Aufgabe 3: Die Siedler

Teilnahme-ID: 69408

Bearbeiter dieser Aufgabe:
Tim Krome

12. April 2024

Lösungsidee

Aufteilen der Aufgabe in zwei Teilprobleme

Die Aufgabe verlangt, dass in einem Polygon möglichst viele Punkte platziert werden, die zueinander einen festen Mindestabstand (im Folgenden als min_distance bezeichnet) haben sollen. Gleichzeitig soll ein Gesundheitszentrum so platziert werden, dass alle Punkte, deren Abstand zum Gesundheitszentrum kleiner als ein fester Wert ist (im Folgenden als secure_distance_ghz bezeichnet), nur einen kleineren Mindestabstand zueinander haben müssen (im Folgenden als

min_distance_ghz bezeichnet). Die Aufgbae lässt sich in zwei miteinander verknüpfte Optimierungsprobleme aufteilen:

1. Teil-Optimierungsproblem:
   Einen Bereich, der durch Polygonkanten und Kreisrandsegmente begrenzt ist, optimal befüllen.
2. Teil-Optimierungsproblem:
   Die optimale Position für das Gesundheitszentrum finden.

Die Teilprobleme dürfen dabei nicht vollständig separat betrachtet werden, da die Position des Gesundheitszentrums die optimale Positionierung von möglichst vielen Ortschaften beeinflusst.

Geometrische Grundlagen

Es handelt sich um eine sehr geometrische Aufgabe. Folglich müssen zum Lösen der Aufgabe einige geometrische Verfahren definiert werden. Dabei wird versucht, so viele geometrische Verfahren wie möglich selbst zu implementieren – dies hat gegenüber der Verwendung einer Bibliothek den Vorteil, dass der Algorithmus genau auf die vorhandenen Bedürfnisse abgestimmt werden kann.
Zur Darstellung von zweidimensionalen Positionen und Vektoren werden im Folgenden Tupel und Vektoren eingesetzt. distance(X, Y) bezeichnet den euklidischen Abstand zwischen zwei Punkten. Es werden im Folgenden nur die Algorithmen erläutert, die besonders interessant sind, z.B. aus algorithmischer Sicht oder weil sie sehr speziell sind. Standardverfahren aus der Schulmathematik gehören zum Grundwissen und werden daher hier nicht erläutert.

Punkt-in-Polygon Test

Unabdingbar ist beim Lösen der Aufgabe ein Algorithmus, mit dem überprüft werden kann, ob sich ein Punkt in einem Polygon befindet. Hierfür existieren mehrere gute Verfahren. Eines dieser Verfahren funktioniert so, dass vom Punkt aus eine horizontale Linie gezeichnet wird und die Schnittpunkte dieser Linie mit den Polygonkanten gezählt werden:

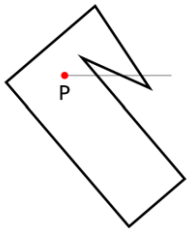


Abbildung 1: Veranschaulichung des zuvor erläuterten Punkt-in-Polygon-Tests

Das Problem an diesem Verfahren ist: Wenn diese Linie zufällig identisch mit einer Polygonkante ist, dann funktioniert es nicht. Daher wird ein alternatives Verfahren verwendet.

Dieses Verfahren trianguliert zunächst das Polygon. Zur Triangulation wird der Clipping-Ear-Algorithm verwendet. Da es sich bei diesem Algorithmus um einen bekannten Algorithmus aus der analytischen Geometrie handelt, wird er hier nicht nochmal extra erläutert. Der Clipping-Ear-Algorithm hat eine Laufzeit von $O(n^2)$, die zwar nicht optimal für eine Punkt-in-Polygon Testmethode ist. Die Triangulation kann allerdings nach einmaligem Berechnen gecached werden

Anschließend wird für jedes der aus der Triangulation hervorgehden Dreiecke überprüft, ob sich Punkt P in dem jeweiligen Dreieck befindet. Um zu überprüfen, ob sich Punkt P im Dreieck ABC befindet, werden zunächst die Dreiecke ABP, APC und PCB definiert. P liegt nun in ABC, wenn gilt: A(ABP) + A(APC) + A(PCB) = A(ABC). A(Dreieck) bezeichnet hierbei die Dreiecksfläche.

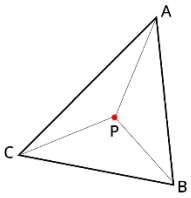


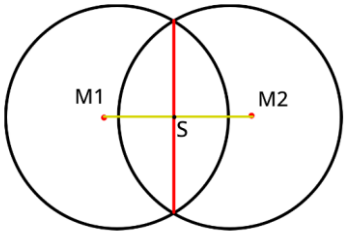
Abbildung 2: Veranschaulichung des zuvor erläuterten Zusammenhangs

Der Flächeninhalt eines Dreiecks lässt sich dabei basierend auf den Seitenlängen a,b,c effizient über den Satz von Heron berechnen:

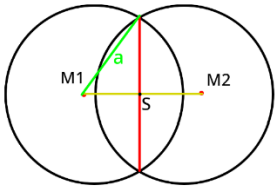
$$A = \sqrt{s * (s-a) * (s-b) * (s-c)} \text{ mit } s = \frac{a+b+c}{2}$$

Schnittpunkte Kreis-Kreis bestimmen

Verfahren zur Berechnung der beiden Schnittpunkte von zwei gleich großen Kreisen:



S = ((M1[0] + M2[0])/2, (M1[1] + M2[1])/2)
b = distance(M1, S)



a = math.sqrt(M1_radius² - b²) | geht aus Satz des Pythagoras hervor

Berechnung der Steigung einer Gerade, die orthogonal zur Strecke M1M2 ist, um die Position der Schnittpunkte zu ermitteln:

falls M1[0] gleich M2[0]:

    orthogonal = 0
ansonsten:
    steigung_M1M2 = (M2[1] – M1[1])/(M2[0]-M1[0])

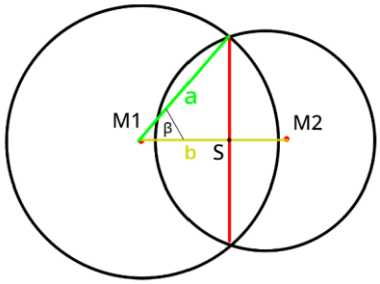
wenn M1[1] gleich M2[1]:
die gesuchten Schnittpunkte sind (S[0], S[1]+a) und (S[0], S[1]-a). Abbruch des Verfahrens
ansonsten:
orthogonal = -1/steigung

a so verrechnen, dass die Positionen der Schnittpunkte bestimmt werden können:

y_abschnitt = -1 * s[0] * orthogonal + s[1]
factor = math.sqrt(1 + orthogonal**2)
die gesuchten Schnittpunkte sind:
(s[0] + a/factor, y_abschnitt + (s[0] + a/factor) * orthogonal)
und
(s[0] + a/factor, y_abschnitt + (s[0] + a/factor) * orthogonal)

Verfahren zur Berechnung der beiden Schnittpunkte von zwei verschieden großen Kreisen:

Um die Schnittpunkte von zwei verschieden großen Kreisen zu bestimmen, müssen nur die ersten beiden Schritte abgeändert werden (S und b müssen anders berechnet werden):

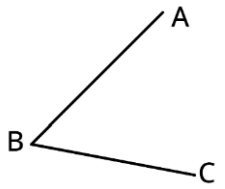


d = distance(M1, M2)
β = math.acos(
(other.radius**2 - self.radius**2 - d**2)/(-2*self.radius*d) | Bestimmung über den Kosinussatz
)

b = math.cos(β) * self.radius | Berechnung von b über den zuvor berechneten Winkel β
S = (M1[0] + (M2[0]-M1[0])*(b/d), M1[1] + (M2[1]-M1[1])*(b/d))

- ab hier kann mit Schritt 3 des Verfahrens zur bestimmung der Schnittpunkte von zwei gleich großen Kreisen fortgefahren werden -

Berechnung eines Polygon-Innenwinkels



Um den Innenwinkel CBA basierend auf den Streckenlängen zu berechnen, kann folgende Formel verwendet werden:

$$\cos\alpha = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$$

mit Vektor a := A-B und Vektor b := C - B

1. Teilproblem: Bereich optimal befüllen

Beim ersten Teilproblem geht es darum, einen Bereich so mit Punkten zu füllen, dass die Punkte alle einen festen Abstand zueinander haben. Es ist bekannt, dass die effizienteste Flächennutzung der euklidischen Ebene dann erreicht ist, wenn die Punkte in einem Muster angeordnet werden, sodass die Punkte miteinander gleichseitige Dreiecke bilden.

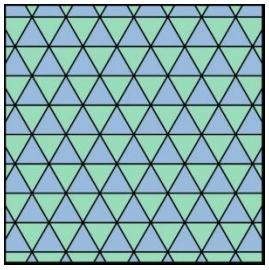


Abbildung 3: Muster bestehend aus gleichseitigen Dreiecken. Werden die Punkte immer an den Ecken der Dreiecke platziert, dann füllen sie die (unendlich große) euklidsche Ebene vollständig aus

Bei den Polygonen, die in der Aufgabe befüllt werden sollen (im Folgenden als Container bezeichnet), handelt es sich allerdings um sehr komplexe Container, bei denen es nicht einfach ausreicht, ein gleichseitiges Raster über das Polygon zu legen o.Ä., um optimale Ergebnisse zu erzielen.

Verwandtschaft mit dem Disk Covering Problem

Wenn man um die in dem Container platzierten Punkte Kreise zeichnet, die einen Radius von min_distance haben, dann sieht dies so aus:

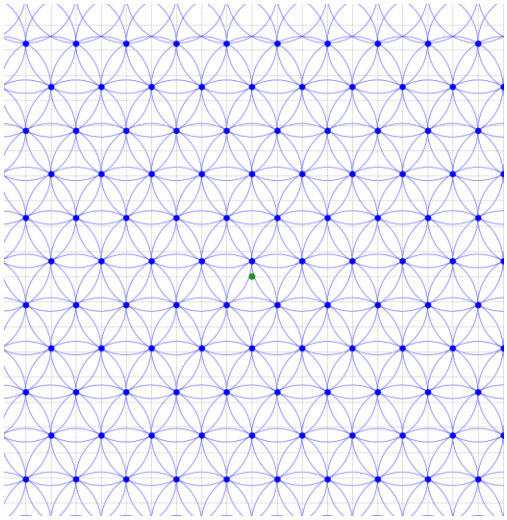


Abbildung 4: Ein befüllter Bereich eines Containers, in dem um jeden platzierten Punkt ein Kreis mit dem Radius min_distance gezeichnet ist

Eine Darstellung dieser Art erinnert an das Disk Covering Problem. Das Disk Covering Problem befasst sich damit, die minimale Anzahl von Disketten (Kreisen) zu finden, die benötigt werden, um eine gegebene Menge von Punkten in einem bestimmten Gebiet zu überdecken:

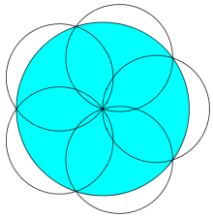


Abbildung 5: Disk Covering Problem

Die Parallelen zwischen Disk Covering Problem und dem Problem aus der Aufgabe sind jedoch nur oberflächlich, eine direkte Reduktion ist nicht möglich.

Reduktion auf das Circle Packing Problem

Wenn man um die in dem Container platzierten Punkte Kreise zeichnet, die einen Radius von min_distance/2 haben, dann sieht dies so aus:

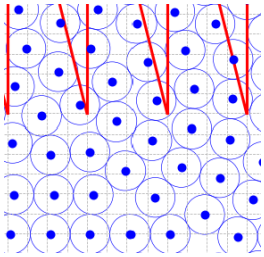


Abbildung 6: Ein befüllter Bereich eines Containers, in dem um jeden platzierten Punkt ein Kreis mit dem Radius min_distance/2 gezeichnet ist.

Diese Darstellung ähnelt optisch sehr stark dem Circle Packing Problem mit gleich großen Kreisen. Beim Circle Packing Problem mit gleich großen Kreisen geht es darum, in einem Container so viele Kreise wie möglich zu platzieren, die alle gleich groß sind. Die platzierten Kreise dürfen sich dabei nicht überlappen und sie dürfen sich auch nicht mit dem Rand überlappen:

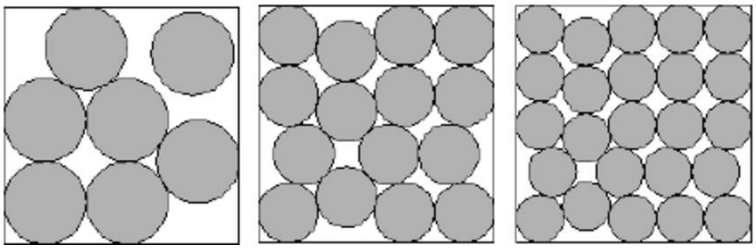


Abbildung 7: Circle packing problem mit einem quadratischen Container

Das Equal Circle Packing Problem ist dabei äquivalent zum 1. Teilproblem der BWinf-Aufgabe. Aus einer Lösung bzw. Punktverteilung, die das 1. Teilproblem optimal löst, lässt sich in polynomialer Zeit die optimale Lösung für das äquivalente Circle Packing Problem ermitteln. Um das BWinf-Problem (Platzieren von Punkten) in das Circle Packing Problem umzuformen, müssen folgende Schritte durchgeführt werden:

1. Zeichne um jeden Punkt, der für die Lösung des BWinf-Problems festgesetzt wurde, einen Kreis mit dem Radius min_distance/2. Wenn die im BWinf-Problem eingezeichneten Punkte einen Abstand von mind. min_distance zueinander hatten, dann werden sich auch die eingezeichneten Kreise nicht überlappen.
2. Bewege die Kanten des Polygons um min_distance/2 LE nach außen.

Dies bedeutet: Wenn das Circle Packing Problem with Equal Circles in polynomieller Zeit optimal gelöst werden kann, dann kann ach das 1. Teilproblem der BWinf-Aufgabe in polynomieller Zeit optimal gelöst werden.

Nun ist allerdings für das Circle Packing Problem with Unequal Circles bewiesen, dass es sich um ein NP-schweres Problem handelt. Ich habe bei meiner Recherche herausgefunden, dass auch beim Circle Packing Problem with Equal Circles stark vermutet wird, dass es sich um ein NP-schweres Problem handelt, d.h. dass kein Algorithmus existiert, der dieses Problem in polynomieller Zeit lösen kann (unter der Voraussetzung, das P ungleich NP ist).

Es ist daher höchstwahrscheinlich, dass auch die BWinf-Aufgabe NP-schwer ist. Es muss also davon ausgegangen werden, dass kein Verfahren gefunden werden kann, das das Problem optimal löst bzw. dies in angemessener Zeit bewerkstelligt.

Dies rechtfertigt den Einsatz von Heuristiken. Brute-Force Verfahren kommen bei einer kontinuierlichen Lösungsmenge nicht infrage.

Greedy-Algorithmus

Um das Polygon mit Punkten zu befüllen, wähle ich ein Verfahren, das einen Punkt nach den anderen platziert. Dabei wird in einer Menge *nodes_to_add* gespeichert, welche Punkte als nächster zu platzierender Punkt infrage kommen.

Am Anfang befinden sich in nodes_to_add nur die Polygonecken. In jedem Schritt wird nun aus nodes_to_add der „beste“ Punkt ausgesucht, hierfür werden verschiedene Heuristiken verwendet, die die Punkte in nodes_to_add bewerten. Nach Platzierung des auserwählten Punkts (im Folgenden als last_added bezeichnet) wird die Menge nodes_to_add mit folgendem Verfahren aktualisiert:

Ein Kreis c wird definiert, der seinen Mittelpunkt im Punkt last_added hat und einen Radius von min_distance hat. Es werden die Schnittpunkte, die dieser Kreis mit den Polygonkanten hat, bestimmt und zu nodes_to_add hinzugefügt. Anschließend wird über alle anderen zuvor platzierten Punkte. Für jeden Punkt p wird ein zweiter Kreis k definiert, der seinen Mittelpunkt in p hat und ebenfalls den Radius min_distance hat. Die Schnittpunkte von k und c werden ebenfalls zu nodes_to_add hinzugefügt.

Dies wird solange fortgeführt, bis keine Punkte mehr hinzugefügt werden können, da jeder hinzufügbare Punkt zu nahe an einem anderen Punkt wäre. Generell muss nach dem Hinzufügen eines Punkts für jeden Punkt in nodes_to_add sichergestellt werden, dass er noch hinzugefügt werden kann und nicht zu nahe an bestehenden Punkten ist, um ungültige Ergebnisse zu vermeiden.

Heuristiken

Es müssen Heuristika entwickelt werden, basierend auf denen die Punkte in nodes_to_add bewertet werden. Ich entwickle zwei Verfahren, die auf jeweils einer anderen Heuristik aufbauen:

1. Heuristik (Flächenheuristik): Die Heuristik bewertet für einen einzelnen Punkt P, wie sinnvoll es ist, diesen Punkt P hinzuzufügen. Hierfür platziert sie auf P einen Kreis C mit dem Radius min_distance. Es wird anschließend die Schnittfläche zwischen C und dem Bereich des Polygons, in dem prinzipiell noch Punkte platziert werden könnten (weil kein anderer Kreis zu nahe ist), bestimmt. Diese Schnittfläche ist eine Art „Maß für den Verlust an Platzierungsmöglichkeiten“, da auf dieser Fläche vor Platzierung von Punkt P noch Punkte hätten platziert werden können, nach Platzierung von P aber nicht mehr. Die 1. Heuristik wählt daher den Punkt aus, für den die niedrigste Schnittfläche berechnet wird.
   Generell lässt sich sagen, dass ein Ausschnitt aus der Fläche des Polygons als ein Ausschnitt aus der kontinuierlichen Menge an Positionen für einen Punkt betrachtet werden kann.
2. Heuristik (Distanzheuristik): Die Heuristik bewertet ebenfalls für eine Punkt P, wie sinnvoll es ist, diesen Punkt hinzuzufügen. Dabei wird für jeden Punkt P die durchschnittliche Distanz zu den nächsten 6 Nachbarpunkten von P bestimmt. Der Hintergedanke hierbei ist, dass je näher die Punkte aneinander platziert werden können, desto mehr Punkte auch insgesamt im Polygon platziert werden können. Daher wählt die 2. Heuristik den Punkt aus, für den die niedrigste durchschnittliche Distanz berechnet wird.

Interessanterweise sorgt eine Verwendung der 2. Heuristik dafür, dass die Punkte meist in einem fast perfekten gleichseitigen Dreiecksmuster angeordnet werden. Eine Verwendung der 1. Heuristik hat hingegen zur Folge, dass die Punkte nahe an den Kanten platziert werden. Um „enge“ Bereiche (z.B. Spitzen mit geringen Innenwinkeln im Polygon gut zu befüllen eignet sich daher die 1. Heuristik am besten, während weiträumige Bereiche im Polygon eher mit der 2. Heuristik befüllt werden sollten. Theoretisch könnte man also einen Sweep-Line-Algorithmus entwickeln, der einmal über das Polygon läuft, „enge“ Bereiche ausfindig macht und diese Bereiche dann gezielt mit der 1. Heuristik befüllt und den Rest mit der 2. Heuristik befüllt. Die Implementierung eines solchen Algorithmus würde den Aufwandrahmen allerdings sprengen und wurde daher nicht durchgeführt. Stattdessen werden die beiden Heuristiken in separaten Programmen implementiert.

Laufzeitüberlegungen

Der Greedy-Algorithmus iteriert in jedem Schritt über alle Polygonkanten und alle bereits platzierten Punkte, um die Schnittpunkte zu bestimmen. Er hat daher eine Laufzeit von ca. O(n*m) + O($\sum_{i=0}^{m} i$) mit n := Anzahl Polygonkanten und m = Anzahl der Punkte, die insgesamt platziert werden.

2. Teilproblem: Gesundheitszentrum möglichst optimal platzieren

Die Platzierung des Gesundheitszentrums erfolgt in mehreren Schritten. Zunächst wird mit folgendem Vorgehen eine initiale Position für das Gesundheitszentrum ermittelt:

1. Das Polygon wird mit einer möglichst günstigen bzw. schellen Heuristik den Regeln entsprechend mit Punkten befüllt.
2. Der Punkt best_P, für den es am meisten Punkte gibt, die weniger oder genau 85 LE von best_P entfernt sind, wird ausgewählt.
3. Initiales Gesundheitszentrum = best_P

(Anmerkung: „Gesundheitszentrum“ wird im Folgenden als Ghz abgekürzt)

Um die Position des Gesundheitszentrums zu verbessern, wird anschließend folgendes Verfahren ausgeührt:

1. Der Bereich um best_P herum, der einen Abstand von max. min_distance zu best_P hat, wird mit einer Genauigkeit von 0.1 diskretisiert, d.h. die Fläche wird in eine Menge M an diskreten Punkten umgewandelt.
2. Es wird der Punkt X aus der Menge M ausgewählt, für den die Schnittfläche zwischen dem Polygon und einem Kreis, der seinen Mittelpunkt in X und den Radius 85 hat, maximal wird. Die Schnittfläche wird hier als Heuristik hierfür verwendet, wie gut ein Punkt als mögliche Ghz-Position das Polygon abdeckt.

Gesamt-Algorithmus

Da im vom Gesundheitszentrum abgedeckten Schutzbereich die Punkte mit einer höheren Flächeneffizienz bzw. dichter platziert werden können, hat die Qualität der Punktplatzierung in diesem Bereich die höchste Priorität. Es wird daher zuerst der Bereich, der von Ghz geschützt ist, mit Punkten gefüllt – danach auch die restlichen Bereiche.

Der Gesamt-Algorithmus folgt also diesem vereinfachten Ablaufplan:

1. Initiale Position für Ghz ermitteln
2. Position von Ghz verbessern
3. nodes_to_add = Ecken, die im Ghz-Schutzgebiet liegen, und Schnittpunkte, die der kreisförmige Rand des Ghz-Schutzgebiets mit den Polygonkanten hat
4. **solange** nodes_to_add nicht leer ist:
5. Über eine Heuristik entscheiden, welcher Punkt aus nodes_to_add am besten ist

6. Diesen Punkt zum Polygon hinzufügen bzw. platzieren
7. Die daraus resultierenden neuen potenziellen Punkte über die Schnittpunke ermitteln und zu nodes_to_add hinzufügen, sofern sie im Ghz-Schutzgebiet sind
8. Für jeden Punkt in nodes_to_add überprüfen, ob er überhaupt noch zum Polygon hinzugefügt werden kann, oder ob andere Punkte inzwischen zu nahe sind
9. nodes_to_add = Ecken, die außerhalb vom Ghz-Schutzgebiet liegen (jetzt wird dieser Bereich gefüllt)
10. **solange** nodes_to_add nicht leer ist:
11. Über eine Heuristik entscheiden, welcher Punkt aus nodes_to_add am besten ist
12. Diesen Punkt zum Polygon hinzufügen bzw. platzieren
13. Die daraus resultierenden neuen potenziellen Punkte über die Schnittpunke ermitteln und zu nodes_to_add hinzufügen
14. Für jeden Punkt in nodes_to_add überprüfen, ob er überhaupt noch zum Polygon hinzugefügt werden kann, oder ob andere Punkte inzwischen zu nahe sind

Es können aber natürlich noch Optimierungen vorgenommen werden, die den Punktplatzierungsprozess effektiver machen bzw. die Anzahl an insgesamt platzierbaren Punkten erhöhen.

Optimierungen

Eckpunkte von spitzen Ecken zu Beginn auf jeden Fall hinzufügen: Wie zuvor erläutert lässt sich sagen, dass ein Ausschnitt aus der Fläche des Polygons als ein Ausschnitt aus der kontinuierlichen Menge an Positionen für einen Punkt betrachtet werden kann. Wird ein Punkt platziert, dann fällt ein kreisförmiger Teil dieser kontinuierlichen Menge an Positionen weg.

Von sollten in Ecken, die einen Winkel kleiner als 30° haben und im Umkreis von min_distance keine weiteren Ecken oder nicht an der Ecke anliegenden Kanten haben, auf jeden Fall sofort Punkte platziert werden. Die Verringerung der kontinuierlichen Positionsmenge ist für ein solches Eckfeld E nämlich minimal. Mit anderen Worten: Wird an einer anderen Position, die weniger als min_distance LE von E entfernt ist, ein Punkt platziert, dann ist die Verkleinerung der kontinuierlichen Positionsmenge immer größer, als wenn man den Punkt in der Ecke E platziert hätte.

Geschicktes Wählen von Startkante und Startecke: Wenn mit der Flächenheuristik gearbeitet wird, dann spielt es nicht wirklich eine Rolle, welche „Startecke“ festgelegt wird, da die Heuristik ohnehin dynamisch arbeitet.

Wird aber mit der Distanzheuristik gearbeitet, die ein gleichseitiges Dreiecksmuster an Punkten generiert, dann sollten Startkante und Startecke so gewählt sein, dass dieses Muster möglichst gut an den Kanten des Polygons ausgerichtet ist.

Def. „Startecke“: Die Ecke, in der als erstes ein Punkt platziert wird.

Def. „Startkante“: Die Kante, auf der der zweite platzierte Punkt liegen soll.

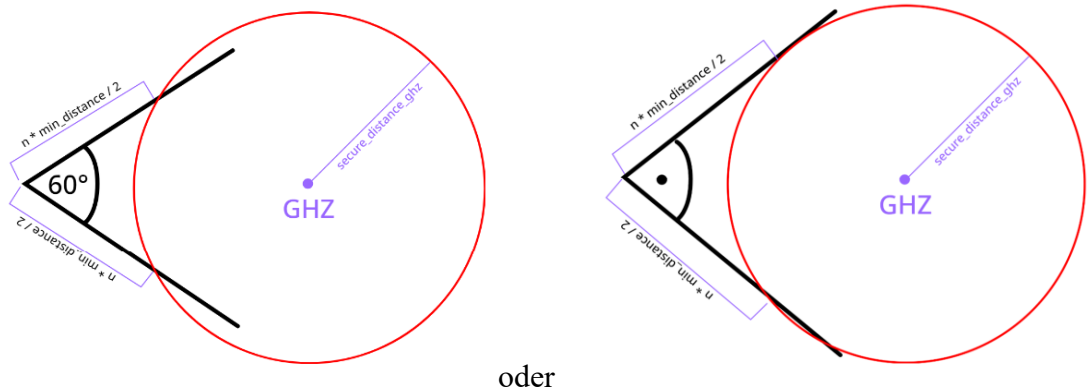
Dies kann mit folgendem Verfahren gewährleistet werden:

1. Es werden alle Randsegmente bestimmt, die innerhalb des Ghz-Schutzbereichs liegen.
2. Es werden die Steigungswinkel (in Grad) aller Randsegmente bestimmt.
3. Die Randsegmente, deren Steigungswinkel modulo 60° das das gleiche ergeben, werden in sog. *trigonalen Winkelsystemen* gespeichert (Hintergrund: Ein gleichseitiges Dreieck hat nur 60°-Winkel und kann daher besonders gut an zwei Kanten gepasst werden, die zueinander im 60°-Winkel stehen.
4. Die Randsegmente, deren Steigungswinkel modulo 90° das das gleiche ergeben, werden in sog. *orthogonalen Winkelsystemen* gespeichert (Hintergrund: Ein gleichseitiges Dreieck ist zugleich auch ein gleichschenkliges Dreieck, und ein gleichschenkliges Dreieck lässt sich in zwei rechtwinklige Dreiecke aufteilen. Ein gleichseitiges Dreieck kann daher sogar auf drei Arten in rechtwinklige Dreiecke aufgeteilt werden. Punkte, die in einem gleichseitigen Dreiecksmuster verteilt sind, lassen sich daher auch sehr gut an zwei Kanten passen, die im rechten Winkel zueinanderstehen.)
5. Das Winkelsystem, dessen kumulierte Kantenlänge am größten ist, wird ausgewählt. (Mit kumulierter Kantenlänge ist die aufaddierte Menge aller Kantensegmente, die sich im Winkelsystem befinden, gemeint)
6. Das längste Kantensegment aus dem zuvor ausgewählten Winkelsystem wird als Startkante ausgewählt. (An der Starkante entlang können maximal viele Punkte platziert werden, da die auf der Startkante liegenden Punkte einen Abstand von min_distance zueinander haben. Daher sollte das längste Segment als Startkante ausgewählt werden.)
7. Die Ecke des Startkantensegments, an der der größere Innenwinkel ist, wird als Startecke ausgewählt. (Wie zuvor erläutert ist eine Befüllung über die Distanzheuristik an den Stellen, wo das Polygon sehr eng ist bzw. wo spitze Ecken sind, nicht sinnvoll. Daher sollte die Befüllung mit der Distanzheuristik an der weniger spitzen Ecke begonnen werden.)

Gesundheitszentrum passend an zueinander orthogonalen oder im 60°-Winkel stehenden Ecken „alignen“:

Damit auch der Bereich außerhalb vom Gesundheitszentrum mit der Distanzheuristik unmittelbar weiter befüllt werden kann, sollte das Gesundheitszentrum – wenn möglich bzw. sinvoll[1] - so platziert werden, dass Folgendes zutrifft:

[1] Dies ist natürlich nicht sinnvoll, wenn durch das „Alignen“ des Gesundheitszentrums die geschützte Fläche deutlich kleiner wird.



oder

Wenn die Position des Gesundheitszentrums so gewählt wird, dass die Längen der oben lila umkasteten Segmente ein Vielfaches von min_distance/2 sind, dann kann das auch beim Befüllen von regelmäßigen Polygonen Vorteile mit sich bringen, wie an Beispiel 3 (siehe Kapitel mit den Beispielen) deutlich wird.

Alternative Vorgehensweisen

Zum Befüllen des Polygons mit Punkten sind auch diese Vorgehensweisen denkbar:

- Auf das Polygon passend zu Startkante und Startecke ein Raster bestehend aus in gleichseitigen Dreiecken angeordneten Punkten legen und die Punkte basierend auf diesem Raster platzieren. Dieser Ansatz wird in der Datei aufgabe3_H3.py implementiert. Da er sich allerdings als deutlich schlechter als die heuristischen Ansätze erwiesen hat, wird er nicht ausführlich dokumentiert. Ein Problem an dem Rasteransatz ist, dass er sich nicht so gut an Anomalien im Polygon (wie z.B. die Einbuchtungen in der Mitte des Polygons bei Beispiel 5) anpassen kann.
- Prinzipiell ist auch denkbar, mehr Punkte als von der Heuristik erzielt zufällig im Polygon zu platzieren – und anschließend eine Physik-Engine zu programmieren und zwischen den Punkten Abstoßungs- und Anziehungskräfte wirken zu lassen, die dafür sorgen, dass sich die Punkte im Polygon in einer Konfiguration anordnen, in der der Mindestabstand erfüllt ist.
- Möglicherweise könnten auch Automatic Mesh Generation Algorithms angewendet werden.

Es wurde versucht, mithilfe von Lloyd‘s Algorithmus eine kontinuierliche Verbesserung der Punktplatzierung in Polygon zu erreichen, um weitere Punkte platzieren zu können. Dies hat sich jedoch in den konkaven und komplexen Polygonen, die durch die Aufgabe gegeben sind, als schwer umsetzbar erwiesen.

Umsetzung

Die Lösungsidee wird in Python 3.10 implementiert. Verwendete Python libraries:
- *math:* Verwendet für mathematische Operationen (atan, Quadratwurzel), Python standard library
- *shapely.geometry*: Verwendet zum Berechnen von Schnittflächen (alle anderen geometrischen Hilfsfunktionen wurden selbst implementiert)
- *matplotlib:* Verwendet zum Visualisieren des Polygons und den in ihm platzierten Punkten

Einlesen des Polygons

Die Eingabedatei wird vom Nutzer als erstes Kommandozeilenargument angegeben und vom Programm eingelesen. Parameter wie der Mindestabstand im Gesundheitszentrum werden direkt im Programmcode angegeben.

```
# Einlesen der Daten
# sys.argv[1] ist das erste Kommandozeilenargument
with open(sys.argv[1], "r") as f:
    data = f.read().split("\n")

# Abstände und Radien festlegen
min_distance = 20 # Mindestabstand der Orte außerhalb vom
Gesundheitszentrum
min_distance_ghz = 10 # Mindestabstand der Orte innerhalb vom
Gesundheitszentrum
secure_distance_ghz = 85 # Radius des vom Gesundheitszentrum geschützten
Gebiets

# Koordinaten der Polygon-Eckpunkte einlesen
num_corners = int(input_lines.pop(0))
coords = []
for i in range(num_corners):
    line = input_lines.pop(0).split(" ")
    coords.append((float(line[0]), float(line[1])))
```

Anschließend werden die Eckpunkte des Polygons in einem Solution-Objekt gespeichert.

```
# Neues Solution Objekt initialisieren
p = Solution(coords)
```

Vererbungsstruktur der Klassen

In der Datei geometric_util.py wird eine Point-Klasse definiert. Die Klasse fungiert als Ersatz für Tupel, da sie zum Vergleich mit einem anderen Point-Objekt immer die ebenfalls in geometric_util.py definierte isclose-Methode verwendet und somit vermeidet, dass durch Fließkommazahlungenauigkeiten zwei gleiche Werte nicht als gleich erkannt werden.
Die Kreis-Klasse erbt von Point und übernimmt somit die Funktionen zum Vergleichen zweier Point-Klassen. Die Node-Klasse, die einen im Polygon platzierten Punkt repräsentiert, erbt wiederum von Kreis und übernimmt somit die Funktionen zum Berechnen von Schnittpunken zwischen zwei Kreisen.

Geometrische Hilfsfunktionen in geometric_util.py

Funktionen:
- isclose
- distance
- crossproduct2d

Klassen:

- Point
- Kreis
- Edge
- Polygon

Zu den Klassen gehören folgende Funktionen: (Die Namen der Funktionen sind dabei ziemlich selbsterklärendend, ggf. Wurden Erläuerungen ergänzt)

Klasse Kreis:
    schnittpunkt_kreis(kreis2)
Klasse Edge:
    distance_to_edge(edge2)
    schnittpunkt_gerade(edge2)
    schnittpunkt_kreis(kreis)
    length()
    is_identic_to (gibt zurück, ob die Kante auf einer bestimmten Gerade liegt)
    steigungswinkel()
    is_on_edge()
    orthogonal_line_through_point
Klasse Polygon:
    get_corner(corner_id) (verwendet eine zyklische Durchnummerierung der Ecken)
    get_edge(edge_id)
    get_corner_angle(corner_id)
    triangulate() (Trianguliert das Polygon unter Verwendung vom Ear-Clipping-Algorithm)
    triangle_area()
    calc_area()
    point_in_polygon(point)
    exact_cut_area ()
    (Funktion, die mithilfe von der Bibliothek shapely die Schnittfläche zwischen dem Kreis und dem Polygon bestimmt – von der Schnittfläche werden die sich in Polygon.circle_union befinden Kreise abgezogen, wenn der Funktion das Argument exclude_union=True gegeben wird)

    add_to_circle_union(circle)

Definieren einer Solution-Klasse

Die Solution Klasse dient zum Speichern von (Teil-)Lösungen. Ihre Funktionen lassen sich dabei folgendermaßen kategorisieren:

Die Klasse `Solution` ist dazu gedacht, (Teil-)Lösungen für ein bestimmtes Problem zu speichern und zu verwalten. In diesem Fall scheint es sich um ein geometrisches Problem zu handeln, bei dem ein Polygon mit bestimmten Kriterien gefüllt werden soll, wie z.B. das Platzieren von Orten innerhalb oder außerhalb eines Gesundheitszentrums-Einzugsgebiets unter Berücksichtigung von Mindestabständen und anderen Einschränkungen.

Die Funktionen innerhalb der Klasse können in mehrere Gruppen unterteilt werden:

1. **Initialisierungsmethoden**:

- reset(self): Setzt alle relevanten Attribute der Lösung auf ihren Anfangszustand zurück.

2. Methoden zur Darstellung und Ausgabe der Lösung:
- render(self, *, title="", draw_small_circles=True, draw_big_circles=False): Zeigt eine grafische Darstellung des Polygons und der platzierten Orte an.
- print(self): Gibt Informationen über die platzierten Orte in der Konsole aus.

3. Methoden zum Umgang mit dem Gesundheitszentrum:
- is_in_ghz_range(self, point): Überprüft, ob ein Punkt im Einzugsgebiet des Gesundheitszentrums liegt.
- implement_ghz(self, ghz): Setzt die Position des Gesundheitszentrums und ermittelt die Schnittpunkte des Einzugsgebiets mit den Polygonkanten.
- implement_ghz_intersections(self): Fügt die Schnittpunkte des Einzugsgebiets mit den Polygonkanten zur Liste der hinzuzufügenden Orte hinzu.
- find_ghz(self): Findet eine initiale Position für das Gesundheitszentrum basierend auf der Anzahl der abgedeckten Orte.
- optimize_ghz(self, ghz_init_pos): Optimiert die Position des Gesundheitszentrums basierend auf dem Schnittflächenanteil mit dem Polygon.
- align_ghz(self, old_ghz_pos): Passt die Ghz-Position an die Kanten an

4. Methoden zum Hinzufügen von Orten zum Polygon:
- add_definite_nodes(self): Fügt Ecken hinzu, die sich an spitzen Innenwinkeln befinden und keine anderen Ecken oder Kanten blockieren.
- add_node(self, new_node): Fügt einen Ort hinzu und aktualisiert die Distanzen zu anderen Orten.
- successors(self): Findet potenzielle Schnittpunkte des letzten hinzugefügten Ortes mit anderen Orten und Kanten.
- update_nodes_to_add(self): Aktualisiert die Liste der hinzuzufügenden Orte basierend auf dem zuletzt hinzugefügten Ort.

5. Heuristiken und Funktionen, die Blockaden überprüfen:
- is_blocked(self, new_point): Überprüft, ob ein Ort platziert werden kann, ohne andere Orte zu nahe zu kommen.
- fill_area_heuristic(self): Befüllt einen Bereich basierend auf der Flächenheuristik.
- area_heuristic(self, x): Berechnet den Flächenanteil, den der Ort x an der gesamten Wegfallfläche hat.
- fill_distance_heuristic(self): Befüllt einen Bereich basieren auf der Distanzheuristik
- distance_heuristic(self): Bewertet den aktuellen Polygon basierend auf der Distanzheuristik

Die Klasse Solution erbt von Polygon. Dies ergibt Sinn, da eine Lösung natürlich immer in Verbindung zu dem Polygon stehen muss, das gelöst wurde.

Programmablauf von aufgabe3_H1.py (auf Flächenheuristik basierendes Programm)

Zunächst wird ein neues Solution Objekt initialisiert:
```
p = Solution(coords)
```

Danach wird die Ghz-Position gefunden und optimiert:

```
p.add_corners_to_nodes_to_add()
p.fill_area_heuristic() # Eine initiale Punktplatzierung erzeugen,
basierend auf der dann das Gesundheitszentrum platziert werden kann (wird
beim Zurücksetzen mit reset() wieder gelöscht)
ghz = p.find_ghz()
ghz = p.optimize_ghz(ghz)
p.reset() # Zurücksetzen auf Anfangszustand
p.implement_ghz(ghz)
```

Anschließend werden an spitzen Ecken, an denen auf jeden Fall Orte zu platzieren sind, die entsprechenden Orte platziert, und die Schnittpunkte des Ghz-Schutzbereiches mit den Kanten werden in der p.implement_ghz_intersections() Funktion zu p.nodes_to_add hinzugefügt:

```
p.add_definite_nodes()
p.implement_ghz_intersections()
```

Danach wird das Ghz-Schutzgebiet befüllt:

```
p.area_heuristic_cache = {}
ghz_area_uncut = len(set(p.ghz_intersection_points)) ==
len(p.ghz_intersection_points) / 2
if ghz_area_uncut:
    for corner in p.corners:
        if corner in p.ghz_intersection_points:
            continue
        if p.is_in_ghz_range(corner):
            ghz_area_uncut = False
            break
if ghz_area_uncut:
    # Wenn der Ghz-Einzugsbereich keine Kanten schneidet, dann wird
direkt der ganze Bereich befüllt, hierbei werden je nach Bereich die
entsprechenden Abstandsregeln eingehalten
    pass # -> kein Befüllen des reinen Ghz-Bereichs
else:
    p.add_corners_to_nodes_to_add()
    p.fill_area_heuristic()
```

Abschließend werden die restlichen Bereiche befüllt, und der endgültige Besiedlungsplan ausgegeben:

```
p.fill_ghz = False
p.nodes_to_add = p.nodes_to_add_outside_ghz
p.add_corners_to_nodes_to_add()
for node in list(p.nodes_to_add):
    if p.is_blocked(node):
        p.nodes_to_add.remove(node)
p.fill_area_heuristic()
p.print()
p.render(title="[H1] Finaler Besiedlungsplan")
```

Beispiele

Grundaufgabe (keine Erweiterungen)

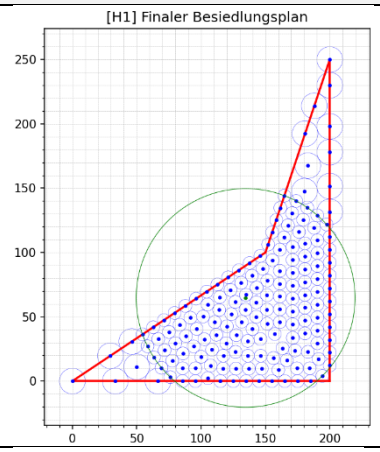
Die in den Beispielen 1 bis 5 verwendeten Eingabedateien stammen von der BWinf-Webseite, für Beispiel 6 und 7 habe ich eigene Dateien ergänzt. Die Eingabedateien und die Programme sind im Ordner *Aufgabe 1* zu finden. Die vollständigen Outputs sind im Ordner *Aufgabe 1/Outputs* gespeichert (die Outputs in der Doku sind gekürzt) und nach folgendem Format benannt:

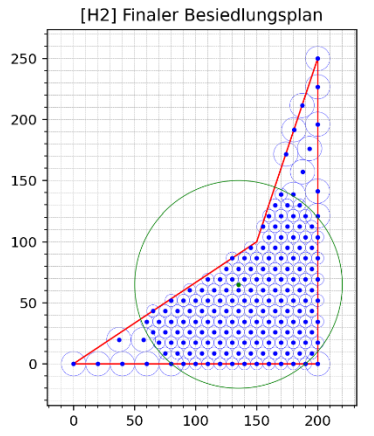
```
aufgabe3_<Programmname>_<Eingabedatei-Name>.txt
```

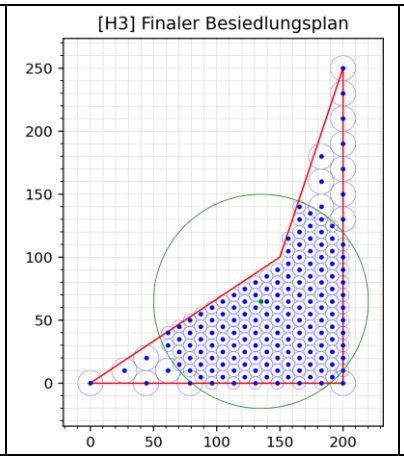
Zu jedem Beispiel ist für jedes Programm dokumentiert, welchen Text in der Konsole und welche graphische Darstellung es ausgibt. Für das Messen der angegebenen Evaluierungszeiten wurde die Programm auf einem Gerät mit 16 GB RAM ausgeführt.

Beispiel 1 bis 5

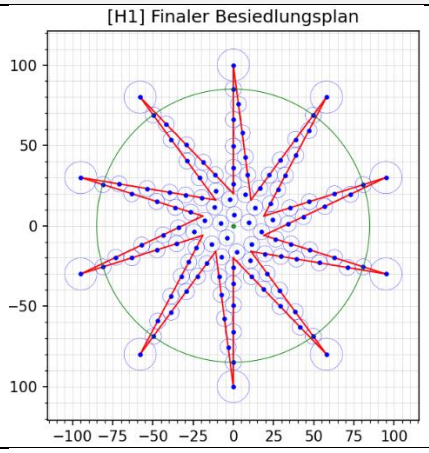
Eingabedatei: siedler1.txt			
Programm	**Ausgabetext**	**Ausgabebild**	**Eval.-dauer**

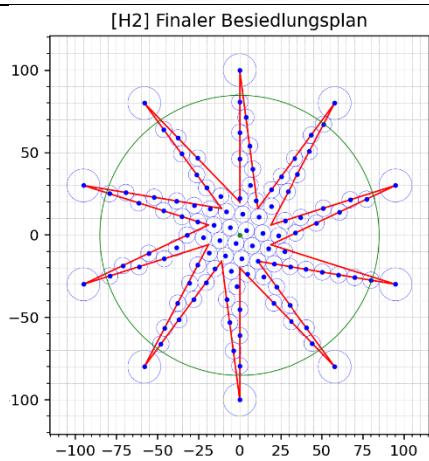
aufgabe3_H1.py	165 Orte insgesamt platziert
	149 Orte im Ghz-Einzugsgebiet platziert
	
	`Position des Gesundheitszentrum: (134.62083, 64.69253)`
	`Positionen der Orte: (200.0, 250.0), (0.0, 0.0), (164.72738, 144.18213), (54.49289, 36.32859), ..., (29.53138, 19.68759), (33.34342, 0.0)`
			4.3 s

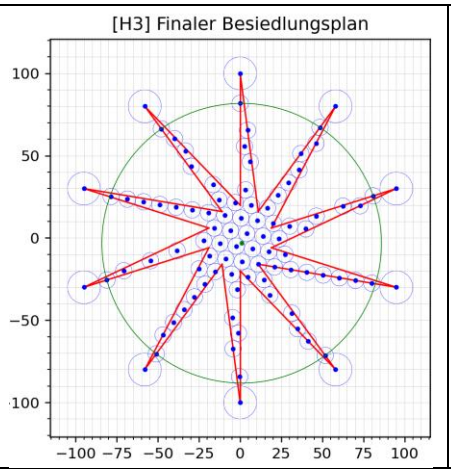
aufgabe3_H2.py	174 Orte insgesamt platziert 155 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (135.10426, 64.89574) Positionen der Orte: (200.0, 250.0), (0.0, 0.0), (200.0, 0.0), (180.0, 0.0), (170.0, 0.0), (160.0, 0.0), (150.0, 0.0), (140.0, 0.0), (130.0, 0.0), (120.0, 0.0), (110.0, 0.0), (100.0, 0.0), (90.0, 0.0),		8 s

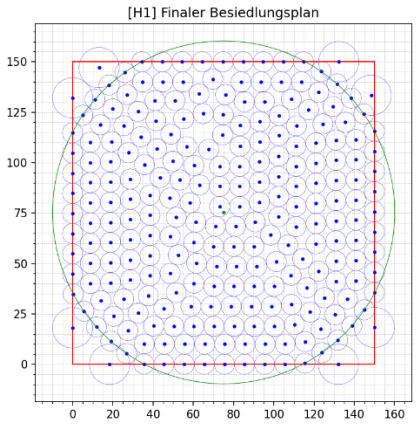
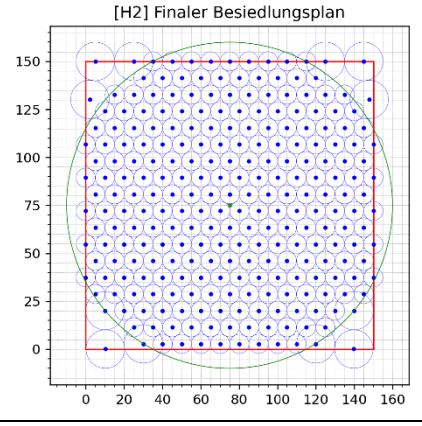
aufgabe3_H3.py	171 Orte insgesamt platziert 154 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (135.10426, 64.89574) Positionen der Orte: (200.0, 250.0), (0.0, 0.0), (200.0, 70.0), (200.0, 80.0), (200.0, 90.0), (200.0, 100.0), (200.0, 110.0),		1 s

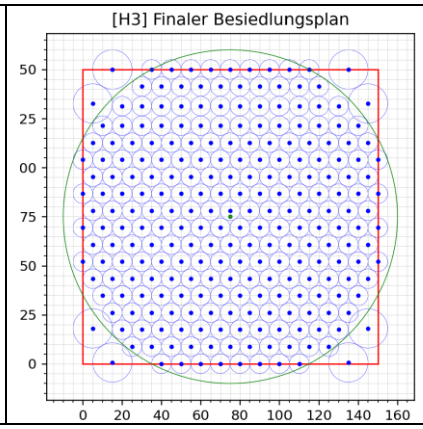
Eingabedatei: siedler2.txt			
Programm	**Ausgabetext**	**Ausgabebild**	**Eval.-dauer**

aufgabe3_H1.py	118 Orte insgesamt platziert
	108 Orte im Ghz-Einzugsgebiet platziert
	
	`Position des Gesundheitszentrum: (-0.12468, 0.04042)`
	`Positionen der Orte: (0.0, 100.0), (58.0, 80.0), (95.0, 30.0), (95.0, -30.0), ..., (0.60351, 6.85052), (-7.96759, 1.69917), (-4.12788, -7.53428)`
			4.1 s

aufgabe3_H2.py	110 Orte insgesamt platziert 100 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (0.01177, -0.29966) Positionen der Orte: (0.0, 100.0), (58.0, 80.0), (95.0, 30.0), (95.0, -30.0), (58.0, -80.0), (0.0, -100.0),		5.1 s

aufgabe3_H3.py	94 Orte insgesamt platziert 84 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (0.94902, -3.22826) Positionen der Orte: (0.0, 100.0), (58.0, 80.0), (95.0, 30.0), (95.0, -30.0), (58.0, -80.0), (0.0, -100.0), (-58.0, -80.0), (-95.0, -30.0),		2s

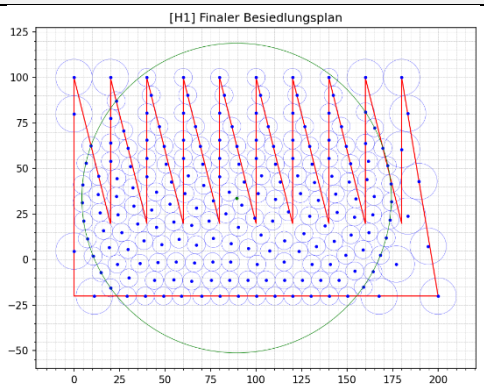
Eingabedatei: siedler3.txt			
Programm	**Ausgabetext**	**Ausgabebild**	**Eval.-dauer**
aufgabe3_H1.py	242 Orte insgesamt platziert 234 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (75.10444, 75.3373) `Positionen der Orte: (34.47832, 150.0), (25.99083, 144.71204), (18.18311, 138.46387), (11.16324, 131.34198), (150.0, 18.30527), (132.1171, 0.0), (18.4405, 0.0), (0.0, 18.07413)`		3.4 s
aufgabe3_H2.py	259 Orte insgesamt platziert 247 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (75.0, 75.0) Positionen der Orte: (115.0, 150.0), (105.0, 150.0), (95.0, 150.0), (85.0, 150.0), (75.0, 150.0), (65.0, 150.0),		16.1 s

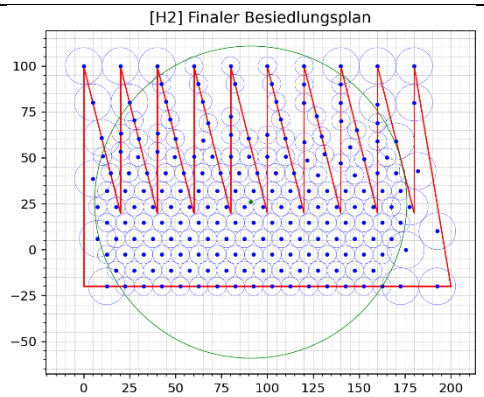
aufgabe3_H3.py	250 Orte insgesamt platziert 242 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (75.0, 75.0) Positionen der Orte: (115.0, 150.0), (105.0, 150.0), (95.0, 150.0), (85.0, 150.0), (75.0, 150.0), (65.0, 150.0),		

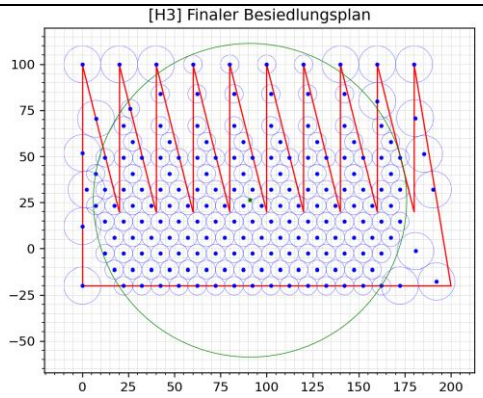
.

Eingabedatei: siedler4.txt

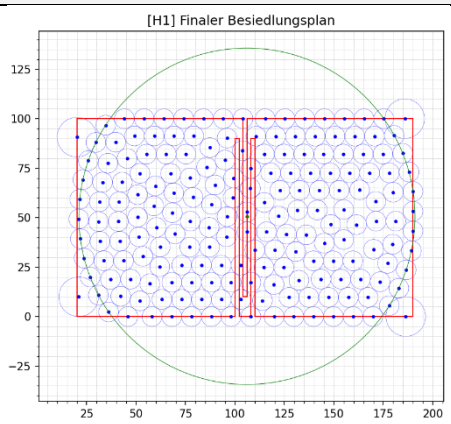
Programm	Ausgabetext	Ausgabebild	Eval.-dauer

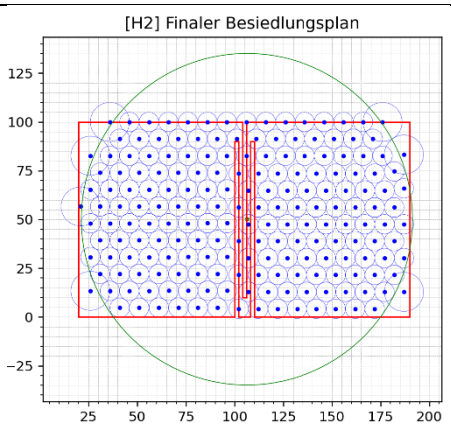
aufgabe3_H1.py	202 Orte insgesamt platziert
	187 Orte im Ghz-Einzugsgebiet platziert
	
	`Position des Gesundheitszentrum: (89.35507, 33.83296)`
	`Positionen der Orte: (0.0, 100.0), (20.0, 100.0), (40.0, 100.0), (60.0, 100.0),`
			7.1 s

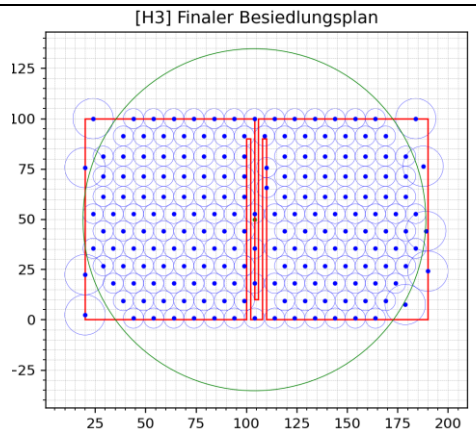
aufgabe3_H2.py	202 Orte insgesamt platziert 181 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (90.99688, 25.83937) Positionen der Orte: (0.0, 100.0), (20.0, 100.0), (40.0, 100.0), (60.0, 100.0), (80.0, 100.0), (100.0, 100.0), (120.0, 100.0), (140.0, 100.0), (160.0, 100.0),		16.1 s

aufgabe3_H3.py	184 Orte insgesamt platziert 166 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (91.0, 26.30127) Positionen der Orte: (0.0, 100.0), (20.0, 100.0), (40.0, 100.0), (60.0, 100.0), (80.0, 100.0), (100.0, 100.0), (120.0, 100.0),		5 s

Eingabedatei: siedler5.txt			
Programm	**Ausgabetext**	**Ausgabebild**	**Eval.-dauer**

aufgabe3_H1.py	191 Orte insgesamt platziert
	187 Orte im Ghz-Einzugsgebiet platziert
	
	`Position des Gesundheitszentrum: (106.00093, 50.67874)`
	`Positionen der Orte: (175.22818, 100.0), (180.54156, 91.5284), (184.82322, 82.4914), (188.01393, 73.01409), (190.0, 63.2133), ...`
			6.6 s

aufgabe3_H2.py	193 Orte insgesamt platziert 186 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (106.35783, 50.13236) Positionen der Orte: (106.0, 100.0), (116.0, 100.0), (96.0, 100.0), (126.0, 100.0), (136.0, 100.0), (86.0, 100.0), (146.0, 100.0), (156.0, 100.0),		10,1 s

aufgabe3_H3.py	188 Orte insgesamt platziert 179 Orte im Ghz-Einzugsgebiet platziert Position des Gesundheitszentrum: (104.0, 49.67949) Positionen der Orte: (104.0, 100.0), (114.0, 100.0), (94.0, 100.0), (124.0, 100.0), (84.0, 100.0), (134.0, 100.0), (74.0, 100.0),		2,3 s

Vergleich

Insgesamt platzierte Orte:
Die höchste platzierte Anzahl an Orten ist jeweils grün hinterlegt. Es fällt auf, dass in den Polygonen, die spitze Ecken enthalten, Heuristik 2 besser abschneidet, während in den anderen Polygonen Heuristik 1 am besten. Heuristik 3 schneidet nie am besten ab.

Programm	siedler1.txt	siedler2.txt	siedler3.txt	siedler4.txt	siedler5.txt
aufgabe3_H1.py	165	118	242	202	191
aufgabe3_H2.py	174	110	259	202	193
aufgabe3_H3.py	171	94	250	184	188

Quellcode

Im Folgenden einige Auszüge aus dem Quellcode meiner Implementierung.

Geometrische Hilfsfunktionen (aus geometric_util.py)

Damit das Dokument nicht zu lang wird, werden hier nur die wesentlichsten Hilfsfunktionen aufgeführt. Bei den hier nicht aufgeführten Funktionen handelt es sich um Standardverfahren aus der analytischen Geometrie, die sogar im normalen Matheunterricht thematisiert werden und daher deutlich weniger interessant sind als speziellere Verfahren aus der analytischen Geometrie (wie z.B. die Bestimmung der Schnittpunkte zweier verschieden großer Kreise), die man nicht beigebracht bekommt, sondern sich selbst herleiten muss.

Hilfsfunktion: Gibt an, ob sich zwei Zahlen ähneln (zum Umgang mit Pythons float-Ungenauigkeit wird diese Funktion anstelle vom == Operator verwendet, wenn zwei Punkte oder Distanzen verglichen werden):

```
def isclose(num1, num2, *, digits=9):
    return abs(num2-num1) < (10**(-digits))
```

Hilfsfunktion: Berechnet das zweidimensionale Kreuzprodukt zweier 2D-Vektoren:

```
def crossproduct2d(vector1 : tuple, vector2 : tuple):
    return vector1[0] * vector2[1] - vector2[0] * vector1[1]
```

Klasse, die einen Punkt im zweidimensionalen Raum repräsentiert:

```
class Point:
    """
    Repräsentiert einen Punkt, bestehend aus x- und y-Koordinate
    Die Klasse wurde als Ersatz für tuple() erstellt, damit eine angepasste
Gleichheitsfunktion (__eq__) eingerichtet werden kann, die zum Vergleichen Pythobs
isclose Funktion verwendet (Umgang mit Pythons Fließkommazahl-Genauigkeit)
    """
    def __init__(self, x,y):
        self.center = (x,y)

    def __str__(self):
        return str(self.center)

    def __eq__(self,other):
        if isinstance(other, tuple):
            other = Point(*other)
        elif isinstance(other, Point):
            return isclose(self.center[0], other.center[0]) and isclose(self.center[1],
other.center[1])
        else:
            return False

    def __getitem__(self,i):
        return self.center[i]

    def __tuple__(self):
        return self.center
```

Klasse, die einen Kreis im zweidimensionalen Raum repräsentiert:

```
class Kreis(Point):
    """
    Repräsentiert einen Kreis
    Implement geometrisch-analytische Methoden zum Umgang mit Kreisen
    (Berechnung der Schnittpunkte zweier Kreise)
    """
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
```

Funktion, die die Schnittpunkte des Kreises mit einem anderen Kreis bestimmt:

```
    def schnittpunkt_kreis(self, other) -> list:
        """
        Gibt in einer Liste die Schnittpunkte zurück, die dieser Kreis mit einem
        anderen Kreis hat
        Args:
        - other (Kreis): der andere Kreis
        """
        other_center = other.center
        if isclose(other.radius, self.radius):
            s = ((self.center[0] + other_center[0]) / 2,
                (self.center[1] + other_center[1])/2)
```

```
            b = distance(s, self.center)
        else:
            d = distance(self.center, other.center)
            if isclose(d, 0):
                return [] # Identischer Kreismittelpunkt
                          # -> keine oder unendlich viele Schnittpnkte
            to_acos = (other.radius**2 - self.radius**2 - d**2)/(-2*self.radius*d)
            if not -1 <= to_acos <= 1:
                return [] # -> keine Schnittpunkte
            beta = math.acos(to_acos)
            b = math.cos(beta) * self.radius
            s = (self.center[0] + (other_center[0]-self.center[0])*(b/d),
                 self.center[1] + (other_center[1]-self.center[1])*(b/d))
        if self.radius ** 2 - b ** 2 < 0:
            return [] # -> keine Schnittpunkte
        a = math.sqrt(self.radius ** 2 - b ** 2)

        if isclose(other_center[0], self.center[0]):
            orthogonal = 0
        else:
            steigung = (other_center[1]-self.center[1])/(other_center[0]-
                self.center[0])
            if isclose(other_center[1],self.center[1] ):
                return [Point(s[0], s[1]+a), Point(s[0], s[1]-a)]
            else:
                orthogonal = -1/steigung

        y_abschnitt =  -1 * s[0] * orthogonal + s[1]
        factor = math.sqrt(1 + orthogonal**2)
        solutions = []
        solutions.append(Point(s[0] + a/factor, y_abschnitt + (s[0] + a/factor) *
                        orthogonal))
        solutions.append(Point(s[0] - a/factor, y_abschnitt + (s[0] - a/factor) *
                        orthogonal))
        return solutions
```

Klasse, die die Kante eines Polygons repräsentiert (und alternativ auch verwendet werden kann, um eine Gerade zu repräsentieren):

```
class Edge:
    """
    Repräsentiert eine Kante eines Polygons
    Implement geometrisch-analytische Funktionen zum Umgang mit Geraden und Kanten
    sowie zur Interaktion mit anderen geometrischen Objekten (Kreisen und Punkten)
    """
    def __init__(self, corner1:Point, corner2:Point=None, *, m=None, c=None):
        """
        Kante definiert sich über Eckpunkte corner1 und corner2.
        Alternativ kann diese Klasse auch zum Repräsentieren einer Gerade verwendet
        werden, in diesem Fall wird die Steigung durch m und der y-Achsenabschnitt
        durch c angegebenen (y = mx + c) und es muss trotzdem mind. corner1 angegeben
        sein (damit im Fall m == unendlich trotzdem die Lage der Gerade rekonstruiert
        werden kann)
        """
        if corner2 is None:
            self.m = m
            self.c = c
            self.corner1 = corner1
```

```
            if self.m is None:
                self.corner2 = (corner1[0],corner1[1]+1)
            else:
                self.corner2 = (corner1[0]+1, self.m*(corner1[0]+1)+self.c)
        else:
            if corner1[0] > corner2[0]:
                # corner1 soll immer der Punkt mit der niedrigeren x-Koordinate sein,
                # dies erleichtert spätere Berechnungen
                helper = tuple(corner1)
                corner1 = tuple(corner2)
                corner2 = helper
            self.corner1 = corner1
            self.corner2 = corner2
            # Steigung und y-Abschnitt der Gerade (mx + c) bestimmen,
              auf der die Kante liegt:
            if corner1[0] == corner2[0]:
                # y-Achsenabschnitt und Steigung nicht bestimmbar,
                  da Kante parallel zur y-Achse
                self.m = None
                self.c = None
            else:

                self.m = ( self.corner2[1] - self.corner1[1] ) / (self.corner2[0] -
self.corner1[0]) # Steigung der Gerade, auf der die Kante liegt (mx + c)
                self.c = -1 * self.corner1[0] * self.m + self.corner1[1] # y-
Achsenabschnitt der Gerade, auf der die Kante liegt (mx + c)

        self.cached_length = None # Hier wird die Länge der Kante gecached, sobald sie
einmal angefordert wurde (zum Vermeiden mehrfacher Berechnungen)

        self.steigungswinkel_cache = None # Hier wird der Steigungswikel der Kante
gecached, sobald er einmal angefordert wurde (zum Vermeiden mehrfacher Berechnungen)
```

Funktion, die die Schnittpunkt der Kante mit einem Kreis bestimmt:

```
    def schnittpunkt_kreis(self, kreis, *, check_validity=True) -> list:
        """
        Bestimmt die Schnittpunkte der Polygonkante mit einem Kreis.
        Die Schnittpunkte werden in einer Liste zurückgegeben.
        Args:
            kreis (Kreis): Der andere Kreis
        """
        solutions = []

        if self.m is None:
            s = (self.corner1[0], kreis.center[1])
        elif math.isclose(self.m, 0):
            s = (kreis.center[0], self.corner1[1])
        else:
            other_m = -(1/self.m)
            other_c = -1 * kreis.center[0] * other_m + kreis.center[1]
            s = self.schnittpunkt_gerade(other_m, other_c, check_validity=False)

        a = distance(kreis.center, s)
        if kreis.radius**2 - a**2 < 0:
            return []
        b = float(math.sqrt(kreis.radius**2 - a**2))
```

```
        if self.m is None:
            solutions.append(Point(self.corner1[0], s[1] + b))
            solutions.append(Point(self.corner1[0], s[1] - b))
        else:
            factor = math.sqrt(1 + self.m**2)
            solutions.append(
                Point(s[0] + b/factor, (s[0] + b/factor) * self.m + self.c))
            solutions.append(
                Point(s[0] - b/factor, (s[0] - b/factor) * self.m + self.c))

        if check_validity:
            for solution in list(solutions):
                if self.corner1[0] <= solution[0] <= self.corner2[0]:
                    if min((self.corner1[1], self.corner2[1])) <= solution[1] <=
                    max((self.corner1[1], self.corner2[1])):
                        continue
                solutions.remove(solution)

        return solutions
```

Funktion, die die herausfindet, ob die Gerade edge 2 auf der Kante liegt:

```
    def is_identic_to(self, edge2):
        if edge2 is None:
            return False
        if self.m is None and edge2.m is None:
            return isclose(self.corner1[0], edge2.corner1[0])
        if self.m is None and edge2.m is not None or self.m is not None and edge2.m is
         None:
            return False
        if not isclose(self.m, edge2.m):
            return False
        return isclose(self.corner1[0] * edge2.m + edge2.c, self.corner1[1])
```

Klasse, die ein Polygon repräsentiert:

```
class Polygon:
    """
    Repräsentiert ein Polygon
    """
    def __init__(self, corners):
        if len(corners) > 3:
            # Sicherstellen, dass die Punkte des Polygons im Uhrzeigersinn vorliegen
              (Voraussetzung für die Triangulation)
            if sum([crossproduct2d(corners[i], corners[(i+1)%len(corners)]) for i in
            range(0,len(corners))]) > 0:
                corners.reverse()

        if len(corners) < 3:
            print("Polygon muss mindestens 3 Ecken haben.")
            exit()

        self.corners = corners
        self.edges = []
        for i in range(len(corners)):
            self.edges.append(Edge(self.get_corner(i), self.get_corner(i+1)))
        self.shapely_polygon = shapely_geometry.Polygon(self.corners)
        self.triangulation = None
```

```
        self.area = None
        self.circle_union_cache = None

    def get_corner(self, corner_index):
        """
        Gibt die Ecke am Index corner_index zurück. Die Indizierung erfolgt dabei
        zyklisch.
        """
        corner_index = corner_index % len(self.corners)
        return self.corners[corner_index]

    def get_edge(self, edge_index):
        """
        Gibt die Ecke am Index corner_index zurück. Die Indizierung erfolgt dabei
        zyklisch.
        """
        edge_index = edge_index % len(self.edges)
        return self.edges[edge_index]
```

Funktion, die das Polygon trianguliert (d.h. das Polygon wird vollständig in Dreiecke aufgespalten):

```
    def triangulate(self):
        """
        Gibt eine Triangulation des Polygons zurück
        Returns:
            list<Polygon>: Eine Liste, die n-2 Dreiecke enthält (mit n := Anzahl Ecken
            vom zu trianguilierenden Polygon)
        """
        if self.triangulation is None:
            polygon_to_divide, self.triangulation = deepcopy(self), []
            while len(polygon_to_divide.corners) > 3:
                for corner_id in range(len(polygon_to_divide.corners)):
                    c0, c1, c2 = polygon_to_divide.get_corner(corner_id-1),
                      polygon_to_divide.get_corner(corner_id),
                      polygon_to_divide.get_corner(corner_id+1)
                    v1, v2 = (c1[0]-c0[0], c1[1]-c0[1]), (c1[0]-c2[0], c1[1]-c2[1])
                    if crossproduct2d(v1, v2) > 0:
                        # -> Ecke konvex
                        triangle = Polygon([c0,c1,c2])
                        for c_check in polygon_to_divide.corners:
                            if c_check in [c0,c1,c2]:
                                continue
                            if triangle.point_in_polygon(c_check):
                                triangle = None
                                break
                        if triangle is not None:
                            self.triangulation.append(triangle)
                            corners = polygon_to_divide.corners
                            corners.remove(c1)
                            polygon_to_divide = Polygon(corners)
                            break
                else:
                    self.triangulation = [] # Triangulation fehlgeschlagen ->
                                            # abbrechen, um Endlosschleife zu vermeiden
                    return []
            self.triangulation.append(polygon_to_divide)
```

```
    return self.triangulation
```

Funktion, die über den Satz des Heron den Flächeninhalt des Polygons berechnet (Voraussetzung: Das Polygon muss dreieckig sein):

```
def triangle_area(self):
    if len(self.corners) == 3:
        s = 0.5*sum([edge.length() for edge in self.edges])
        to_sqrt = s*(
           s-self.edges[0].length())*(s-self.edges[1].length())*
           (s-self.edges[2].length()
        )
        if to_sqrt < 0:
            return 0
        return math.sqrt(to_sqrt)
```

Funktion, die bestimmt, ob sich ein Punkt im Polygon befindet oder nicht:

```
def point_in_polygon(self, point : Point):

    if len(self.corners) == 3:
        if min([self.corners[0][0], self.corners[1][0], self.corners[2][0]])-1 <=
               point[0] <= max([self.corners[0][0], self.corners[1][0],
               self.corners[2][0]])+1:
            if min([self.corners[0][1], self.corners[1][1], self.corners[2][1]])-1
             <= point[1] <= max([self.corners[0][1], self.corners[1][1],
             self.corners[2][1]])+1:
                  big_area = self.triangle_area()
                  small_triangles = [Polygon([point, self.get_corner(i),
                                  self.get_corner(i+1)]) for i in range(3)]
                  small_area = sum([triangle.triangle_area() for triangle in
                                    small_triangles])
                  if isclose(big_area, small_area):
                      return True
        return False

    else:

        triangulation = self.triangulate()
        for triangle in triangulation:
            if triangle.point_in_polygon(point):
                return True
        return False
```

Funktion, die mithilfe von der Bibliothek shapely die Schnittfläche zwischen dem Kreis und dem Polygon bestimmt
(von der Schnittfläche werden die sich in Polygon.circle_union befinden Kreise abgezogen, wenn exclude_union wahr ist):

```
def exact_cut_area(self, circle : Kreis, *, exclude_union=False, intersect_with =
    None, radius_reduction=0) -> float:
    # Define polygon and circle
    shapely_circle =
        shapely_geometry.Point(tuple(circle.center)).buffer(circle.radius-
        radius_reduction)
```

```
        # Get the intersection
        initial_intersection = self.shapely_polygon.intersection(shapely_circle)
        if initial_intersection.is_empty:
            return 0

        if intersect_with is not None:

            initial_intersection =
initial_intersection.intersection(shapely_geometry.Point(tuple(intersect_with.center)).
buffer(intersect_with.radius))

            if initial_intersection.is_empty:
                return 0

        circle_union = self.circle_union_cache

        if circle_union is None or exclude_union is False:
            return initial_intersection.area
        intersection = initial_intersection.intersection(circle_union)
        if intersection.is_empty:
            return initial_intersection.area
        return initial_intersection.area - intersection.area

    def add_to_circle_union(self, object):

        if self.circle_union_cache is None:
            self.circle_union_cache =
shapely_geometry.Point(tuple(object.center)).buffer(object.radius)

        else:
            self.circle_union_cache =
self.circle_union_cache.union(shapely_geometry.Point(tuple(object.center)).buffer(objec
t.radius))
```

Node-Klasse

Klasse, die einen im Polygon platzierten Ort mit allen an den Ort gebundenen Informationen repräsentiert:

```
class Node(Kreis):
    def __init__(self, center, *, in_ghz=False, origin="", origin_edge=None):
        super().__init__(center, min_distance_ghz if in_ghz else min_distance)
        self.in_ghz = in_ghz
        self.closest_neighbors = {}
        # --- Diese Attribute hat die Node-Klasse nur in aufgabe2_H2.py --- #
        self.origin = origin # Hier werden Informationen über den Ursprung des Orts
                               gespeichert
        self.origin_edge = origin_edge

    def is_in_ghz_range(self, point):
        """
        Gibt zurück, ob sich der Punkt point im Einzugsgebiet des Gesundheitszentrums
befindet
        """
        if self.ghz is None:
            return True
        d = distance(point, self.ghz)
```

```
        return d < secure_distance_ghz or isclose(d, secure_distance_ghz)
```

Funktionen, die an der Platzierung und Optimierung des Gesundheitszentrums beteiligt sind

Funktion, die das zuvor von find_ghz() gefundene Gesundheitszentrums-Position implementiert:

```
def implement_ghz(self, ghz):
    """
    self.ghz wird auf die Position des Gesundheitszentrums gesetzt und die
    Schnittpunkte des Ghz-Einzugsgebiets mit den Polygonkanten werden ermittelt
    Args:
        ghz (tuple oder Point): Position des Gesundheitszentrums
    """
    self.ghz = ghz
    self.ghz_intersection_points = []
    for edge in self.edges:
        intersections = edge.schnittpunkt_kreis(Kreis(tuple(self.ghz),
         secure_distance_ghz))
        for s in intersections:
            self.ghz_intersection_points.append(tuple(s))
```

Funktion, die die Schnittpunkte des Ghz-Kreises mit den Polygonkanten zu Solution.nodes_to_add hinzufügt:

```
def implement_ghz_intersections(self):
    """
    Die Schnittpunkte des Ghz-Einzugsgebiets mit den Polygonkanten werden zur
    nodes_to_add Liste hinzugefügt, wenn sie nicht durch andere bereits platzierte
    Orte blockiert werden
    Dieser Schritt erfolgt erst jetzt, da nach der Implementierung des
    Gesundheitszentrums noch Orte hinzugefügt werden, die möglicherweise die
    Schnittpunkte als mögliche Orte blockieren
    """
    for point in self.ghz_intersection_points:
        new_node = Node(point, in_ghz=True)
        if not self.is_blocked(new_node):
            self.nodes_to_add.append(new_node)
```

Funktion, die basierend auf der aktuellen Platzierung der Orte eine initiale Ghz-Position findet:

```
def find_ghz(self):
    # Nutzt die aktuelle Ortskonfiguration zum Finden einer initialen Ghz-Position:
    Gibt den Ort zurück, von dem aus die meisten Nachbarorte im Ghz-
    Sicherheitsabstand liegen
    best_node = None
    best_score = -1
    for node in self.nodes:
        # Iteriert über alle im Polygon platzierten Punkte, um den zu finden, der
        (wenn dort ein Ghz platziert wird) am meisten andere Punkte im
         secure_distance_ghz-Radius abdeckt
        score = len(list(filter(lambda x : node.closest_neighbors[tuple(x)] <=
         secure_distance_ghz, list(node.closest_neighbors.keys()))))
        if score > best_score:
            best_score = score
```

```
                best_node = tuple(node.center)
        self.ghz_score = best_score
        return best_node
```

Funktion, die die von Solution.find_ghz() gefundene Ghz-Position verbessert:

```
    def optimize_ghz(self, ghz_init_pos):
        # Optimiert die zuvor mit self.find_ghz() gefundene initiale Position des Ghzs,
        # indem alle Punkte, die im Kreis mit dem Radius min_distance_ghz um das
initiale Ghz liegen, mit einer Genauigkeit von 1 (Diskretisierungswert) durchprobiert
werden
        # Der Punkte, bei dem ein Kreis um den Punkte mit dem Radius
secure_distance_ghz die größte Schnittfläche mit dem Polygon hat, wird ausgewählt
        best_node = None
        best_score = 0
        step = 1
        radius = min_distance_ghz
        # Kreis diskretisieren
        x = ghz_init_pos[0] - radius
        for i1 in range(radius*2):
            # In jeder Iteration dieser Schleife wird der x Wert um step erhöht
            # und es wird über alle für diesen x-Wert im Kreis liegenden y-Werte
iteriert
            x += step
            to_sqrt = 1-((x-ghz_init_pos[0])/radius)**2
            if to_sqrt < 0:
                y_max = 0
            else:
                y_max = math.sqrt(to_sqrt) * radius # y-Bereich an im Kreis liegenden
Punkten an diesem x-Wert finden

            y = ghz_init_pos[1]
            while y < ghz_init_pos[1] + y_max:
                cut_area = self.exact_cut_area(Kreis((x,y), secure_distance_ghz))
                if cut_area > best_score:
                    if self.point_in_polygon((x,y)):
                        best_node = (x,y)
                        best_score = float(cut_area)
                y += step

            y = ghz_init_pos[1]-step
            while y > ghz_init_pos[1] - y_max:
                cut_area = self.exact_cut_area(Kreis((x,y), secure_distance_ghz))
                if cut_area > best_score:
                    if self.point_in_polygon((x,y)):
                        best_node = (x,y)
                        best_score = float(cut_area)
                y -= step

        m = (sum([n[0] for n in self.nodes])/len(self.nodes), sum([n[1] for n in
self.nodes])/len(self.nodes))
        cut_area = self.exact_cut_area(Kreis(m, secure_distance_ghz))
        if cut_area > best_score:
            if self.point_in_polygon(m):
                best_node = m
                best_score = float(cut_area)
        self.ghz_score = best_score # "Score" bzw. Schnittflächenanteil speichern
        if best_node is None:
```

```
            return ghz_init_pos
        else:
            return best_node
```

Funktion, die die Position des Gesundheitszentrums an die Kanten anpasst:

```
    def align_ghz(self, ghz_init_pos):
        # Passt die Position des Ghz so an, dass Eckpunkte, die in einer 60°- oder 90°-
          Ecke sind, leichter in die Dreiecksstruktur der Befüllung miteinbezogen
          werden können
        new_ghz_positions = []
        new_ghz_position_nodes = []
        for corner_id in range(len(self.corners)):
            angle = self.get_corner_angle(corner_id)
            if angle == math.pi/2 or angle == math.pi/3:
                if not self.is_in_ghz_range(self.corners[corner_id]):
                    edge1 = self.get_edge(corner_id-1)
                    edge2 = self.get_edge(corner_id)
                    s1 = edge1.schnittpunkt_kreis(Kreis(self.ghz, secure_distance_ghz))
                    s2 = edge2.schnittpunkt_kreis(Kreis(self.ghz, secure_distance_ghz))
                    if s1 == [] or s2 == []:
                        continue
                    else:
                        i1 = min(s1, key=lambda x :
distance(self.get_corner(corner_id),x))
                        i2 = min(s2, key=lambda x :
distance(self.get_corner(corner_id),x))
                    d1, d2 = distance(i1, self.get_corner(corner_id)), distance(i2,
self.get_corner(corner_id))
                    reference_distance = min_distance_ghz/2 if angle == math.pi/2 else
min_distance_ghz
                    new_d1 = round(d1/reference_distance)*reference_distance
                    new_d2 = round(d2/reference_distance)*reference_distance
                    new_i1 = edge1.schnittpunkt_kreis(Kreis(self.get_corner(corner_id),
new_d1), check_validity=True)
                    if new_i1 == []:
                        continue
                    else:
                        new_i1 = new_i1[0]
                    new_i2 = edge2.schnittpunkt_kreis(Kreis(self.get_corner(corner_id),
new_d2), check_validity=True)
                    if new_i2 == []:
                        continue
                    else:
                        new_i2 = new_i2[0]
                    possible_ghz_positions = Kreis(new_i1,
secure_distance_ghz).schnittpunkt_kreis(Kreis(new_i2, secure_distance_ghz))
                    for ghz_position in possible_ghz_positions:
                        if not self.is_in_ghz_range(ghz_position):
                            continue
                        new_score = self.exact_cut_area(Kreis(ghz_position,
secure_distance_ghz))
                        if new_d1 == min_distance_ghz and new_d2 == min_distance_ghz:
                            new_score += Polygon([new_i1, new_i2,
self.corners[corner_id]]).triangle_area()
                        if new_score >= self.ghz_score:
                            self.implement_ghz(ghz_position)
```

```
                        if new_d1 == min_distance_ghz and new_d2 ==
                         min_distance_ghz:
                            new_ghz_positions.append(ghz_position)
                            new_ghz_position_nodes.append(self.corners[corner_id])
                        break
                else:
                    continue

    return new_ghz_positions, new_ghz_position_nodes
```

Heuristikfunktionen und Funktionen, die das Polygon basierend auf diesen Heuristiken befüllen

Funktion, die die sich in Solution.nodes_to_add befindenden Punkte aktualisieren bzw. neue Punkte hinzufügen:

```
def successors(self, *, ignore_circle_circle_intersections=False):
    """
    Schnittpunkte vom letzten hinzugefügten Ortsumkreis mit den bereits
    existierenden Ortsumkreisen und den Begrenzungsflächenkanten finden
    """
    if self.last_added is not None:
        # Schnittpunkte mit den Ortsumkreisen der bereits existierenden Orte finden
        if not ignore_circle_circle_intersections:
            for node in self.nodes:
                if node.center == self.last_added.center:
                    continue
                result = self.last_added.schnittpunkt_kreis(node) # Schnittpunkte
                                mit dem Ortsumkreis von node ermitteln
                for point in result:
                    if self.point_in_polygon(point): # Überprüfen, ob der
                                        Schnittpunkt im Polygon liegt
                        in_ghz = self.is_in_ghz_range(point)
                        if (not in_ghz and self.fill_ghz):
                            self.nodes_to_add_outside_ghz.append(Node(point,
                            in_ghz=in_ghz))
                        else:
                            new_node = Node(point, in_ghz=in_ghz)
                            if (not self.is_blocked(new_node)) and (not new_node in
                            self.nodes_to_add):
                                self.nodes_to_add.append(new_node)

        # Schnittpunkte mit den Kanten finden
        for edge in self.edges:
            result = edge.schnittpunkt_kreis(self.last_added, check_validity=False)
            for point in result:
                if self.point_in_polygon(point):
                    # Überprüfen, ob der Schnittpunkt im Polygon liegt
                    in_ghz = self.is_in_ghz_range(point)
                    new_node = Node(point, in_ghz=in_ghz)
                    if (not in_ghz and self.fill_ghz):
                        self.nodes_to_add_outside_ghz.append(new_node)
                    elif not new_node in self.nodes_to_add:
                        if not self.is_blocked(new_node):
                            self.nodes_to_add.append(new_node)
```

```
            # Falls der Bereich des Ghz befüllt wird: Schnittpunkte mit dem Ghz-
              Einzugesbereich-Umfang finden
            if not ignore_circle_circle_intersections:
                if self.fill_ghz and self.ghz is not None:
                    result = self.last_added.schnittpunkt_kreis(Kreis(self.ghz,
                       secure_distance_ghz))
                    for point in result:
                        if self.point_in_polygon(point): # Überprüfen, ob der
                                                  Schnittpunkt im Polygon liegt

                            new_node = Node(point, in_ghz=self.is_in_ghz_range(point))

                            if not self.is_blocked(new_node) and (not (not
                            new_node.in_ghz and self.fill_ghz)) and not (new_node in
                            self.nodes_to_add):

                                self.nodes_to_add.append(new_node)


    def update_nodes_to_add(self):

        # Überprüft für alle Punkte aus nodes_to_add, ob sie nach Hinzufügen von
        self.last_added noch hinzugefügt werden können
        if not self.last_added is None:
            for point in list(self.nodes_to_add):
                if max(abs(point[0] - self.last_added[0]), abs(point[1] -
                self.last_added[1])) > point.radius + self.last_added.radius:
                    continue
                d = distance(point, self.last_added)
                use_ghz_distance = self.last_added.in_ghz or point.in_ghz
                if not (d > (min_distance_ghz if use_ghz_distance else min_distance) or
                isclose(d, min_distance_ghz if use_ghz_distance else min_distance)):
                    self.nodes_to_add.remove(point)

    def add_corners_to_nodes_to_add(self):

        # Fügt die Ecken des Polygons als potentiell hinzufügbare Orte zu
          self.nodes_to_add hinzu, wenn sie im gerade zu befüllenden Bereich liegen und
          nicht durch bereits platzierte Orte blockiert werden

        for corner in self.corners:
            new_node = Node(corner, in_ghz=self.is_in_ghz_range(corner))
            if self.fill_ghz and self.ghz is not None:
                if not new_node.in_ghz:
                    continue
            if not self.is_blocked(new_node):
                self.nodes_to_add.append(new_node)
```

Funktion, die Punkte hinzufügen:

```
    def add_definite_nodes(self):
        """
        Fügt Eckenpunkte hinzu, die sich an spitzen Innenwinkeln <60° befinden und in
dem durch die Platzierung wegfallenden Bereich keine Schnittpunkte mit anderen Kanten
als den aus dem Scheitelpunkt hervorgehenden Kanten haben.
        Dieser Schritt kann erst jetzt erfolgen, da die Platzierung des
Gesundheitszentrums die Größe des bei der PLatzierung wegfallenden Bereichs beeinflusst
(wegen min_distance != min_distance_ghz)
```

```
        """
        for corner_id in range(len(self.corners)):
            # Hinzufügen von gesichert zu platzierenden Orten
            if self.get_corner_angle(corner_id) <= math.pi * 1/3:
                new_node = Node(self.corners[corner_id],
                in_ghz=self.is_in_ghz_range(self.corners[corner_id]))
                # Überprüfen, ob im wegfallenden Bereich andere Ecken sind
                contains_corner = False
                for corner2 in self.corners:
                    if corner2 == self.corners[corner_id]:
                        continue
                    d = distance(corner2, self.corners[corner_id])
                    if d < new_node.radius:
                        contains_corner = True
                        break
                if contains_corner:
                    continue
                # Überprüfen, ob der wegfallende Bereich andere Ecken schneidet
                num_intersecting_edges = 0
                for edge in self.edges:
                    result = edge.schnittpunkt_kreis(new_node, check_validity=True)
                    for r in list(result):
                        if not distance(r,new_node.center) < new_node.radius:
                            result.remove(r)
                    if len(result) > 0:
                        num_intersecting_edges += 1
                if num_intersecting_edges > 2:
                    continue
            else:
                continue
            self.add_node(new_node)
            self.last_added = new_node
            self.successors(ignore_circle_circle_intersections=True)

    def add_node(self,new_node):
        """
        Fügt den Ort new_node hinzu und berechnet die Distanzen
        """
        for node in self.nodes:
            d = distance(new_node, node)
            new_node.closest_neighbors[tuple(node)] = d
            node.closest_neighbors[tuple(new_node)] = d
        self.nodes.append(new_node)
        if new_node in self.nodes_to_add:
            self.nodes_to_add.remove(new_node)
        self.last_added = new_node
        self.update_nodes_to_add()
```

Heuristiken:

```
    def fill_area_heuristic(self):
        """
        Befüllt einen Bereich (entweder das ganze Polyon oder nur den Bereich um das
        Gesundheitszentrum herum) basierend auf der Flächenheuristik
        """
        self.circle_union_cache = None
        for node in self.nodes:
            self.add_to_circle_union(node)
```

```
        while self.nodes_to_add != []:
            next_node = min(self.nodes_to_add, key=self.area_heuristic)
            self.add_node(next_node)
            self.add_to_circle_union(next_node)
            self.successors()

    def area_heuristic(self, x):
        """
        Gibt den Flächenanteil der durch Ort x wegfallenden Polygonfläche an der durch
        Ort x wegfallenden Gesamtfläche zurück
        """
        if tuple(x) in self.area_heuristic_cache and (not self.last_added is None):
            if distance(self.last_added, x) < 2*(min_distance_ghz if x.in_ghz else
min_distance):

                self.area_heuristic_cache[tuple(x)] = self.exact_cut_area(x,
exclude_union=True, intersect_with=Kreis(self.ghz, secure_distance_ghz) if
(self.fill_ghz and self.ghz is not None) else None, radius_reduction=0.1) / (math.pi *
(x.radius**2))

        else:

            self.area_heuristic_cache[tuple(x)] = self.exact_cut_area(x,
exclude_union=True, intersect_with=Kreis(self.ghz, secure_distance_ghz) if
(self.fill_ghz and self.ghz is not None) else None, radius_reduction=0.1) / (math.pi *
(x.radius**2))

        return self.area_heuristic_cache[tuple(x)]
```

Es existieren noch weitere den Gruppen zuordnebare Funktionen, die aber weggelassen werden, damit das Dokument nicht zu lang wird.