                    randomCandle = random.Next(NumberOfCandles);
                    oldX = cake.Candles[randomCandle].X;
                    oldY = cake.Candles[randomCandle].Y;
                }

                newX = oldX + (int)(Math.Max(((cake.Size * 2) /
cooldown) * random.NextDouble(), 1) //Verschiebungsrang mit cooldown
                *

```

```

        (random.NextDouble() >= 0.5 ? 1 : -1));
//Zufälliges Vorzeichen
        newY = oldY + (int)(Math.Max(((cake.Bounds.Height /
2) / cooldown) * random.NextDouble(), 1)
        *
        (random.NextDouble() >= 0.5 ? 1 : -1));

//Wenn eine ungültige Mutation erzeugt wird, dann
wird sie nicht mitgezählt
    if (!cake.Contains(newX, newY))
    {
        i--;
        currentTries--;
        continue;
    }

//Mal wieder der selbe Workaround
    var tmp2 = cake.Candles[randomCandle];
    tmp2.X = newX; tmp2.Y = newY;
    cake.Candles[randomCandle] = tmp2;

//Abbruchbedingung
    if (cancellationToken.IsCancellationRequested || i
    >= iterations)
        break;

    if (cake.CalculateScore() > lastScore)
        break;
}

lastScore = cake.CalculateScore();
if (cancellationToken.IsCancellationRequested)
{
    //Es sollte sichergestellt werden, dass der Kuchen
wirklich aktualisiert wird, wenn die Optimierung abgeschlossen ist
    //Sonst kann es sein, dass ein veralteter Kuchen
gerendert wird
    UpdateCake(cake.Clone(),
lastScore).GetAwaiter().GetResult();
    break;
}

//Es ist egal wann der beste Kuchen aktualisiert wird
UpdateCake(cake.Clone(), lastScore);
}

}

//Ermöglicht es den besten gefundenen Kuchen threadsafe zu aktualisieren
private async Task UpdateCake(Cake cake, float newScore)
{
    if (cake == null)
        throw new TheCakeIsALieException();

    await Task.Run(() =>
    {
        lock (this.cake)
        {
            if (newScore > bestScore)
            {
                bestScore = newScore;
                this.cake = cake;
            }
        }
    });
}

```

```
        });  
    }  
  
    //Verhindert lustige NullReferenceExceptions beim Schließen des  
Programms  
    public void Cancele()  
    {  
        for (int i = 0; i < DegreeOfParallelization; i++)  
        {  
            threads[i]?.Abort();  
        }  
    }  
}
```