

# Aufgabe 3: Die Siedler

Teilnahme-ID: 70408

Bearbeiter/-in dieser Aufgabe:  
Robert Vetter

14. April 2024

## Inhaltsverzeichnis

1	Lösungsidee	1
2	Umsetzung	3
3	Beispiele	5
4	Wichtigste Teile des Quellcodes	7

## 1 Lösungsidee

Dieses Problem lässt sich in kleinere Abschnitte untergliedern.

1. Verteile möglichst viele Punkte mit dem Mindestabstand von  $10km$  in einem Polygon.
2. Suche innerhalb des Polygons einen Punkt, in welchem im Radius von  $85km$  möglichst viele Punkte liegen.
3. Platziere dieses und lasse Punkte außerhalb dieses Kreises weg, sodass jeder Punkt außerhalb des Kreises einen Mindestabstand von  $20km$  besitzt.

Hierbei stellte sich der 1. Abschnitt als am schwierigsten heraus. Es gibt verschiedene Ansätze, wie man diesen lösen kann. Nachfolgend möchte ich einige von Ihnen vorstellen und eine begründete Aussage tätigen, warum ich mich explizit für den einen entschieden habe.

**Dart-Throwing** Der erste Algorithmus, den ich anführen möchte, ähnelt der *Brute – Force* Methode. Hier werden einfach zufällig Städte innerhalb des Polygon platziert, wobei bei jedem "Wurfsichergestellt wird, dass der Mindestabstand eingehalten wird. Diese Methode ist jedoch sehr ineffizient und garantiert auch keine optimale Lösung, weswegen ich nun mit der Vorstellung verschiedener Ansätze fortfahren möchte.

**Zerlegung in Dreiecke** Darüber hinaus wäre es auch möglich, das Polygon in mehrere nicht überlappende Dreiecke zu zerlegen (siehe *Ear – Clipping – Algorithmus*) und dann für jedes Dreieck eine optimale Konstellation zu berechnen. Hier ergeben sich jedoch zwei Probleme:

- Insbesondere an den Berührungsstellen der Dreiecke kommt es zu einer Verletzung des Kriteriums des Mindestabstands. Häufig werden bei der Berechnung der optimalen Konstellation Punkte direkt auf die Kanten gesetzt, was unvorteilhaft ist.
- Dreiecke können sehr verschieden aussehen. Bei sehr ungleichmäßigen Dreiecken kann es passieren, dass ein großer Teil der Fläche des Dreiecks nicht mit Punkten ausgefüllt ist.

Aufgrund dieser Schwachstellen, entschied ich mich auch gegen diesen Algorithmus.

**Auffüllen des Polygons von außen nach innen** Ein weitere Antizipation, über die ich nachdachte, ist das Befüllen des Polygons von innen nach außen. Hierbei werden vorerst Punkte auf die Kanten des Polygons gelegt, wonach anschließend die Schnittpunkte zweier benachbarter Kreisradien berechnet werden und auf diese wiederum ein Punkt gesetzt wird. Diese Berechnung der Schnittpunkte und das Platzieren eines Punktes auf diesem wird solange fortgeführt, bis das Polygon vollständig gefüllt ist. Ich implementiere dieses Verfahren, jedoch ergaben sich grundsätzlich zwei Fehlerquellen, die sich auch nicht so leicht beheben lassen:

- Füllt man das Polygon von außen nach innen auf, so kann es passieren, dass bei den letzten Punkten viele freie Stellen in der Mitte entstehen, in welche auch kein Punkt platziert werden kann, da sonst das Mindestabstandskriterium verletzt wird.
- Die Berechnung der Schnittpunkte zweier benachbarter Städte ist sehr rechen- und zeitintensiv.

Dennoch liefert dieses Verfahren sehr gute Ergebnisse, wie beispielhaft in der folgenden Abbildung zu sehen ist:

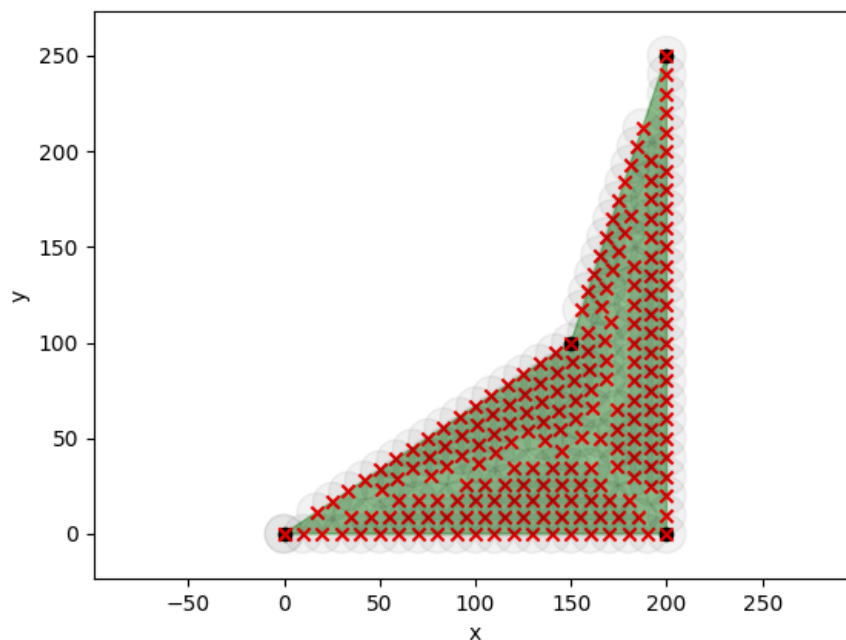


Abbildung 1: Spiralförmiges platzieren der Städte

**Umschließen des Polygons mit einem Rechteck** Das nächste Verfahren, welches mich im Endeffekt auf meine finale Idee brachte, war das Umschließen des Polygons mit einem Rechteck, für welches ich schon die maximale Punkteverteilung berechnet habe. Anschließend müsste ich einfach nur die Punkte, die außerhalb des Rechtecks liegen, entfernen. Das Problem, welches sich hier ergeben hat, liegt in den Feinheiten. So kann es je nach Lage und Form des Polygons passieren, dass einige der Punkte innerhalb des Rechtecks gerade so nicht mehr Teil des Polygons waren, wodurch am Rand des Polygons 'kahle' Stellen entstanden, welche man auch nicht mehr mit Städten besetzen könnte, da sonst der Mindestabstand nicht mehr eingehalten werden würde.

**Hexagonales Gitter** Um dieses Problem zu umgehen, entschied ich mich für die Verwendung eines hexagonalen Gitters. Diese hexagonale Anordnung von Punkten innerhalb des Rechtecks war sowieso schon die effizienteste, welche die maximale Anzahl von Punkten erreichte. Zusätzlich wird dieses Gitter horizontal und vertikal verschoben, um die maximale Anzahl an Punkten innerhalb des Polygons zu finden:

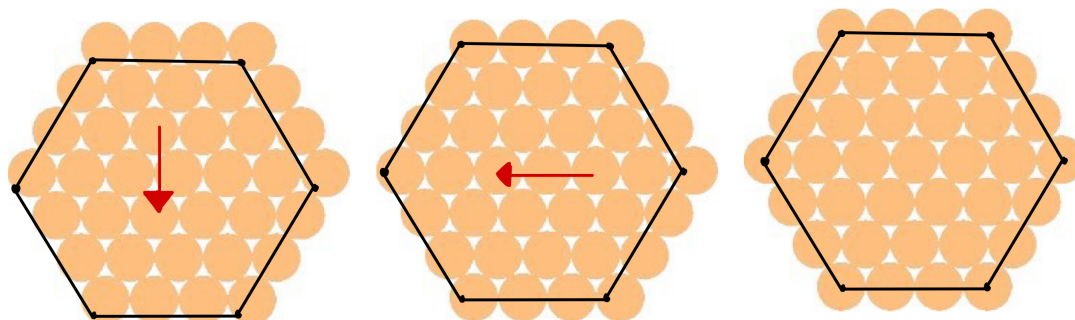


Abbildung 2: Verschiebung des Gitters

Hier ist hervorzuheben, dass die Befüllung des Polygons sehr effizient ist. Häufig stößt das Gitter bei unregelmäßigen Polygonen aufgrund seiner festen Struktur jedoch auf seine Grenzen: Somit müssen die

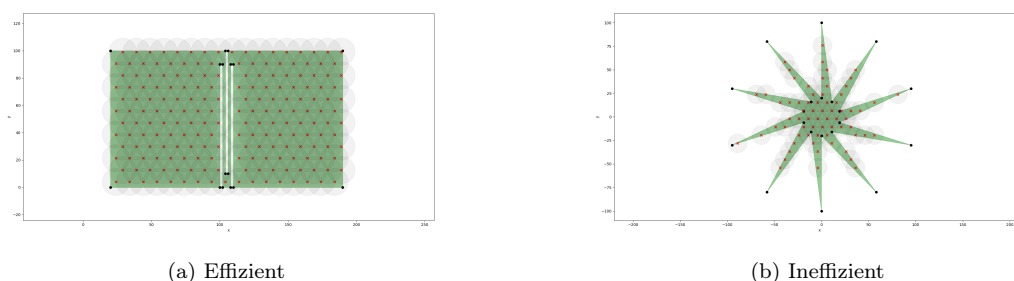


Abbildung 3: Vergleich einer effizienten Packweise mit einer ineffizienten

Punkte außerhalb noch einmal zusätzlich komprimiert werden. Hierbei kam mir die Idee, Gravitation zu simulieren. Die Punkte werden alle mit einer Kraft in Richtung eines Zielpunktes bewegt, wobei jedoch konstant der Mindestabstand eingehalten wird. Dadurch könnte Platz für zusätzliche Punkte geschaffen werden. Zur besseren Verwaltung und schnelleren Kalkulation implementierte ich zudem einen *Quadtree*. Aufgrund der jedoch nur minimalen Verbesserung der Laufzeit entschied ich mich, diesen wegzulassen. Eine weitere Frage, die ich mir stellte, zielte auf folgendes ab: Nach der Simulation der Gravitation entstehen im besten Fall einige freie Stellen. Nur wie können diese jetzt berechnet und aufgefüllt werden?

Zur Lösung dieses Problems kam mir die Idee, das Polygon mit seinen Kreisen mit den Städten als Mittelpunkt als Bild zu exportieren. Das Polygon wird mit schwarz gefüllt und die Kreise erhalten eine weiße Füllung. Nun kann man durch alle Pixel gehen und jedes mal, wenn man einen schwarzen Pixel findet, lässt sich auf ihm eine Stadt platzieren.

Um eine Lösung für den zweiten Abschnitt (Suche nach Punkt, in dessen Radius von  $85km$  maximale Anzahl an Punkten liegen) zu finden, entschied ich mich für die Erstellung eines Gitters. Anschließend wird jeder Punkt in diesem Gitter durchgegangen und die Zahl der Punkte innerhalb des Radius bestimmt. Ist die optimale Position für das Gesundheitszentrum gefunden, wird diese noch leicht variiert, um potenziell noch mehr Punkte in dem Radius zu erhalten.

Für den dritten Abschnitt wird ein Großteil der Punkte außerhalb des Kreises entfernt und der Rest wird mithilfe der soeben beschriebenen Analyse der Pixel aufgefüllt, wobei jetzt jede Stadt einen Abstand von  $20km$  besitzen muss.

## 2 Umsetzung

Die Lösungsidee wird in ein Programm in der Sprache Python umgesetzt. Zuerst wird hier das hexagonale Gitter erstellt, wobei der horizontale Abstand immer  $10km$  beträgt. Der vertikale Abstand kann über die Formel der Höhe eines gleichseitigen Dreiecks mit der Seitenlänge  $s$  berechnet werden:

$$h = \frac{\sqrt{3}}{2} \cdot s \quad (1)$$

Nach der Erstellung dieses Gitters wird die Position des hexagonalen Gitters jeweils variiert und sich für diese entschieden, welche die höchste Anzahl an Punkten im Polygon erzielt. Nun kann mit der Simulation der Gravitation begonnen werden. Dafür schrieb ich zwei Klassen. Die erste Klasse, *Particle*, modelliert individuelle Partikel, die innerhalb der Grenzen eines definierten Polygons agieren. Jedes Partikel besitzt eine Startposition, eine Geschwindigkeit, und interagiert mit anderen Partikeln durch abstoßende Kräfte, falls sie sich zu nahe kommen. Die Methode *update* berechnet diese Interaktionen und aktualisiert die Position und Geschwindigkeit basierend auf Gravitations-, Reibungs- und Zufallskräften, um ein realistisches Partikelbewegungsverhalten zu simulieren. Die zweite Klasse, *ParticleSwarm*, verwaltet eine Gruppe von Partikeln, die in einem Polygon verteilt sind. Sie bestimmt ein zentrales Ziel basierend auf dem Schwerpunkt der initialen Partikelpositionen und steuert die kollektive Bewegung der Partikel durch wiederholte Aufrufe der *update*-Methode für jedes Partikel über eine festgelegte Anzahl von Iterationen. Ziel ist es, die Endpositionen der Partikel nach Abschluss der Simulation zu ermitteln, wobei die Partikel innerhalb des Polygons verbleiben und gleichzeitig einen Mindestabstand zueinander wahren. Zur besseren Veranschaulichung ist nachfolgend ein Klassendiagramm beider Klassen abgebildet:

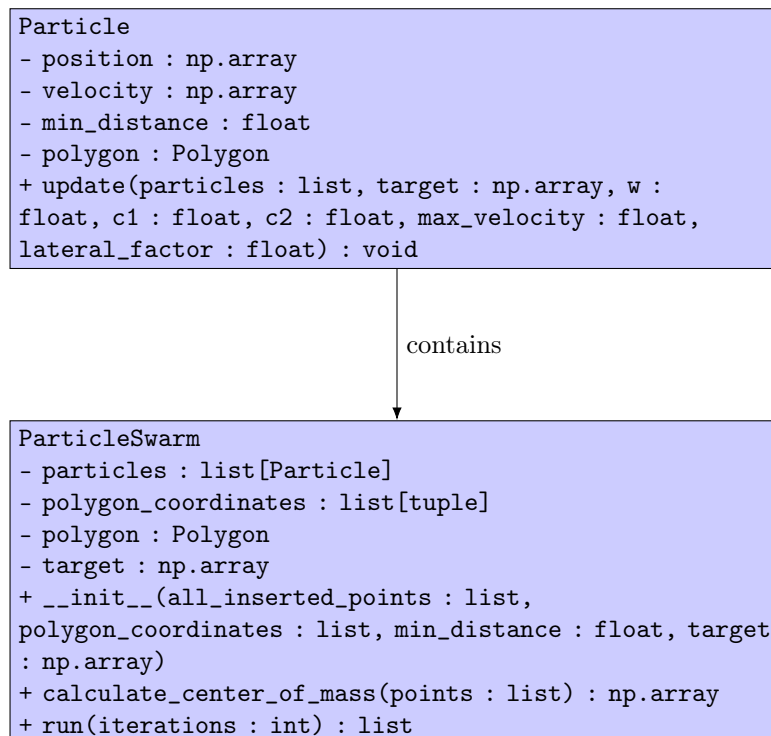


Abbildung 4: Klassendiagramm der Klassen *Particle* und *ParticleSwarm*

Nachdem die Gravitation simuliert wurde und die Punkte verdichtet sind, wird die Pixelanalyse des Bildes durchgeführt. Diese Phase ist entscheidend, da sie darauf abzielt, den Raum innerhalb des Polygons maximal effizient zu nutzen, indem sie potenzielle neue Standorte für die Platzierung weiterer Punkte ermittelt. Die Methode *save\_image* wird zunächst genutzt, um ein visuelles Bild der aktuellen Situation zu erzeugen. Sie zeichnet das Polygon und die vorhandenen Punkte in einer Bilddatei, wobei die Punkte als weiße Kreise auf einem schwarzen Hintergrund dargestellt werden. Dieses Bild dient als Grundlage für die anschließende Pixelanalyse.

Die Funktion *find\_black\_pixel\_and\_place\_city* wird eingesetzt, um im Bild nach schwarzen (nicht-weißen) Pixeln zu suchen, die potenzielle Orte für neue Punkte darstellen könnten. Diese Pixelanalyse durchsucht das Bild systematisch von der zuletzt analysierten Position aus, um sicherzustellen, dass jeder Bereich des Bildes effizient untersucht wird. Die Funktion verwendet dabei eine Toleranzschwelle, um geringfügige Variationen in der Farbe zu ignorieren und echte nicht-weiße Bereiche zu erkennen.

Sobald ein passender Pixel gefunden wird, wandelt die Funktion die Bildkoordinaten in geographische Koordinaten um, basierend auf den Skalierungen `x_scale` und `y_scale`, die aus dem Verhältnis der

realen Maße des Polygons zur Bildgröße berechnet werden. Diese neuen Koordinaten repräsentieren einen potenziellen neuen Punkt, der zur Liste der Punkte hinzugefügt werden könnte.

Die Hauptfunktion *process\_and\_save\_image* steuert diesen gesamten Prozess. Nach jeder Erstellung eines Bildes und jeder Pixelanalyse überprüft sie, ob ein neuer Punkt gefunden wurde. Ist dies der Fall, wird überprüft, ob der neue Punkt mindestens 20km von allen bestehenden Punkten entfernt ist, um die Einhaltung der Mindestabstandsregel zu gewährleisten. Falls der Punkt gültig ist, wird er zur Liste hinzugefügt und das Bild wird aktualisiert, um den neuen Punkt zu reflektieren. Dieser Zyklus wiederholt sich, bis keine weiteren Punkte mehr hinzugefügt werden können, oder bis eine bestimmte Anzahl von Iterationen erreicht ist. Nun kann die Position des Gesundheitszentrums bestimmt werden. Nun kann die Position des Gesundheitszentrums bestimmt werden. Dabei wird ein optimierter Standort ausgewählt, der auf der höchsten Dichte von Städten innerhalb eines vorgegebenen Radius basiert. Diese Methode ermöglicht es, den Standort zu identifizieren, der die größte Anzahl an Städten abdeckt und somit theoretisch die effektivste Versorgung für die umliegende Gemeinschaft bietet. Die beiden Funktionen *grid\_find\_center* und *find\_optimal\_center* arbeiten zusammen, um diesen optimalen Standort zu bestimmen. Zuerst erzeugt *grid\_find\_center* eine initiale Schätzung basierend auf einer Raster-Suche über das Polygon, während *find\_optimal\_center* diese Schätzung verfeinert, indem systematisch die Umgebung des initialen Zentrums durchsucht wird, um sicherzustellen, dass das endgültig gewählte Zentrum die maximale Anzahl an Städten innerhalb des gewählten Radius erreicht. Abschließend werden alle Punkte außerhalb des Radius des Gesundheitszentrums entfernt und mithilfe der Bildanalyse nach und nach aufgefüllt (diesmal mit einem Abstand von 20km).

### 3 Beispiele

Im nachfolgenden werde ich das Ergebnis meines Programms mit den gegebenen BWInf-Beispielen darstellen.

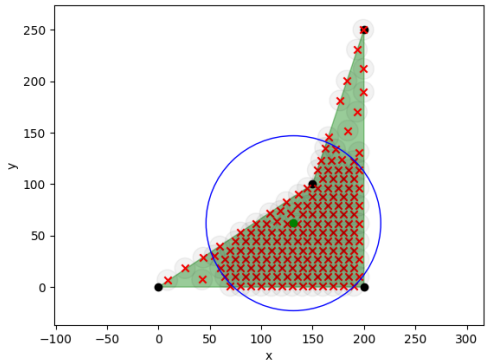
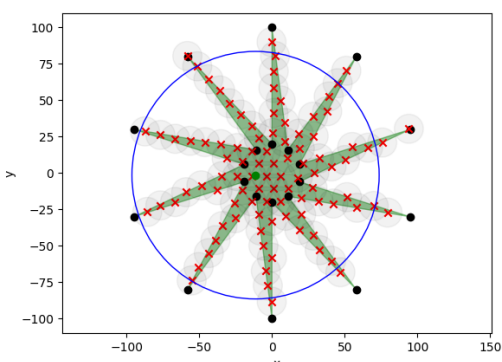
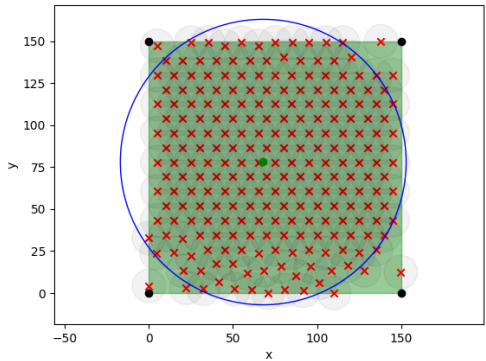
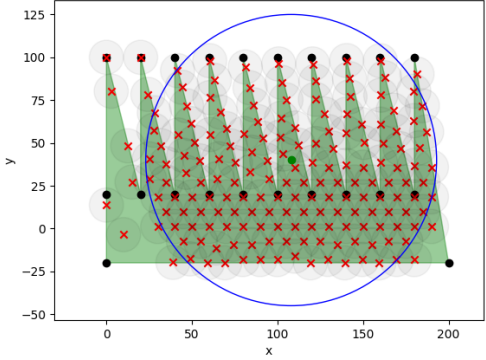
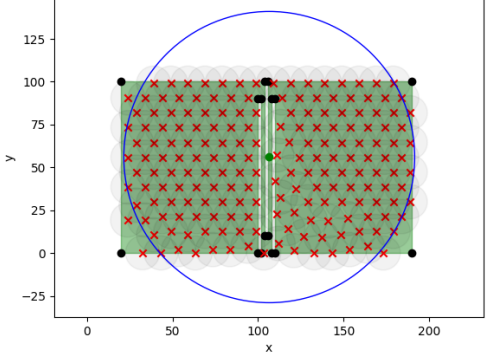
Abbildung der platzierten Städte	Daten zum Bild
	<p>Anzahl der Städte: 167            Laufzeit des Programms: 40 Sekunden</p>
	<p>Anzahl der Städte: 99            Laufzeit des Programms: 25 Sekunden</p>

Abbildung der platzierten Städte	Daten zum Bild
	Anzahl der Städte: 238 Laufzeit des Programms: 55 Sekunden
	Anzahl der Städte: 185 Laufzeit des Programms: 65 Sekunden
	Anzahl der Städte: 189 Laufzeit des Programms: 30 Sekunden

## 4 Wichtigste Teile des Quellcodes

```

1 class Particle:
2     def __init__(self, position, polygon_coordinates, target, min_distance=10):
3         self.polygon = Polygon(polygon_coordinates) # Polygon-Objekt fuer Grenzen
4         self.position = np.array(position) # Startposition
5         self.velocity = np.random.randn(2) * 0.1
6         self.min_distance = min_distance # Mindestabstand zu anderen Partikeln
7
8         # Update-Methode zur Berechnung neuer Position basierend auf Kraefte
9     def update(self, particles, target, w=0.5, c1=0.7, c2=0.5, max_velocity=2, lateral_factor=1.5):
10         r1, r2, r3 = np.random.rand(3)
11         direction_to_target = target - self.position
12         lateral_direction = np.array([-direction_to_target[1], direction_to_target[0]])
13         lateral_movement = lateral_factor * r3 * lateral_direction
14
15         # Neuberechnung der Geschwindigkeit unter Beruecksichtigung der verschiedenen Kraefte
16         new_velocity = (w * self.velocity +
17                        c1 * r1 * direction_to_target +
18                        c2 * r2 * direction_to_target +
19                        lateral_movement)
20
21         # Repulsion von anderen Partikeln
22         for other in particles:
23             if other != self:
24                 distance = np.linalg.norm(other.position - self.position)
25                 if distance < self.min_distance:
26                     repulsion = (self.position - other.position) / distance**2
27                     new_velocity += repulsion * 0.1
28
29         # Geschwindigkeitsbegrenzung
30         if np.linalg.norm(new_velocity) > max_velocity:
31             new_velocity = new_velocity / np.linalg.norm(new_velocity) * max_velocity
32
33         # Berechnung der neuen Position
34         new_position = self.position + new_velocity
35         if not self.polygon.contains(Point(new_position)):
36             return
37
38         # Ueberpruefung auf zu nahe Nachbarn
39         for other in particles:
40             if other != self and np.linalg.norm(new_position - other.position) < self.min_distance:
41                 return
42
43         # Aktualisierung von Position und Geschwindigkeit
44         self.position = new_position
45         self.velocity = new_velocity
46
47     '''Klasse, die eine Gruppe von Partikeln verwaltet und das Verhalten steuert'''
48 class ParticleSwarm:
49     def __init__(self, all_inserted_points, polygon_coordinates, min_distance=10, target=None):
50         self.polygon_coordinates = polygon_coordinates # Polygon fuer Partikelbegrenzung
51         self.polygon = Polygon(polygon_coordinates)
52         self.target = target or self.calculate_center_of_mass(all_inserted_points) # Zielzentrum
53         self.particles = [Particle(point, polygon_coordinates, self.target, min_distance)
54                           for point in all_inserted_points]
55
56         # Berechnung des Schwerpunkts der Punkte
57         @staticmethod
58         def calculate_center_of_mass(points):
59             x_coords = [p[0] for p in points]
60             y_coords = [p[1] for p in points]
61             center_x = sum(x_coords) / len(points)
62             center_y = sum(y_coords) / len(points)
63             return np.array([center_x, center_y])
64
65         # Ausfuehrung der Simulation
66     def run(self, iterations=1000):
67         for _ in range(iterations):
68             for p in self.particles:
69                 p.update(self.particles, self.target)
70
71         final_positions = [p.position for p in self.particles]
72         return final_positions

```

```

# erzeugt hexagonales Gitter, auf welchem die Staedte liegen
73 def generate_hexagonal_grid(bounds, side_length, offset):
    min_x, max_x, min_y, max_y = bounds
75     vertical_distance = math.sqrt(3) / 2 * side_length

77     points = []
    y = min_y - offset[1]
79     while y <= max_y:
        x_offset = (0 if (round(y / vertical_distance) % 2 == 0) else side_length / 2) - offset[0]
81         x = min_x + x_offset
        while x <= max_x:
83             points.append((x, y))
            x += side_length
85         y += vertical_distance

87     return points

89 def save_image(coordinates, points, radius=10):
    """Zeichnet das Polygon und die Punkte in einer Bilddatei."""
91     filename = "process.png"
    polygon = Polygon(coordinates)
93     x, y = polygon.exterior.xy

95     plt.figure()
    bounds = get_bounds_polygon(coordinates)
97     plt.xlim(bounds[0], bounds[1])
    plt.ylim(bounds[2], bounds[3])
99     plt.fill(x, y, alpha=1, color='black') # Polygon fuellen

101    for p in points:
        circle = Circle((p[0], p[1]), radius, color='white', alpha=1)
103        plt.gca().add_patch(circle) # Kreise fuer Punkte zeichnen

105    plt.gca().set_axis_off()
    plt.subplots_adjust(top=1, bottom=0, right=1, left=0, hspace=0, wspace=0)
107    plt.margins(0, 0)
    plt.savefig(filename)
109    plt.close()

111 def find_black_pixel_and_place_city(image_path, plot_x_range, plot_y_range,
    x_min, y_min, last_position, tolerance=10):
113     """Sucht den naechsten nicht-weissen Pixel und konvertiert ihn in Koordinaten."""
    with Image.open(image_path) as img:
115         pixels = img.load()
        x_scale = plot_x_range / img.width
117         y_scale = plot_y_range / img.height
        last_x, last_y = last_position

119         for y in range(last_y, img.height):
            for x in range(last_x if y == last_y else 0, img.width):
121                 r, g, b = pixels[x, y][:3]
                if not all(255 - tolerance <= channel <= 255 for channel in (r, g, b)):
123                     scaled_x = x * x_scale + x_min
                     scaled_y = (img.height - y) * y_scale + y_min
125                     return Point(scaled_x, scaled_y), (x, y)
127         last_x = 0

129     return None, (last_x, last_y)

131 def process_and_save_image(coordinates, points, last_position=(0, 0), radius=10):
    """Verarbeitet die Bilddatei zur Platzierung neuer Punkte und aktualisiert das Bild."""
133     min_x, max_x, min_y, max_y = get_bounds_polygon(coordinates)
    plot_x_range = abs(max_x - min_x)
135     plot_y_range = abs(max_y - min_y)

137     save_image(coordinates, points, radius) # Bild mit aktuellen Punkten speichern
    new_pixel, new_position = find_black_pixel_and_place_city("process.png", plot_x_range,
139     plot_y_range, min_x, min_y, last_position)

141     if new_pixel: # Wenn ein neuer Punkt gefunden wurde
        points.append((new_pixel.x, new_pixel.y)) # Neuen Punkt zur Liste hinzufuegen
143     save_image(coordinates, points, radius) # Bild aktualisieren
    return True, new_position

```



```

145     return False, last_position

147 def grid_find_center(crosses, polygon, grid_size, radius=85):
148     """Sucht das Zentrum mit der hoechsten Dichte von Staedten innerhalb eines gegebenen Radius
149     ueber ein Raster"""
150     min_x, max_x, min_y, max_y = get_bounds_polygon(crosses)
151
152     best_center = None
153     max_count = 0
154
155     for x in np.arange(min_x, max_x, grid_size):
156         for y in np.arange(min_y, max_y, grid_size):
157             if polygon.contains(Point(x, y)):
158                 count = count_cities_in_circle((x, y), crosses, polygon, radius)
159                 if count > max_count:
160                     max_count = count
161                     best_center = (x, y)
162
163     return best_center

165 def find_optimal_center(crosses, initial_center, polygon, radius=85):
166     """Passt das Zentrum an, um die hoechste Anzahl von Staedten in einem Kreis zu maximieren"""
167     step_size = 2
168     center = initial_center
169     max_count = count_cities_in_circle(center, crosses, polygon, radius)
170
171     def adjust_center(center, crosses, polygon, step_size, radius=85):
172         best_center = center
173         max_count = count_cities_in_circle(center, crosses, polygon, radius)
174
175         for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
176             new_center = (center[0] + dx * step_size, center[1] + dy * step_size)
177             if polygon.contains(Point(new_center)):
178                 count = count_cities_in_circle(new_center, crosses, polygon, radius)
179                 if count > max_count:
180                     max_count = count
181                     best_center = new_center
182
183     return best_center, max_count

185     while True:
186         new_center, new_count = adjust_center(center, crosses, polygon, step_size, radius)
187         if new_count <= max_count:
188             break
189         center, max_count = new_center, new_count
190
191     return center

193 # Hauptmethode
194 if __name__ == "__main__":
195     current_directory = os.path.dirname(os.path.abspath(__file__))
196     file = os.path.join(current_directory, 'siedler1.txt')
197     coordinates = read_coordinates(file)
198     polygon = Polygon(coordinates)
199
200     start = time.time()
201
202     bounds = get_bounds_polygon(coordinates)
203     extreme_points = find_extreme_points(polygon)
204
205     best_grid = None
206     max_points_inside = 0
207
208     for vertex in coordinates:
209         hexagonal_grid = generate_hexagonal_grid(bounds, 10, vertex)
210         points_inside_polygon = filter_points_inside_polygon(hexagonal_grid, polygon)
211
212         if len(points_inside_polygon) > max_points_inside:
213             max_points_inside = len(points_inside_polygon)
214             best_grid = points_inside_polygon
215
216     all_inserted_points = best_grid
217

```

```
print(f"Urspruengliche Anzahl an Staedten: {len(all_inserted_points)}")

219

221 for point in extreme_points:
    particle_swarm = ParticleSwarm(all_inserted_points, coordinates, target=point)
223     all_inserted_points = particle_swarm.run(iterations=50)

225     last_position = (0, 0)
    points_added = True
227     while points_added:
        points_added, last_position = process_and_save_image(coordinates, all_inserted_points,
229         last_position)
        if points_added:
231             print(f"Neue Stadt hinzugefuegt (Abstand 10km), aktuelle Anzahl:
                {len(all_inserted_points)}")
233

235     best_center = grid_find_center(all_inserted_points, polygon, 2)
    best_center = find_optimal_center(all_inserted_points, best_center, polygon)
237

    all_inserted_points = [tuple(point) for point in all_inserted_points]

239

    cities_outside = find_cities_outside_center(all_inserted_points, best_center)
241     cities_inside = []
    for city in all_inserted_points:
243         if tuple(city) not in cities_outside:
            cities_inside.append(city)
245     cities_direkt_am_kreisrand = find_nearest_outside_points(cities_inside, cities_outside)

247     all_cities = cities_inside + cities_direkt_am_kreisrand

249

    last_position = (0, 0)
    points_added = True
251     while points_added:
        points_added, last_position = process_and_save_image(coordinates, all_cities, last_position,
253         radius=20)
        if points_added:
255             print(f"Neue Stadt hinzugefuegt (Abstand 20km), aktuelle Anzahl: {len(all_cities)}")

257     print(f"Finale Anzahl an Staedten: {len(all_cities)}")
    end = time.time()
259     print("Vergangene Zeit: ", end - start, " Sekunden")

261

263     visualise_polygon(coordinates, all_cities, best_center)
```