

Aufgabe 3: Die Siedler

Teilnahme-ID: 71358

Bearbeiter/-in dieser Aufgabe:
Florian Werth

15. April 2024

Inhaltsverzeichnis

1 Lösungsidee	2
1.1 reguläre Parkettierungen	2
1.2 0-1 ILP-Formulierung	3
1.2.1 zusätzliche Heuristik	3
1.3 einfache Heuristik (Marmorplatte)	4
1.4 Optimierungen	5
2 Umsetzung	5
2.1 Allgemein	5
2.2 Auswahl Gesundheitszentrum	8
2.3 0-1 ILP Formulierung	8
2.4 reguläre Parkettierungen	9
2.5 Marmorplatte	9
3 Laufzeitanalyse	10
3.1 reguläre Parkettierungen	11
3.2 0-1 ILP	11
3.3 Marmorplatte	11
3.4 praktische Laufzeitanalyse	12
4 Beispiele	12
4.1 Siedler1.txt	13
4.2 Siedler2.txt	14
4.3 Siedler3.txt	15
4.4 Siedler4.txt	16
4.5 Siedler5.txt	17
4.6 Labyrinth	18
4.7 Mona Lisa	20
4.8 Kreis	20
4.9 kleiner Kreis	21
5 Quellcode	22
5.1 solver.py	22
5.2 main.py (nicht ganz)	25

1 Lösungsidee

Das Gesundheitszentrum ist ein Punkt innerhalb des Polygon. Siedlungen sind Punkte innerhalb des Polygon. Siedlungen, die wegen des Gesundheitszentrums geschützt sind, gehören zur Menge G für geschützt. Siedlungen, die nicht geschützt sind gehören zur Menge U für ungeschützt. Die Aufgabenstellung fordert die Anzahl an Siedlungen zu maximieren. Die Aufgabenstellung gibt außerdem folgende Bedingungen:

- Für jede Siedlung $\in G$ gilt: $\text{dist}(\text{Siedlung}, \text{Gesundheitszentrum}) \leq 85$
- Für jede Siedlung $s_1 \in G$ gilt: $\forall s_2 \in G \cup U \setminus \{s_1\} : \text{dist}(s_1, s_2) \geq 10$
- Für jede Siedlung $s_1 \in U$ gilt: $\forall s_2 \in U \setminus \{s_1\} : \text{dist}(s_1, s_2) \geq 20$

Die Siedlungen können auch als Kreise betrachtet werden. Siedlungen aus G haben den Radius 5 und Siedlungen aus U den Radius 10. Kreise aus G dürfen untereinander sich nicht überschneiden. Dasselbe gilt für Kreise aus U. Kreise aus G dürfen jedoch mit Kreisen aus U überschneiden, da sie geschützt sind, solange sie den Mindestabstand von 10 einhalten.

Da diese Aufgabe quasi das Packen von gleich großen Kreisen in ein Polygon beinhaltet, wofür ich kein optimales Verfahren in der Literatur gefunden habe, gehe ich davon aus, dass die Nutzung einer Heuristik gerechtfertigt ist.

Die Problemlösung lässt sich in zwei Aufgaben aufteilen:

- Ausfüllen einer gegebenen Fläche mit möglichst vielen gleich großen nicht überschneidenden Kreisen. Die Kreise können jedoch mit den Polygongrenzen überschneiden.
- Finden eines Standortes für das Gesundheitszentrum.

Der Standort des Gesundheitszentrums wird durch folgende Heuristik gewählt: Das Polygon wird gefüllt mit Kreisen, die den Radius 5 haben. In dem Polygon wird ein feines Gitter aus Punkten erstellt. Der Punkt, der als Standort für das Gesundheitszentrum die meisten Kreise beinhalten würde, wird als Standort für das Gesundheitszentrum gewählt.

Der gesamte Ablauf sieht dann wie folgt aus:

1. Polygon mit Radius 5 Kreisen befüllen
2. Standort für das Gesundheitszentrum auswählen
3. Die vorherigen Kreise entfernen und die Fläche innerhalb des Wirkungsbereiches des Gesundheitszentrums mit Radius 5 Kreisen ausfüllen oder die vorher platzierten Kreise benutzen und die außerhalb des Wirkungsbereiches entfernen.
4. Die noch übrige Fläche mit Radius 10 Kreisen ausfüllen.

1.1 reguläre Parkettierungen

Die Kreise werden nach dem Muster eines Quadrat-/Seckseckgitters auf der auszufüllenden Fläche platziert. Das ist schnell und kann gute Ergebnisse liefern. Ist jedoch nicht sehr anpassungsfähig. Für offene, große, simple Flächen, wie bei einem Rechteck sind die Ergebnisse jedoch sehr gut. Das Seckseckmuster ist, je nachdem meist besser, weil die Kreise enger aneinander liegen.

Für das Quadratmuster sind die Abstände trivial. Beim Seckseckmuster ist der horizontale Abstand von einer Spalte zur nächsten $\text{Radius} \cdot \frac{2}{\sqrt{3}}$. Zwei nebeneinander liegenden Spalten sind vertikal um den Radius versetzt.

Punkte der Parkettierungen, die nicht auf dem Polygon liegen oder mit vorher bereits platzierten Radius 5 Kreisen überschneiden würden, werden nicht hinzugefügt.

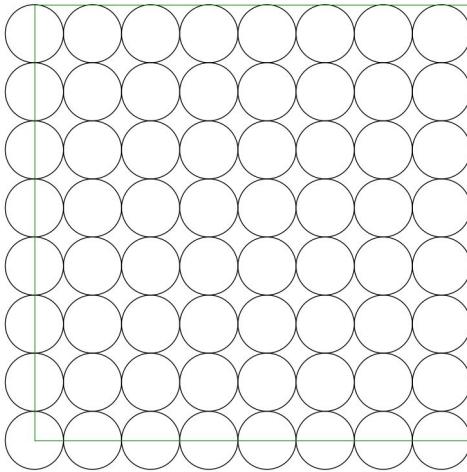


Abbildung 1: Quadratmuster 64 Kreise

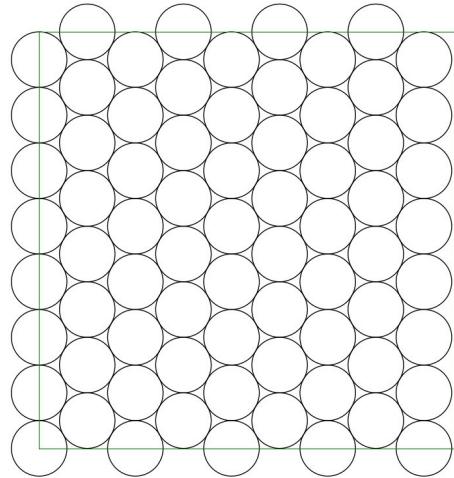


Abbildung 2: Seckseckmuster 72 Kreise

1.2 0-1 ILP-Formulierung

Das Problem wird mit 0-1-Integer-Linear-Programming¹ approximativ gelöst. Man erstellt ein Gitter über die auszufüllende Fläche. Jeder Gitterpunkt p_i entspricht einer binären Variable b_i .

$$b_i = \begin{cases} 1, & \text{ein Kreis mit Radius } r \text{ liegt auf } p_i. \\ 0, & \text{kein Kreis.} \end{cases} \quad (1)$$

Wird die Fläche mit Radius 10 Kreisen ausgefüllt, werden nur Gitterpunkte mit einbezogen, deren Abstand zu den bereits platzierten Radius 5 Kreisen größer gleich 10 ist.

Für jedes b_i berechnen wir die Menge N_i , aller $b_j (i \neq j)$ für die gilt: Abstand p_i zu p_j kleiner als $2 \cdot r$ ist. Die Menge N_i enthält also die jeweiligen Kreispositionen, die mit einem bei p_i platzierten Kreis überschneiden würden.

Für alle b_i fügen wir folgende Ungleichung hinzu, damit Kreise sich nicht überschneiden dürfen:

$$b_i \cdot |N_i| + \sum N_i \leq |N_i| \quad (2)$$

Ist $b_i = 1$, dann kann kein $b_j \in N_i = 1$ sein, weil wenn z.B. nur ein $b_j = 1$ ist $|N_i| + 1 \leq |N_i|$ nicht erfüllt werden kann.

Zu maximieren ist die Summe aller b_i , also die Anzahl an platzierten Kreisen.

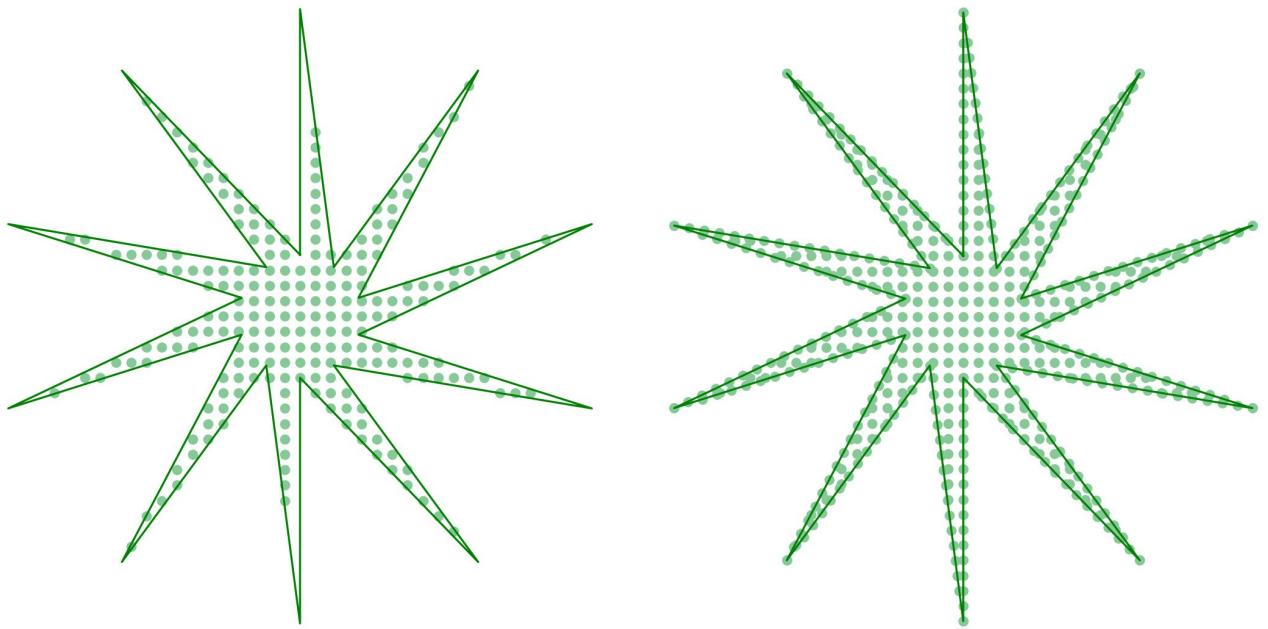
Das benutzte Gitter ist rechtwinklig. Der Abstand zwischen bildlich gesehen im Gitter direkt verbundenen Punkten ist Δ . Damit auch sehr kleine Gebiete bei einem zu großem Δ erfasst werden können, werden zu den Gitterpunkten noch Kantenpunkte im Abstand von Δ hinzugefügt.

Gitterpunkte, Polygonpunkte und Kantenpunkte, die nicht innerhalb der auszufüllenden Fläche liegen oder mit vorher bereits platzierten Radius 5 Kreisen überschneiden würden, werden nicht in Betracht gezogen.

1.2.1 zusätzliche Heuristik

Aufgrund der hohen Laufzeit bei einem zu dichten Gitter oder einem zu großen Polygon wird die zu füllende Fläche in mehrere Rechtecke eingeteilt, welche getrennt gefüllt werden. Als zusätzliche Heuristik wird eingeführt, dass Kreise möglichst tief und links gepackt werden sollen, sodass in nachfolgenden Rechtecken möglicherweise mehr Fläche zur Verfügung steht. (Die Rechtecke werden von links nach

¹Das gesamte Verfahren stammt aus folgendem Artikel: <https://doi.org/10.1016/j.ejor.2013.04.050>

Abbildung 3: Siedler2.txt Gitterpunkte mit Δ
5 ohne KantenpunktenAbbildung 4: Gitterpunkte mit Kantenpunkten
 Δ 5

rechts und von unten nach oben abgearbeitet) Ein Rechteck ist durch *bottom*, *top*, *left*, *right* definiert. Wir bestimmen *breite* und *höhe* des Rechtecks. $breite = right - left$ und $höhe = top - bottom$. Für eine Variable b_i mit Punkt p_i wird die relative Position (x_r, y_r) berechnet.

$$x_r = right - p_{i_x} \quad (3)$$

$$y_r = top - p_{i_y} \quad (4)$$

Der Kostenfunktion wird für b_i folgender Koeffizient k hinzugefügt.

$$k = 1 + (breite - x_r + höhe - y_r) \cdot 0.0000001 \quad (5)$$

Je weiter oben oder rechts man ist, desto geringer wird der Koeffizient. Eine optimale Lösung wird daher versuchen die Kreise eher weiter links und weiter unten zu platzieren. Die Anzahl an Kreisen bleibt davon ungestört, weil die Koeffizienten nur eine geringe Änderung herbeiführen und daher die Anzahl an Kreisen Priorität hat. In der Praxis habe ich meist auch einfach nur weit unten packen genommen.

1.3 einfache Heuristik (Marmorplatte)

Es gibt eine Menge B mit Punkten, die Kandidaten für einen Kreis sind. Ist die Menge B leer wird, ein Punkt gesucht, der sich so weit wie möglich unten und zweitrangig weit links befindet und auf dem ein Kreis platziert werden kann. Ein Kreis wird auf dem Punkt platziert. Für jeden neu platzierten Kreis mit Radius r und Mittelpunkt m wird B um n gleichmäßig verteilte Punkte erweitert, die genau einen Abstand von $2 \cdot r$ von m haben. (Also Punkte die auf dem Kreis mit Radius $2 \cdot r$ und Mittelpunkt m liegen). Die Punkte werden mit ein bisschen Einheitskreismagie berechnet (weiter unten näher beschrieben) und B wird gefiltert, sodass nur noch Punkte übrig bleiben, auf denen ein Kreis platziert werden könnte. Ist B nicht leer, wird der Kandidat der am weitesten links und zweitrangig weit unten ist ausgewählt und ein Kreis wird an diese Stelle gesetzt. Das alles wiederholt sich bis B nicht mehr erweitert werden kann. Die Heuristik stammt aus einem Artikel über ein effizientes Verwenden von Marmorplatten.² Daher heißt die Heuristik ab jetzt Marmorplattenheuristik.

²<https://iopscience.iop.org/article/10.1088/1757-899X/399/1/012059>

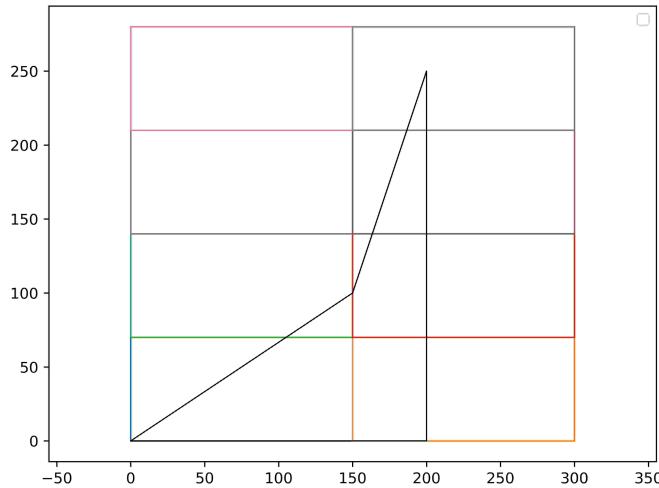


Abbildung 5: Einteilung in mehrere Rechtecke

1.4 Optimierungen

Als Optimierung wird eine räumliche Einteilung der Kreise vorgenommen. Die Fläche wird in Quadrate mit Seitenlänge 86 eingeteilt. Jeder Kreis wird dem Quadrat innerhalb dessen er sich befindet zugeordnet. Zu Berechnungen bei den Abstandsbedingungen müssen nun nicht alle Kreise, sondern nur die Kreise im eigenen und die in den 8 umliegenden Quadranten betrachtet werden. Dasselbe zahlt sich auch bei der Einhaltung der Minimumabstände aus, weil dann nur noch Kreise innerhalb dieser Quadrate betrachtet werden müssen. Somit sollte sich die Laufzeit von $\mathcal{O}(n)$ zu $\mathcal{O}(1)$ verbessern, da es dann eine konstante obere Grenze für die Anzahl an betrachteten Kreisen gibt.

2 Umsetzung

Die Lösungsideen wurde in Python geschrieben. Für ILP wird die Python Bibliothek PuLP 2.8 benutzt. Die verschiedenen Wege eine Fläche auszufüllen, sowie das Auswählen des Gesundheitszentrums ist in der Klasse **Solver** (Siehe Abbildung unten) implementiert.

2.1 Allgemein

`Solver.precision_error`

Fließpunktarithmetik ist nicht unbedingt sehr genau, daher wird bei z.B. Abstandsvergleichen das Attribut `Solver.precision_error` mit eingeführt.

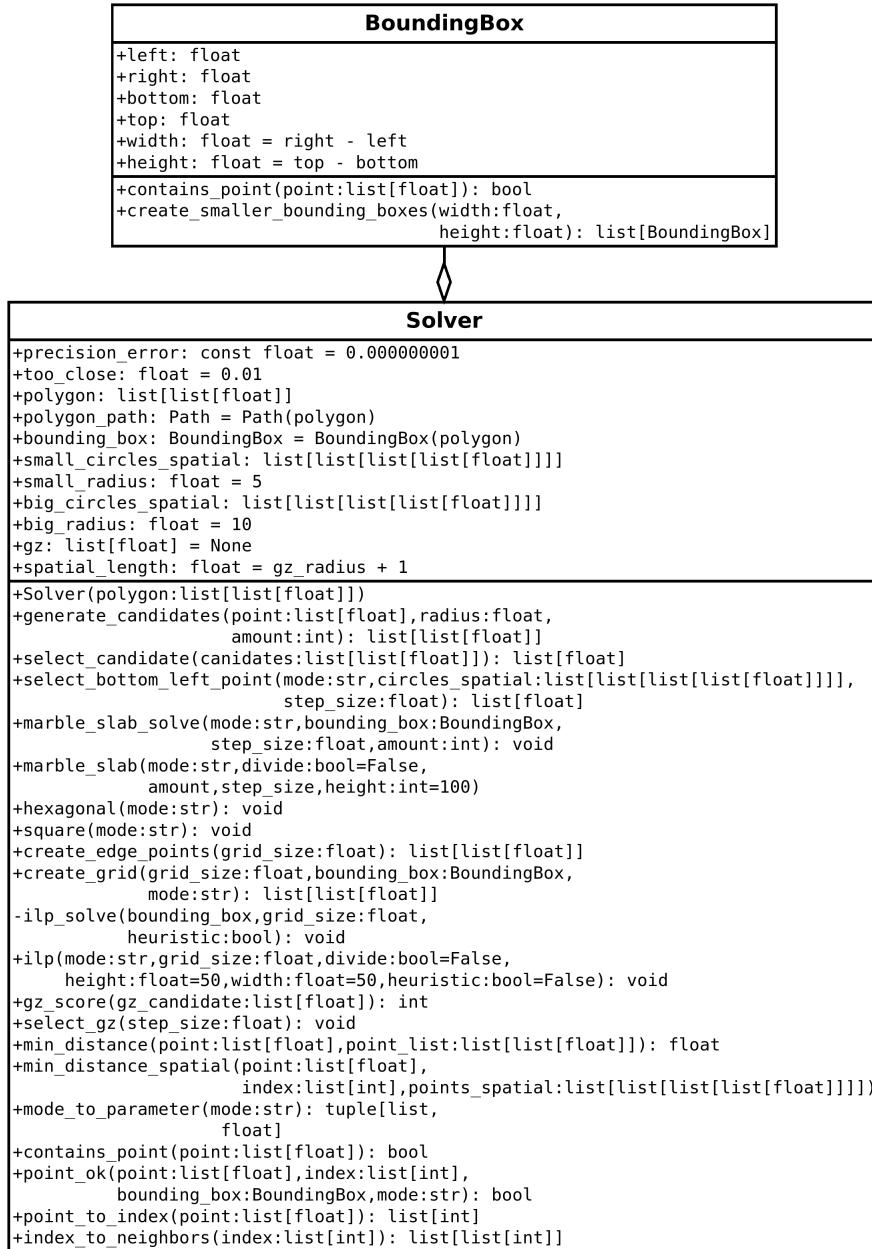
`Solver.too_close`

Gibt an, um wie viele die zu den Gitterpunkten hinzugefügten Kantenpunkten, mindestens von den Gitterpunkten entfernt. Der Mindestabstand ist $\Delta \cdot \text{Solver}.too_close$

`Solver.spatial_length`

Seitenlänge der Quadrate bei der räumlichen Datenstruktur.

`Solver.small_circles_spatial`



Beinhaltet die kleinen Kreise bzw. später die kleinen Kreise im Gesundheitszentrum, in einer räumlich geteilten 2-dimensionalen Liste bzw. Matrix, deren Elemente eine Liste an Kreisen ist. Die Anzahl an Zeilen ist $\text{ceil}(\text{Solver.bounding_box.height} / \text{Solver.spatial_length}) + 1$ und die Anzahl an Spalten ist $\text{ceil}(\text{self.bounding_box.width} / \text{self.spatial_length}) + 1$.

`Solver.big_circles_spatial`

Beinhaltet die Kreise außerhalb des Gesundheitszentrums.

`def min_distance(self, point: list[float], point_list: list[list[float]]) -> float:`

Gibt den geringsten Abstand von *point* zu den Punkten aus *point_list* zurück. Sinnvoll um die Bedingungen an die Mindestabstände einzuhalten

`def point_to_index(self, point) -> list[int]:`

Gibt für einen Punkt den Spalten- und Zeilenindex für das Quadrat, zu welchem er gehören würde in der räumlichen Liste, zurück.

```
def index_to_neighbors(self, index) -> list[list[int]]:
```

Gibt für einen Index mit (Zeile, Spalte), den Index sowie alle Indizes der umliegenden Quadrate als Liste zurück.

```
def min_distance_spatial(self, point, index, points_spatial):
    indices = self.index_to_neighbors(index)
    min_dist = 9999999
    for index in indices:
        min_dist = min(min_dist, self.min_distance(point, points_spatial[index[0]][index[1]]))
    return min_dist
```

Gibt für einen Punkt und dessen Index, den Minimumabstand zu den Punkten innerhalb `points_spatial` zurück.

```
def Solver.contains_point(self, point) -> bool:
    return self.polygon_path.contains_point(point, radius=self.precision_error) or \
           self.polygon_path.contains_point(point, radius=-self.precision_error)
```

Gibt zurück, ob `point` innerhalb des Polygons ist. Die `contains_point` Methode der `mplPath.Path` Klasse aus Matplotlib wird dafür verwendet. Damit auch Punkte auf den Grenzen erkannt werden, wird die Methode mit $\pm precision_error$ aufgerufen. Dadurch wird das Polygon vergrößert und/oder geschrumpft, deshalb wird sie zweimal aufgerufen.

```
def BoundingBox.contains_point(self, point):
```

Gibt zurück, ob `point` innerhalb der Boundingbox ist. Ist sinnvoll, wenn mehrere einzelne Rechtecke ausgefüllt werden, damit man keine Kreise außerhalb des derzeitigen Rechteckes platziert.

```
def mode_to_parameter(self, mode) -> tuple[list, float]:
    radius = self.small_radius
    circles = self.small_circles
    if mode == "outside":
        radius = self.big_radius
        circles = self.big_circles
    return circles, radius
```

Es gibt drei verschiedene Modi:

- “both”: Das gesamte Polygon wird mit Radius 5 Kreisen gefüllt.
- “inside”: Der Wirkungsbereich des Gesundheitszentrums wird mit Radius 5 Kreisen gefüllt.
- “outside”: Der noch übrige Bereich wird mit Radius 10 Kreisen ausgefüllt.

Je nach Modus bekommt man den Radius, sowie die Liste der jeweiligen Kreise, die man für den Modus braucht.

```
def point_ok(self, point, bounding_box, mode="both"):
```

Gibt zurück, ob man auf `point` einen Kreis im Modus `mode` platzieren darf.

Für alle Modi wird überprüft:

- Ist `point` innerhalb der Boundingbox

- Ist *point* innerhalb des Polygon
- Ist der geringste Abstand zu allen anderen gleichen Kreisen mindestens $2 \cdot r$

Für “inside” wird überprüft:

- Ist der Abstand zwischen *point* und dem Gesundheitszentrum höchstens 85

Für “outside” wird überprüft:

- Ist der geringste Abstand zu allen anderen kleinen Kreisen mindestens $2 \cdot Solver.small_radius$

Werden alle Bedingungen erfüllt wird True zurückgegeben, ansonsten False.

2.2 Auswahl Gesundheitszentrum

```
def gz_score(self, gz_candidate) -> int:
```

Gibt die Anzahl an Kreisen zurück, die ein Gesundheitszentrum bei *gz_candidate* beinhalten würde. Betrachtet nur die umliegenden Kreise in *Solver.small_circles.spatial*

```
def select_gz(self, step_size)
```

Durchgeht in zwei verschachtelten while-Schleifen alle Punkte innerhalb *Solver.bounding_box* im Abstand von *step_size*. Der Punkt mit dem größten *gz_score* wird als Gesundheitszentrum gespeichert

2.3 0-1 ILP Formulierung

```
def create_edge_points(self, grid_size) -> list[list[float]]:
```

Gibt die Kantenpunkte im Abstand von *grid_size* erstellt zurück.

```
def create_grid(self, grid_size, bounding_box, mode) -> list[list[float]]:
```

Erstellt mit zwei verschachtelten while-Schleifen das Punktegitter im Abstand von *grid_size*, innerhalb der gegebenen *bounding_box*. Die Punkte werden in einer 1-Dimensionalen Liste gespeichert. Die Punkte aus *create_edge_points* werden hinzugefügt. Alle Punkte in der Liste müssen die *point_ok* Methode bestehen.

```
def ilp_solve(self, grid_size, bounding_box, mode, divide=False):
```

Die möglichen Kreisstandorte werden mit *create_grid* erstellt und in der Liste *grid* gespeichert. Für jedes Element in *grid* wird eine zugehörige binäre Variable in der Liste *variables* gespeichert. In zwei verschachtelten for-Schleifen werden die Ungleichungen hinzufügt. In einer for-Schleife wird die Kostenfunktion erstellt. Ist *divide* True wird jeweils noch der Koeffizient für die Heuristik hinzufügt. Dann wird der ILP-Solver gestartet. Zum Schluss werden alle Punkte deren binäre Variable 1 ist als Kreis hinzufügt.

```
def BoundingBox.create_smaller_bounding_boxes(self, width, height) -> list[BoundingBox]:
```

Teilt die das Polygon umgebende Boundingbox in mehrere Boundingboxen mit breite *width* und Höhe *height* und gibt diese in einer Liste zurück.

```
def ilp(self, grid_size, mode="", divide=False, width=50, height=50):
    if divide == True:
        for bb in self.bounding_box.create_smaller_bounding_boxes(width, height):
            self.ilp_solve(grid_size, bb, draw=draw, divide=divide, mode=mode)
```

```
    else:
        self.ilp_solve(grid_size, self.bounding_box, divide=divide, mode=mode)
```

Mit der Methode kann man entweder ILP auf das gesamte Polygon oder auf die Rechteckteile anwenden. Der Parameter `divide` bestimmt was davon gemacht wird.

2.4 reguläre Parkettierungen

```
def square(self, mode):
```

Füllt die Fläche gemäß dem Viereckmuster auf. Das geschieht in zwei verschachtelten while-Schleifen

```
def hexagonal(self, mode):
```

Füllt die Fläche nach Seckseckmuster auf. Das geschieht in zwei verschachtelten while-Schleifen. Die Variable `alternate` bewirkt, dass zwei nebeneinander Spalten bezogen auf die Y-Koordinate versetzt anfangen. Die Variable wechselt nämlich ständig zwischen 0 und 1.

2.5 Marmorplatte

```
def generate_candidates(self, point, radius, amount) -> list[list[float]]
```

Gibt die Liste `candidates` zurück, die `amount` viele gleichmäßig verteilte Punkte auf dem Kreis mit Mittelpunkt `point` und Radius $2 \cdot \text{radius}$ beinhaltet. Die Liste `candidates` ist anfangs leer. Die Variable `step_size` wird auf 2π durch `amount` gesetzt und die Variable `curr` auf 0. Solange $curr \leq 2\pi$ ist, wird der Punkt $(point_x + 2 \cdot \text{radius} \cdot \cos(curr), point_y + 2 \cdot \text{radius} \cdot \sin(curr))$ der Liste `candidates` hinzugefügt und `curr` wird um `step_size` erhöht.

```
def select_candidate(self, candidates) -> list[float]:
```

Gibt den Punkt $\in candidates$ zurück, der die geringste x-Koordinate hat und zweitrangig die geringste y-Koordinate hat.

```
def select_bottom_left_point(self, mode, bounding_box, step_size) -> list[float]:
```

Gibt den Punkt auf der auszufüllenden Fläche zurück innerhalb der `bounding_box`, der die geringste y-Koordinate hat und zweitrangig die geringste x-Koordinate hat und außerdem die `point_ok` Methode besteht. Im Abstand `step_size` werden den dafür die Punkte durchsucht. Der erste gefundene Punkt wird zurückgegeben. (Da links nach rechts und von unten nach oben gesucht wird)

```
def marble_slab_solve(self, mode, bounding_box, amount, step_size):
```

Platziert die Kreise nach der oben erklärten Marmorbrettheuristik. Die Variable *bottom_left* wird auf *select_bottom_left_point(mode, bounidng_box, step_size)* gesetzt. Dann solange *bottom_left* nicht None ist wird: *bottom_left* als platzierten Kreis hinzugefügt. Die Liste *candidates* wird auf *generate_candidates(bottom_left, amount)* gesetzt. Solange *candidates* nicht leer ist wird: *candidates* gefiltert, indem alle entfernt werden die, die *point_ok* Methode nicht bestehen. Ist *candidates* danach nicht leer wird die Variable *circle* auf *select_candidate(candidates)* gesetzt. *circle* wird als Kreis hinzugefügt und *candidates* wird um die Elemente der Liste *generate_candidates(circle, amount)* erweitert. Ist *candidates* leer wird *bottom_left* wieder neu berechnet.

```
def marble_slab(self, mode, divide, amount, step_size, height):
```

Wie bei ilp eine Wrapperfunktion um *marble_slab_solve*, damit wenn *divide=True* das Polygon in Rechtecken mit Höhe *height* und Breite gleich der Breite des Polygons einzeln ausgefüllt wird.

3 Laufzeitanalyse

Sei *n* die Anzahl an Polygonecken und *r* der Radius der Kreise mit denen wir das Polygon befüllen und die Boundingbox des Polygons hat Breite *b* und Höhe *h*. Der Einfachheit halber nehmen wir an, dass unser gegebenes Polygon ein Quadrat mit Seitenlänge *a* ist.

Solver.contains_point ist wahrscheinlich³ $\mathcal{O}(n)$

BoundingBox.contains_point ist $\mathcal{O}(1)$

min_distance(point, point_list) ist $\mathcal{O}(\text{len}(point_list))$. Für jeden Punkt ausfüllen der *point_list* wird die Distanz berechnet.

point_to_index ist $\mathcal{O}(1)$

index_to_neighboors ist $\mathcal{O}(1)$

min_distance_spatial(point, index, points_spatial) ist $\mathcal{O}(1)$. Zwar ruft die Funktion min_distance auf, jedoch kann man sagen, dass die Länge der übergebenen Punkteliste konstant ist, da es eine obere Grenze für die Anzahl an Kreisen innerhalb eines Quadrates mit Seitenlänge 86 gibt.

mode_to_parameter ist $\mathcal{O}(1)$

point_ok ist $\mathcal{O}(n)$ wegen Solver.contains_point

gz_score ist $\mathcal{O}(1)$, weil auch hier die Anzahl an betrachteten Kreisen konstant bzw. nach oben beschränkt ist.

select_gz(step_size) ist daher $\mathcal{O}(\frac{b \cdot h}{step_size^2} \cdot 1)$. Ist die Anzahl an betrachteten Punkten für die jeweils gz_score aufgerufen wird.

³Ja, ich habe es nicht herausgefunden

3.1 reguläre Parkettierungen

Für jeden betrachteten Punkt wird `point_ok` aufgerufen und die Anzahl an betrachtet Punkten ist $b \cdot h$ square ist $\mathcal{O}(b \cdot h \cdot n)$

hexagonal ist $\mathcal{O}(b \cdot h \cdot n)$

3.2 0-1 ILP

Sei l die Länge der längsten Kante.

`create_edge_points` ist $\mathcal{O}(n \cdot \frac{l}{\Delta})$. Es gibt n Kanten und für jede werden $\frac{\text{Kantelänge}}{\Delta}$ viele Kantenpunkte erstellt.

`create_grid` ist $\mathcal{O}((\frac{b \cdot h}{\Delta^2} + n \cdot \frac{l}{\Delta}) \cdot n + (\frac{b \cdot h}{\Delta^2} \cdot (\frac{l}{\Delta})^2))$

Der linke Summand beschreibt die gesamte Anzahl an Gitterpunkten, für die alle `point_ok` aufgerufen werden muss. Der rechte Summand beschreibt, dass für jeden Kantenpunkt geschaut werden muss, ob er den Mindestabstand zu den Gitterpunkten einhält.

Wenn die Kantenpunkte hinzugefügt werden und damit die Mindestabstände zu bereits bestehenden Gitterpunkten eingehalten werden, wird die Methode `min_distance` benutzt, ohne einer räumlichen Datenstruktur. Das könnte man optimieren ist jedoch eigentlich egal, da die Anzahl an Gitterpunkten praktisch nach oben beschränkt ist, weil die ILP-Lösung für große Flächen ohne Einteilung nicht verwendet wird.

Sei Δ der Gitterabstand. Die Anzahl an Variablen beträgt ungefähr $\frac{a^2}{\Delta^2}$ ohne Kantenpunkte und mit $\frac{a^2}{\Delta^2} + n \cdot \frac{a}{\Delta}$. Der Gitterabstand geht quadratisch ein. Halbieren wir ihn vervierfacht sich $\frac{a^2}{\Delta^2}$. Die Anzahl an Ungleichungen ist gleich der Anzahl an Variablen. Bei ILPs handelt es sich um NP-schwere Probleme. Die Laufzeit ist im schlechtesten Fall exponentiell. Eine zu geringer Gitterabstand bzw. zu viele Variablen und der Algorithmus braucht Ewigkeiten. Interessant ist auch wie viele Variablen in einer Ungleichung maximal auftauchen. Das den Kreis mit Radius r umgebende Quadrat hat eine Seitenlänge von $2r$. Die Anzahl an Variablen die innerhalb dem Quadrat wären sind $\frac{4r^2}{\Delta^2}$. Fläche vom Kreis geteilt durch die des Quadrates ist $\frac{\pi r^2}{4r^2} = \frac{\pi}{4}$. Innerhalb des Kreises bzw. einer Ungleichung sind deshalb ungefähr maximal $\frac{\pi}{4} \cdot \frac{4r^2}{\Delta^2} = \frac{\pi r^2}{\Delta^2}$ Variablen. (Notiz: möglicherweise ein komischer Umweg)

Sei b' die Breite und h' die Höhe der Rechtecke, die nach der Einteilung entstehen sollen.

`create_smaller_bounding_boxes` ist $\mathcal{O}(\frac{b \cdot h}{b' \cdot h'})$

Benutzt man die zusätzliche Heuristik, dass man das Polygon in mehrere gleich große Rechtecke teilt, könnte man vielleicht sagen, dass die Laufzeit zum Lösen der einzelnen ILPs konstant ist und die Laufzeit daher polynomial ist.

3.3 Marmorplatte

`generate_candidates` ist $\mathcal{O}(1)$, da die Anzahl an generierten Punkten konstant ist.

`select_candidate(candidates)` ist $\mathcal{O}(\text{len}(\text{candidates}))$, weil wir alle Kandidaten durchgehen.

`select_bottom_left(mode, step_size)` ist $\mathcal{O}(\frac{b \cdot h}{step_size^2})$, quasi wie bei `select_gz` nur das wir hier im nicht worst-case nicht das gesamte Polygon durchgehen, sondern beim ersten gefundenen Punkt abbrechen.

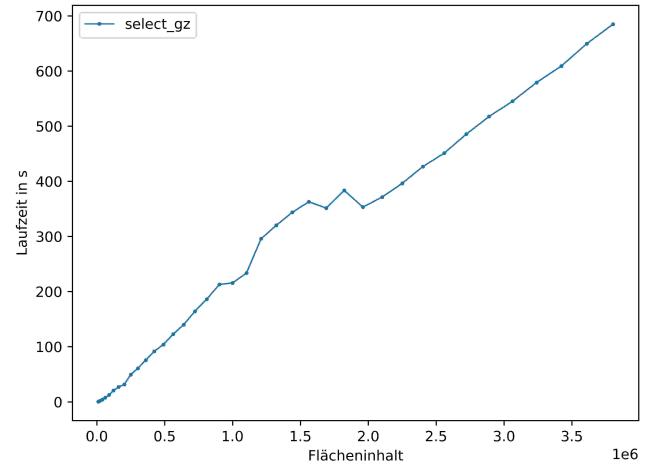
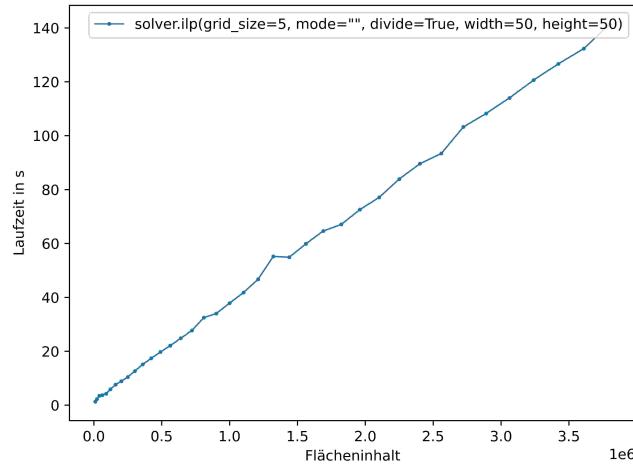
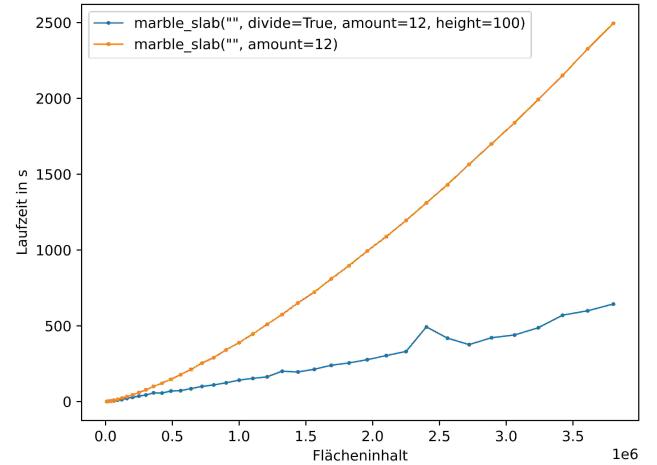
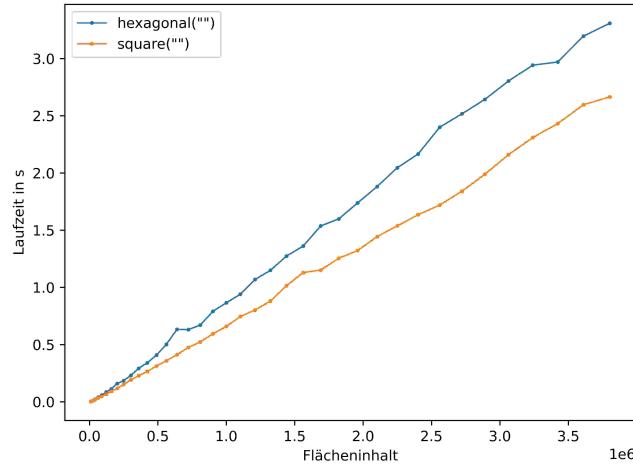
`marble_slab_solve(mode, step_size)` ist wahrscheinlich polynomial. Die Anzahl an Wiederholungen der

ersten while-Schleife ist ungefähr der Anzahl an nicht verbundenen Flächen (z.B. durch vorher platzierte Kreise getrennt oder so). In der zweiten while-Schleife wird an sich so gut wie jedes Mal ein Kreis platziert. Ich sag jetzt Mal, dass die Größe der Liste *candidates* durch die Höhe h beschrieben werden kann. Da wir immer soweit wie möglich links platzieren, müssen die alten Kandidaten bereits gelöscht sein, wenn wir uns weiter nach rechts begeben. `marble_slab` ist also $\mathcal{O}(\text{Anzahl getrennte Flächen} \cdot (\frac{b \cdot h}{\text{step_size}^2} + b \cdot h \cdot h \cdot n))$. $b \cdot h$ soll die Anzahl an Wiederholungen der zweiten while-Schleife sein und $h \cdot n$ die zugehörige Laufzeit eines Schleifendurchgangs.

Wird bei `marble_slab` das Polygon in einzelne Rechtecke geteilt, mit fester Höhe und Breite gleich der breitesten Stelle des Polygon, ist die Laufzeit: $\mathcal{O}(h \cdot \text{Anzahl getrennte Flächen} \cdot (\frac{b}{\text{step_size}^2} + b \cdot n))$.

3.4 praktische Laufzeitanalyse

Als Ergänzung habe ich als Eingabe noch Quadrate mit Seitenlänge 100 bis 1950 im Abstand von 50 generieren lassen. Die Laufzeit habe ich in Bezug zum Flächeninhalt dargestellt. Die Laufzeit wurde für den ersten Schritt das gesamte Polygon mit kleinen Kreisen zu befüllen gemessen. Nebenbei habe ich teilweise Garrys Mod gespielt. Die Ergebnisse könnten also leicht beeinträchtigt sein.



4 Beispiele

Das Programm wird mit den Parametern

<eingabedatei> <ausgabedatei> <Modus zum ausfuellen mit kleinen Kreisen> <Modus zum ausfuellen des Gesundheitszentrums> <Modus fuer das restliche Ausfuellen>

Die Modi sind

- s: Quadratisch ausfüllen
- h: Hexagonal ausfüllen
- m: mit Marmorbrettheuristik ausfüllen
- md: mit Marmorbrettheuristik Teilstreifen ausfüllen
- i: mit ILP ausfüllen
- d mit ILP Teilrechtecke ausfüllen

mit dem optionalen Parameter --keep werden die Kreise aus dem ersten Ausfüllen direkt weiterverwendet, indem nur die Kreise außerhalb des Gesundheitszentrums entfernt werden. Das wirkt sich bei Nutzung der Marmorbrettheuristik oft positiv aus, da sie nicht so gut beim alleinigen Ausfüllen des Gesundheitszentrums ist.

Eingabedatei	beste Lösung	Modi
siedler1.txt	171	h h m
siedler2.txt	109	i i i Δ2.5
siedler3.txt	260	h h m
siedler4.txt	193	i i i Δ5
siedler5.txt	201	h h m

Die Ausgabedateien sind unter ausgabe/ und dann im Ordner des verwendeten Modus als Bild und Textdatei zu finden. Beispiele bei denen Verfahren gemischt wurden, sind unter output/mixed zu finden. In der Textdatei steht zuerst das GZ, dann kommen Punkte innerhalb des GZ und nach einer Leerzeile Punkte außerhalb des GZ.

4.1 Siedler1.txt

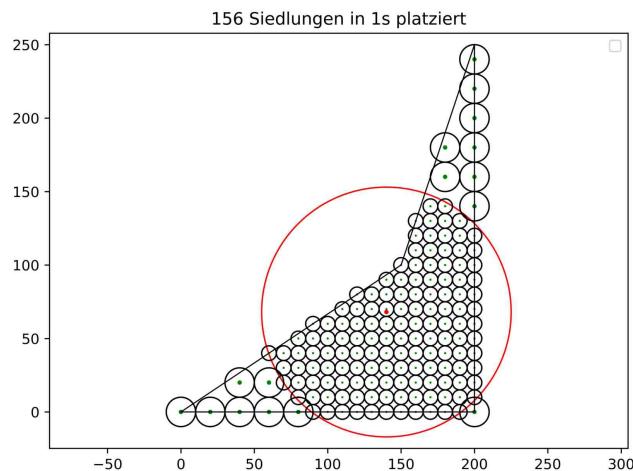


Abbildung 6: Quadratmuster mit s s s

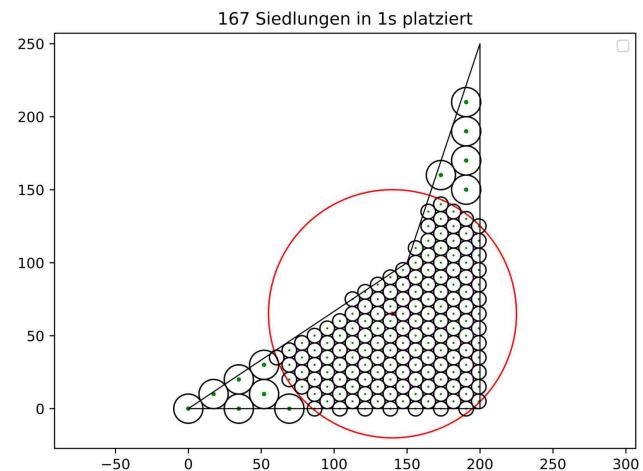


Abbildung 7: Seckseckmuster mit h h h

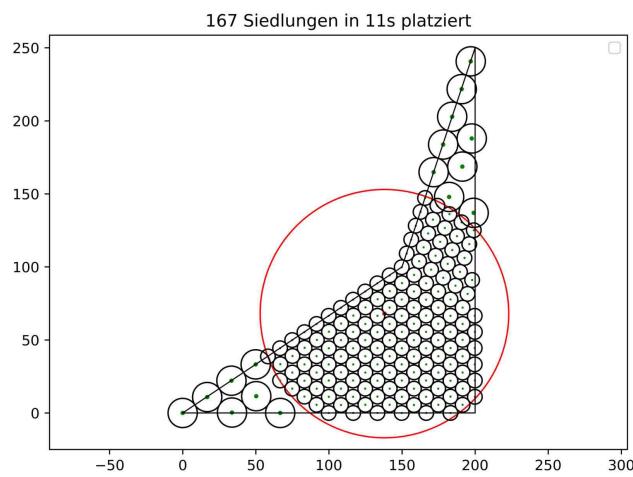


Abbildung 8: Marmorbrett mit m m --keep

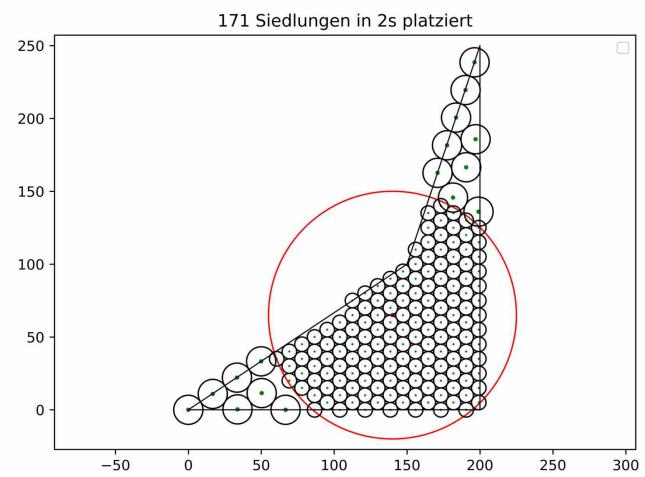
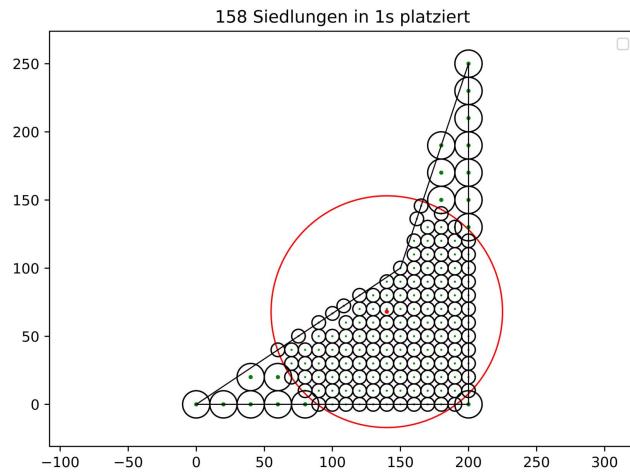
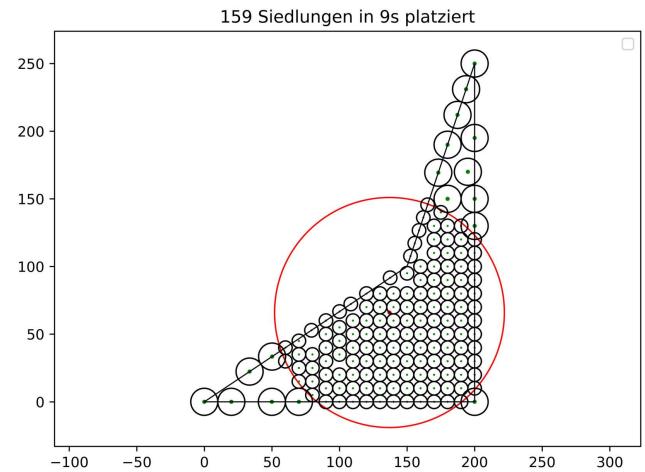


Abbildung 9: Gemischt mit h h m

Abbildung 10: ILP Δ 10 mit i i iAbbildung 11: ILP Δ 5 mit i i i

4.2 Siedler2.txt

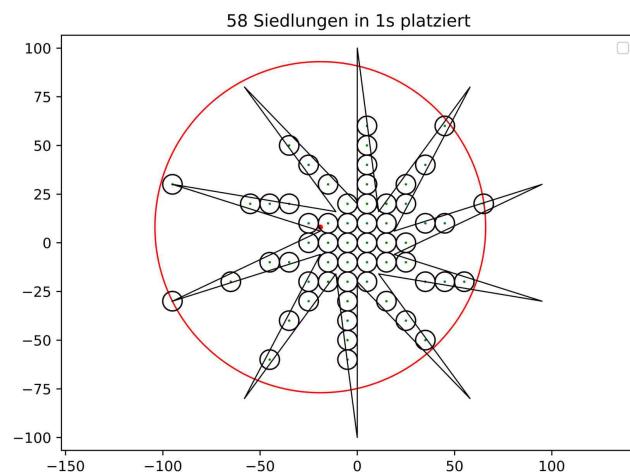


Abbildung 12: Quadratmuster mit s s s

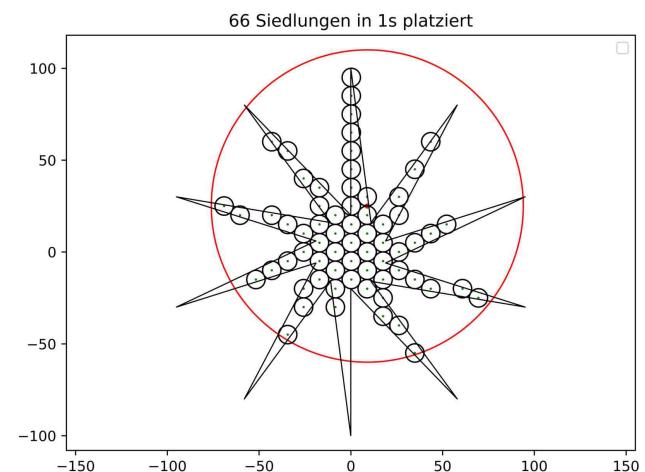


Abbildung 13: Seckseckmuster mit h h h

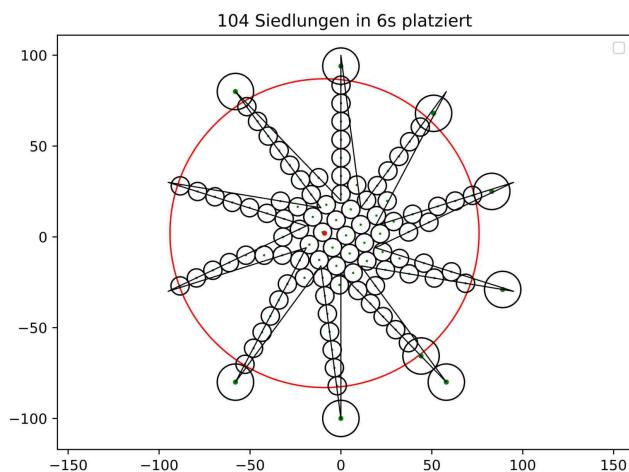
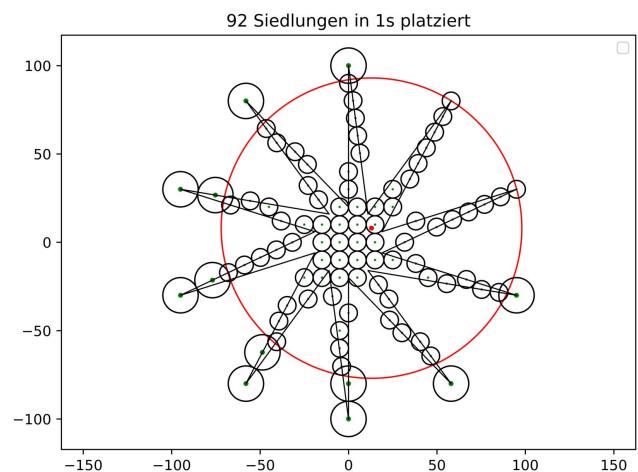
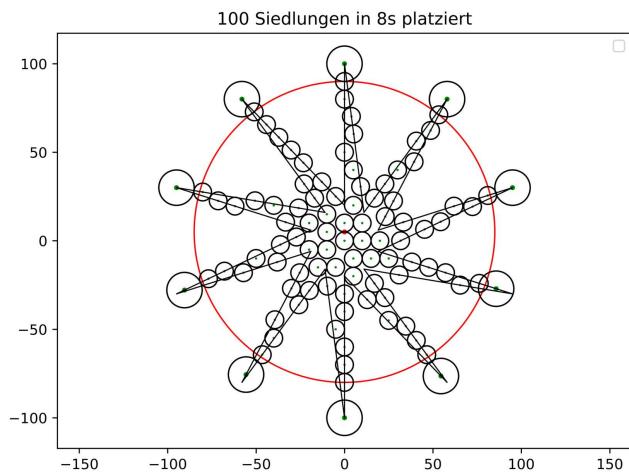
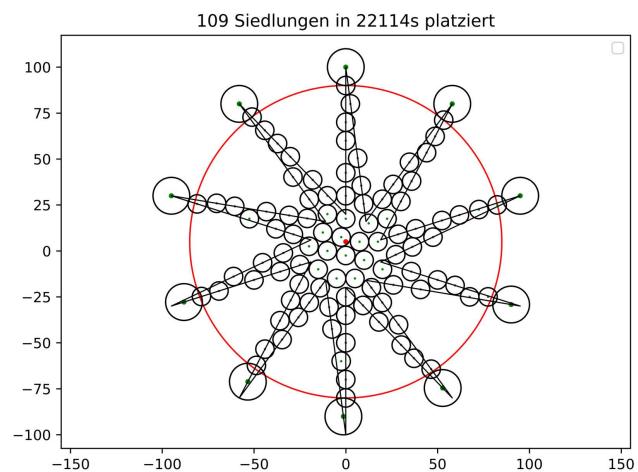


Abbildung 14: Marmorbrett mit m m m

Abbildung 15: ILP Δ 10 mit i i iAbbildung 16: ILP Δ 5 mit i i iAbbildung 17: ILP Δ 2.5 mit i i i

4.3 Siedler3.txt

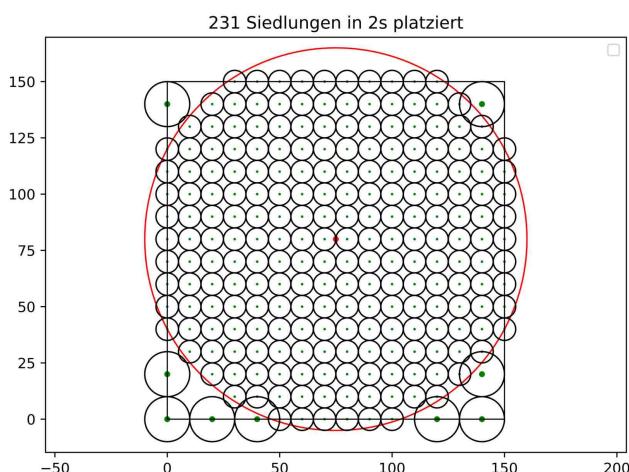


Abbildung 18: Quadratmuster s s s

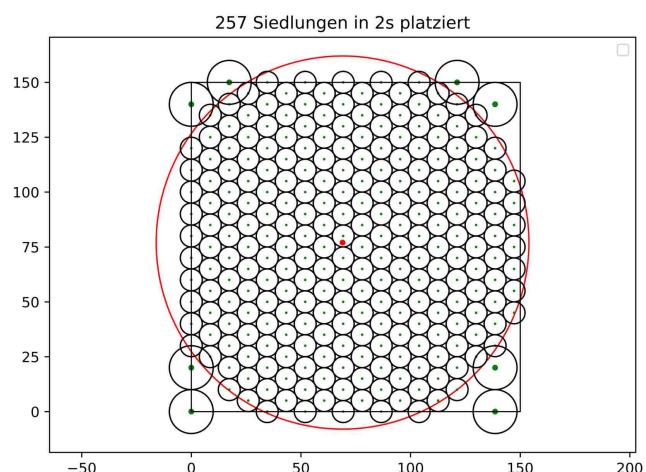


Abbildung 19: Seckseckmuster h h h

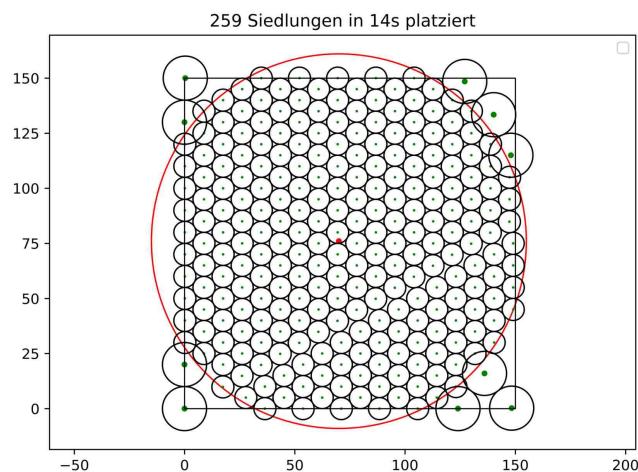


Abbildung 20: Marmorbrett m m --keep

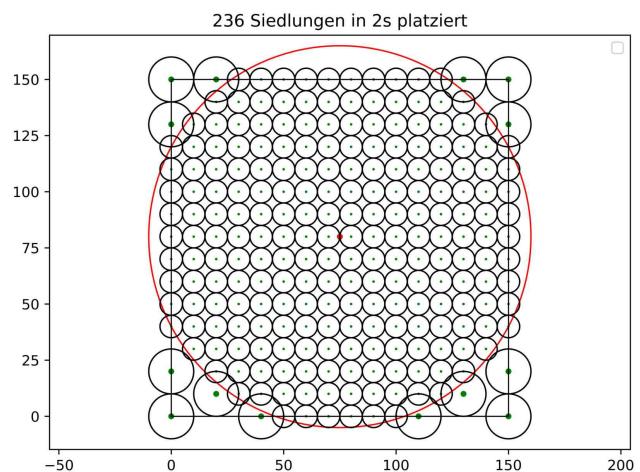


Abbildung 21: ILP Δ 10 i i i

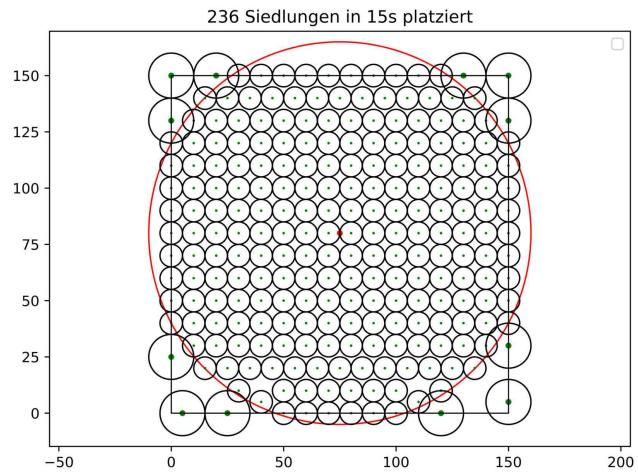


Abbildung 22: ILP Δ 5 i i i

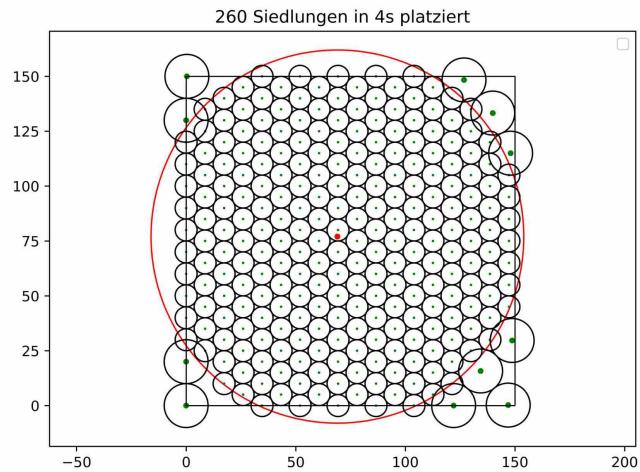


Abbildung 23: Gemischt h h m

4.4 Siedler4.txt

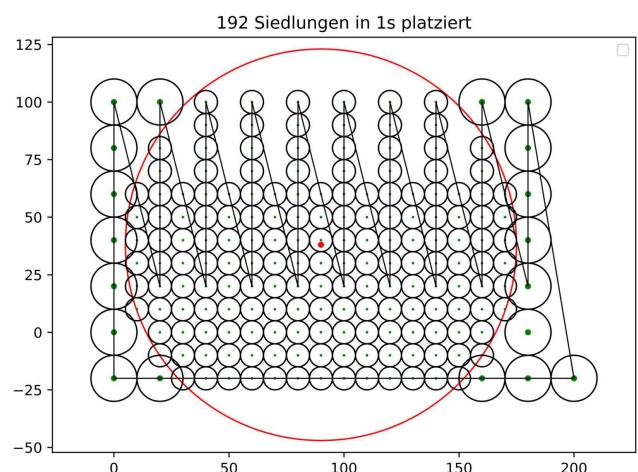


Abbildung 24: Quadratmuster mit s s s

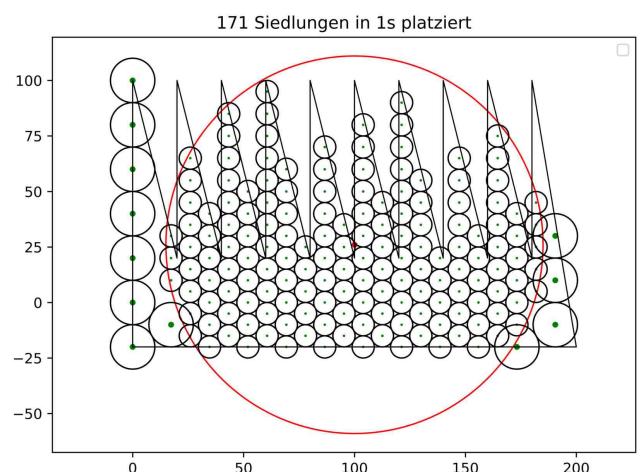


Abbildung 25: Seckseckmuster mit h h h

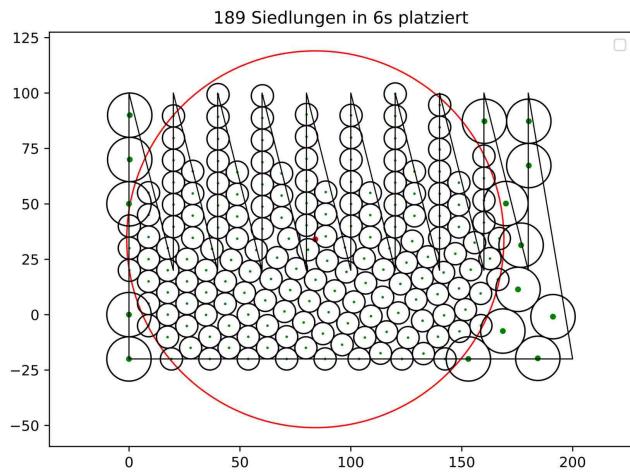
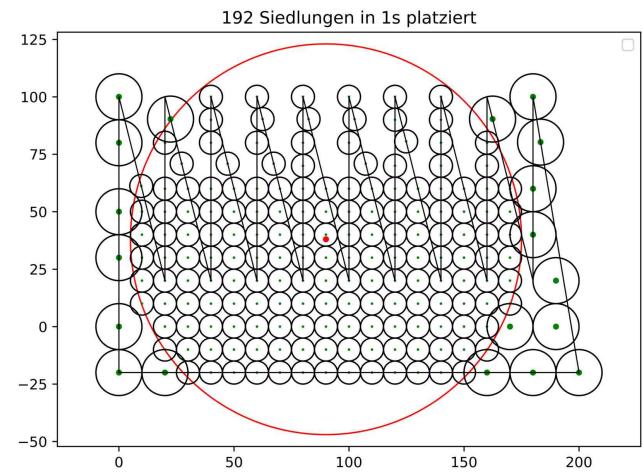
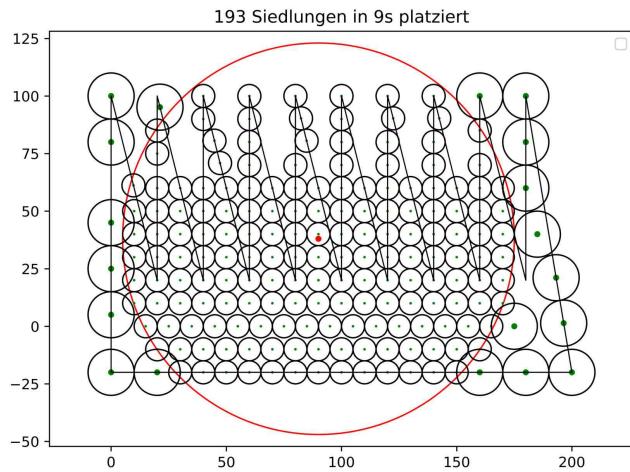


Abbildung 26: Marmorbrett mit m m --keep

Abbildung 27: ILP Δ 10 mit i i iAbbildung 28: ILP Δ 5 mit i i i

4.5 Siedler5.txt

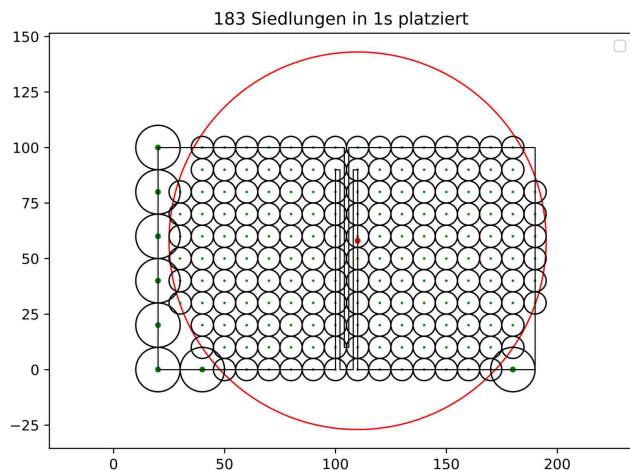


Abbildung 29: Quadratmuster mit s s s

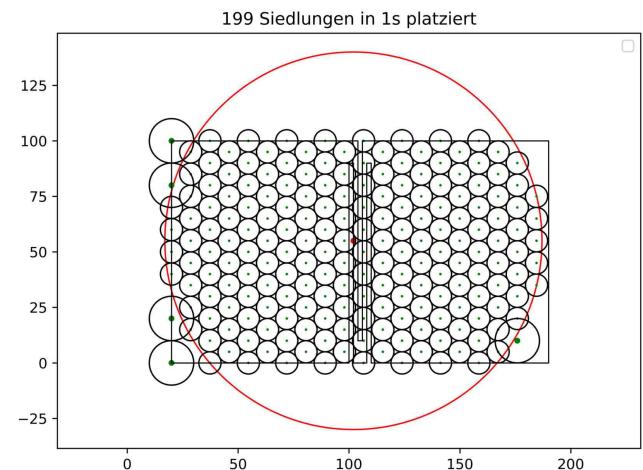


Abbildung 30: Seckseckmuster mit h h h

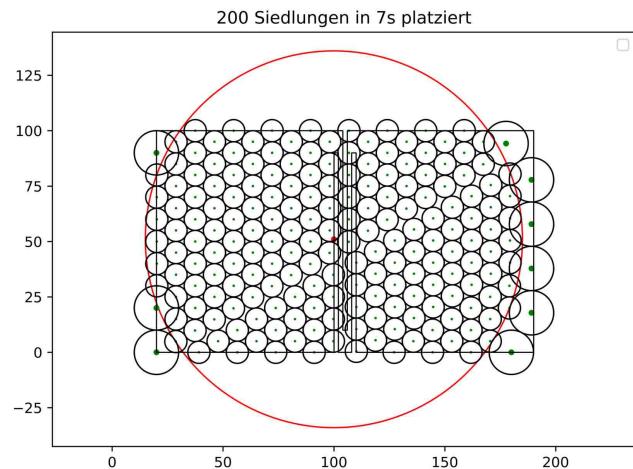


Abbildung 31: Marmorbrett mit m m --keep

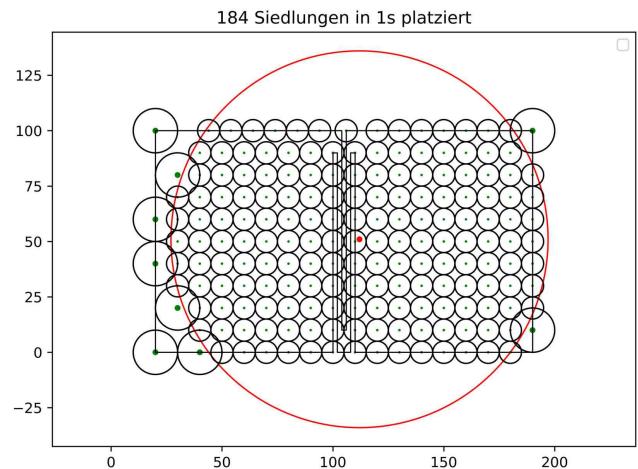


Abbildung 32: ILP Δ 10 mit i i i

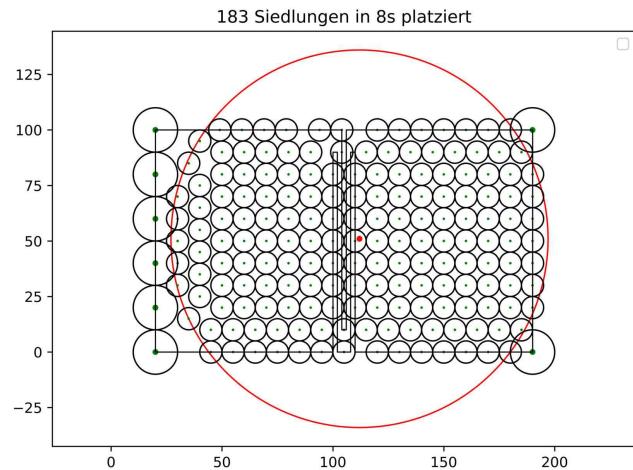


Abbildung 33: ILP Δ 5 mit i i i

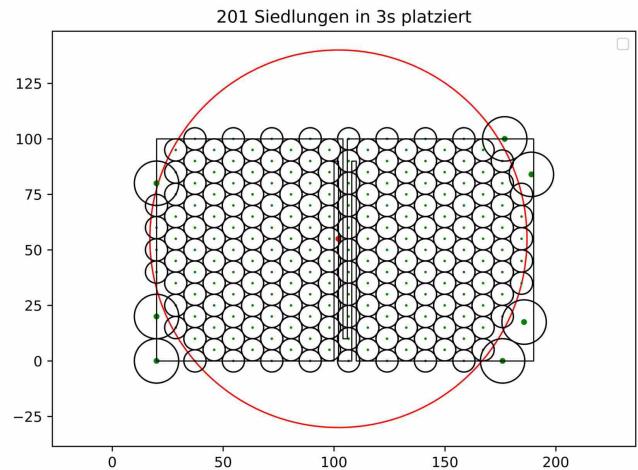


Abbildung 34: Gemischt mit h h m

4.6 Labyrinth

Die zusätzlichen Beispiele hat ChatGPT für mich generiert. Die Beispiele sind deshalb ein bisschen komisch und irritierend. Bei dem Labyrinth wüsste ich jetzt z.B. nicht überall was innen und was außen ist.

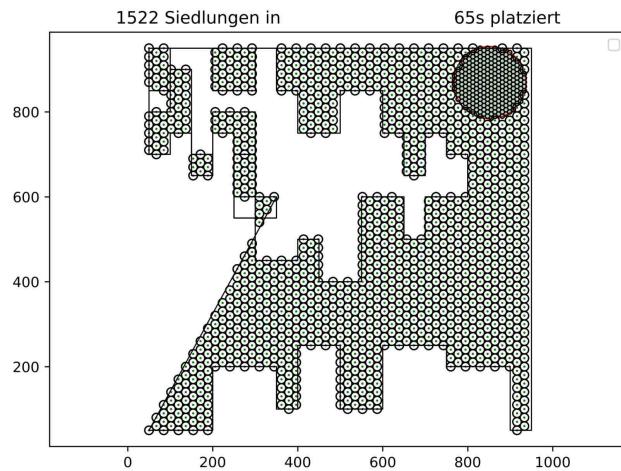


Abbildung 35: Seckseckmuster mit h h h

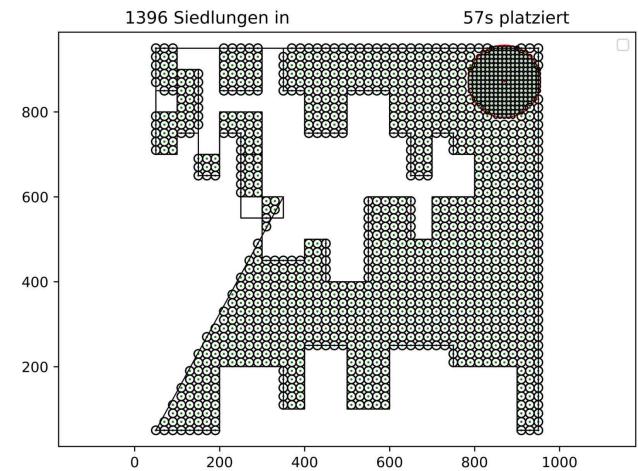


Abbildung 36: Quadratmuster mit s s s

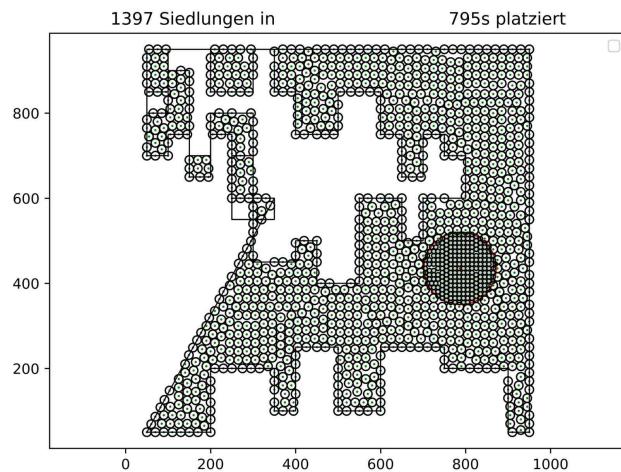
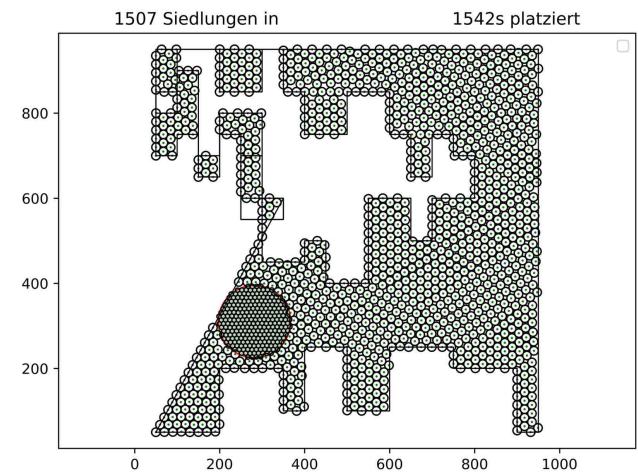
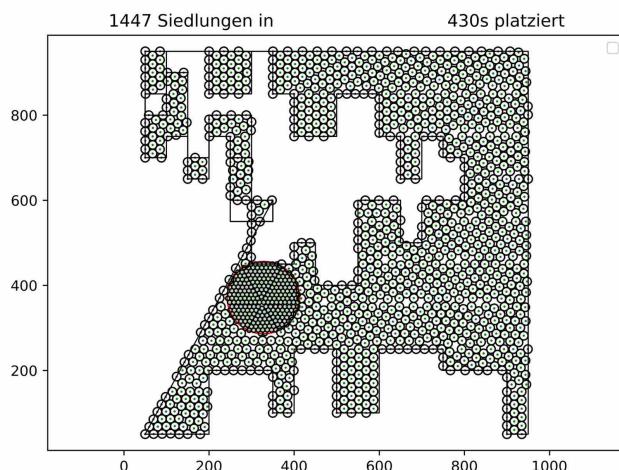
Abbildung 37: ILP Δ 5 geteilt in 100x100 mit i i

Abbildung 38: Marmorbretttheuristik mit m m --keep

Abbildung 39: Marmorbretttheuristik geteilt
höhe 100 mit md md md

4.7 Mona Lisa

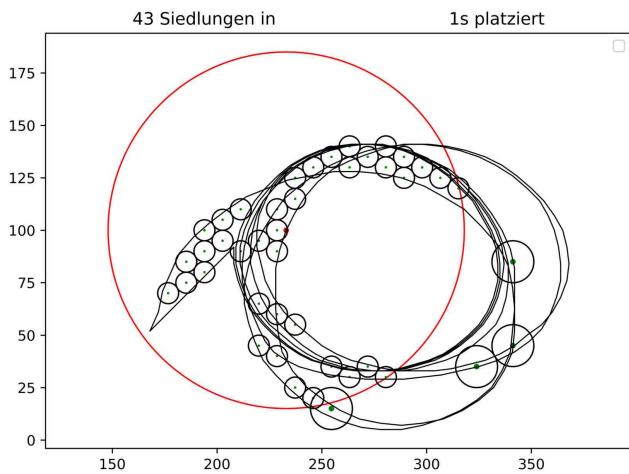


Abbildung 40: Seckseckmuster mit h h h

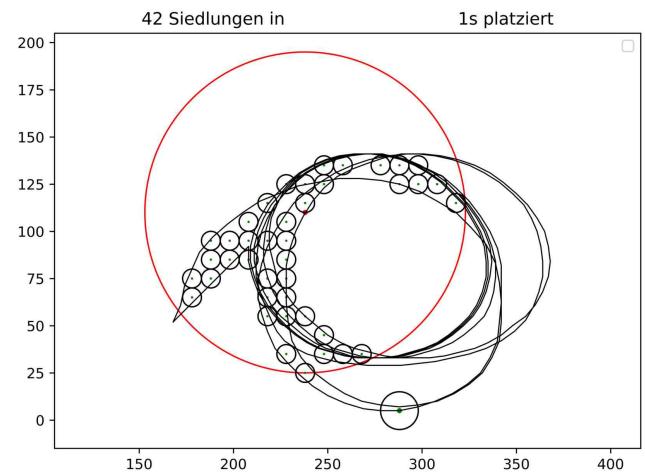


Abbildung 41: Quadratmuster mit s s s

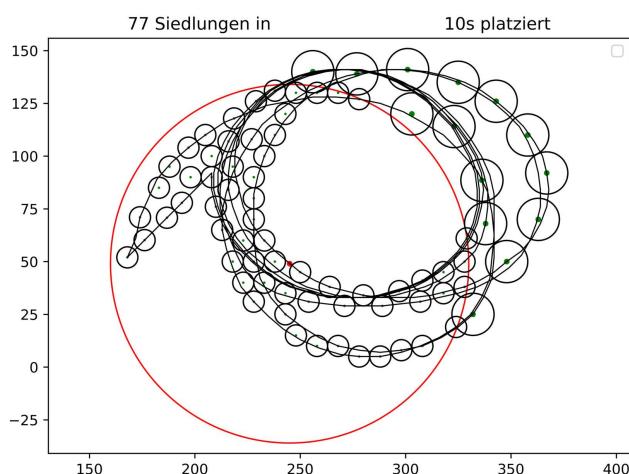
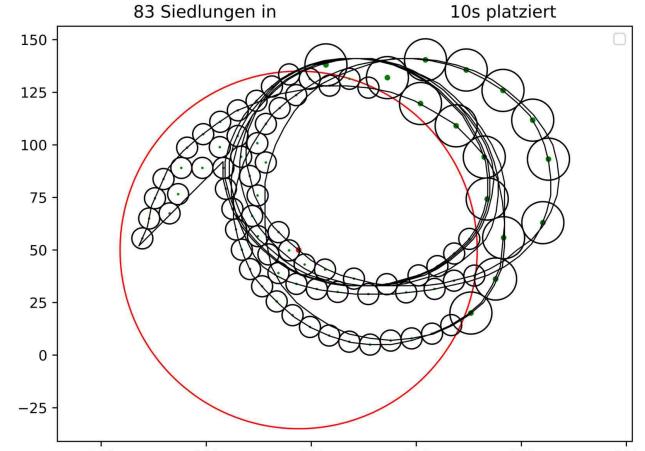
Abbildung 42: ILP Δ 5 mit i i i

Abbildung 43: Marmorbrettheuristik mit m m m

4.8 Kreis

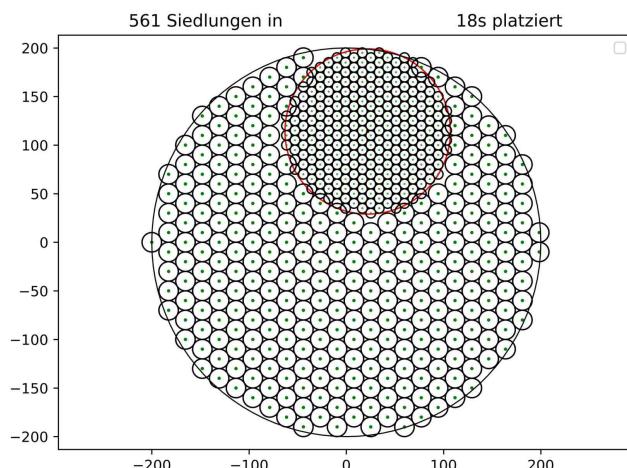


Abbildung 44: Seckseckmuster mit h h h

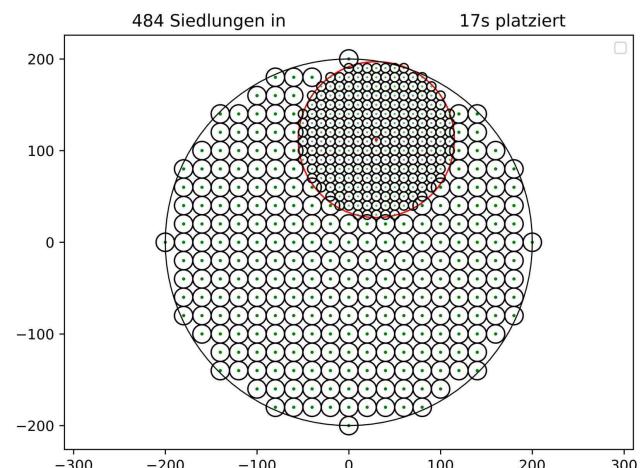


Abbildung 45: Quadratmuster mit s s s

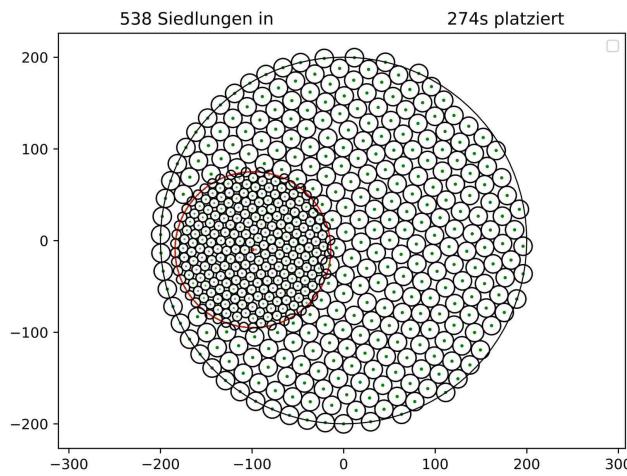
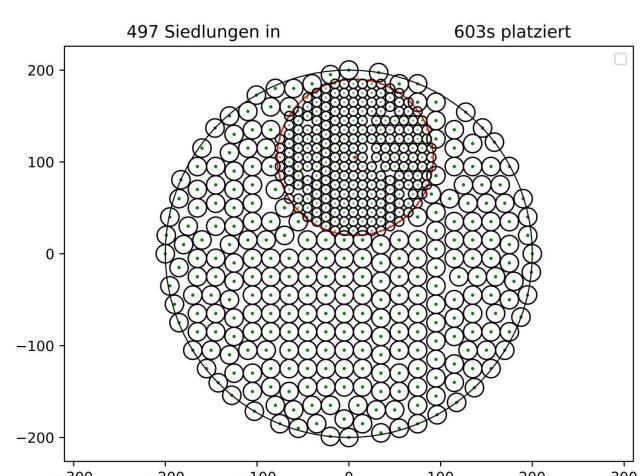


Abbildung 46: Marmorbretttheuristik mit m m m

Abbildung 47: ILP Δ 100x100 mit d d d

4.9 kleiner Kreis

Das besondere an dem Beispiel ist, dass der Kreis den Radius 80 hat und das somit dasselbe ist, wie einen Kreis mit Radius 85 mit Kreisen mit Radius 5 zu befüllen, wobei die Kreise nicht mit den Grenzen überschneiden dürfen.

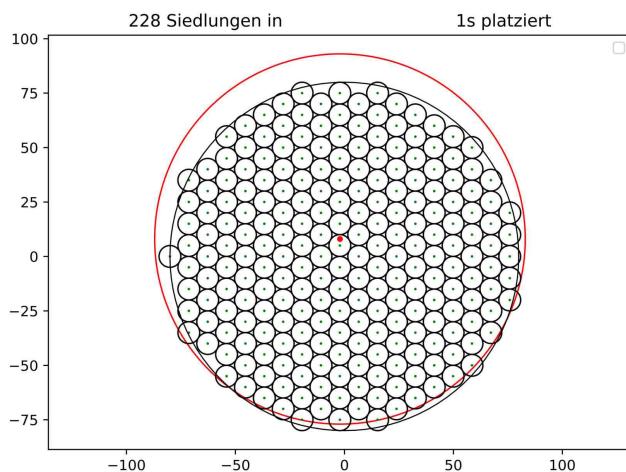


Abbildung 48: Seckseckmuster mit h h h

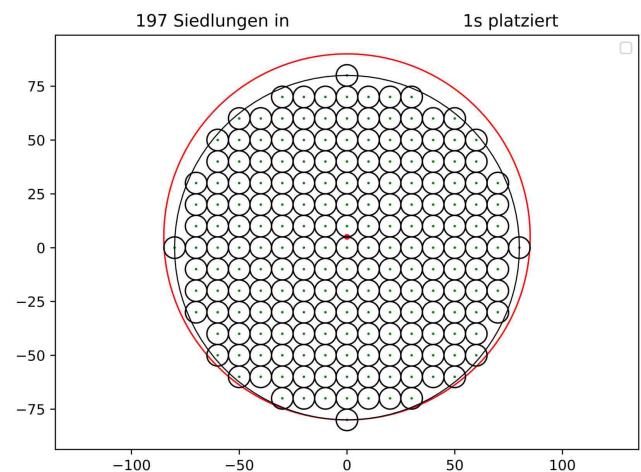


Abbildung 49: Quadratmuster mit s s s

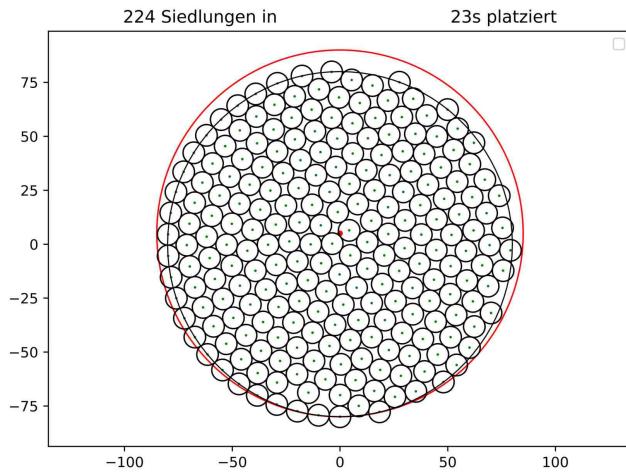
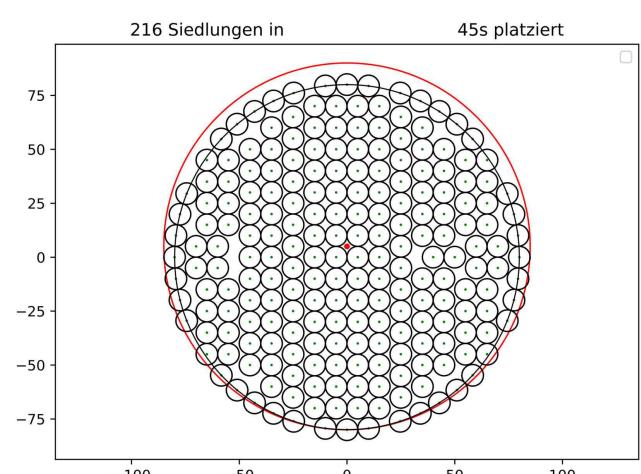


Abbildung 50: Marmorbretttheuristik mit m m m

Abbildung 51: ILP Δ 5 mit i i i

5 Quellcode

Kürzungen sind mit ... gekennzeichnet.

5.1 solver.py

```
...
class Solver():

    ...
    def __init__(self, polygon) -> None:
        ...
        self.big_circles_spatial = [[[x for x in range(self.horizontal_amount)] for y in
            range(self.vertical_amount)]
        self.small_circles_spatial = [[[x for x in range(self.horizontal_amount)] for y in
            range(self.vertical_amount)]
```

```

...
def marble_slap(self, mode, step_size = 1) -> None:
    circles_spatial, radius = self.mode_to_parameter(mode)

    # select starting bottom-left position inside polygon
    bottom_left = self.select_bottom_left_point(mode, circles_spatial, step_size)
    while bottom_left is not None:

        candidates = self.generate_candidates(bottom_left, radius, 360)
        while len(candidates) > 0:

            candidates = [x for x in candidates
                          if self.point_ok(x, self.point_to_index(x), self.bounding_box, mode)]

            if len(candidates) == 0:
                break

            circle = self.select_candidate(candidates)
            index = self.point_to_index(circle)
            circles_spatial[index[0]][index[1]].append(circle)
            candidates = candidates + self.generate_candidates(circle, radius, 360)

        bottom_left = self.select_bottom_left_point(mode, circles_spatial, step_size)

def hexagonal(self, mode) -> None:
    circles_spatial, radius = self.mode_to_parameter(mode)

    width = radius / (math.sqrt(3) / 2)
    height = radius

    alternate = 0
    x = self.bounding_box.left
    y = self.bounding_box.bottom

    while x <= self.bounding_box.right:
        y = self.bounding_box.bottom + alternate * height
        while y <= self.bounding_box.top:
            index = self.point_to_index([x, y])
            if self.point_ok([x, y], index, self.bounding_box, mode=mode):
                circles_spatial[index[0]][index[1]].append([x, y])
            y += 2 * height
        x += 1.5 * width
        alternate = (alternate + 1) % 2

    ...
    ...

def create_grid(self, grid_size, bounding_box, mode) -> list[list[float]]:
    grid = []
    # add grid points
    y = bounding_box.bottom
    while y < bounding_box.top:
        x = bounding_box.left
        while x < bounding_box.right:
            index = self.point_to_index([x, y])
            if self.point_ok([x, y], index, bounding_box, mode=mode):
                grid.append([x, y])

            x += grid_size
        y += grid_size
    ...

    return grid

def ilp_solve(self, grid_size, bounding_box, divide, draw=False, mode="") -> None:
    grid = self.create_grid(grid_size, bounding_box, mode=mode)

    circles, radius = self.mode_to_parameter(mode)

    model = LpProblem(name="circle-packing", sense=LpMaximize)

```

```

# binäre Variablen erstellen
variables = [None] * len(grid)
for i in range(len(grid)):
    variables[i] = LpVariable(str(i), cat="Binary")

# Ungleichungen hinzufügen
for i in range(len(grid)):
    sum_close_variables = 0
    amount_close_variables = 0
    for j in range(len(grid)):
        if i == j:
            continue

        if math.dist(grid[i], grid[j]) + self.precision_error < 2 * radius:
            sum_close_variables += variables[j]
            amount_close_variables += 1

    if amount_close_variables == 0:
        continue

    model += variables[i] * amount_close_variables + sum_close_variables <=
amount_close_variables

# wird benötigt, wenn die Rechtecke einzeln gefüllt werden.
height = bounding_box.top - bounding_box.bottom
width = bounding_box.right - bounding_box.left

cost_function = 0
for i in range(len(variables)):
    if divide:
        scale_y = 2 * (height - (grid[i][1] - bounding_box.bottom))
        scale_x = width - (grid[i][0] - bounding_box.left)
        cost_function += (1 + (scale_y + scale_x) * self.precision_error) * variables[i]
    else:
        cost_function += variables[i]

model += cost_function

...

# returns the index (x, y) of the square the point belongs to
def point_to_index(self, point) -> list[int]:
    # Punkt muss relativ zur boundingbox vorliegen,
    # wegen negativen Koordinaten
    point_relative = [self.bounding_box.top - point[1],
                      self.bounding_box.right - point[0]]

    # floor, weil wir bei 0 anfangen
    indices = [int(point_relative[1] / self.spatial_length),
               int(point_relative[0] / self.spatial_length)]

    return indices

# returns the index of the own and neighboring squares of the indexed square
def index_to_neighbors(self, index) -> list[list[int]]:
    indices = [index]

    if index[0] > 0:
        # unten mitte
        indices.append([index[0] - 1, index[1]])
        if index[1] > 0:
            # unten links
            indices.append([index[0] - 1, index[1] - 1])
        if index[1] < self.horizontal_amount - 1:
            # unten rechts
            indices.append([index[0] - 1, index[1] + 1])

    ...

    return indices

```

5.2 main.py (nicht ganz)

```
# Beispiel, wie man die Funktionen benutzt
polygon = load_polygon(sys.argv[1])
solver = Solver(polygon)
# Polygon mit kleinen Kreisen ausfüllen
solver.hexagonal("")
# Gesundheitszentrum auswählen
solver.select_gz(1)
# Alle Kreise löschen
solver.small_circles_spatial = [[[] for x in range(solver.horizontal_amount)]
                                  for y in range(solver.vertical_amount)]
# Gesundheitszentrum mit kleinen Kreisen ausfüllen
solver.hexagonal("inside")
# Rest mit großen Kreisen ausfüllen
solver.hexagonal("outside")
```
