Aufgabe 3: Die Siedler

Máté Tirpák

Teilnahme-ID: 71669

15. April 2024

Inhaltsverzeichnis

1. Lösungsidee

1.1 Problembeschreibung und Intuition

In dem beschriebenen Problem geht es darum, möglichst viele Siedlungen in ein Polygon, gegeben den Eckpunkten, zu platzieren. Diese müssen im Regelfall einen Abstand von 20km zueinander einhalten, wobei ein medizinisches Zentrum, ebenfalls im Polygon liegend, mit einem Radius von 85km, innenliegenden Siedlungen einen gegenseitigen Abstand von 10km erlaubt. Dieser Abstand von 10km gilt ebenfalls für zwei Siedlungen, wenn sich eine der beiden innerhalb des medizinischen Zentrums befindet.

Dementsprechend lässt sich das Problem in folgende zwei Teilprobleme unterteilen, die im weiteren Verlauf weiter differenziert werden sollen:

Es muss ein Muster für die optimale Aufstellung der Siedlungen gefunden werden, welches sowohl den verschwendeten Platz minimiert als auch einen flüssigen Übergang zwischen dem inneren und äußeren Bereich des medizinischen Zentrums erlaubt.

Die Position des medizinischen Zentrums ist ausschlaggebend für die Maximierung der Siedlungsanzahl, da in seinem Radius eine besondere Menge an Siedlungen platziert werden können. Folglich ist es essenziell eine Position zu finden, bei der möglichst viele Siedlungen Platz in diesem Radius finden.

1.2 Reduktion von NP-Hard Schwere

Die Reduktion soll mithilfe des als NP-schwer anerkannten „k-center problem“ [3] durchgeführt werden. Es beschreibt eine bestimmte Anzahl Städte n, wobei zwischen diesen ein vorgeschriebener Mindestabstand vorliegen muss. Aus diesen n Städten sollen k zu Centren transformiert werden, mit dem Ziel den größten Abstand zwischen einer Stadt und dem naheliegendsten Center zu minimieren.

Das Problem der Siedler weist entscheidende Übereinstimmungen auf. Es soll zunächst nur das Gebiet eingeschlossen vom Gesundheitszentrum betrachtet werden. Die Städte des „k-center problem“ entsprechen dabei den Siedlungen und k ließe sich zur Konstante 1 umformen, da genau 1 Gesundheitszentrum existiert. Der Mindestabstand zwischen den Siedlungen entspricht dabei 10km. Abgesehen von der Tatsache, dass im Problem der Siedler alle Siedlungen in einem Polygon liegen müssen, was eine nennenswerte Erschwerung darstellt, unterscheiden sich noch die Ziele der beiden Probleme. So müssen bei den Siedlern die Anzahl Siedlungen maximiert und bei k-centers der größte Abstand zwischen Stadt und Centrum minimiert werden. Dieses Ziel ist dennoch auch bei den Siedlern präsent. So soll der Abstand der Siedlungen zum Gesundheitszentrum generell minimiert werden, sodass möglichst viele Siedlungen in diesen Bereich eingefügt werden können.

Zusammenfassend können die Städte des „k-center problem“ zu den Siedlungen des Siedler Problems übersetzt werden, der minimale Abstand wäre übertragbar und würde 10km übertragen und die Anzahl k Center wäre 1 für das eine Gesundheitszentrum. Das Ziel, den größten Abstand zum Zentrum zu minimieren ist gleichzeitig auch ein Ziel der Siedler, damit möglichst viele Siedlungen vom geringeren Abstand profitieren.

Mit dem Gedanken, das darüber hinaus die Fläche als Polygon betrachtet werden muss und außerhalb des Zentrums weitere Siedlungen mit einem neuen Mindestabstand bestehen, ließe sich sagen, dass „Die Siedler“ aufgrund des Fakts, dass „k-center“ NP-hard ist, ebenfalls NP-hard ist.

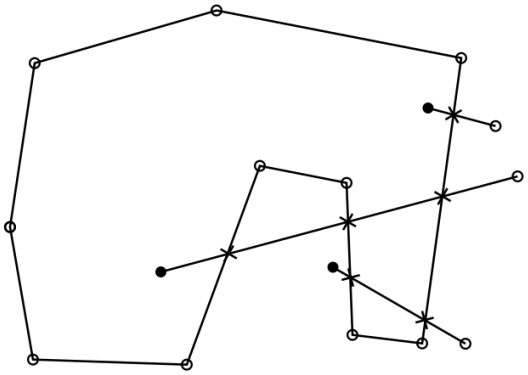
1.3 Allgemeine Funktionalitäten

1.3.1 Erstellen der Seiten

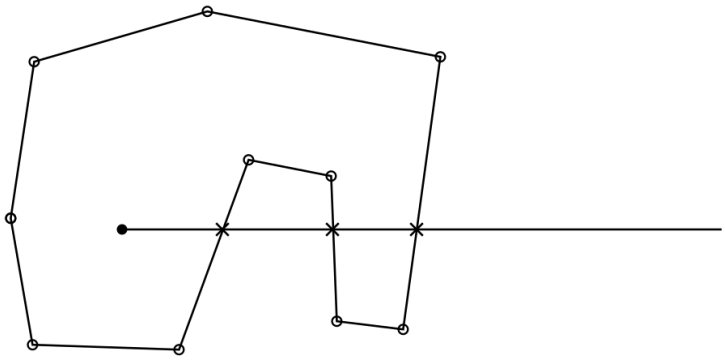
Ausgehend aus den Eckpunkten des Polygons sollen Seiten erstellt werden, indem durch die Eckpunkte iteriert und diese jeweils mit dem nächsten Eckpunkt zusammengefasst werden sollen. Die Betrachtung des Polygons in Seiten statt Eckpunkten bietet eine vereinfachte Darstellung für die Lösung komplexerer Probleme.

1.3.2 Punkt im Polygon

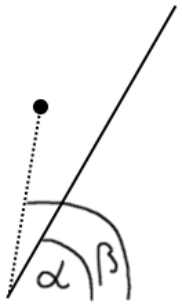
Ein Punkt liegt genau dann im Polygon, wenn eine Linie zu einem sicher außenliegenden Punkt eine ungerade Anzahl Seiten des Polygons schneidet. Jeder Schnitt beschreibt einen Aus- bzw. Eintritt bezüglich des Polygons, weshalb eine ungerade Anzahl einen Austritt beschreibt, woraus sich schlussfolgern lässt, dass der potenziell im Polygon liegende Punkt tatsächlich in diesem liegt.



Dabei lässt sich dieses Problem zusätzlich mit Ray-Casting [1] vereinfachen. So existiert der außenliegende Punkt nur imaginär und befindet sich auf gleicher y-Koordinate bei unendlich großem x-Wert.



Es wird ein potenzieller Schnitt mit jeder Seite untersucht. Wenn der Punkt bezüglich beider Punkte der Seite oberhalb, unterhalb oder rechts liegt, existiert sicher kein Schnitt. Anderenfalls muss der Winkel der Verbindungslinie zwischen dem unteren Punkt der Seite und dem zu untersuchenden Punkt, und der Winkel der Seite miteinander verglichen werden.



Da der Winkel der Verbindungslinie Beta größer als der der Seite Alpha ist, liegt ein Schnitt vor. Bei kleinerem Beta würde der Punkt leicht rechts versetzt von der Seite liegen und somit keinen Schnitt vorweisen.

Das gleiche Verfahren gilt auch für negative Werte des Winkels, etwa wenn die Seiten nicht nach rechts, sondern nach links kippen. Auch hier wird der Betrag des Winkels größer, je näher die x-Koordinate des Variablen Punktes der betrachteten Seite/Verbindungslinie an dem x-Wert des festen Punktes liegt. Ein größerer Betrag im negativen Raum symbolisiert allerdings ein Sinken des Wertes, weshalb sich auch hier ein größerer Winkel Beta als ein Schnitt deuten lässt.

1.3.3 Punkt in Reichweite des Gesundheitszentrums

Um für den weiteren Verlauf analysieren zu können, ob ein Eckpunkt des Polygons im Gesundheitszentrum liegt, muss die Entfernung dieses zum Ursprung des Zentrums betrachtet werden. Da der Radius des medizinischen Zentrums 85km beträgt, muss die euklidische Distanz zwischen Punkt und Mittelpunkt kleiner als 85km sein. Er berechnet sich wie folgt mithilfe des Pythagoras:

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

1.3.4 Rotation von Punkten

Für die Optimierung der Siedlungspositionen ist es essenziell, diese um einen weiteren Punkt, etwa das Gesundheitszentrum, rotieren zu können. Folgende Formel gilt für die Rotation von Punkten um den Ursprung. Daher müssen die Koordinaten des festen Punktes vom zu untersuchenden Punkt subtrahiert werden, sodass eine Verschiebung zum Ursprung simuliert wird. Anschließend gilt Rotation des Punktes (x, y):

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & +\cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Schlussendlich müssen die Koordinaten des festen Punktes zu x’ und y‘ hinzuaddiert werden müssen, um die Verschiebung zum Ursprung rückgängig zu machen.

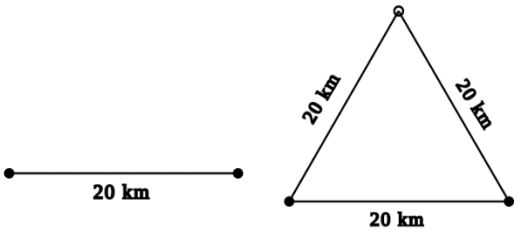
Zusätzlich zur Rotation soll auch der bereits bestehende Winkel eines Punktes im Vergleich zu einem anderen errechnet werden können. Es wird dabei der Einheitskreis betrachtet und beide Punkte so verschoben, dass der nicht zu untersuchende im Ursprung liegt. Anschließend wird der Arkustangens(y/x) mit den Koordinaten des zu untersuchenden Punktes berechnet. Die Werte von y/x des 2. und 4. Quadranten sind dabei negativ, weshalb im 2. zum Winkel π und im 4. 2π hinzuaddiert

werden müssen. Die Ergebnisse y/x der Punkte des 3. Quadranten sind zwar positiv, allerdings ist der Winkel um π kleiner als erwartet, weshalb wie auch im 2. Quadranten π addiert werden muss.

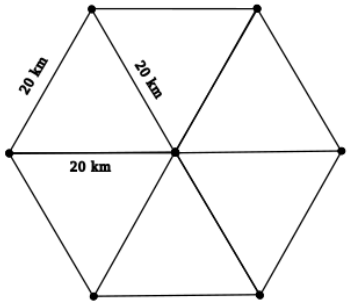
1.4 Aufstellung der Siedlungen

1.4.1 Muster

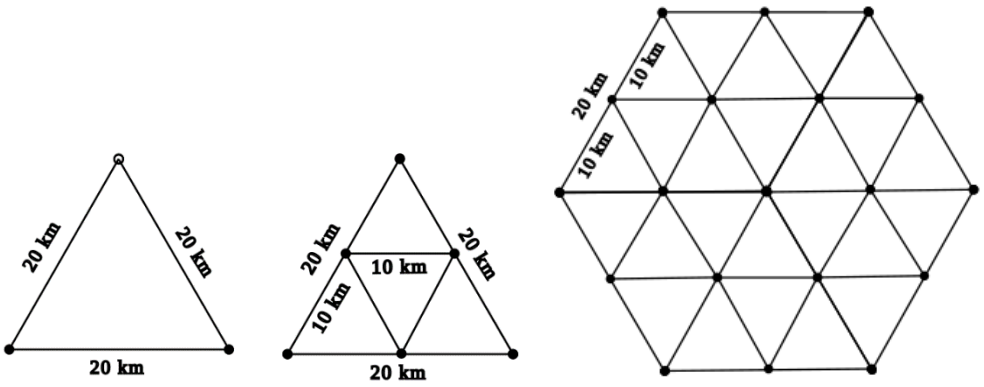
Betrachtet werden die Siedlungen, für die ein Abstand von 20km gilt. Werden zwei von diesen in diesem Abstand voneinander platziert und die Position für eine dritte Siedlung gesucht, die möglichst nahe zu den beiden Bestehenden liegt, bildet sich ein gleichseitiges Dreieck mit der Seitenlänge 20km.



Mit jedem Positionieren einer Siedlung neben zwei bereits bestehenden, die eine Seite eines Dreiecks bilden, entsteht durch diese drei Siedlungen ein weiteres gleichseitiges Dreieck, sodass ein beliebig erweiterbares Muster vorliegt.



Diese Form erlaubt zudem einen flüssigen Übergang zwischen einer Siedlungs-Distanz von 20 und 10km, indem jedes Dreieck der Seitenlänge 20km in 4 gleichseitige Dreiecke der Seitenlänge 10km unterteilt werden kann.

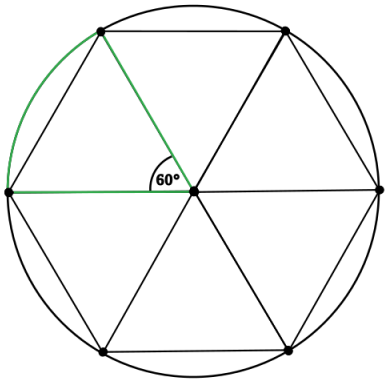


Damit möglichst viele Siedlungen in den Radius des medizinischen Zentrums hineinpassen, ist der Ursprung des Musters äquivalent zur Position des Zentrums. Folglich besteht eine feste Aufstellung der Siedlungen innerhalb und außerhalb der Reichweite des Gesundheitszentrums. Dabei ist die Länge des Musters zu definieren, um zu verhindern, dass dieses weitgreifender als nötig ausgebaut wird

Das Muster soll dabei diagonal aufgebaut werden.

1.4.2 Optimierung

Um möglichst viele mögliche Aufstellungen zu berücksichtigen, können die Siedlungen entlang ihres Ursprungs rotiert werden. Da das Muster punktsymmetrisch in Abständen von 60° bzw. π/3 ist, reicht es nur diesen Wertebereich zu betrachten.



Mit einer bestimmten variablen Genauigkeit sollen alle Siedlungen von 0° bis 60° immer gleichzeitig rotiert werden, um anschließend auszuwerten, wie viele Siedlungen bei dem zugehörigen Winkel im Polygon liegen.

1.4.3 Herausfiltern der korrekten Siedlungen

Ausgehend aus der optimierten Aufstellung der Siedlungen müssen diese durchgegangen und die außerhalb des Polygons liegenden entfernt werden. Deren Anzahl beschreibt dabei das Endergebnis der Problemfrage.

1.5 Position des medizinischen Zentrums

1.5.1 Heuristic

Um die bestmögliche Position zu finden, muss eine Heuristic existieren, die jeder Position einen eindeutigen Wert zuordnen kann. Mithilfe dieser lassen sich verschiedene Positionen vergleichen und die optimale herausfiltern. Dabei soll das Gesundheitszentrum als Kreis mit einem Radius von 85km betrachtet werden. Es erschließen sich dabei zwei Optionen:

Option 1 wäre eine Annäherung, bei der die maximale Anzahl Siedlungen in den Kreis platziert und anschließend die Anzahl der Siedlungen, die tatsächlich im Polygon liegen, als Wert festgelegt werden.

Option 2 wäre die genaue Berechnung der Schnittfläche des Kreises und Polygons.

Beide Optionen sollen in Erwägung gezogen und im folgenden Verlauf miteinander verglichen werden.

1.5.1.1 Annäherung durch Siedlungen

Es soll die maximale Anzahl Siedlungen in dem vorbestimmten Muster in den Kreis platziert werden. Anschließend wird die Rotation optimiert, sodass ein realistischeres Bild über das Potenzial der Position gewonnen werden kann. Diese Annäherung des Potenzials erscheint vereinfacht, ist allerdings die realistischste Option, da nicht von der übergestellten Problemstellung abgewichen wird, etwa im

Gegensatz zum Berechnen der Schnittfläche, welches keine Auskunft über die Platzierung der Siedlungen bietet.

1.5.1.2 Berechnung der Schnittfläche

Mithilfe der Schnittfläche von Kreis und Polygon wird ein genauer Überblick über das räumliche Potenzial zum späteren Platzieren der Siedlungen gewonnen, daher soll es trotz leichter Abweichung von der übergeordneten Problemstellung zum Vergleich in Erwägung gezogen werden. Durch das Platzieren von zusätzlichen Punkten soll die Schnittfläche zu einem concave Polygon reduziert werden, sodass Triangulation zur Berechnung der Fläche anwendbar wird.

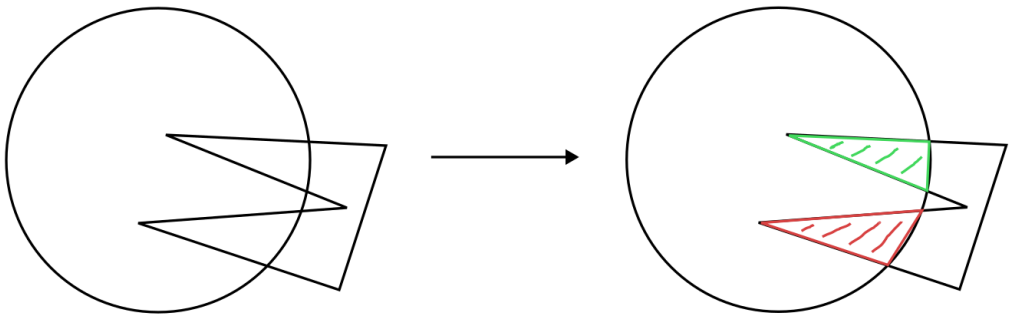
1.5.1.2.1 Berechnung der Fläche eines Polygons

Durch Triangulation soll das Polygon auf Dreiecke aufgeteilt werden, sodass anschließend deren Flächeninhalte berechnet werden können. Dazu wird Ear Clipping genutzt [2]. Jedem Eckpunkt des Polygons wird ein Status, reflex, convex, ear, zugeordet. Reflex beschreibt einen Innenwinkel größer als 180° während convex kleiner gleich 180° beschreibt. Convexe Punkte können dabei auch ein ear sein, wenn sie mit ihren Nachbarpunkten ein Dreieck bilden, indem kein anderer Punkt des Polygons liegt. Diese ears können anschließend ohne Auswirkung auf das restliche Konstrukt entfernt werden, wobei die Nachbarpunkte ebenfalls gespeichert werden, sodass ein Dreieck gebildet wurde. Anschließend muss der Status der Nachbarpunkte aufgrund der Änderung neu klassifiziert werden. Das Resultat ist eine Liste Dreiecke, die in ihrer Gesamtheit das Polygon repräsentieren. Deren Flächeninhalte werden mit folgender Formel berechnet und zusammenaddiert, sodass als Ergebnis der Flächeninhalt des Polygons bleibt.

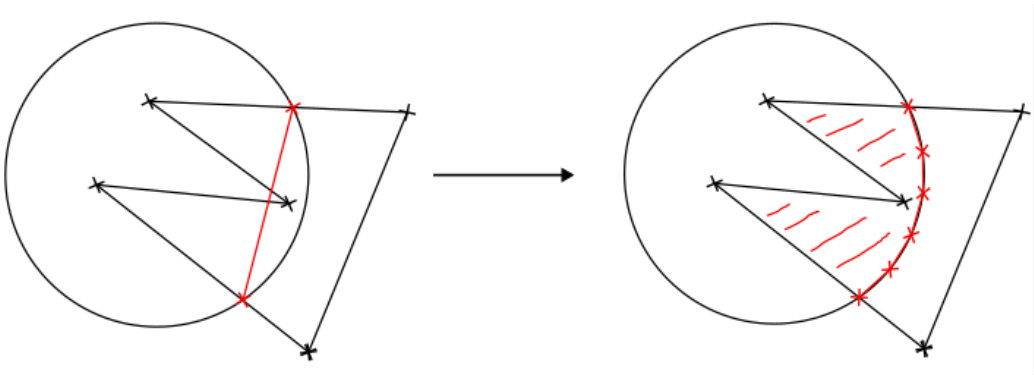
$$A = 0.5 * abs((x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2)))$$

1.5.1.2.2 Aufteilen der Schnittfläche in Polygone

Aus der Schnittfläche sollen einzelne unabhängige Abschnitte separiert und zu eigenen Polygons zusammengefasst werden.



Der Abschnitt zwischen zwei Kreisschnittpunkten soll dabei durch zusätzliche Punkte gefüllt werden, sodass die Berechnung durch Triangulation sowohl möglichst genau wird als auch nicht eine große Fläche auslässt, sodass beim Eintreten des Polygons in diese eine Überkreuzung der Seiten stattfinden würde.



Zuerst sollen die Schnittpunkte von Kreis und Polygon berechnet werden, wobei jeder der vielen Schnittpunkte genau einen anliegenden Abschnitt zum nächsten Schnittpunkt besitzt, welcher im Polygon und der andere außerhalb des Polygons liegt. Dabei bilden zwei benachbarte Schnittpunkte ein Intervall, welches, wenn es zu den im Polygon liegenden zählt, mit einer hohen Anzahl in Reihe gelegenen zusätzlichen Punkten gefüllt werden soll.

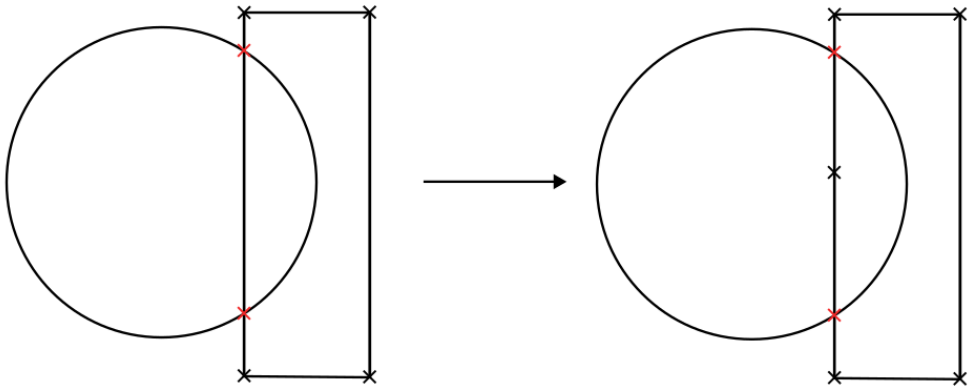
Um die Schnittpunkte zu berechnen, wird mithilfe der „sympy“ library der Kreis als folgende Funktion modelliert und jede Seite des Polygons auf einen möglichen Schnitt untersucht.

$$85^2 = (x - \text{circle\_x})^2+(y - \text{circle\_y})^2$$

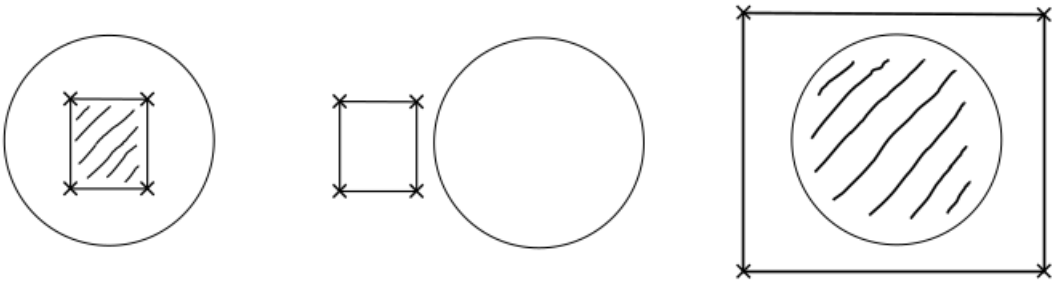
Neben der Speicherung der Schnittpunkte selbst wird auch ein Verzeichnis geführt, welches diesen ihre entsprechenden Seiten des Polygons zuordnet, sodass sich der Schnitt im Polygon verorten lässt. Ein Spezialfall wäre eine Seite, welche den Kreis als Sekante durchquert und zu zwei Schnittpunkten führt. In dem Fall soll diese Seite in zwei separate unterteilt werden. Folgende Formel bestimmt die Koordinaten des neuen Eckpunkts, welcher ins Polygon eingeschoben werden soll.

$$x = (x_1+x_2) / 2$$

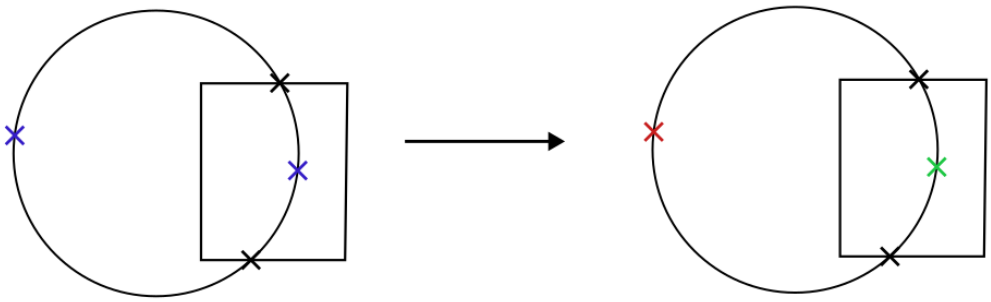
$$y = (y_1+y_2) / 2$$



Falls die Anzahl Schnittpunkte kleiner gleich 1 ist, soll das spezielle Verhältnis des Polygons zum Kreis untersucht werden. Wenn der Mittelpunkt des Kreises nicht im Polygon liegt, befindet sich der Kreis außerhalb des Polygons und die Schnittfläche ist 0, andernfalls $85\pi^2$, die gesamte Fläche des Kreises. Falls ein beliebiger Eckpunkt des Polygons im Kreis liegt, liegt das gesamte Polygon im Kreis und es soll sein Flächeninhalt ausgerechnet werden.

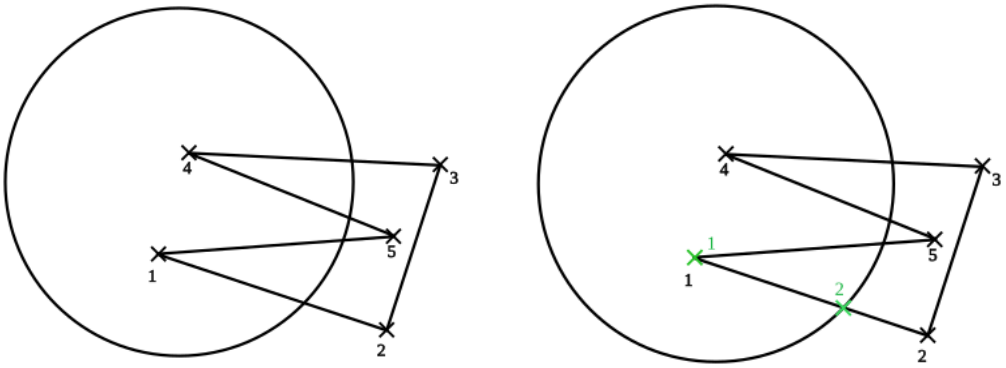


Bei einer Anzahl Schnittpunkte die 1 übersteigt, sollen die Winkel dieser anhand des Einheitskreises berechnet werden. Mithilfe dieser lassen sich Punkte des Kreises eindeutig klassifizieren und vergleichen. Zwischen je zwei Winkel soll genau mittig ein Probewinkel platziert werden. Wenn der repräsentierte Punkt im Polygon liegt, beschreiben die zwei anliegenden Winkel ein Intervall. Dabei besitzt jeder Schnittpunkt genau ein anliegendes Intervall, das im Polygon und eines das außerhalb liegt. Anschließend können die Winkel der Intervalle in Punkte rückübersetzt und ein Verzeichnis erstellt werden, wobei für jeden Schnittpunkt sein zugehöriges Intervall zugeordnet wird und abfragbar ist.

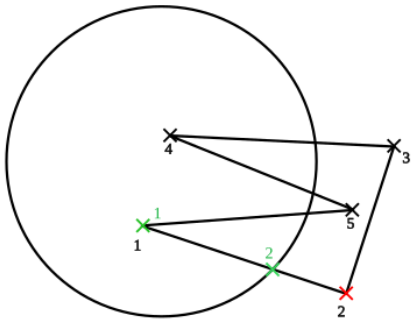


Anschließend soll eine Liste erstellt werden, die außerhalb des Kreises liegende Punkte ignoriert und Schnittpunkten eine Identifikationsnummer zuordnet. Es wird dabei jeder Eckpunkt des Polygons untersucht, wobei die Position des nächsten Punktes ebenfalls eine Rolle spielt. Wenn der derzeitig betrachtete Punkt im Polygon liegt, wird dieser direkt in die Liste übernommen. In dem Fall, wo zusätzlich genau einer der beiden im und der andere außerhalb des Kreises liegt, liegt ein Schnitt des Kreises vor und es wird der entsprechende Schnittpunkt abgefragt. Ihm wird eine Identifikationsnummer, anhand des Indexes im Verzeichnis der Intervalle, zugeordnet, sodass der andere Schnittpunkt des Intervalls eindeutig zu erkennen und gruppieren sein wird.

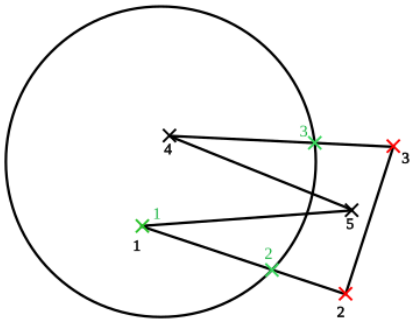
Ein beispielhafter Ablauf wäre folgender. Dabei wird die Reihenfolge der Eckpunkte des Polygons mit schwarzen und die der neuen Liste mit grünen Nummern gekennzeichnet.



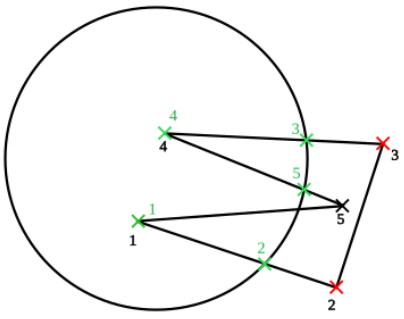
Bei der Seite (1,2) wird ein Schnitt verzeichnet, sodass der entsprechende Schnittpunkt herausgesucht und in die neue Liste eingefügt wird.



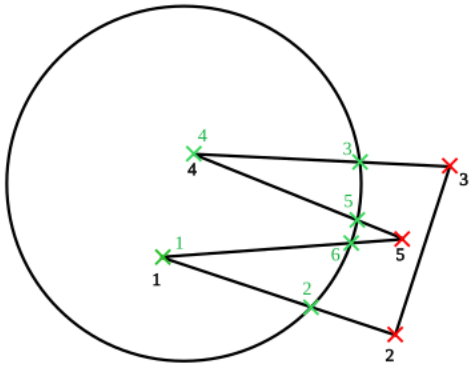
2 liegt außerhalb des Kreises und wird daher ignoriert.



3 selbst wird ebenfalls ignoriert, allerdings wird der Schnittpunkt seiner Seite mit 4 an die Liste angefügt.

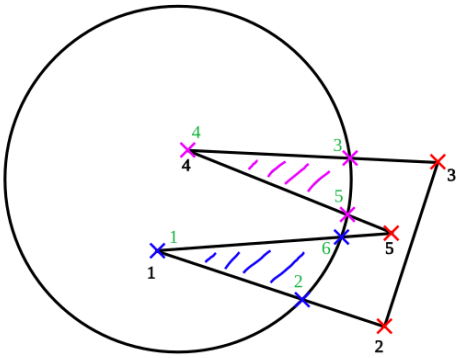


4 liegt im Kreis und seine Seite mit 5 hat einen Schnittpunkt, beide werden angefügt.

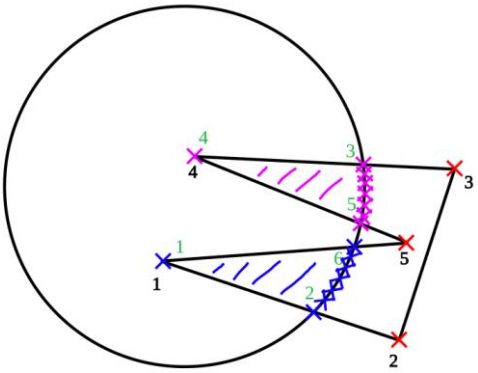


5 selbst wird nicht angefügt, allerdings der durch ihn beschriebene Schnittpunkt.

Wenn zwei Schnittpunkte nebeneinanderliegen, begrenzen sie sicherlich kein eigenes Polygon, sondern biete nur einen Übergang, sodass deren Identifikationsnummer auf -1 geändert werden soll. Anschließend können die einzelnen Teilflächen/-polygone entnommen werden. Es liegt eine Liste vor, in der sich verschachtelte Intervalle befinden. Solange die Liste noch Elemente besitzt, soll ein Abschnitt gefunden werden, der von einem Intervall begrenzt wird und kein anderes begrenzendes Intervall enthält, Übergangsintervalle mit der Nummer -1 werden dabei nicht berücksichtigt. Sobald ein solches gefunden wurde, wird dieses entfernt und die Suche beginnt erneut. Zu Beginn soll überprüft werden, ob noch unerforschte Intervalle existieren, wenn nicht endet der Vorgang. Dabei ist zu berücksichtigen, dass sich vermeintlich begrenzende Intervalle inmitten des Ablaufs als Übergangsintervalle herausstellen können, so sollen zwei aufeinanderfolgende begrenzende Intervalle gemerkt und im weiteren Verlauf ignoriert werden.



Die Schnittfläche liegt bereits in einzelne Polygone unterteilt vor, allerdings befinden sich in diesen noch Intervalle, zwischen welchen die zusätzlichen Punkte platziert werden müssen. Die Schnittpunkte sollen in Winkel rückübersetzt werden, sodass zwischen den beiden Winkeln die zusätzlichen Punkte generiert und anschließend wieder zu Koordinaten rückübersetzt werden können. Diese werden in passender Reihenfolge zwischen die Schnittpunkte ins Polygon eingefügt.



Anschließend muss der Flächeninhalt der einzelnen Polygone mithilfe von Triangulation und Ear Clipping berechnet und zusammenaddiert werden. Diese Summe entspricht der Schnittfläche von Kreis und Polygon,

1.5.2 Grid Search

Die beste Position des Gesundheitszentrums kann angenähert werden, indem systematisch bestimmte Position des Zentrums getestet und auf deren Wert untersucht werden, sodass schlussendlich die beste Position gewertet herausgefiltert werden kann. Dabei soll immer die beste Position weiter untersucht werden, indem neue Punkte in das Umfeld platziert und in Betrachtung gezogen werden.

Zu Beginn soll der gültige Bereich der Untersuchung begrenzt werden, indem ein achsenparalleles Rechteck, welches das gesamte Polygon einschließt, erstellt wird. Es sollen dazu der Definitions- und Wertebereich des Polygons errechnet werden:

$$D_f = [\min(x)\,;\max(x)]$$

$$W_f = [\min(y)\,;\max(y)]$$

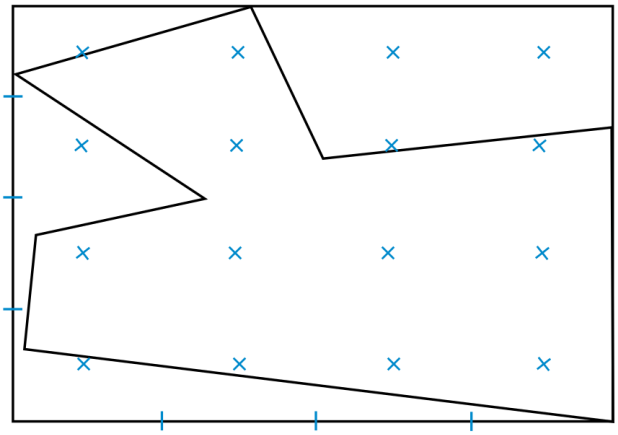
Die Grenzen der Intervalle beschreiben in ihren vier Permutationen die Eckpunkte des Rechtecks.

Um die ersten Test-Punkte in dieses Rechteck zu platzieren, müssen zwischen diesen die Abstände in x- und y-Richtung festgelegt werden. Dazu werden die jeweiligen Intervalllängen durch n geteilt. n soll dabei flexibel sein, um den Vorgang optimieren zu können.

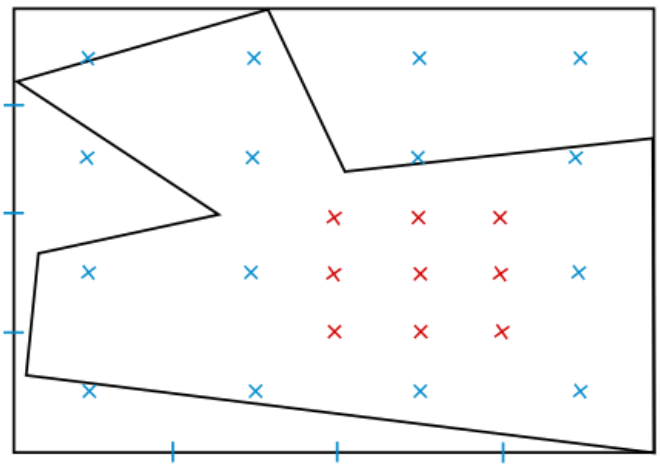
$$d_x = (\max(x) - \min(x)) \,/\, n$$

$$d_y = (\max(y) - \min(y)) \,/\, n$$

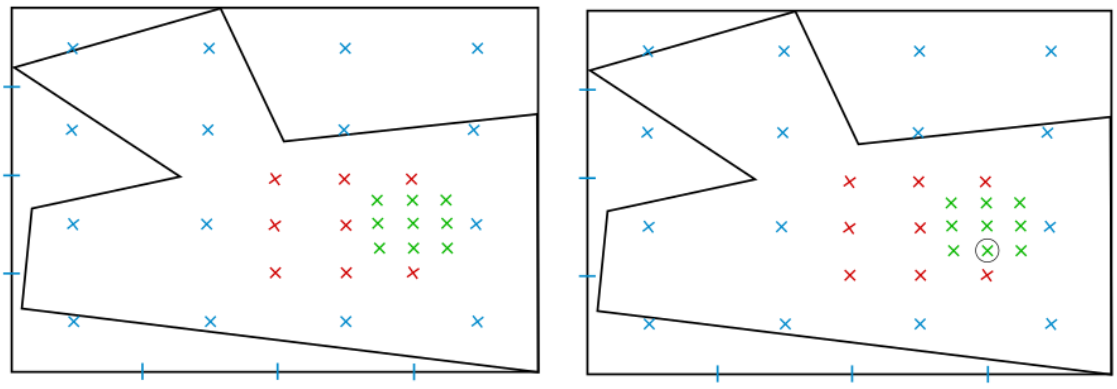
Es werden anschließend n^2 Punkte in folgendem Muster in das Rechteck platziert:



Der beste Punkt wird mithilfe der Heuristic ermittelt und die Untersuchung dort fortgesetzt. Dieser Punkt wird aus dem Speicher entfernt und 9 neue gehen aus diesem hervor, welche die Tiefe 2, im Gegensatz zu den ursprünglichen Punkten mit der Tiefe 1, besitzen. Diese werden in der folgenden Abbildung rot markiert. Einer dieser 9 besitzt die gleiche Position wie der Ursprüngliche, während die anderen 8 in einem 3x3 Feld um diesen angeordnet sind. Die Abstände in x- und y-Richtung entsprechen dabei der Hälfte der vorherigen Tiefe.



Dieser Ablauf wird wiederholt, bis ein Punkt einer bestimmten Tiefe t als bester klassifiziert wird und die Position des Gesundheitszentrums endgültig festlegt.



Neben n und t bietet dieser Prozess einen weitere Optimierungsmöglichkeit, etwa indem eine weitere Variable s für Stopp eingeführt wird, die besagt, dass ab dieser Tiefe s nur noch Punkte registriert werden dürfen, die sich innerhalb des Polygons befinden. Somit werden außenliegende Punkte nicht bereits am Anfang außer Betracht gezogen, sondern erst nach einer bestimmten Tiefe, die eine bestimmte Ausbreitung des ursprünglichen Punktes beschreibt, sodass ab diesem Zeitpunkt aussortiert werden kann, um die Laufzeit zu verbessern.

2. Zeitkomplexität

2.1 Position des Gesundheitszentrums

Zu Beginn werden p Punkte in O(Heuristik) betrachtet und in log(p) Laufzeit in einen Heap gepusht. Bis ein Lösungspunkt im Heap gefunden wurde, müssen a Punkte aus dem Heap in log(p) entnommen und unter der Laufzeit der Heuristik untersucht werden. Die mathematische, ungekürzte Laufzeit laute:

$$O(p * (O(Heuristik) + log(p)) + a * (log(p) + O(Heuristik))$$

Die Anzahl zu Beginn betrachteter Punkte beträgt p^2, wobei p durch die Anzahl Teilungen in x- und y-Richtung bestimmt wird. Zumeist überwiegt diese Anzahl zu Beginn betrachteter Punkte die Anzahl an später betrachteten Punkten, sodass die Laufzeit zu folgender reduziert wird:

$$O(p * (O(Heuristik) + log(p)))$$

Anschließend müssen die Laufzeiten der Heuristiken betrachtet werden.

2.1.1 Annäherung durch Siedlungen

Abgesehen von der Erstellung der Seiten des Polygons, welche in O(n) für n Eckpunkte des Polygons abläuft, besteht das Grundgerüst aus drei Abläufen. Das Platzieren der im Kreis liegenden Siedlungen, das Optimieren der Rotation dieser Siedlungen und das Entfernen der in Betracht vom Polygon außenliegenden Siedlungen.

2.1.1.1 Platzierung der inneren Siedlungen

Dies geschieht in O(1) Laufzeit, da das Muster linear unter simplen Rechnungen aufgebaut wird und sich in der Menge der Siedlungen nicht unterscheidet.

2.1.1.2 Entfernung der außenliegenden Siedlungen

Für jede Siedlung muss überprüft werden, ob sie sich im Polygon befindet. Dies wird durch ray-casting realisiert, welches einen Schnitt mit jeder Seite des Polygons überprüft. Folglich müsste die Laufzeit O(n*m) für n Seiten, äquivalent zu der Anzahl Eckpunkte, und m Siedlungen lauten. Die Anzahl Siedlungen im Kreis ist jedoch fest, sodass hier zu O(n) reduziert werden darf.

2.1.1.3 Optimierung der Rotation

Es wird in der Reichweite 0°-60° unter vorbestimmter Genauigkeit eine Anzahl Winkel bestimmt, bei denen untersucht werden soll, bei welchem Winkel sich die größte Anzahl Siedlungen im Polygon befindet. Die Laufzeit würde O(n*m*w) n Ecken des Polygons, m Siedlungen und w Winkel, allerdings sind sowohl die Anzahl Siedlungen im Kreis als auch die Anzahl Siedlungen fest, sodass die Laufzeit im Allgemeinen zu O(n) mit n Ecken als einzige zwingend verändernde Variable gekürzt werden darf.

2.1.1.4 Resultat

Die höchste Teillaufzeit beträgt O(n), sodass dies als finale Laufzeit anerkannt werden kann. In der Realität übersteigt sie jedoch die Linearität, so ist n zwar die einzige sich wahrhaftig verändernde

Variable, jedoch verlängern Berechnungen, die die Anzahl Siedlungen oder verschiedene Winkel als Faktoren beinhalten die Laufzeit erheblich, allerdings in konstanter Natur.

2.1.2 Berechnung der Schnittfläche

Die Laufzeit wird hierbei durch die Triangulation und Flächeninhaltsberechnung der einzelnen Teilpolygone dominiert. Bis dahin werden etwa nur Operationen durchgeführt, die eine Umformung von Werten in O(1) bewirken oder die Daten in etwa O(n) umstrukturieren. Für jedes der m Polygone muss sein Flächeninhalt berechnet werden. So werden diese in Dreiecke aufgeteilt und deren Flächeninhalte in O(1) berechnet. Diese Aufteilung erfolgt durch Ear Clipping. Es werden schrittweise alle n Eckpunkte des Polygons entfernt, wobei nach jedem Entfernen die Eckpunkte neu klassifiziert werden müssen, resultierend in O(n^2). Somit ergibt sich eine finale Laufzeit von O(m * n^2) für m Teilpolygone und n Eckpunkte dieser.

2.1.3 Finale Zeitkomplexitäten

Wenn die Annäherung durch Siedlungen als Heuristik benutzt wird, ergibt sich eine Laufzeit von

$$O(p * (n + \log(p)))$$

n übersteigt dabei log(p), sodass es zum finalen Ergebnis reduziert werden kann:

$$O(p*n)$$

Bei der exakten Berechnung der Schnittfläche als Heuristik ergibt sich folgende Laufzeit.

$$O(p*m*n^2)$$

2.1.4 Erkenntnis

Das Nutzen der Annäherung durch Siedlungen als Heuristik erweist sich gegenüber der Berechnung der Schnittflächen als deutlich überlegen. Im Folgenden sollen beide Heuristiken auf die Qualität der Ergebnisse untersucht und daraus ein Urteil gefällt werden.

3. Platzkomplexität

Da sich die Annäherung der Siedlungen als Heuristik gegenüber der Ermittlung der Schnittfläche deutlich überlegen erwies, soll diese im weiteren Verlauf betrachtet und auch zur Lösung der Probleme verwendet werden. Im Folgenden soll Die Platzkomplexität des Programms mit Betracht dieses Verfahrens bestimmt werden. Zuerst wird eine Datei eingelesen und die daraus entnommenen Eckpunkte zusätzlich zu Seiten umformatiert. Beides entspricht einer Platzkomplexität von O(n) für n Eckpunkte des Polygons, da aus der Datei nur diese gespeichert werden und die Seitenzahl äquivalent zur Anzahl Eckpunkte ist. Anschließend soll die Position des Gesundheitszentrums ermittelt werden. Dort dominiert der Heap den verwendeten Platz. Zwar werden auch etwa vorherige Speicherungen, wie etwa eine Liste aller Werte der anfänglichen Punkte, vorgenommen, allerdings werden all diese Punkte ebenfalls in den Heap eingefügt, sodass dieser bereits mindestens einen ebenso großen Speicherbedarf besitzt als die restlichen Strukturen. Zudem wird der Heap nicht unbeachtet gelassen. So erhöht sich bei jeder Untersuchung eines Punktes die Anzahl Einträge im Regelfall um 8, da einer entfernt und 9 neue hinzugefügt werden. Eine Ausnahme würden Punkte bilden, die außerhalb des rechteckigen Feldes oder

außerhalb des Polygons nach der Abbruchstiefe liegen und deswegen nicht hinzugefügt werden. Somit dominiert die Platzkomplexität des Heap mit O(h) für h Einträge.

Nach dem Herausfinden der Position des Gesundheitszentrums müssen die Siedlungen platziert werden. Dabei müssen diese jetzt auch über den Radius des Gesundheitszentrums hinaus betrachtet werden. Dabei belaufe sich die Zeitkomplexität etwa auf O(d^2) für d als maximale Distanz und ebenso die Platzkomplexität, da in jedem einzigen Zeitabschnitt genau eine Siedlung platziert wird. Anschließend wird die Rotation optimiert, was keinen zusätzlichen Platz benötigt und daher konstant ist, O(1). Zum Schluss werden außenliegende Siedlungen entfernt und eine neue Liste der korrekten Siedlungen zurückgegeben. Die Platzkomplexität ist folglich O(s) für s Siedlungen. Diese Verfahren überbieten nicht die Platzkomplexität des Suchens der Position des Gesundheitszentrums, sodass die insgesamte Platzkomplexität des Programms bei O(h) für h Einträge im Heap bleibt. Die genaue Bestimmung mithilfe der Parameter ist schwer, aber lässt sich ungefähr annähern durch die Anzahl Teilungen pro Seite im Quadrat, äquivalent zur Anzahl anfänglicher Punkte, und die Abbruchstiefe multipliziert mit 8. Dies würde allerdings den besten Fall beschreiben, wobei ein durchschnittlicher deutlich mehr Punkte betrachtet und ausgehend aus diesen 8 neue schöpft, was die Zeitkomplexität erhöhen würde. Allgemein bleibt diese jedoch bei O(h) für h Einträge im Heap.

4. Beispiele

Sobald das Programm ausgehend aus __main__.py gestartet wird, ermöglicht es dem Nutzer, zusätzliche Visualisierungen zu aktivieren. Nach dieser Frage wird ein Dateipfad abgefragt, nach dessen Eingeben der Lösungsprozess startet.

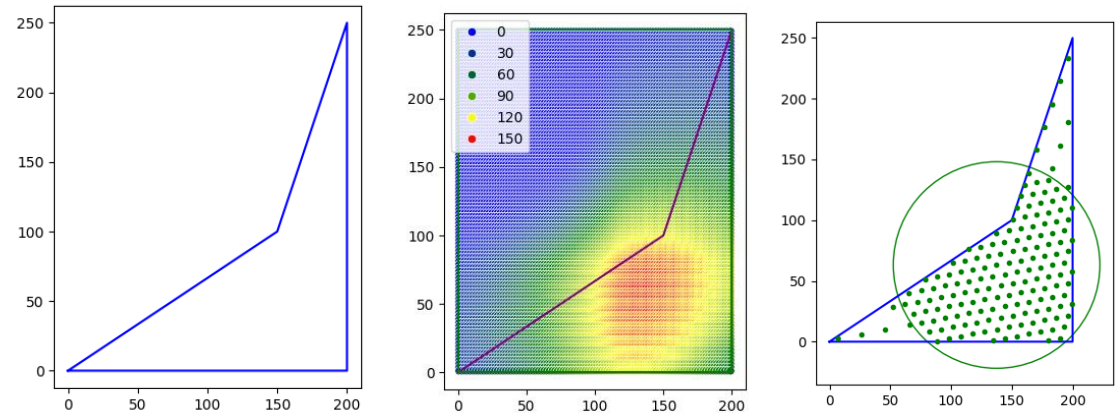
4.1 Vorgegebene Beispiele

Erstes Bild stellt das Polygon dar, letztes die finale Platzierung und mittleres eine Heatmap für die Platzierung des Gesundheitszentrums nach dem Wert der Heuristik, in diesem Fall die Annäherung durch Siedlungen. Die einzelnen Messpunkte sind die der ersten Tiefe.

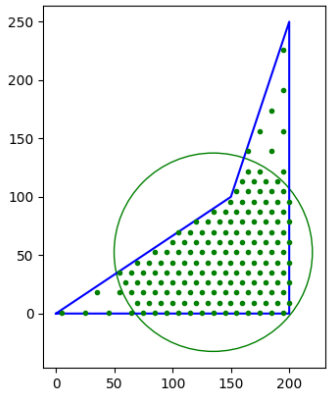
1.

Parameter des Grid-Search: splits=**100**, depth_finish=**3**, depth_stop=**2**

```
Anzahl Siedlungen:  163
Laufzeit:  6.98 s
```



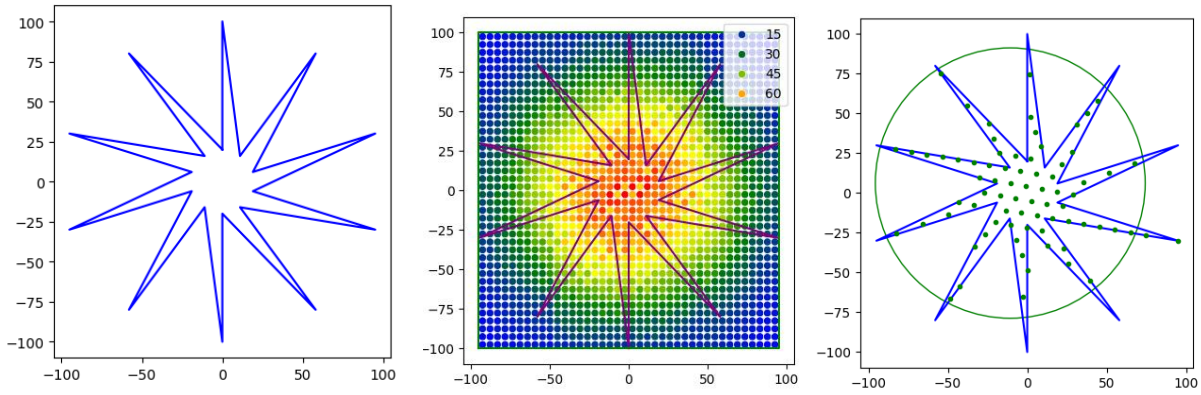
Folgende Lösung mit 165 Siedlungen wurde ebenfalls gefunden, leider ohne gespeicherte Parameter.



2.

Parameter des Grid-Search: splits=**40**, depth_finish=**3**, depth_stop=**2**

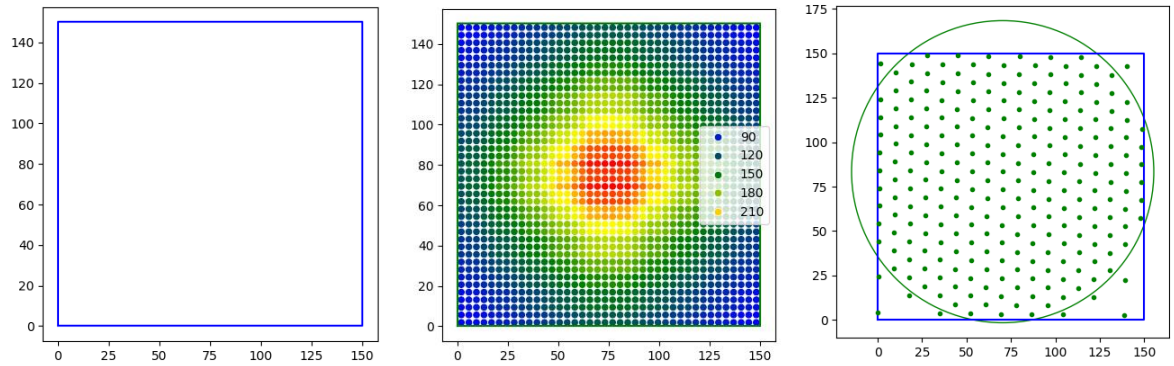
```
Anzahl Siedlungen:  75
Laufzeit:  3.30 s
```



3.

Parameter des Grid-Search: splits=**40**, depth_finish=**3**, depth_stop=**2**

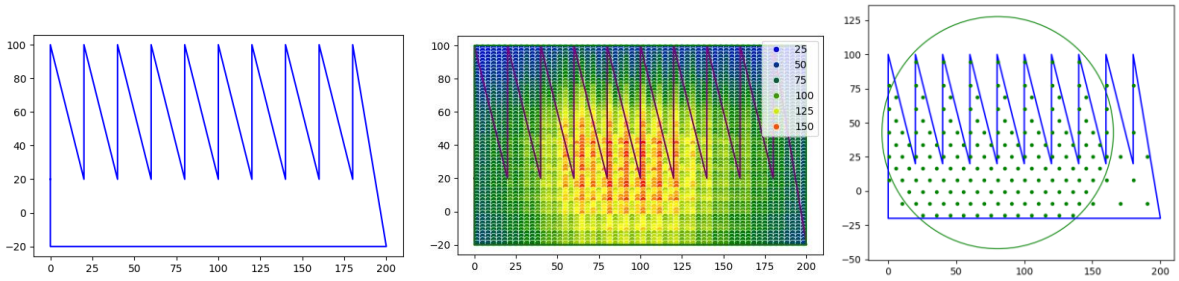
```
Anzahl Siedlungen:  245
Laufzeit:  1.19 s
```



4.

Parameter des Grid-Search: splits=**60**, depth_finish=**4**, depth_stop=**3**

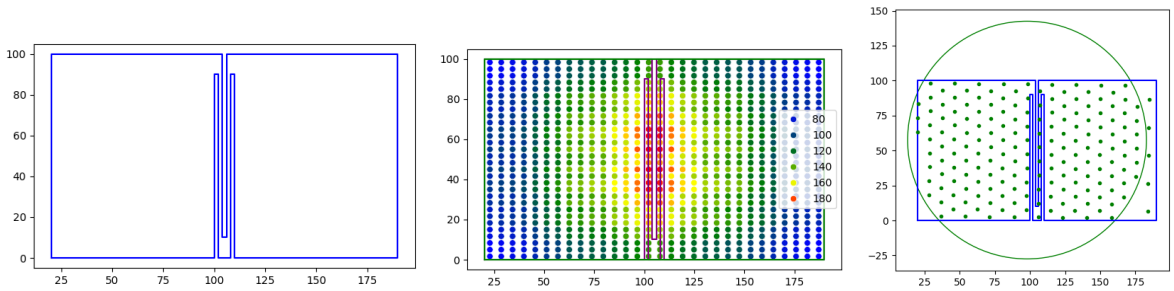
```
Anzahl Siedlungen:  172
Laufzeit:  24.03 s
```



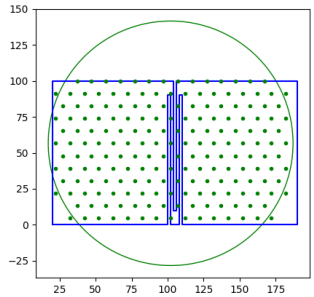
5.

Parameter des Grid-Search: splits=**30**, depth_finish=**3**, depth_stop=**2**

```
Anzahl Siedlungen:  182
Laufzeit:  1.43 s
```



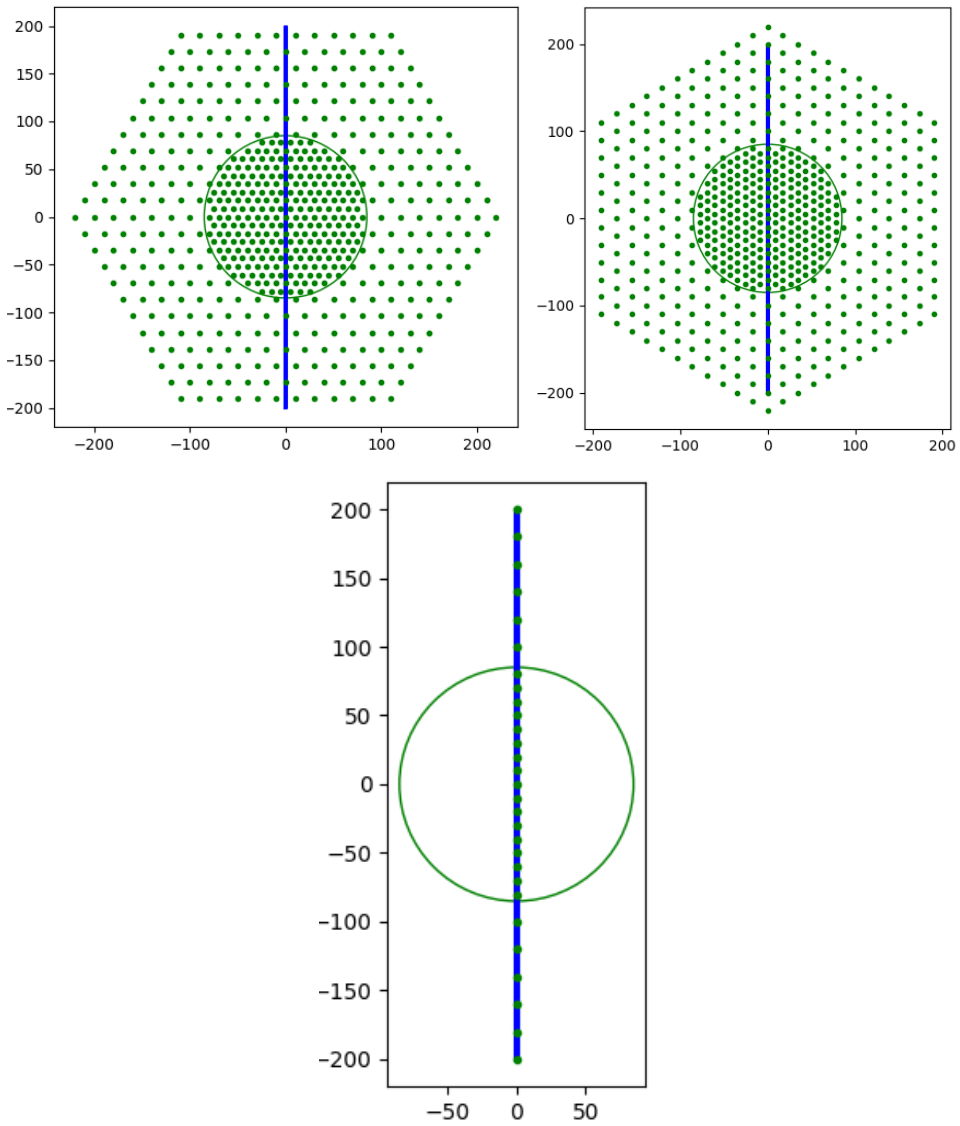
Zudem kann folgende Lösung von 192 Siedlungen erreicht werden, wenn die Optimierung der Rotation bereits bei der Positionssuche des Gesundheitszentrums ausgelassen und somit dieser besonders effektive Fall der Aufstellung gefunden wird.



4.2 Eigene Beispiele

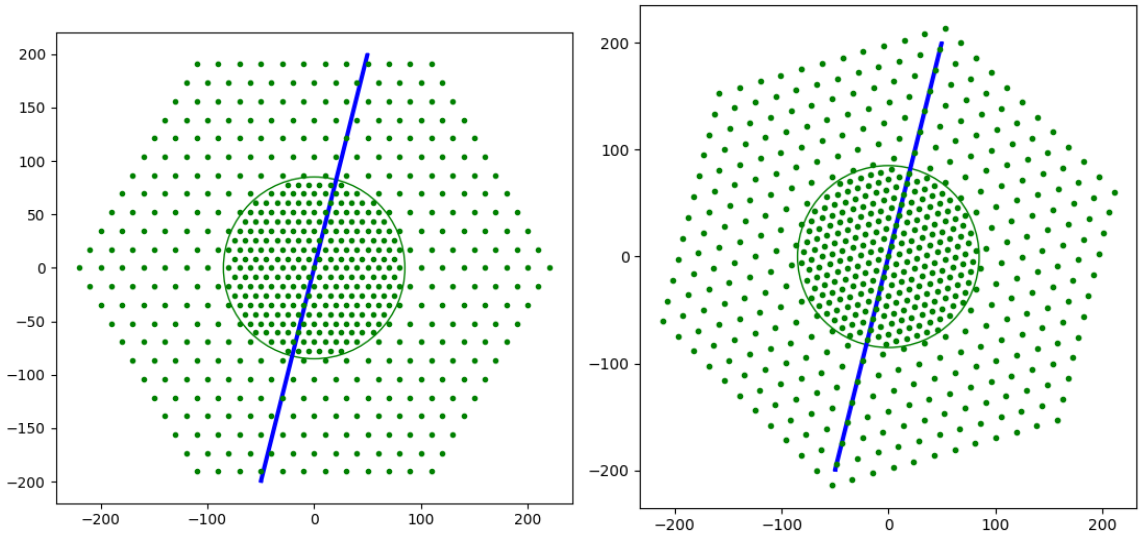
Aus den Lösungen der vorgegebenen Beispiele ergibt sich, dass alle Funktionalitäten des Programms funktionieren und zu nennenswerten Ergebnissen führen. Im Folgenden sollen spezieller Funktionalitäten weitgreifender getestet und deren Potenzial veranschaulicht werden. Alle Beispiele sind in „test.py“ enthalten und ausführbar.

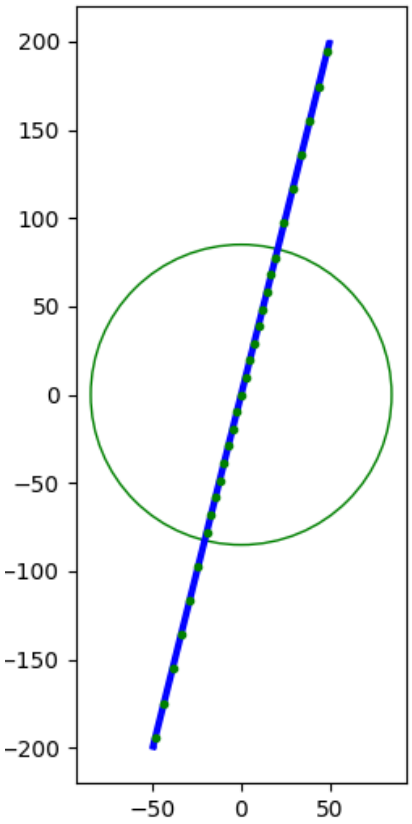
4.2.1 Rotation der Siedlungen




```
Eckpunkte: [[-1,-200],[1,-200],[1,200],[-1,200]]
```

Die oben links liegende Abbildung stellt die erstmalige Platzierung der Siedlungen dar. Diese ist nicht optimal und wird daher durch die passende Rotation in der rechts davon gelegenen Abbildung verbessert. Die untere Grafik bildet nur noch die innenliegenden Siedlungen ab. Deren Zahl hat sich von 15, bei der ursprünglichen Platzierung, zu 29 erhöht. Noch deutlicher wird die Verbesserung, wenn das Polygon schräg und nicht senkrecht vorliegt.





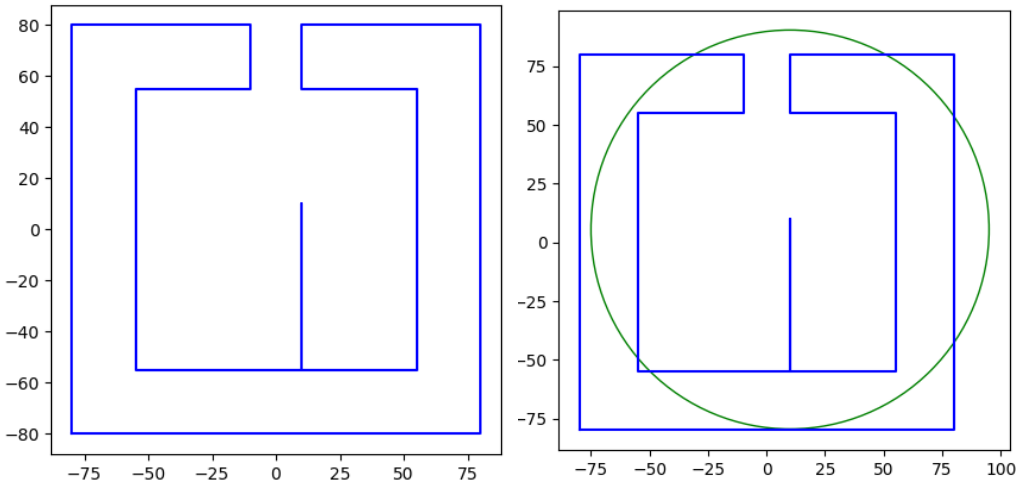
`Eckpunkte: [[-49,-200],[-51,-200],[49,200],[51,200]]`

Hier lässt sich eine Veränderung von anfänglich 5 Siedlungen zu ebenfalls 29 Siedlungen nach der Rotation verzeichnen.

Somit lässt sich sagen, dass eine Rotation des Polygons keine Veränderung des Ergebnisses hervorruft und das Potenzial des Polygons in dem Aspekt stets vollständig genutzt werden kann.

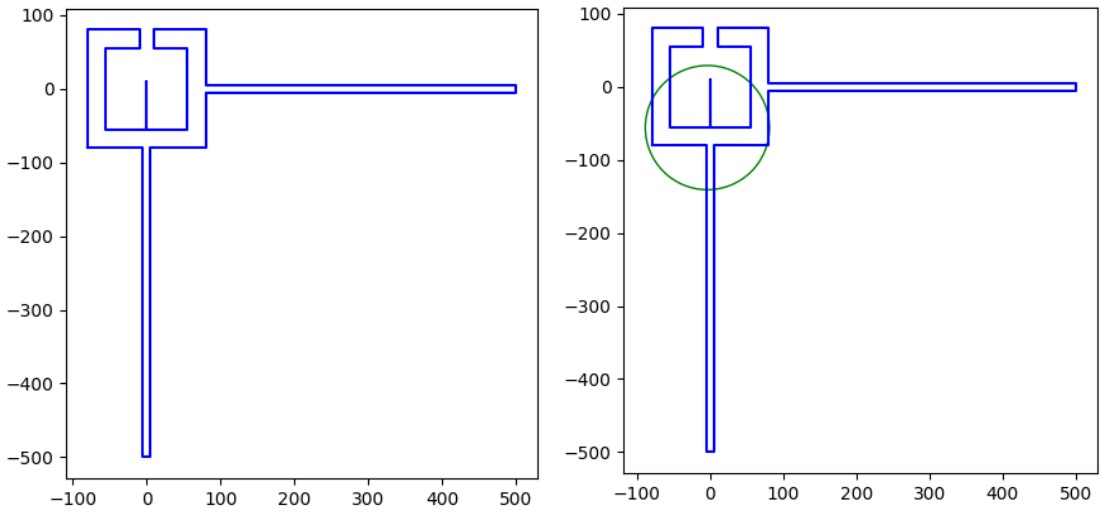
4.2.2 Position des Gesundheitszentrums

Beim Ermitteln der Position des Gesundheitszentrums ist es von großer Bedeutung die Parameter an die Form des Polygons anzupassen und einen optimalen Ausgleich zwischen Genauigkeit und Geschwindigkeit zu finden. So sollen verschieden Polygone im Folgenden getestet werden.



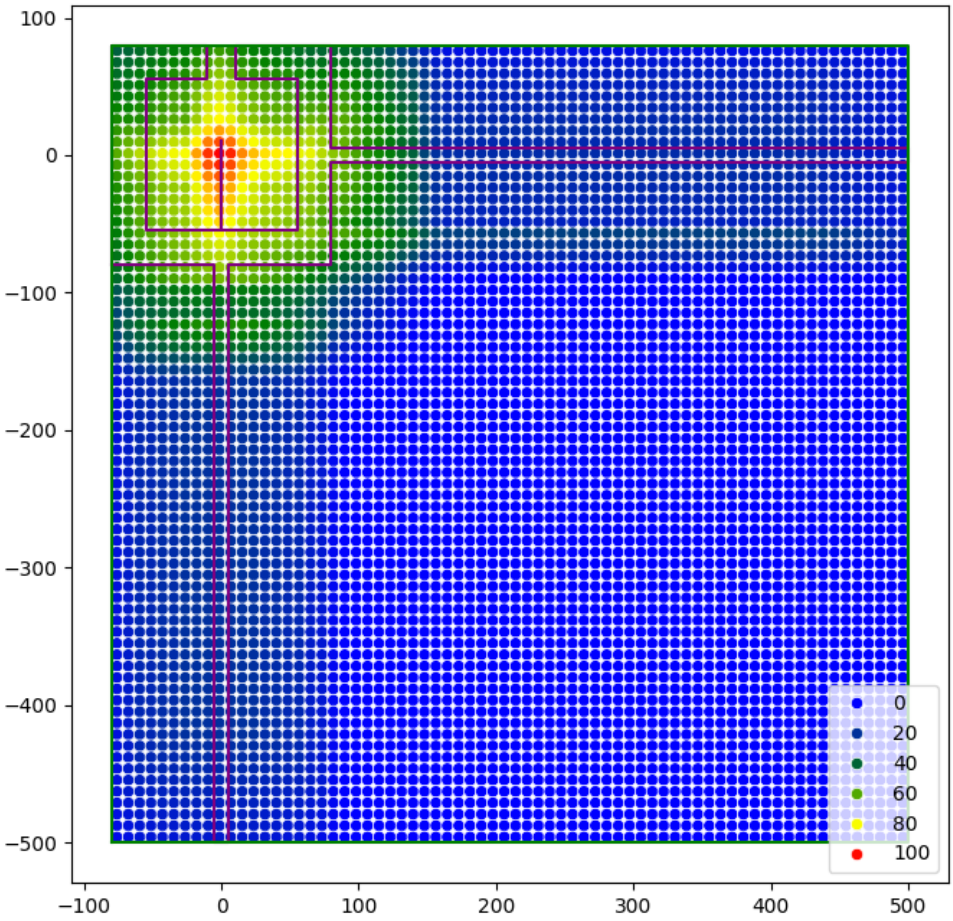
```
Eckpunkte: [[-80,-80],[80,-80],[80,80],[10,80],[10,55],[55,55],[55,-
55],[10.0000001,-55],[10.0000001,10],[9.9999999,10],[9.9999999,-55],[-55,-55],[-
55,55],[-10,55],[-10,80],[-80,80]]
```

Dieses Beispiel sollte die Genauigkeit des Programms testen. Dabei besitzt der mittlere Weg des Polygons eine Breite von 0,0000002km. Die optimale Position des Gesundheitszentrums wurde erfolgreich gefunden, mit einer beeindruckenden Laufzeit von 1,81s. Die anfänglichen Stichpunkte beliefen sich auf 20^2, die Abschlusstiefe auf 5 und die Tiefe, ab der nur noch innenliegende Punkte weitergeführt werden, 3. Allerdings ist die allgemeine Größe des Polygons recht gering, sodass es fraglich ist, ob eine ähnliche räumlich kleine Position in einem größeren Polygon zu finden ist. Dazu soll folgendes Polygon verwendet werden. Die Breite des Wegs soll auf 0.02km gekürzt werden, um keine unrealistischen Erwartungen zu haben.

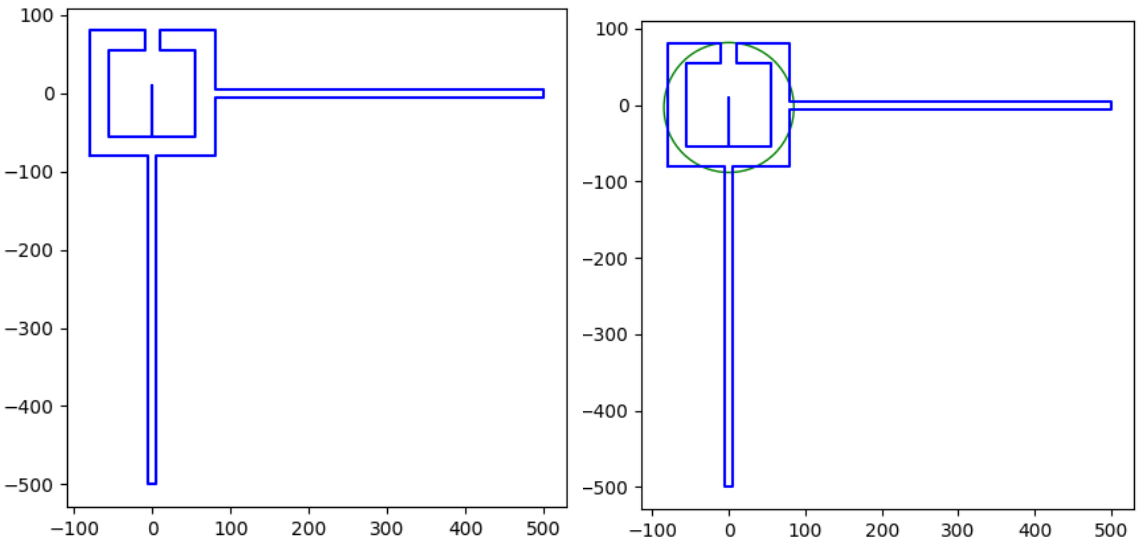


```
Eckpunkte: [[-80,-80],[-5,-80],[-5,-500],[5,-500],[5,-80],[80,-80],[80,-5],[500,-
5],[500,5],[80,5],[80,80],[10,80],[10,55],[55,55],[55,-55],[0.01,-55],[0.01,10],[-
0.01,10],[-0.01,-55],[-55,-55],[-55,55],[-10,55],[-10,80],[-80,80]]
```

Unter den gleichen Parameter wurde in diesem Beispiel nicht die optimale Position gefunden. Es soll eine Heatmap aufgerufen werden, um das Ergebnis zu werten.



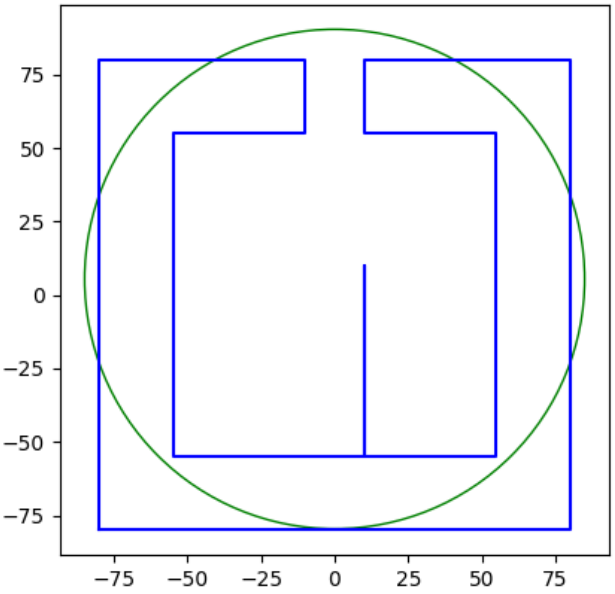
Das Ergebnis ist folglich akzeptabel, aber stark verbesserungswürdig. Die Parameter sollen also wie folgt angepasst werden. Die Anzahl anfänglicher Punkte soll sich auf 70^2 belaufen, um die gewachsene Größe zu kompensieren. Zusätzlich sollen sowohl die Abschluss- als auch die, Abbruchstiefe um 2 erhöht werden.



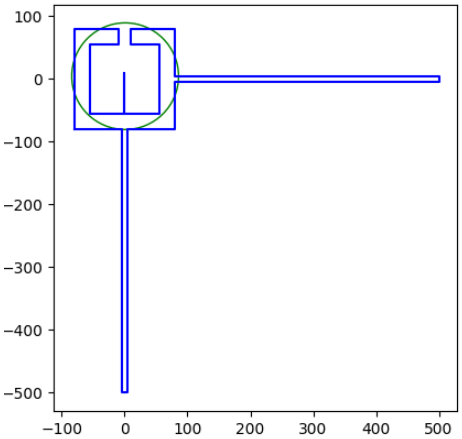
Diese Parameter erwiesen sich erfolgreich, sodass diesmal die beste Position gefunden wurde. In Betrachtung der Qualität, führen höhere Parameter stets zu besseren Ergebnissen, allerdings muss auf die Laufzeit achtgegeben werden. So erhöhte sich diese durch das Ändern der Parameter von 3,48s zu 6,45s. Beide sind durchaus akzeptabel, sodass 70, 7, 5 als Parameter für die Positionssuche des

Gesundheitszentrums ideal sind. Interessant wäre zudem die Erhöhung der Laufzeit beim ersten Beispiel, wo das beste Ergebnis auch mit schlechteren Parametern erzielt werden konnte.

Erstaunlicherweise kann in akzeptabler Zeit kein Ergebnis gefunden werden, da der Weg leicht versetzt zur Mitte vorliegt und daher ständig Punkte mittig des Polygons untersucht werden, welche zwar das meiste Potenzial bieten, aber keine korrekten Lösungen sind. Daher sollen sowohl die Abschluss- und Abbruchstiefe auf 4 gesetzt werden.



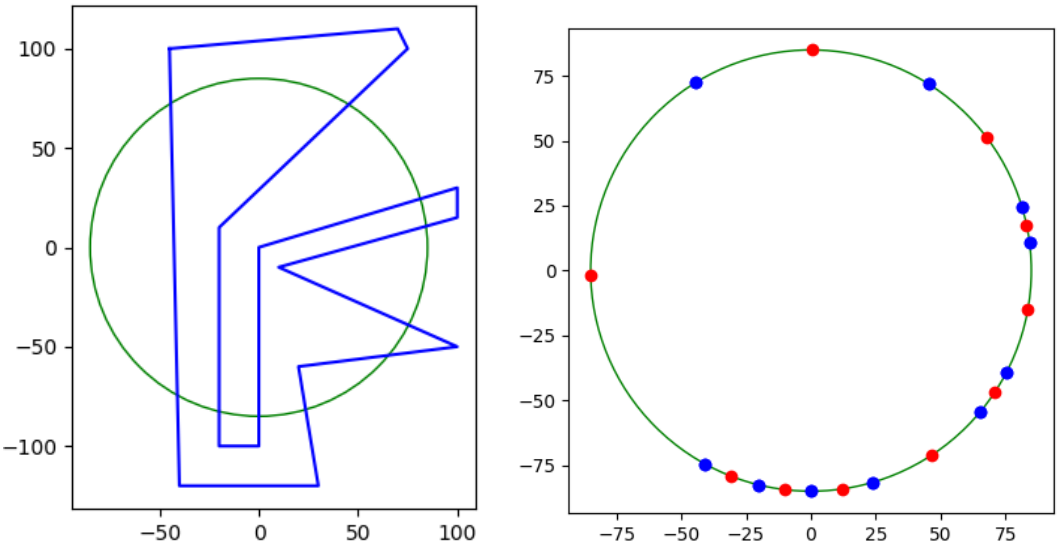
Das Ergebnis ist jetzt korrekt und die Laufzeit mit 4,95s bei einem Weg einer Breite von 0,0000002km zufriedenstellend. Es stellt sich allerdings die Frage, ob diese Veränderung der Parameter das vorherige Beispiel verändert. Allerdings bleibt die Lösung weiterhin korrekt und die Laufzeit mit 6,42s fast identisch.



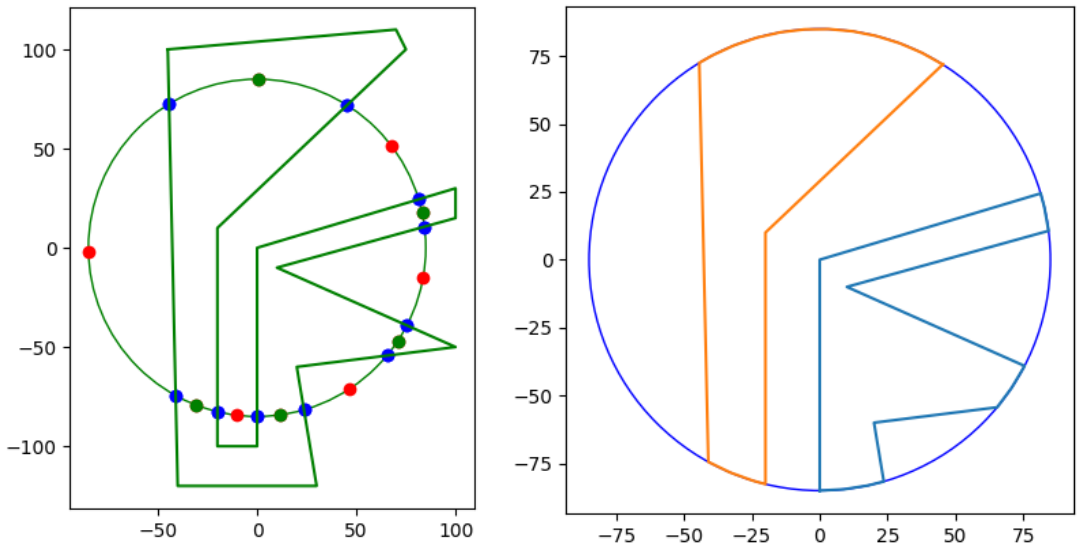
4.2.3 Ermittlung der Schnittfläche

Das Berechnen des Flächeninhalts eines Dreiecks ist simpel, ebenso wie das Teilen eines Polygons in Dreiecke durch Ear Clipping. Daher besteht die größte Herausforderung darin, die Schnittfläche des Polygons und Kreises in kleinere Polygone aufzuteilen, sodass anschließend mit den eben genannten Methoden deren Flächeninhalte berechnet und addiert werden können. Folglich soll getestet werden, ob das Programm Schnittflächen erfolgreich in Polygone unterteilen kann.

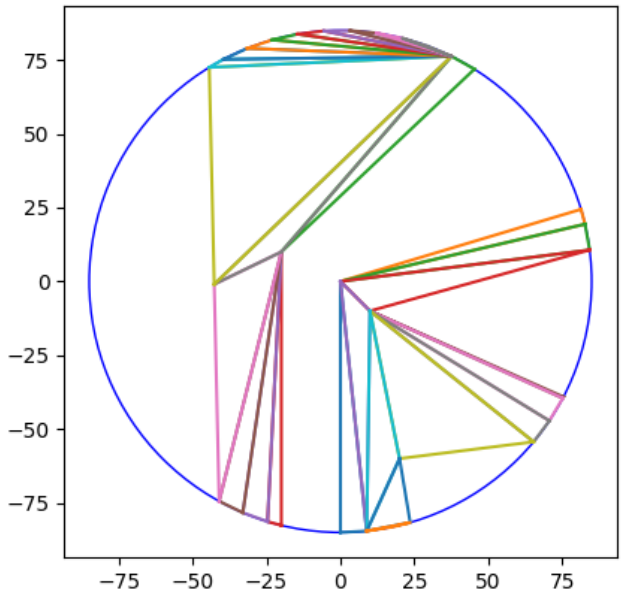
Folgendes Beispiel soll dabei ausführlich betrachtet werden.



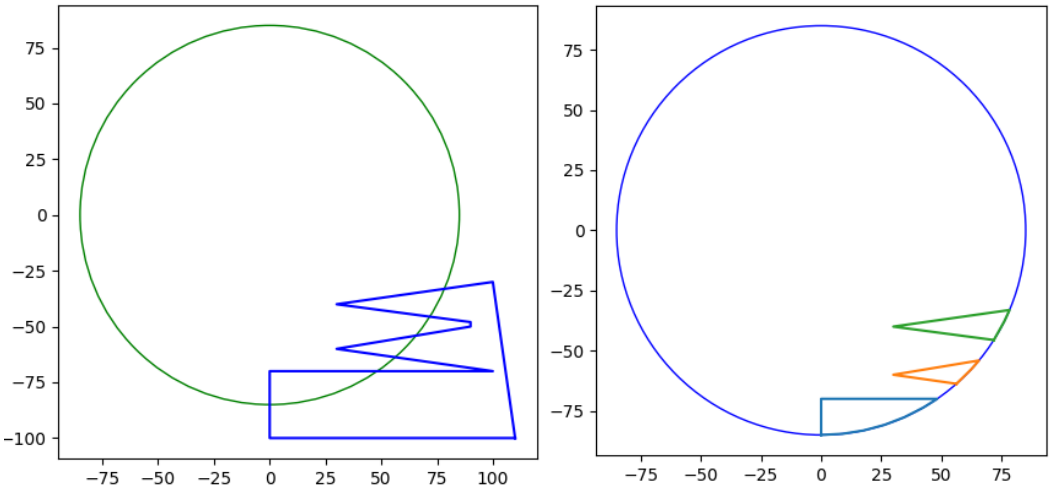
Mit blau sind die Schnittpunkte markiert und anhand dessen, ob die roten Punkte im Polygon liegen, was das Programm welcher Abschnitt im Polygon liegt.

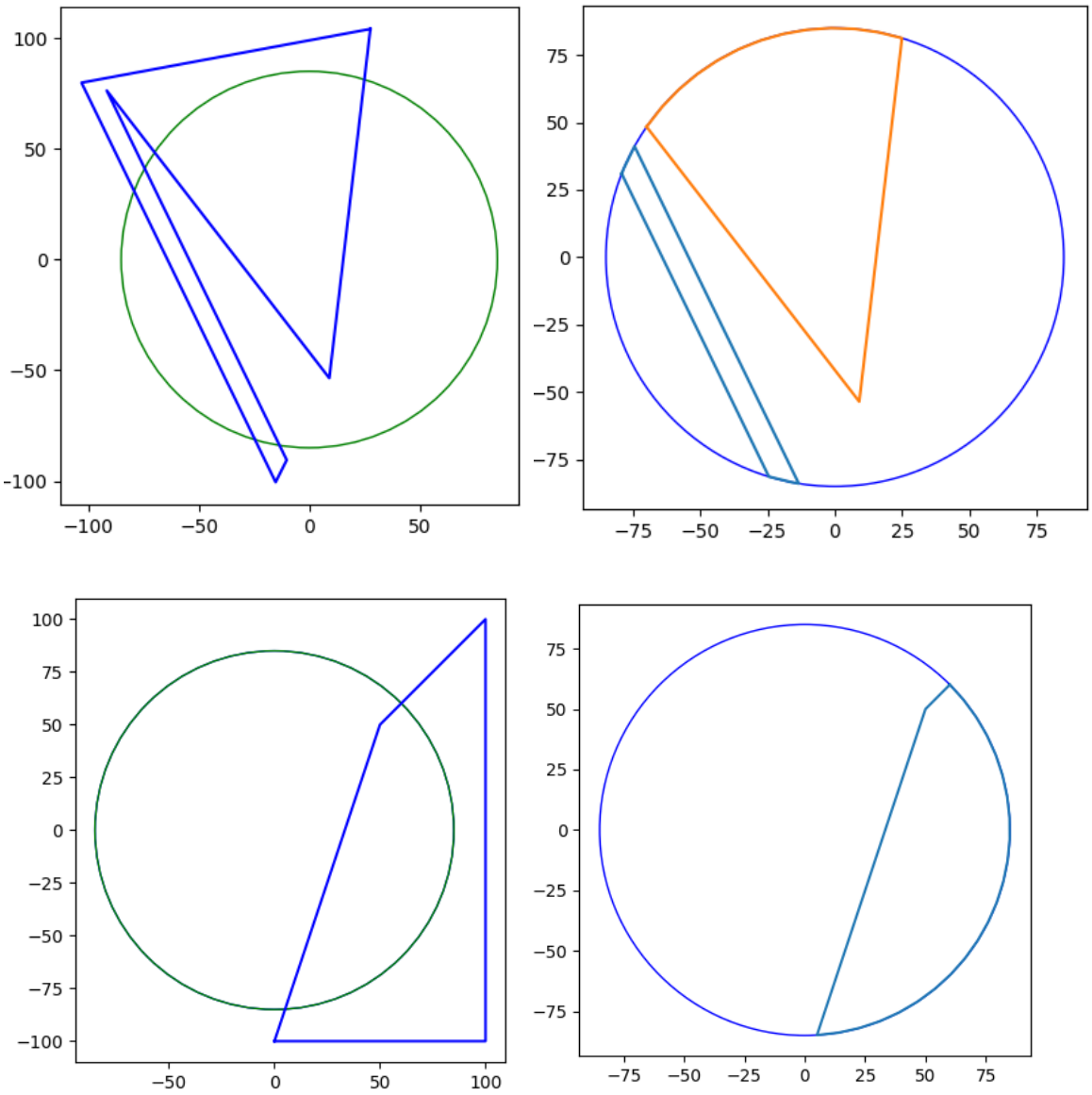


Diese beiden Polygone wurden separiert und anschließend soll Triangulation angewandt werden.



Abschließend kann der Flächeninhalt der einzelnen Dreiecke berechnet werden. Somit erweist sich die Berechnung der Schnittfläche als erfolgreich und beträgt 10446.77km^2. Im Folgenden sollen weitere Beispiele aufgeführt werden.





5. Implementation

Es sollen wesentliche Teile des Codes gezeigt werden. Dabei werden sehr simple Funktionen und Funktionen zur reinen Visualisierung ausgelassen.

5.1 do_lines_intersect

```
def do_lines_intersect(point, edge):
    '''ray-casting mit einem Strahl nach rechts'''
    '''Time Complexity: O(1), Auxiliary Space: O(1)'''
    p_x, p_y = point
    low_x, low_y = edge[0]
    high_x, high_y = edge[1]

    # Stellt sicher, dass 'high' gewiss der höhere Punkt ist
    if low_y > high_y:
        low_x, high_x = high_x, low_x
        low_y, high_y = high_y, low_y
```

```
    # Falls der Punkt in einem Eckpunkt liegt, sollen nicht 2
Schnitte für 2 anliegende Seiten registriert werden
    if p_y == low_y or p_y == high_y:
        p_y += 0.00001

    # Punkt liegt oberhalb, unterhalb oder weit rechts von der
Seite, Schnitt nicht möglich
    if p_y > high_y or p_y < low_y or p_x > max(low_x, high_x):
        return False

    # Punkt liegt weit links von der Seite, Schnitt
    if p_x < min(low_x, high_x):
        return True

    # Sonderfälle werden behandelt
    if high_x - low_x == 0:
        edge_angle = sys.float_info.max
    else:
        edge_angle = (high_y - low_y) / (high_x - low_x)
    if p_x - low_x == 0:
        if edge_angle > 0:
            point_angle = sys.float_info.max
        else:
            point_angle = sys.float_info.min
    else:
        point_angle = (p_y - low_y) / (p_x - low_x)

    if abs(edge_angle - sys.float_info.max) < 100 and point_angle <
0:
        edge_angle = sys.float_info.min

    # Größerer Winkel des Punktes deutet darauf hin, dass dieser
links von der Seite liegt
    return point_angle > edge_angle
```

5.2 is_point_in_polygon

```
def is_point_in_polygon(polygon_edges, point):
    '''Bei einer ungeraden Anzahl geschnittener Seiten liegt der
Punkt im Polygon'''
    '''Time Complexity: O(n) für n Seiten, Auxiliary Space: O(1)'''
    count = 0
    for edge in polygon_edges:
        if do_lines_intersect(point, edge):
            count += 1
    return count % 2 == 1
```

5.3 place_inner_settlements

```
def place_inner_settlements(circle_anchor):
    '''Platzierung der Siedlungen innerhalb des Gesundheitszentrums
in einer Hexagon-Struktur'''
```

```
    '''Time Complexity: O(1), Auxiliary Space: O(s) für s
Siedlungen'''
    anchor_x, anchor_y = circle_anchor
    settlements = []

    # Anhand des Abstandes 10 zweier Siedlungen, Rundung von 85 auf
90
    dist = 90

    # Platzierung der Siedlungen links vom Mittelpunkt
    for x in range(-dist, 0+1, 10):
        # Platzierung der oben gelegenen Siedlungen
        for y in range(dist//10+1):
            pos_x = anchor_x + x + y * 5
            pos_y = anchor_y + y * 8.660254037844386
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        # Platzierung der unten gelegenen Siedlungen
        for y in range(1, (dist+x)//10+1, 1):
            pos_x = anchor_x + x - y * 5
            pos_y = anchor_y - y * 8.660254037844386
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    # Platzierung der Siedlungen rechts vom Mittelpunkt
    for x in range(10, dist+1, 10):
        # Platzierung der unten gelegenen Siedlungen
        for y in range(dist//10+1):
            pos_x = anchor_x + x - y * 5
            pos_y = anchor_y - y * 8.660254037844386
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        # Platzierung der oben gelegenen Siedlungen
        for y in range(1, (dist-x)//10+1, 1):
            pos_x = anchor_x + x + y * 5
            pos_y = anchor_y + y * 8.660254037844386
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    # Entfernen der außerhalb des Radius von 85 liegenden
Siedlungen, da die Hexagon-Struktur an den Ecken aus dem Kreis
hinausragt
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-2)
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-3)
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-1-dist//10)
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-2-dist//10)
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-3-dist//10)
```

```
    settlements.pop(((dist//10+1)**2)*2 + factorial_sum(dist//10-
1)*2-4-(dist+1)//10)
    settlements.pop((dist//10+1)**2+factorial_sum(dist//10-1)-
1+dist//10*3)
    settlements.pop((dist//10+1)**2+factorial_sum(dist//10-
1)+dist//5)
    settlements.pop((dist//10+1)**2+factorial_sum(dist//10-1)-
1+dist//10)
    settlements.pop((dist//10+1)**2+factorial_sum(dist//10-1)-1)
    settlements.pop((dist//10+1)*(dist//10)+factorial_sum(dist//10-
1)-1)
    settlements.pop((dist//10+1)*(dist//10)+factorial_sum(dist//10-
2)-1)
    settlements.pop((dist//10+1)*2)
    settlements.pop((dist//10+1)*2-1)
    settlements.pop(dist//10)
    settlements.pop(dist//10-1)
    settlements.pop(1)
    settlements.pop(0)

    return settlements
```

5.4 place_outer_settlements

```
def place_outer_settlements(circle_anchor, max_dist):
    '''Platzierung der Siedlungen außerhalb des
Gesundheitszentrums'''
    '''Time Complexity: etwa O(d^2) für d als maximale Distanz,
Auxiliary Space: O(s) für s Siedlungen'''
    anchor_x, anchor_y = circle_anchor
    settlements = []
    dist = 90
    if max_dist-dist < 10:
        return []

    dist = int(max_dist//20)*20
    for x in range(-dist, -80, 20):
        for y in range(dist//20+1):
            pos_x = anchor_x + x + y * 10
            pos_y = anchor_y + y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        for y in range(1, (dist+x)//20+1, 1):
            pos_x = anchor_x + x - y * 10
            pos_y = anchor_y - y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    for x in range(-80, 0+1, 20):
        for y in range(5, dist//20+1):
            pos_x = anchor_x + x + y * 10
            pos_y = anchor_y + y * 8.660254037844386*2
```

```
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        for y in range((80+x)//20+1, (dist+x)//20+1, 1):
            pos_x = anchor_x + x - y * 10
            pos_y = anchor_y - y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    for x in range(20, 80+1, 20):
        for y in range(5, dist//20+1):
            pos_x = anchor_x + x - y * 10
            pos_y = anchor_y - y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        for y in range((80-x)//20+1, (dist-x)//20+1, 1):
            pos_x = anchor_x + x + y * 10
            pos_y = anchor_y + y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    for x in range(100, dist+1, 20):
        for y in range(dist//20+1):
            pos_x = anchor_x + x - y * 10
            pos_y = anchor_y - y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)
        for y in range(1, (dist-x)//20+1, 1):
            pos_x = anchor_x + x + y * 10
            pos_y = anchor_y + y * 8.660254037844386*2
            s = Settlement(pos_x, pos_y)
            settlements.append(s)

    return settlements
```

5.5 place_settlements

```
def place_settlements(circle_anchor, polygon_vertices,
visualization=False):
    '''Platziert alle Siedlungen in Abhängigkeit von der maximalen
Distanz im Polygons'''
    '''Time Complexity: etwa O(d^2) für d als maximale Distanz,
Auxiliary Space: etwa O(d^2) für d als maximale Distanz'''
    #abs(vertex[0] - circle_anchor[0]) + abs(vertex[1] -
circle_anchor[1])
    max_dist, pos = max(((euclidean_distance(vertex, circle_anchor),
vertex) for vertex in polygon_vertices), key=lambda x: x[0])
    max_dist += 20
    if visualization:
        visualize_polygon(polygon_vertices,
draw_line=(circle_anchor, pos), title='longest distance')
    settlements = []
    inner_settlements = place_inner_settlements(circle_anchor)
```

```
    outer_settlements = place_outer_settlements(circle_anchor,
max_dist)

    settlements += inner_settlements + outer_settlements


    if visualization:
        visualize_polygon(polygon_vertices, settlements=settlements,
circle_anchor=circle_anchor)
    return settlements
```

5.6 optimize_rotation

```
def optimize_rotation(polygon_edges, settlements, anchor,
precision=1):
    '''Findet die beste Rotation der Siedlungen'''
    '''Time Complexity: O(w*s*n) w untersuchte Winkel, s Siedlungen
und n Seiten des Polygons, Auxiliary Space: O(1)'''
    precision = int(precision)
    best_angle = 0
    max_settlements = 0

    # Ermitteln der optimalen Rotation
    for angle_degrees in range(0, 60*precision, 1):
        angle = radians(angle_degrees/precision)
        settlements_in_polygon = 0
        for settlement in settlements:
            # Anwendung der zu testenden Rotation
            new_x, new_y = rotate_point((settlement.pos_x,
settlement.pos_y), anchor, angle)
            if is_point_in_polygon(polygon_edges, (new_x, new_y)):
                # Zählen der gültigen Siedlungen
                settlements_in_polygon += 1
        if settlements_in_polygon > max_settlements:
            # Merken der besten Daten
            max_settlements = settlements_in_polygon
            best_angle = angle

    # Anwendung der optimalen Rotation
    for settlement in settlements:
        settlement.pos_x, settlement.pos_y =
rotate_point((settlement.pos_x, settlement.pos_y), anchor,
best_angle)

    return settlements

def remove_outside_settlements(polygon_edges, settlements):
    '''Entfernt alle Siedlungen außerhalb des Polygons'''
    '''Time Complexity: O(s*n) für s Siedlungen und n Seiten des
Polygons, Auxiliary Space: O(s) für s Siedlungen'''
    correct_settlements = []
    for settlement in settlements:
```

```
        if is_point_in_polygon(polygon_edges, (settlement.pos_x,
settlement.pos_y)):
            # Korrekte Siedlungen werden gespeichert
            correct_settlements.append(settlement)
    return correct_settlements
```

5.7 calculate_triangle_area

```
def calculate_triangle_area(vertices):
    '''berechnet den Flächeninhalt eines Dreiecks in O(1)'''
    '''Time Complexity: O(1), Auxiliary Space: O(1)'''
    x1, y1 = vertices[0]
    x2, y2 = vertices[1]
    x3, y3 = vertices[2]

    area = 0.5 * abs((x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2)))
    return area
```

5.8 calculate_polygon_area

```
def calculate_polygon_area(polygon_vertices):
    '''Berechnet den Flächeninhalt eines Polygons'''
    '''Time Complexity: O(n^2) für n Eckpunkte des Polygons,
Auxiliary Space: O(n) für n Eckpunkte des Polygons'''
    triangles = ear_clipping(polygon_vertices) # Zeitkomplexität
O(n^2)
    area_num = 0
    for triangle in triangles:
        area_num += calculate_triangle_area(triangle)
    return area_num
```

5.9 calculate_multiple_polygon_area

```
def calculate_multiple_polygon_area(polygons):
    '''Berechnet den insgesamten Flächeninhalt'''
    '''Time Complexity: O(m*n^2) für m Polygons mit n Eckpunkten,
Auxiliary Space: O(m*n) für m Polygone mit m Eckpunkten'''
    area = 0
    for polygon in polygons:
        for i, vertex in enumerate(polygon):
            if isinstance(vertex, tuple):
                polygon[i] = [vertex[0]/(10**decimal_precision),
vertex[1]/(10**decimal_precision)]
        area += calculate_polygon_area(polygon) # Zeitkomplexität
O(n^2)
    return area
```

5.10 convex_or_reflex_vertex

```
def convex_or_reflex_vertex(vertex1, main_vertex, vertex2):
    '''Gibt an, ob der Innenwinkel eines Eckpunkts convex (<=180
Grad) oder reflex (> 180 Grad) ist'''
```

```
    '''Time Complexity: O(1), Auxiliary Space: O(1)'''
    a_x, a_y = main_vertex
    b_x, b_y = vertex1
    c_x, c_y = vertex2

    if (a_x - b_x) * (c_y - a_y) - (c_x - a_x) * (a_y - b_y) > 0:
        return 'convex'
    else:
        return 'reflex'
```

5.11 ear_clipping

```
def ear_clipping(old_polygon_vertices, tries=0):
    '''Unterteilt ein Polygon in Dreiecke'''
    '''Time Complexity: O(n^2) für n Eckpunkte des Polygons, da n
Ecken entfernt werden müssen, wobei nach jedem Entfernen neu
aktualisiert werden muss,
       Auxiliary Space: O(n) für n Eckpunkte des Polygons'''
    if tries>1:
        raise Exception()
    try:
        polygon_vertices = deepcopy(old_polygon_vertices)
        triangles = []

        # Winkel <= 180°
        convex_vertices = []
        # Winkel > 180°
        reflex_vertices = []
        # convex + zusammen mit den Nachbarn wird ein Dreieck
gebildet, in welchem keine anderen Punkte liegen
        ears = []

        for i in range(len(polygon_vertices)):
            # Polygon baut sich gegen den Uhrzeigersinn auf, bei
einer Linksdrehung ist der Eckpunkt convex
            vertex = polygon_vertices[i]
            result = convex_or_reflex_vertex(polygon_vertices[i-1],
vertex, polygon_vertices[(i+1) % len(polygon_vertices)])

            if result == 'convex':
                convex_vertices.append(vertex)
            elif result == 'reflex':
                reflex_vertices.append(vertex)

        # Herausfinden der ears aus den convex Eckpunkten
        for vertex in convex_vertices:
            i = polygon_vertices.index(vertex)
            triangle_vertices = [polygon_vertices[i-1],
polygon_vertices[i], polygon_vertices[(i+1) %
len(polygon_vertices)]]
            if no_point_in_triangle(triangle_vertices,
polygon_vertices):
```

```
                ears.append(vertex)

        while len(polygon_vertices) > 3:
            # Entnehmen des erstmöglichen Dreiecks
            ear = ears.pop()
            i = polygon_vertices.index(ear)
            neighbor1 = polygon_vertices[i-1]
            neighbor2 = polygon_vertices[(i+1) %
len(polygon_vertices)]
            neighbors = [neighbor1, neighbor2]
            triangles.append([ear, neighbor1, neighbor2])
            polygon_vertices.remove(ear)

            # die benachbarten Eckpunkte müssen neu kategorisiert
werden
            for neighbor in neighbors:
                i = polygon_vertices.index(neighbor)

                # neuer Typ wird ermittelt
                neighbor_type =
convex_or_reflex_vertex(polygon_vertices[i-1], neighbor,
polygon_vertices[(i+1) % len(polygon_vertices)])

                if neighbor_type == 'convex':
                    # falls der Punkt vorher nicht convex war, wird
er umgeschrieben
                    if neighbor not in convex_vertices:
                        reflex_vertices.remove(neighbor)
                        convex_vertices.append(neighbor)
                    # Überprüfung ob der Punkt ear ist
                    triangle_vertices = [polygon_vertices[i-1],
neighbor, polygon_vertices[(i+1) % len(polygon_vertices)]]
                    # Punkt ist ein ear
                    if no_point_in_triangle(triangle_vertices,
polygon_vertices):
                        # Punkt ist nicht schon bereits ear
                        if neighbor not in ears:
                            ears.append(neighbor)
                    # Punkt hat Status ear verloren
                    elif neighbor in ears:
                        ears.remove(neighbor)
                # falls der Punkt vorher nicht reflex war, wird er
umgeschrieben
                elif neighbor_type == 'reflex' and neighbor not in
reflex_vertices:
                    convex_vertices.remove(neighbor)
                    reflex_vertices.append(neighbor)

        triangles.append(polygon_vertices)
        return triangles

    except Exception:
```

```
        # Polygon ist in falscher Richtung aufgebaut, Konflikt bei
der Betrachtung von Innen- und Außenwinkeln, Neuversuch
        polygon_vertices.reverse()
        return ear_clipping(polygon_vertices, tries=tries+1)
```

5.12 area_settlement_approximation

```
def area_settlement_approximation(polygon_vertices, circle_anchor,
visualize=False):
    '''Ermittelt die Siedlungen, die in einen Kreis vom Radius 85
passen'''
    '''Time Complexity: O(n) für n Eckpunkte, Auxiliary Space:
O(d^2) für d als maximale Distanz im Polygon, äquivalent zur Anzahl
Siedlungen'''
    polygon_edges = create_edges(polygon_vertices) # Zeitkomplexität
O(n) für n Eckpunkte des Polygons
    settlements = place_inner_settlements(circle_anchor) #
Zeitkomplexität O(1)
    optimal_settlements = optimize_rotation(polygon_edges,
settlements, circle_anchor, precision=0) # Zeitkomplexität O(n) für
n Eckpunkte des Polygons
    optimal_settlements = remove_outside_settlements(polygon_edges,
optimal_settlements) # Zeitkomplexität O(n) für n Eckpunkte des
Polygons

    if visualize:
        visualize_polygon(polygon_vertices,
circle_anchor=circle_anchor, settlements=optimal_settlements)

    return len(optimal_settlements)
```

5.13 get_circle_intersection_points

```
def get_circle_intersection_points(polygon_vertices, circle_anchor):
    '''Bestimmt die Schnittpunkte des Kreises und Polygons in O(n*m)
für n Seiten bzw. Eckpunkte und das Lösen des Gleichungssystems in
m'''
    '''Time Complexity: O(n*m) für n Eckpunkte des Polygons und m
als Zeitkomplexität der solve Funktion aus sympy, Auxiliary Space:
O(n+s) für n Eckpunkte des Polygons und s Schnittpunkte'''
    global decimal_precision
    sols = []
    intersection_edges_point = dict()

    circle_x, circle_y = circle_anchor
    r = 85
    x, y = symbols ('x y', real=True)
    circle_equation = Eq((x-circle_x)**2+(y-circle_y)**2, r**2)

    polygon_edges = create_edges(polygon_vertices)
    i = 0
    while i < len(polygon_edges):
```

```
        edge = polygon_edges[i]
        x1, y1 = edge[0]
        x2, y2 = edge[1]
        if x2-x1 == 0:
            edge_equation = Eq(x, x1)
        else:
            slope = (y2-y1) / (x2-x1)
            edge_equation = Eq(y - y1, slope * (x - x1))
        potential_sol = solve((circle_equation, edge_equation), (x,
y))

        sol = []
        for s in potential_sol:
            s_x = s[0]
            s_y = s[1]
            if min(x1,x2) <= s_x and s_x <= max(x1,x2) and
min(y1,y2) <= s_y and s_y <= max(y1,y2):
                sol.append(s)


        if len(sol) == 2:
            sol_1 = sol[0]
            sol_2 = sol[1]
            new_point = [(sol_1[0]+sol_2[0])/2,
(sol_1[1]+sol_2[1])/2]
            polygon_vertices.insert(i+1, new_point)
            polygon_edges = create_edges(polygon_vertices)
            i-=1

        elif len(sol) == 1:
            if is_point_in_circle(edge[0], circle_anchor) or
is_point_in_circle(edge[1], circle_anchor):
                x_sol = custom_round(sol[0][0], decimal_precision)
                y_sol = custom_round(sol[0][1], decimal_precision)
                new_sol = (x_sol,y_sol)
                sols += [new_sol]

                edge_str =
str(custom_round(edge[0][0],decimal_precision)) + ' ' +
str(custom_round(edge[0][1],decimal_precision)) + '_' +
str(custom_round(edge[1][0],decimal_precision)) + ' ' +
str(custom_round(edge[1][1],decimal_precision))
                intersection_edges_point[edge_str] = new_sol
        i+=1


    return sols, circle_anchor, intersection_edges_point,
polygon_vertices
```

5.14 get_circle_area

```
def get_circle_area(polygon_vertices, circle_anchor,
visualize=False, visualize_result=False):
    '''Berechnet die Schnittfläche eines Polygons und Kreises des
Radius 85'''
    '''Time Complexity: O(m*n^2) für m Polygone mit n Eckpunkten,
Auxiliary Space: O(e+s+z) für e Eckpunkt des Polygons im Kreis, s
Schnittpunkte und z zusätziche Punkte'''
    global decimal_precision

    intersection_points, _, intersection_edges_point,
polygon_vertices = get_circle_intersection_points(polygon_vertices,
circle_anchor)
    polygon_edges = create_edges(polygon_vertices)

    # Es gibt höchstens einen Berührungspunkt
    if len(intersection_points) <= 1:
        # Polygon liegt im Kreis
        if euclidean_distance(polygon_vertices[0], circle_anchor) <
85:
            return calculate_polygon_area(polygon_vertices)
        # Polygon umschließt den Kreis
        elif is_point_in_polygon(polygon_edges, circle_anchor):
            return (85**2)*math.pi
        # Poly
        else:
            return 0

    # Alle Winkel der Schnittpunkte werden errechnet
    angles = []
    for point in intersection_points:
        angle = angle_of_point(circle_anchor,
(point[0]/(10**decimal_precision),point[1]/(10**decimal_precision)))
        angles.append(angle)
    all_angles = []
    new_angles = []
    angles = sorted(angles)

    # Zwischen die Schnittpunkte werden zu untersuchende Punkte
positioniert
    for i, angle in enumerate(angles):
        new_angle = (angle + angles[(i+1) % len(angles)]) / 2
        if len(new_angles) != 0:
            if new_angle <= max(new_angles):
                new_angle += math.pi
        new_angles.append(new_angle)
        angle_tuple = (angle, angles[(i+1) % len(angles)],
new_angle)
        all_angles.append(angle_tuple)
    if visualize:
        visualize_circle(all_angles=all_angles,
circle_anchor=circle_anchor)
```

```
    # Bei korrektem Zwischenpunkt wird ein Intervall bestehend aus
den anliegenden Schnittpunkten gespeichert
    intervals = []
    correct_test_points = []
    for angle_tuple in all_angles:
        test_angle = angle_tuple[2]
        test_point =
rotate_point((85+circle_anchor[0],circle_anchor[1]),circle_anchor,te
st_angle)
        if is_point_in_polygon(polygon_edges, test_point):
            intervals.append((angle_tuple[0], angle_tuple[1]))
            correct_test_points.append(test_point)
    if visualize:
        visualize_circle(all_angles=all_angles,
correct_test_points=correct_test_points,
polygon_vertices=polygon_vertices, circle_anchor=circle_anchor)


    # die Intervalle, gekennzeichnet durch Winkel, sollen zu Punkten
transformiert werden
    intervals_points = []
    # das zugehörige Intervall soll durch beide Schnittpunkte
abfragbar sein
    intersection_to_interval = dict()

    for interval in intervals:
        point1 =
rotate_point((circle_anchor[0]+85,circle_anchor[1]),circle_anchor,
interval[0])
        point2 =
rotate_point((circle_anchor[0]+85,circle_anchor[1]),circle_anchor,
interval[1])

        point1 = (custom_round(point1[0], decimal_precision),
custom_round(point1[1], decimal_precision))
        point2 = (custom_round(point2[0], decimal_precision),
custom_round(point2[1], decimal_precision))

        intervals_points.append((point1,point2))

        point1 = str(point1[0]) + ' ' + str(point1[1])
        intersection_to_interval[point1] = interval

        point2 = str(point2[0]) + ' ' + str(point2[1])
        intersection_to_interval[point2] = interval


    # Punkte außerhalb des Kreises sollen abgeschnitten werden
    vertices_inside_circle = []
    for i, vertex in enumerate(polygon_vertices):
        # erster Punkt liegt im Kreis
```

```
        bool_1 = euclidean_distance(vertex, circle_anchor) < 85
        if bool_1:
            vertices_inside_circle.append(vertex)
        # zweiter Punkt liegt im Kreis
        bool_2 =
euclidean_distance(polygon_vertices[(i+1)%len(polygon_vertices)],
circle_anchor) < 85
        # ein Punkt liegt innerhalb, der andere außerhalb, folglich
existiert ein Schnitt
        if bool_1 != bool_2:
            # es wird der Schnittpunkt gesucht
            edge = [vertex,
polygon_vertices[(i+1)%len(polygon_vertices)]]
            edge_str =
str(custom_round(edge[0][0],decimal_precision)) + ' ' +
str(custom_round(edge[0][1],decimal_precision)) + '_' +
str(custom_round(edge[1][0],decimal_precision)) + ' ' +
str(custom_round(edge[1][1],decimal_precision))

            intersection_point = intersection_edges_point[edge_str]
            # es wird die Nummer des Intervalls, zu dem der Punkt
gehört, gesucht
            intersection_num = -1
            for j, interval_point in enumerate(intervals_points):
                point1 = interval_point[0]
                point2 = interval_point[1]
                if (abs(point1[0] - intersection_point[0]) <
decimal_precision*10 and abs(point1[1] - intersection_point[1]) <
decimal_precision*10) or (abs(point2[0] - intersection_point[0]) <
decimal_precision*10 and abs(point2[1] - intersection_point[1]) <
decimal_precision*10):
                    intersection_num = j
            if intersection_num == -1:
                raise Exception()
            # der Schnittpunkt wird zusammen mit seiner
Intervallnummer gespeichert
            vertices_inside_circle.append((intersection_point[0],
intersection_point[1], intersection_num))

    # wenn zwei Schnittpunkte nebeneinander liegen, wird dies
gekennzeichnet, indem die Intervallnummer zu -1 geändert wird
    for i, vertex in enumerate(vertices_inside_circle):
        next_vertex =
vertices_inside_circle[(i+1)%len(vertices_inside_circle)]
        if isinstance(vertex, tuple) and isinstance(next_vertex,
tuple):
            if vertex[2] == next_vertex[2]:
                vertices_inside_circle[i] = (vertex[0],vertex[1],-1)
                vertices_inside_circle[(i+1)%len(vertices_inside_cir
cle)] = (next_vertex[0],next_vertex[1],-1)

    # einzelne Teilflächen werden entnommen
```

```
    polygons = []
    silent_interval = []
    while len(vertices_inside_circle) > 0:
        end = True
        unique_intervals = set()
        # überprüft, ob das Endstadium erreicht ist
        for vertex in vertices_inside_circle:
            if isinstance(vertex, tuple) and vertex[2] != -1 and
vertex[2] not in silent_interval:
                unique_intervals.add(vertex[2])
                if len(unique_intervals) > 0:
                    end = False
                    break
        if end:
            polygons += [vertices_inside_circle]
            vertices_inside_circle = []
            break

        # es wird eine Teilfläche gesucht
        objective = -2
        objective_i = -2
        i = 0
        while True:
            if i>len(vertices_inside_circle)*2-2:
                break
            vertex =
vertices_inside_circle[i%len(vertices_inside_circle)]

            if isinstance(vertex, tuple) and vertex[2] != -1 and
vertex[2] not in silent_interval:
                if objective == vertex[2]:
                    if abs(i-objective_i) == 1 and objective not in
silent_interval:
                        # Intervall bildet keine eigene Teilfläche
ab
                        silent_interval.append(objective)
                        continue
                    # Intervall begrenzt eine Teilfläche und wird
separiert
                    if i<len(vertices_inside_circle):
                        polygons +=
[vertices_inside_circle[objective_i:i+1]]
                        vertices_inside_circle =
vertices_inside_circle[:objective_i] + vertices_inside_circle[i+1:]
                        break
                    else:
                        polygons +=
[vertices_inside_circle[:i%len(vertices_inside_circle)+1] +
vertices_inside_circle[objective_i:]]
                        vertices_inside_circle =
vertices_inside_circle[i%len(vertices_inside_circle)+1:objective_i]
                        break
```

```
                objective = vertex[2]
                objective_i = i
            i+=1

    # Genaugkeit der Platzierung der Extra-Punkte zwischen 2
Schnittpunkten
    distance = 30
    angle_diff = math.pi/distance

    # für jedes Polygon werden die Intervalle der Kreisschnittpunkte
mit Extra-Punkten gefüllt
    for i, polygon in enumerate(polygons):
        complicated = False
        for vertex in polygon:
            if isinstance(vertex, tuple):
                if vertex[2] != -1:
                    complicated = True
                    break

        new_polygon = deepcopy(polygon)

        # Polygon beinhaltet mehrere Schnittpunktsegmente
        if complicated:
            j = len(new_polygon)-2

            if isinstance(polygon[0], tuple):
                intersection_1 = polygon[0]
                intersection_2 = polygon[-1]
                segment = get_intersection_segment(intersection_1,
intersection_2, intersection_to_interval, angle_diff, circle_anchor)

                new_polygon += segment

            while j > 1:
                if isinstance(new_polygon[j], tuple):
                    intersection_1 = [new_polygon[j][0],
new_polygon[j][1]]
                    intersection_2 = [new_polygon[j-1][0],
new_polygon[j-1][1]]
                    segment =
get_intersection_segment(intersection_1, intersection_2,
intersection_to_interval, angle_diff, circle_anchor)

                    new_polygon = new_polygon[:j] + segment +
new_polygon[j:]
                    j-=1
                j -= 1
        # Polygon beinhaltet nur 1 Schnittpunktsegment
        else:
            if isinstance(polygon[0], tuple) and
isinstance(polygon[-1], tuple):
                j = len(new_polygon)-2
```

```
                intersection_1 = polygon[0]
                intersection_2 = polygon[-1]
                segment = get_intersection_segment(intersection_1,
intersection_2, intersection_to_interval, angle_diff, circle_anchor)
                new_polygon += segment
                stop = 1
            else:
                j = len(new_polygon)-1
                stop = 0
            while j > stop:
                if isinstance(new_polygon[j], tuple):
                    intersection_1 = [new_polygon[j][0],
new_polygon[j][1]]
                    intersection_2 = [new_polygon[j-1][0],
new_polygon[j-1][1]]
                    segment =
get_intersection_segment(intersection_1, intersection_2,
intersection_to_interval, angle_diff, circle_anchor)

                    new_polygon = new_polygon[:j] + segment +
new_polygon[j:]
                    j-=1
                j -= 1
        polygons[i] = new_polygon

    # Visualisierung des Polygons aufgebaut durch die einzelnen
Teilflächen
    if visualize or visualize_result:
        circle = plt.Circle(circle_anchor, 85, color='b',
fill=False)
        ax = plt.gca()
        for polygon in polygons:
            if len(polygon) > 0:
                for i, vertex in enumerate(polygon):
                    if isinstance(vertex, tuple):
                        polygon[i] =
[vertex[0]/(10**decimal_precision),
vertex[1]/(10**decimal_precision)]

                x, y = zip(*(polygon + [polygon[0]]))
                ax.plot(x, y)
        ax.set_aspect('equal')
        ax.add_patch(circle)
        plt.show()

        ax = plt.gca()
        ax.set_aspect('equal')
        circle = plt.Circle(circle_anchor, 85, color='b',
fill=False)
        ax.add_patch(circle)
        for polygon in polygons:
            triangles = ear_clipping(polygon)
```

```
            for triangle in triangles:
                x, y = zip(*(triangle + [triangle[0]]))
                plt.plot(x,y)
        plt.show()


    # Berechnung des insgesamten Flächeninhalts
    area = calculate_multiple_polygon_area(polygons)

    return area
```

5.15 get_intersection_segment

```
def get_intersection_segment(intersection_1, intersection_2,
intersection_to_interval, angle_diff, circle_anchor):
    '''Erstellt zusätzliche Punkte zwischen zwei Schnittpunkten des
Kreises'''
    '''Time Complexity: O(w*d) für w als Winkeldifferenz zwischen
denen zweier zusätzlicher Punkten und d als Differenz der Winkel der
Schnittpunkte,
       Auxiliary Space: äquivalent zur Zeitkomplexität, da pro
Zeiteinheit genau ein Punkt geschaffen wird'''
    # Intervall wird abgefragt
    intersection_str = str(intersection_1[0]) + ' ' +
str(intersection_1[1])
    interval = None
    try:
        interval = intersection_to_interval[intersection_str]
    except KeyError:
        for key, value in intersection_to_interval.items():
            num_key = key.split()
            if abs(int(num_key[0]) - intersection_1[0]) <
decimal_precision*10 and abs(int(num_key[1]) - intersection_1[1]) <
decimal_precision*10:
                interval = value
                break
    if interval == None:
        print('No interval')
        raise Exception()

    # Es wird der im Einheitskreis kleinere Winkel gesucht
    # Punkten werden ihr Winkel zugeordnet
    intersection_1 = (intersection_1[0]/(10**decimal_precision),
intersection_1[1]/(10**decimal_precision))
    intersection_2 = (intersection_2[0]/(10**decimal_precision),
intersection_2[1]/(10**decimal_precision))
    potentially_smaller_angle = angle_of_point(circle_anchor,
intersection_1)
    potentially_bigger_angle = angle_of_point(circle_anchor,
intersection_2)

    # Index des kleineren Winkels
```

```
    smaller_angle = 0
    if potentially_smaller_angle > potentially_bigger_angle:
        # Erster Winkel ist größer, folglich ist Index 1 der
kleinere
        smaller_angle = 1

    # Fälschlich größerer Winkel des Intervalls überschreitet 360°
und ist daher im Einheitskreis kleiner. Indexe müssen getauscht
werden
    if interval[0] > interval[1]:
        if smaller_angle == 0:
            smaller_angle = 1
        else:
            smaller_angle = 0

    # Es werden die Extra-Punkte generiert
    current = interval[0] + angle_diff
    segment = []
    stop_condition = interval[1]
    if interval[1] < interval[0]:
        stop_condition += math.pi*2
    while current < stop_condition:
        new_point =
rotate_point((85+circle_anchor[0],circle_anchor[1]), circle_anchor,
current)
        segment.append([new_point[0],new_point[1]])
        current += angle_diff

    # Segment wird in die richtige Richtung gedreht
    if smaller_angle == 0:
        reversed_segment = deepcopy(segment)
        reversed_segment.reverse()
        return reversed_segment
    else:
        return segment
```

5.16 get_facility_location

```
def get_facility_location(polygon_vertices, splits=10,
depth_finish=5, depth_stop=3, force_area_calculation=False,
visualize=False, visualize_beginning=False):
    '''Findet die beste Position des Gesundheitszentrums'''
    '''Time Complexity: O(p * O(Heuristik)), Auxiliary Space: O(h)
für h Einträge im Heap'''
    polygon_edges = create_edges(polygon_vertices)

    # Eckpunkte des rechteckigen Untersuchungsbereichs werden
ermittelt
    max_x = max(vertex[0] for vertex in polygon_vertices)
    min_x = min(vertex[0] for vertex in polygon_vertices)
    max_y = max(vertex[1] for vertex in polygon_vertices)
    min_y = min(vertex[1] for vertex in polygon_vertices)
```

```
    bottom_left = (min_x, min_y)
    bottom_right = (max_x, min_y)
    top_left = (min_x, max_y)
    top_right = (max_x, max_y)

    # Rechteckiger Untersuchungsbereich
    vertices = [bottom_left, bottom_right, top_right, top_left]

    distance_x = (max_x - min_x) / splits
    distance_y = (max_y - min_y) / splits

    # Erstellen der ersten Schicht Punkte
    points = []

    current_y = max_y-(distance_y/2)
    stop_y = min_y
    step_y = -distance_y

    stop_x = max_x
    step_x = distance_x

    while current_y > stop_y:
        current_x = min_x+(distance_x/2)
        while current_x < stop_x:
            points.append([current_x,current_y])
            current_x += step_x
        current_y += step_y


    # Heuristic für das Vergleichen der Kreispotenziale
    if force_area_calculation:
        get_circle_value = get_circle_area
    else:
        get_circle_value = area_settlement_approximation # O(n*m)


    if depth_stop >= depth_finish:
        depth_stop == depth_finish-1

    areas = []
    # Einfügen der Untersuchungspunkte in einen Heap
    # [Heuristic, Tiefe, ID, Koordinaten]
    max_heap = deepcopy(points)
    id = 0
    for i, point in enumerate(max_heap):
        #print(point)
        point_data = deepcopy(point)
        try:
            area = get_circle_value(polygon_vertices, point,
visualize=False)
        except Exception:
```

```
            print(f'error: {point}')
            area = 0
        point_data = [-area, 1, id] + [point_data]
        max_heap[i] = point_data
        areas.append(int(area))
        # Verhindert, dass der Heap bei gleicher Fläche und Tiefe
möglicherweise im nächsten Index unvergleichbare Datentypen zu
vergleichen versucht
        id += 1

    elapsed_time_during_visualization = None
    if visualize or visualize_beginning:
        start = time.time()
        visualize_grid(polygon_vertices, vertices, points,
areas=areas)
        end = time.time()
        elapsed_time_during_visualization = end - start

    heapify(max_heap)

    while True:
        # Daten des derzeitig wertvollsten Punktes
        area, depth, _, point = heappop(max_heap)

        if depth == depth_finish:
            # Lösung zurückgeben
            return point, elapsed_time_during_visualization

        # zu untersuchender Punkt
        x = point[0]
        y = point[1]

        # 9 neue Punkte werden in einem Gitter platziert, die
Abstände sind halb so groß wie die der vorherigen Tiefe
        current_y = y + distance_y/(2**(depth))
        for _ in range(3):
            current_x = x - distance_x/(2**(depth))
            for _ in range(3):
                new_point = [current_x, current_y]
                # Punkt muss um Untersuchungsbereich liegen
                if point_in_area(new_point, max_x, min_x, max_y,
min_y):
                    try:
                        area = get_circle_value(polygon_vertices,
new_point, visualize_result=False)
                    except Exception:
                        area = 0
                    # Punkt wird nur zum Heap hinzugefügt wenn er
nicht die Stop-Tiefe überschritten hat und außerhalb des Polygons
liegt
                    if not (depth+1 > depth_stop and not
is_point_in_polygon(polygon_edges, new_point)):
```

```
                        point_data = [-area, depth+1, id] +
[new_point]
                        points.append(new_point)
                        heappush(max_heap, point_data)
                        id += 1
                current_x += distance_x/(2**(depth))
            current_y -= distance_y/(2**(depth))
        if visualize:
            visualize_grid(polygon_vertices, vertices, points)
```

5.17 main

```
def main():
    visualize = ask_for_visualization()
    polygon_vertices = read_file()

    # Starten der Laufzeitmessung
    start_time = time.time()

    # Visualisierung des Polygons
    if visualize:
        start = time.time()
        visualize_polygon(polygon_vertices)
        end = time.time()
        start_time += end - start

    # Zusätzliches Umformatieren der Eckpunkte in Seiten für
leichtere Weiterverarbeitung
    polygon_edges = create_edges(polygon_vertices)

    # Ermitteln der optimalen Position des Gesundheitszentrums
    facility_anchor, time_in_visualization =
get_facility_location(polygon_vertices, splits=70, depth_finish=5,
depth_stop=4, visualize_beginning=visualize,
force_area_calculation=False)#, debugging=False)
    if time_in_visualization != None:
        start_time += time_in_visualization

    # Visualisierung des Gesundheitszentrums
    if visualize:
        start = time.time()
        visualize_polygon(polygon_vertices,
circle_anchor=facility_anchor)
        end = time.time()
        start_time += end - start

    # Platzierung der Siedlungen
    settlements = place_settlements(facility_anchor,
polygon_vertices, visualization=False)

    # Visualisierung der aufgestellten Siedlungen
```

```
    if visualize:
        start = time.time()
        visualize_polygon(polygon_vertices, settlements=settlements,
circle_anchor=facility_anchor)
        end = time.time()
        start_time += end - start

    # Ermitteln der optimalen Rotation
    rotated_settlements = optimize_rotation(polygon_edges,
settlements, facility_anchor, 3)
    # Entfernen außerhalb liegender Siedlungen
    correct_settlements = remove_outside_settlements(polygon_edges,
rotated_settlements)

    # Stoppen der Laufzeitmessung
    end_time = time.time()

    # Visualisieren und Ausgeben der Lösung
    visualize_polygon(polygon_vertices,
settlements=correct_settlements, circle_anchor=facility_anchor)
    print('Anzahl Siedlungen: ', len(correct_settlements))
    runtime = end_time - start_time
    print('Laufzeit: ', "{:.2f}".format(runtime), "s")
```

5.18 rotate_point

```
def rotate_point(point, anchor, angle_radians):
    '''Rotiert einen Punkt um einen anderen'''
    '''Time Complexity: O(1), Auxiliary Space: O(1)'''
    point_x, point_y = point
    anchor_x, anchor_y = anchor

    # Der Punkt wird so verlegt, dass der Anchorpunkt dem Ursprung
entspricht
    translated_x = point_x - anchor_x
    translated_y = point_y - anchor_y

    s = math.sin(angle_radians)
    c = math.cos(angle_radians)

    # Ermitteln der rotierten Koordinaten
    rotated_x = translated_x * c - translated_y * s
    rotated_y = translated_x * s + translated_y * c

    # Zurückverschieben des Punktes
    final_x = rotated_x + anchor_x
    final_y = rotated_y + anchor_y

    return (final_x, final_y)
```

5.19 angle_of_point

```
def angle_of_point(anchor, point):
    '''Winkel eines Punkts im Einheitskreis'''
    '''Time Complexity: O(1), Auxiliary Space: O(1)'''
    translated_point = (point[0] - anchor[0], point[1] - anchor[1])
    x, y = translated_point

    # Sonderfälle werden behandelt
    if x == 0:
        if y > 0:
            return math.pi/2
        else:
            return math.pi/2*3
    if y == 0:
        if x > 0:
            return 0
        else:
            return math.pi

    angle = math.atan(y/x)

    # Punkt liegt im 2. oder 3. Quadranten
    if x < 0:
        angle += math.pi
    # Punkt liegt im 4. Quadranten
    if x > 0 and y < 0:
        angle += math.pi*2

    return angle
```

5.20 create_edges

```
def create_edges(vertices):
    '''Erstellt Seiten aus den n Eckpunkten des Polygons'''
    '''Time Complexity: O(n) für n Eckpunkte, Auxiliary Space: O(n)
für n Eckpunkte'''
    edges = []
    for i, _ in enumerate(vertices):
        edges.append([vertices[i], vertices[(i+1) % len(vertices)]])
    return edges
```

5.21 def no_point_in_triangle(triangle_vertices, polygon_vertices):

```
    '''Untersucht ob sich kein Punkt im Dreieck befindet, genutzt
für ear clipping'''
    '''Time Complexity: O(n) für n Eckpunkte des Polygons, Auxiliary
Space: O(n) für n Seiten'''
    triangle_edges = create_edges(triangle_vertices)
    for vertex in polygon_vertices:
        # Punkt entspricht einem Punkt des Dreiecks, überspringen
        if vertex in triangle_vertices:
```

```
            continue
        # Punkt liegt im Dreieck
        if is_point_in_polygon(triangle_edges, vertex):
            return False
    # Kein Punkt liegt im Dreieck
    return True
```

6. Literatur

[1] “Ray-casting algorithm”, (2024).

https://rosettacode.org/wiki/Ray-casting_algorithm

[2] David Eberly, “Triangulation by Ear Clipping”, (2023)

https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

[3] Shiva Chaudhuri, Naveen Garg, R. Ravi, “Generalized k -Center Problems”

https://pure.mpg.de/rest/items/item_1827380_4/component/file_2574029/content