Institut für Formale Methoden der Informatik

Abteilung Algorithmik

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2455

Algorithmen zur Optimierung von Packungsproblemen

Markus Hildinger

Studiengang:	Dipl. Informatik
Prüfer:	Prof. Dr.-Ing. Stefan Funke
Betreuer:	Prof. Dr.-Ing. Stefan Funke
begonnen am:	27.03.2014
beendet am:	26.09.2014
CR-Klassifikation:	D.2.2, D.2.3, D.2.6, E.1

Inhaltsverzeichnis

1 Aufgabenstellung

In vielen industriellen Anwendungsgebieten sind geometrische Packungsprobleme zu lösen, so möchte man zum Beispiel bei der Verarbeitung von Blech oder Stoffen oft den Verschnitt minimieren. Dass heißt, es wird eine Form vorgegeben, die möglichst oft auf einer größeren Fläche platziert werden soll. Die platzierten Formen müssen komplett sein, dass heißt, sie dürfen sich nicht überschneiden und nicht über den Rand hinausragen.

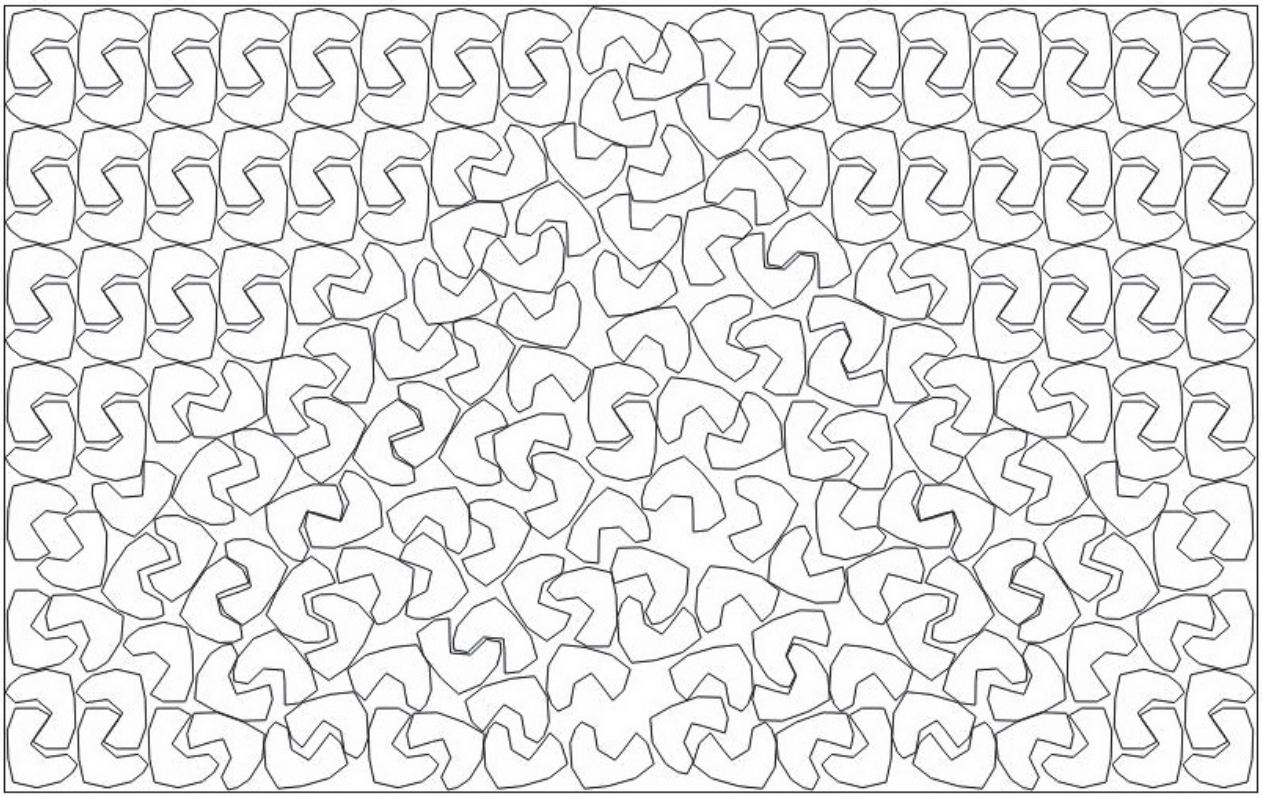


Abb.1: Beispiel eines Packungsproblems

Packungsprobleme gehören zur Klasse der NP-vollständigen Probleme. Dies bedeutet, es gibt keinen effizienten Algorithmus (Polynomialzeit-Algorithmus), der in jedem Fall eine optimale Lösung liefern kann, es sei denn, es kann bewiesen werden, dass die Komplexitätsklasse NP gleich der Komplexitätsklasse P ist. Bisher gibt es allerdings weder einen Beweis für die Gleichheit noch einen dagegen. Da für keines der vielen bekannten NP-schweren Probleme bislang ein effizienter Algorithmus gefunden worden ist, liegt die Vermutung jedoch nahe, dass die Komplexitätsklassen nicht gleich sind.
Da also keine optimale Lösung in vertretbarer Zeit gefunden werden kann, gilt es, Algorithmen zu entwickeln, welche Ergebnisse liefern, die einer optimalen Lösung so nahe wie möglich kommen.
Ziel dieser Studienarbeit ist es, solche Algorithmen zu entwickeln.
Um die Komplexität des Problems im Rahmen zu halten, wurden einige Vereinfachungen vorgenommen. So wird das Feld, auf dem die Figuren platziert werden sollen, stets ein Rechteck sein. Des weiteren wird es auch nur eine Figur geben, die auf der Fläche verteilt werden kann und, damit keine gekrümmten Begrenzungen beachtet werden müssen, muss die Figur ein Polygon sein.

2 Grundlagen Funktionen

Da, wie bereits erwähnt, aus Gründen der Vereinfachung, in dieser Arbeit ausschließlich Polygone als Figuren zugelassen sind, gilt es zunächst einige Funktionen für den Umgang mit dieser Datenstruktur zu betrachten. Ein Polygon wird als eine Liste aus Punkten gespeichert, sodass der Eintrag $List_{(i)}$ verbunden ist mit den Einträgen $List_{(i-1)}$ und $List_{(i+1)}$. Das letzte Element wird zusätzlich noch mit dem ersten verbunden, um das Polygon zu schließen.

2.1 Positionierung des Polygons

In erster Linie wird es darum gehen, Polygone in allen erdenklichen Positionen auf einem Feld anzuordnen. Deshalb sollte ein gegebenes Polygon verschoben, gedreht und gespiegelt werden können.

Zum Verschieben eines Polygons im zweidimensionalen Raum genügt es bei der oben genannten Darstellung, jeden Punkt des Polygons um ein Δx in x-Richtung und ein Δy in y-Richtung zu verschieben. Auf diese Weise kann jeder Punkt auf dem Feld erreicht werden.

Für eine Drehung des Polygons muss zunächst jedoch festgelegt werden, um welchen Punkt das Polygon gedreht werden sollte. Hier scheint es sich zunächst anzubieten, einen der Eckpunkte des Polygons zu wählen, da diese Punkte bereits bekannt sind und der gewählte Punkt bei einer Drehung gleich bleibt, sodass ein Punkt weniger gedreht werden muss.
Einen großen Nachteil gibt es dabei jedoch. Wird ein Polygon an die Stelle (x,y) geschoben und dort um 180° um einen seiner Eckpunkte gedreht, werden sich die Lagen der beiden Polygone in vielen Fällen deutlich voneinander unterscheiden. Dass heißt, es lässt sich aus der Stelle (x,y) nicht sonderlich viel über die eigentliche Lage des Polygons sagen.
Aus diesem Grund wird in dieser Arbeit der geometrische Schwerpunkt oder auch Flächenschwerpunkt als Angelpunkt für eine Drehung gewählt.
Es handelt sich dabei gleichzeitig um den Massenmittelpunkt, das heißt, teilt man das Polygon entlang einer beliebigen Geraden durch diesen Punkt, werden beide Teile die gleiche Größe haben. Nach einer Drehung um diesen Punkt wird der geometrische Schwerpunkt des gedrehten Polygons gleich dem des Ausgangspolygons sein. Die Lage dieser Polygone wird sich also nur minimal voneinander unterscheiden. Aus diesem Grund wird im weiteren Verlauf der geometrische Schwerpunkt eines Polygons als Drehangelpunkt verwendet, und gleichzeitig wird durch ihn auch die Position des Polygons auf dem Feld bestimmen.

Für manche Anwendungsgebiete, in denen das Material keine unterschiedlichen Ober- und Unterseiten hat, kann das Polygon zusätzlich zur Drehung noch gespiegelt werden. Da eine Punktspiegelung einer Drehung um 180° entspricht, muss eine Spiegelung an einer Achse vorgenommen werden. Vor allem bei sehr asymmetrischen Polygonen werden sich diese gespiegelten Polygone sehr von den nicht gespiegelten unterscheiden (siehe Abb.2).
Um auch bei der Spiegelung die Lage des Polygons nicht allzu sehr zu verfälschen, sollte das Polygon an einer Achse gespiegelt werden, die durch den geometrischen Schwerpunk verläuft. Auch nach einer solchen Spiegelung wird sich der ursprüngliche geometrische Schwerpunkt nicht von dem nach der Spiegelung unterscheiden. Gespiegelt wird im Folgenden immer an der Senkrechten.

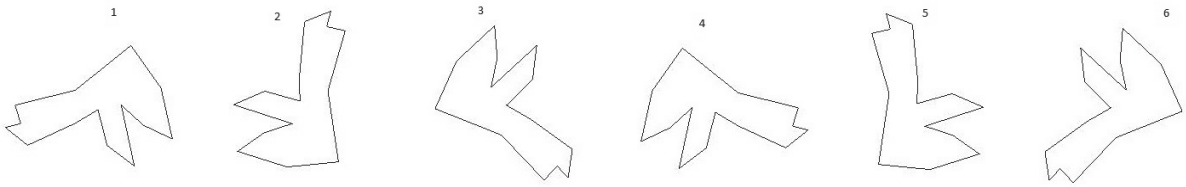


Abb.2: Ausgangspolygon 1 rotiert zu 2 und 3. 4, 5 und 6 sind Spiegelungen von 1, 2 und 3 an einer senkrechten Achse

2.2 Schnitttests

Da wir nun die Polygone auf dem Feld in unterschiedlichsten Positionen anordnen können, gilt es als nächstes dafür zu sorgen, dass sie sich nicht gegenseitig überschneiden oder über den Rand hinausragen, da wir im Ergebnis nur vollständige Figuren erhalten möchten.

Der einfache Fall ist hierbei die Randbetrachtung:
Da als Figuren nur Polygone betrachtet werden, sind alle Kantensegmente Geraden. Das Feld wird stets ein Rechteck sein und deshalb ist es nicht möglich, dass ein Kantensegment eines Polygons den Rand schneidet, wenn seine beiden Endpunkte innerhalb des Feldes liegen. Aus diesem Grunde wird ein Polygon genau dann über den Rand hinausragen, wenn mindestens einer der Punkte auf der falschen Seite der Begrenzungslinien des Feldes liegt. Somit muss bei der Randbetrachtung jeder Punkt des Polygons mit den vier Begrenzungslinien des Feldes verglichen werden. Wird ein Punkt gefunden, der außerhalb liegt, befindet sich das Polygon entweder komplett außerhalb des Feldes oder es schneidet den Rand. In beiden Fällen kommt es nicht als Teil der Lösung in Frage.

Der Test, ob sich zwei Polygone überschneiden, ist nicht ganz so einfach. Für die Lösung dieses Problems sind drei unterschiedliche Algorithmen entstanden:
Ein Linearzeit-Algorithmus, der nur mit konvexen Polygonen funktioniert, ein Sweepline-Algorithmus der in O(nlogn) auf beliebigen Polygonen arbeitet und ein naiver $O(n^2)$ Algorithmus, ebenfalls auf beliebigen Polygonen. Sollten sich zwei Polygone an einem Punkt oder einer Geraden berühren, wird dies auch als Überschneidung betrachtet.
Diese Algorithmen werden nur oberflächlich beschrieben, da die beiden ersten im Programm nicht genutzt wurden und der naive Algorithmus wenig an Erklärung benötigt.

Zentraler Baustein bei der Berechnung, ob zwei Polygone sich schneiden, wird die Frage nach dem Schnitt zweier Segmente sein. Sowohl beim naiven, als auch beim Sweepline-Algorithmus werden die Polygone mehr oder weniger in Segmente zerlegt und die Segmente dann miteinander verglichen.
Es ist schnell klar, dass die folgende Aussage zutrifft:
Zwei gerade Segmente schneiden sich genau dann, wenn für beide Segmente gilt, dass ihre jeweiligen Endpunkte auf gegenüberliegenden Seiten der vom jeweils anderen Segment aufgespannten Geraden liegen.
Die Umsetzung dieser Idee findet sich im Kapitel Implementierung.

2.2.1 Linearzeit-Algorithmus auf konvexen Polygonen

Die Frage, ob ein Polygon konvex ist lässt sich anschaulich an einem Bild beschreiben. Stellen

wir uns die Eckpunkte des Polygons als Nägel vor, die aus einem Brett ragen. Nun legen wir ein Gummi um diese Nägel, sodass alle Nägel umschlossen sind. Lassen wir das Gummi los, zieht es sich so weit wie möglich zusammen. Verläuft das Gummi nun entlang der Kanten des Polygons und alle Punkte werden vom Gummi berührt, so ist das Polygon konvex.
Eine andere, etwas theoretischere Betrachtung wäre folgende: Beginne bei einem beliebigen Punkt des Polygons und laufe entlang der Kanten immer in die gleiche Richtung. Das Polygon ist konvex, falls bei allen Punkten eine Linkskurve oder bei allen eine Rechtskurve gelaufen werden muss.

Die Voraussetzung für den folgenden Algorithmus ist, dass beide Polygone konvex sind.
Die Idee ist nun, ein „oberes“ und ein „unteres“ Polygon zu definieren und dann zu testen, ob einer der Punkte des oberen Polygons unterhalb der oberen Begrenzung des unteren Polygons liegt oder umgekehrt.
Zunächst wird bei beiden Polygonen der Punkt mit dem kleinsten x-Wert gesucht. Nun kann das Polygon mit dem kleineren Wert (poly1) von links nach rechts an der oberen und der unteren Begrenzungslinien entlanggegangen werden bis der Punkt mit dem kleinsten x-Wert des anderen Polygons (poly2) erreicht wird. Hier gilt es nun 4 Fälle zu unterscheiden:
Fall 1: Der erste Punkt von poly2 liegt unterhalb der beiden Begrenzungslinien von poly1
Fall 2: Der Punkt liegt oberhalb der beiden Begrenzungslinien
Fall 3: Der Punkt liegt zwischen den beiden Begrenzungslinien
Fall 4: Der Punkt liegt auf einer der beiden Begrenzungslinien
Da wir stets nur Polygone von gleicher Form betrachten, bedeutet der dritte Fall automatisch, dass sich die beiden Polygone schneiden. Ein Polygon nämlich niemals komplett in einem anderen Polygon gleicher Größe enthalten sein kann.
Im vierten Fall liegt natürlich auch eine Überschneidung vor, da sich die Polygone nicht berühren dürfen.
Die Fälle eins und zwei sind identisch, aber spiegelverkehrt. Deshalb wird im folgenden nur auf Fall 1 eingegangen. Hierbei liegt nun also der erste Punkt von poly2 unterhalb von poly1. Das heißt zunächst, wir müssen nur noch die untere Begrenzungslinie von poly1 betrachten und die Punkte auf der oberen Begrenzungslinie von poly2. Wir laufen also nun parallel entlang der beiden Polygone und testen für jeden Punkt aus poly2, ob er oberhalb der Begrenzungslinie von poly1 liegt. Wird ein solcher Punkt gefunden, schneiden sich die Polygone. Sind wir am rechten Rand eines der beiden Polygone angelangt, dann schneiden sich die beiden Polygone nicht und der Algorithmus kann beendet werden.

2.2.2 Sweepline-Algorithmus in O (nlogn)

Beim Sweepline Algorithmus werden die Punkte der beiden zu vergleichenden Polygone sortiert und von links nach rechts in einer Event-Liste gespeichert (x-Struktur). Bei einem senkrechen Segment werden beide Endpunkte an der gleichen x-Stelle liegen, das heißt, es wird in diesem Fall zwei Events mit dem gleichen x-Wert geben. Deshalb muss dafür gesorgt werden, dass das Segment beim zuerst bearbeiteten Event in die y-Struktur aufgenommen und beim zweiten Event wieder entfernt wird. Es muss also einmal als rechtes und einmal als linkes Segment betrachtet werden.
Das Durchlaufen dieser Event-Liste kann man sich als eine senkrechte Linie vorstellen, die von links nach rechts gezogen wird; daher auch der Name des Algorithmus. Während die Sweepline über die Polygone wandert, werden alle Segmente, die die Sweepline schneiden, in einer Baumstrukturen gespeichert (y-Struktur). Für jedes Polygon wird eine solche Baumstruktur aufgebaut.
Wird ein neues Event erreicht, wird der entsprechende Punkt dem richtigen Polygon zugeord-

net. Danach gilt es 3 Fälle zu unterscheiden:
Fall 1: Beide Nachbarn liegen rechts der Sweepline
Fall 2: Ein Nachbar liegt rechts, einer liegt links der Sweepline
Fall 3: Beide Nachbarn liegen links der Sweepline
Im ersten Fall werden zwei neue Segmente in die y-Struktur aufgenommen. Anhand des y-Wertes des Punktes, für welchen das Event erstellt wurde, kann die y-Struktur durchlaufen und die neuen Segmente können an der richtigen Position eingefügt werden. Gleichermaßen wird die y-Struktur des anderen Polygons durchlaufen, um diejenigen Segmente zu finden, die am nächsten zu den neu eingefügten Segmenten liegen. Diese Segmente werden dann auf Schnitt mit den eingefügten Segmenten verglichen.
Im Zweiten Fall kann das Segment, welches von der Sweepline aus nach links geht, einfach durch das neue, welches nach rechts geht ersetzt werden. Da sich die Segmente, die zu einem Polygon gehören nicht kreuzen, bleibt auch nach dem Ersetzen die Baumstruktur korrekt und es muss nicht umsortiert werden. Jeweils das alte und das neue Segment werden wieder mit den „Nachbarn“ aus der y-Struktur des andren Polygons auf Schnitt getestet.
Im dritten Fall werden für die beiden zu entfernenden Segmente noch einmal die Schnitttests ausgeführt, bevor sie aus der y-Struktur gelöscht werden. In diesem Moment bekommen die benachbarten Segmente aus der anderen y-Struktur neue „Nachbarn“, da die alten eben entfernt wurden. Auch für diese neu entstandenen Paare müssen Schnitttests durchgeführtwerden.

2.2.3 Naiver Algorithmus in O (n²)

Der naive Algorithmus arbeitet, wie der Name schon sagt, ohne große Raffinesse. In einer geschachtelten Schleife werden jeweils alle Segmente des einen Polygons mit allen Segmenten des anderen Polygons verglichen. Wird ein Segment-Paar gefunden, welches sich überschneidet, schneiden sich logischerweise auch die Polygone und es kann abgebrochen werden. Wird bis zum Schluss kein solches Paar gefunden, schneiden sich die Polygone nicht.
Abgesehen vom Sonderfall eines oder mehrerer senkrechter Segmente ist dieser Algorithmus sehr einfach und bietet kaum Platz für Fehler.
Da hauptsächlich Polygone mit wenigen Punkten verwendet wurden, schlägt die quadratische Laufzeit nicht so sehr zu Buche wie das bei Polygonen mit sehr vielen Punkten der Fall wäre. Um möglichen Fehlen in der Implementierung des Sweepline-Algorithmus aus dem Wege zu gehen, wurde deshalb dieser Algorithmus für die Schnitttests verwendet.

3 Lösungsansätze

Zur Lösung des Optimierungsproblems wurden zwei Ansätze betrachtet. Zunächst ein einfacher, naiver Algorithmus, der bei den meisten Eingaben eher schlechte Ergebnisse liefert. Der weitaus spannendere Ansatz beruht darauf, das Packungsproblem auf ein Graphenproblem zu übertragen, woraufhin mit bekannten Graphenalgorithmen eine Lösung errechnet werden kann.

3.1 Naiver Algoithmus

Die Grundidee des naiven Algorithmus ist es, die Polygone von oben links nach unten rechts zu stapeln. Man beginnt mit dem ersten und schiebt es so weit wie möglich in die obere linke Ecke. Im folgenden werden die weiteren Polygone von rechts an das erste angefügt, bis das nächste Polygon nicht mehr auf das Feld passt. Die so entstandene Reihe kann nun einfach ko-

piert und unterhalb der ersten erneut eingefügt werden, bis auch hier kein Platz mehr für eine weitere Reihe übrig bleibt. Am einfachsten ist dieses Vorgehen wohl zu realisieren, indem man ein minimales umschließendes Rechteck für das Polygon berechnet und die Rechtecke aneinander fügt. Auf diese Weise ergibt sich die Position des jeweils nächsten Polygons, bzw. Rechtecks fast wie von selbst.
Um möglichst viele Rechtecke auf dem Feld unterzubringen, wird das Polygon zunächst in eine möglichst optimale Position gedreht, sodass der Flächeninhalt des umschließenden Rechtecks (bestehend aus senkrechten und waagerechten Kanten) minimal ist.
Mögliche weitere Optimierungen des Algorithmus, die in dieser Arbeit nicht realisiert wurden, sind zum Beispiel:
Das Auffüllen des Randes. Falls die Rechtecke ungleiche Kantenlängen haben wäre es möglich, dass auf einer Seite noch genug Platz bleibt um weitere Rechtecke, die um 90° gedreht wurden, einzufügen.
Des weiteren könnte man schon zu Beginn testen, ob möglicherweise bei einer Drehung der Rechtecke um 90° weniger Rand übrig bleiben würde.
Auch wäre es möglich das Polygon so zu drehen, dass die Rechtecke das Feld von links nach rechts Optimal ausfüllen. Auf dies Weise könnte es sein, dass durch die verminderte Höhe der Rechtecke von oben nach unten mehr Polygone Platz finden.
Es gibt mit Sicherheit noch weitere Optimierungsmöglichkeiten für diesen einfach Algorithus, das Hauptaugenmerk dieser Arbeit wurde allerdings mehr auf die Independent Set Algorithmen gelegt.

3.2 Independent Set Algorithmen

Die Idee hinter diesen Algorithmen liegt, wie bereits erwähnt, auf der Abbildung des eigentlichen Problems auf das Maximum Independent Set Problem. Bevor die eigentlichen Algorithmen erläutert werden, wird zunächst kurz erklärt, was unter einem Independent Set verstanden werden kann und wie sich mit dessen Hilfe ein graphisches Packungsproblem lösen lässt.

3.2.1 Das Maximum Independent Set Problem

Sei G = (V,E) (V = Knoten, E = Kanten) ein ungerichteter Graph ohne Mehrfachkanten und U eine Teilmenge von V. U ist genau dann eine unabhängige Menge (oder auch ein Independent Set), wenn für alle Knoten v,w ϵ U gilt, dass sie nicht benachbart sind, also keine direkte Kante zwischen ihnen existiert.
Eine Maximale unabhängige Menge, also ein Maximum Independent Set, ist demnach eine Teilmenge U aus V für die gilt, U $\geq$ W für alle unabhängigen Teilmengen W aus V. In Worten: Es existiert keine unabhängige Teilmenge W aus V, die mehr Elemente enthält, als die unabhängige Teilmenge U. Dann ist U ein Maximum Independent Set.
Auch dieses Problem ist ein NP -schweres Problem und es gibt demnach auch hierfür keine effizienten Algrorithmen, um ein solches Maximum Independent Set zu berechnen.
Es geht also darum, möglichst gute Lösungen zu finden. Hierzu wurden in dieser Arbeit einfache Greedy-Strategien implementiert.

3.2.2 Abbildung einer Probleminstanz auf einen Konflikt-Graphen

Um Algorithmen zum Finden eines Maximum Independent Set anwenden zu können, muss zu-

nächst aus einer Probleminstanz unseres Packungsproblems ein Graph aus Knoten und Kanten entstehen.
Um möglichst keine Einschränkungen zu machen, muss das Polygon in möglichst kleinen Schritten über das Feld geschoben und an jeder Stelle für jeden möglichen Drehwinkel ein Knoten für den Graphen erstellt werden. Zusätzlich kann das Polygon auch noch gespiegelt werden und es kann für die gespiegelte Version ebenfalls an jeder Stelle mit jedem Drehwinkel ein weiterer Knoten erstellt werden.
Unser Ziel ist es, möglichst viele komplette Polygone auf dem Feld unterzubringen. Das bedeutet, dass von zwei Polygonen, die sich schneiden, höchstens eines der beiden Teil des Independent Sets sein kann. Deshalb muss die Information, ob sich die Polygone schneiden, auch im Graph enthalten sein. Dies lässt sich natürlich mit Hilfe der Kanten optimal lösen.
Für alle Knotenpaare des Graphen wird also getestet, ob sich die beiden jeweiligen Polygone überschneiden. Falls ja, wird eine Kante zwischen den Knoten erstellt, falls nein, wird keine Kante erstellt.
Den so entstandenen Graph nennt man auch einen Konfliktgraph.

Da nun klar ist, wie ein graphischen Packungsproblem in einen Graphen umgewandelt werden kann, sollte dieser Graph noch etwas unter die Lupe genommen werden. Bei genauerer Betrachtung lässt sich nämlich feststellen, dass ein solcher Graph sehr schnell sehr groß wird, wie im folgenden Beispiel zu sehen ist:
Angenommen, die Feldgröße beträgt 800 auf 800 Pixel und unser Polygon hat eine Größe von ca. 10 auf 10 Pixel. Je mehr Positionen man auf dem Feld erlaubt, desto besser wird das Ergebnis am Ende voraussichtlich sein. Versuchen wir es also zunächst mit einer Position für jedes Pixel und 360 unterschiedliche Drehwinkel, also einer Drehung um jeweils 1°. Dadurch würden wir 800*800, also 640.000 unterschiedliche Positionen erhalten. Für jede Position 360 unterschiedliche Drehungen, also 640.000 * 360 = 230.400.000 Knoten und noch einmal genau so viele für das gespiegelte Polygon, also insgesamt 460.800.000 Knoten.
Das eigentliche Problem stellt allerdings die Anzahl der Kanten dar. Jeder Knoten kann eine Kante zu jedem Knoten in seinem Umfeld haben. Bei einer Größe des Polygons von ca. 10 auf 10 Pixel lässt sich die maximale Anzahl der Kanten pro Knoten abschätzen mit der Kreisformel für den Flächeninhalt (Π * Radius²). Der Radius muss hier gleich 10 gewählt werden (Erklärung, siehe „dismax“ im nächsten Kapitel). Jedes Polygon kann sich demnach mit Polygonen an $\Pi * 10^2 = 314$ Positionen schneiden. Für jede dieser Positionen existieren 720 Knoten (360 Drehungen und 360 für die Spiegelungen); also 314*720 = 226.080 mögliche Kanten. Damit würden wir 460.800.000 * 226.080 = 104.177.664.000.000 Kanten erhalten. Um mit einem Graphen mit 460 Mio. Knoten und 104.177 Mrd. Kanten zu arbeiten, wird eine Menge Zeit und sehr viel Speicherkapazität benötigt. Ein normaler PC ist dabei bei weitem überfordert.
Um diesem Problem Herr zu werden, könnte man die Feldgröße verringern, indem das Feld in mehrere kleine Felder gleicher Größe unterteilt wird. So könnte man beim obigen Beispiel das Feld in 64 kleinere Felder von der Größe 100* 100 aufteilen. Auf diese Weise würde sich die Anzahl der Kanten und Knoten um den Faktor 64 verringern. Womit wir noch bei ca. 7 Mio. Knoten und 1.600 Mrd. Kanten wären.
Weitaus Effektiver wäre es, zusätzlich die Distanz zwischen den Positionen und die Anzahl der Drehungen zu verringern. In diesem Fall verringert sich nämlich nicht nur die Anzahl der Knoten und Kanten linear zu Feldgröße, sondern zusätzlich wird es auch noch weniger Kanten pro Knoten geben. Vergrößern wir die Distanz von einem auf nur zwei Pixel und drehen wir das Polygon in jedem Schritt um 3° anstatt um 1°, bleiben uns 600.000 Knoten und 11 Mrd. Kanten Damit wäre die Anzahl der Knoten noch einmal um einen Faktor von 12 verringert und die Anzahl der Kanten sogar um einen Faktor von 144.
Je nach dem wie nahe man der optimalen Lösung kommen will, wie viel Rechenzeit zur Verfügung seht und was die Rechenumgebung in der Lage ist zu leisten, sollte der Algorithmus also angepasst werden können.

3.2.3 Optimierungen beim Aufbau des Konfliktgraphen

Um die Kanten des Graphen zu erstellen, müssen zunächst alle Knotenpaare darauf getestet werden, ob sich die jeweiligen Polygone schneiden. Bei einer großen Anzahl von Knoten wird dies eine Menge Zeit in Anspruch nehmen. Es gilt also, die Anzahl dieser Tests zu verringern. Zunächst ist wohl klar, dass in den meisten Fällen bei weitem nicht alle Polygone miteinander verglichen werden müssen. Falls das Feld nämlich wesentlich größer als die Figur ist, wovon bei den meisten Problemen wohl ausgegangen werden kann, müssen nur diejenigen Polygone, die in der näheren Umgebung zueinander liegen, getestet werden. Da nur ein Polygon betrachtet wird, kann die maximale Distanz, bei der sich zwei Kopien dieses Polygons noch schneiden können, leicht errechnet werden:
Die Position des Polygons wird durch den geometrischen Schwerpunkt bestimmt, also berechnen wir die Distanz zwischen diesem und allen Punkten des Polygons. Die größte Distanz wird daraufhin verdoppelt und wir haben den gewünschten Wert „dismax“. Dementsprechend müssen wir also nur Polygone testen, deren geometrischer Schwerpunkt innerhalb des Kreises mit dem Radius „dismax“ um die aktuelle Position liegt.

Des weiteren fällt auf, dass sich durch die regelmäßige Anordnung der Polygone auf dem Feld ein Muster ergibt. Die Knoten für eine Position werden also genau gleich viele adjazente Knoten haben wie die Knoten der Position daneben, es sei denn, man befindet sich in der Nähe des Randes, da hier das Polygon eines möglichen adjazenten Knotens den Rand schneiden kann und somit nicht Teil des Graphen ist. Ansonsten entstehen Unterschiede alleine durch die Drehung und Spiegelung des Polygons.
Daher bietet es sich an, ein Muster für jede mögliche Lage des Polygons an einer einzigen Position zu erstellen und dieses Muster auf alle anderen Positionen zu übertragen. Dabei müssen dann nur noch die Positionen in der Nähe des Randes gesondert betrachtet und die Kanten zu Nachbarknoten, die gar nicht existieren, wieder entfernt oder gleich weggelassen werden.
Mit Hilfe dieser beiden Ideen lässt sich die Anzahl der Schnitttests von $|\text{Anzahl Knoten}|^2$ auf ein paar wenige reduzieren.

3.2.4 Greedy-Strategie

Wie bereits erwähnt, wurden zur Berechnung eines gültigen Independent Sets einfache Greedy-Strategien verfolgt. Das bedeutet, dass die Algorithmen in jedem Schritt denjenigen Folgezustand wählen, welcher dem Ziel am nächsten kommt. Um zu entscheiden welcher Folgezustand das sein wird, wird eine Bewertungsfunktion benötigt. Greedy-Algorithmen finden meistens nicht die optimale Lösung, für unser Problem scheinen die Ergebnisse in den meisten Fällen allerdings recht gut zu sein.
Um eine Greedy-Strategie anwenden zu können, muss der Algorithmus in Schritte unterteilt werden. Die Zustände die durch solche Schritte erreichbar sind, müssen klar definiert sein und miteinander verglichen werden können.
Unser Ziel ist das Finden eines möglichst großen Independent Sets. Es wird also ein Untergraph gesucht, der keine Kanten, aber möglichst viele Knoten enthält. Da Kanten nicht einfach entfernt werden dürfen (die Tatsache, dass sich zwei Polygone schneiden, lässt sich nicht entfernen), kann der Graph nur verkleinern werden, indem Knoten entfernt werden (Ein Polygon vom Feld zu nehmen ist möglich). Als ein Schritt im Algorithmus bietet es sich demnach an, einen oder mehrere Knoten und logischerweise deren adjazente Kanten aus dem Graphen zu entfernen. Somit wäre ein Folgezustand als Untergraph des Ursprungsgraphen klar definiert.

Die Bewertungsfunktion kann nun unterschiedlich definiert werden. Zwei einfache Varianten sind im Folgenden erklärt.

3.2.4.1 Pruning Algotihmus

Der Begriff Pruning lässt sich wohl am besten mit „beschneiden“ oder „zurecht schneiden“ übersetzten. Gemeint ist in diesem Fall, dass der Graph Schritt für Schritt verkleinert wird, indem ein „schlechter“ Knoten nach dem anderen mit den dazugehörigen Kanten aus dem Graphen entfernt wird.
Das Ziel, ein Independent Set zu finden, ist genau dann erreicht, wenn im verbliebenen Graphen keine Kanten mehr existieren. Das Bewertungskriterium wird in diesem Fall also die Anzahl der verbliebenen Kanten im Graphen sein. Je weniger Kanten ein Untergraph hat, desto besser wird er bewertet. Nach dem „Greedy“-Prinzip wählen wir demnach immer denjenigen Knoten mit den meisten Kanten aus, um ihn zu entfernen. In jedem Schritt wird somit die größtmögliche Anzahl an Kanten entfernt und der Folgezustand ist logischerweise derjenige mit den wenigsten verbleibenden Kanten.
Die Hoffnung ist, dass so am Ende möglichst wenige Knoten entfernt wurden und die übrigen ein möglichst großes Independent Set bilden.

3.2.4.2 Cherry-Pick Algorithmus

Dieser Algorithmus geht genau umgekehrt vor. „Cherry-Pick“ bedeutet, „sich das beste herauspicken“ oder „sich die Rosinen herauspicken“. Es wird also kein Knoten gewählt, der aus dem Graphen entfernt werden soll, sondern es wird ein besonders viel versprechender Knoten gewählt, der am Ende als Teil des Independent Sets im Graphen verbleibt. Logischerweise können alle Nachbarknoten des gewählten Knotens nicht mehr Teil des Independent Sets sein und müssen aus dem Graphen entfernt werden. Das Bewertungskriterium ist in diesem Fall die Anzahl der verbliebenen Knoten im Graphen. Demnach wird die Wahl, entsprechend der Greedy-Strategie, jeweils auf den Knoten mit den wenigsten Nachbarn fallen. Es werden minimal viele Knoten aus dem Graphen entfernt und der Graph des Folgezustandes ist derjenige mit den meisten verbliebenen Knoten.
Bildlich kann man sich diesen Algorithmus als eine Art Auffüllen von außen nach innen vorstellen. Da von den Polygonen, die am Rand und in den Ecken liegen, einige in der Nähe befindliche Polygone den Rand schneiden und nicht in den Graphen aufgenommen werden, haben die entsprechenden Knoten der Polygone weniger Nachbarknoten als die Knoten, deren Polygone in der Mitte des Feldes liegen. Demnach werden diese zuerst gewählt. Die adjazenten Knoten werden entfernt und die Nachbarn dieser entfernten Knoten, die nun am Rand liegen, sind die möglichen Kandidaten als nächstes ausgewählt zu werden, da sie nun auch weniger Nachbarknoten haben.

4 Implementierung

Nachdem nun die grundlegenden Ideen erklärt sind, wird im folgenden etwas mehr ins Detail gegangen und es werden einige zentrale Aspekte der Implementierung vorgestellt.
Als Programmiersprache wurde C++ verwendet. Als Entwicklungsumgebung kam der QtCreator zum Einsatz, welcher sich aufgrund seines GUI-Designers und der Qt-Klassenbibliotheken für das Arbeiten mit Graphiken angeboten hat.

4.1 Probleminstanz

Der erste wichtige Aspekt ist wohl die Darstellung der Figur und die Größe des Feldes, die vom Nutzer eingegeben werden können. Diese definieren eine Instanz des Problems.

4.1.1 Datenstrukturen der Probleminstanz

Für das Feld bietet sich der von Qt vorgegeben Datentyp eines QRectF an. Der Nutzer kann die gewünschte Breite und die Höhe des Feldes angeben. Daraufhin wird ein Rechteck, bestehend aus den Koordinatenachsen und der Senkrechten bzw. Waagerechten, die durch die Breite und die Höhe definiert werden, erstellt. Um Rechenzeit zu sparen kann das „field“ in kleinere Rechtecke unterteilt werden. Die Größe eines dieser Teile wird in „divFeld“ gespeichert und soll ungefähr Platz für zehn Polygone in jeder Richtung bieten.

```
QRectF field;
QRectF divFeld;
```

Um das Problem möglichst einfach zu halten, werden als Figuren nur Polygone zugelassen, sodass keine gekrümmten Begrenzungen beachtet werden müssen. Auch für Polygone gibt es bereits einen vom QtCreator vordefinierten Datentyp namens QPolygonF. Hier werden die einzelnen Punkte des Polygons in einer Liste gespeichert und beim zeichnen durch Liniensegmente miteinander verbunden. Der Letzte Punkt wird mit dem ersten verbunden, um das Polygon zu schließen. Zusätzlich zum Polygon werden die Anzahl der Punkte des Polygons in „count“ gespeichert. Des weiteren wird „dismax“ gespeichert, dessen Berechnung und Bedeutung bereits im Kapitel 3.2.3 erläutert wurde.
Diese drei Variablen werden in der Datenstruktur „p“ für Polygon zusammengefasst und es wird eine Instanz „protyp“ dieser Struktur erzeugt, in der die Eingabe vom Benutzer hinterlegt wird.

```
struct p {
        QPolygonF poly;
        int count;
        double dismax;
};
struct p protyp;
```

4.1.2 Eingabe der Probleminstanz

Für die Eingabe des Polygons hat der Nutzer zwei Möglichkeiten. Er kann es sich einerseits mit der Maus im Vorschau-Fenster zusammenklicken. Hierbei muss die Reihenfolge beachtet werden, da neue Punkte nur am Ende des Polygons angefügt werden. Die Alternative dazu besteht darin, einen Punkt und seinen Index explizit in der dafür vorgesehenen Maske anzugeben. Auf diese Weise kann ein Punkt an jeder beliebigen Stelle des Polygons hinzugefügt werden. Zu beachten ist, dass nur Punkte, die im Vorschau-Fenster sichtbar sind eingefügt werden können. Reicht der Platz nicht aus, kann das Fenster hoch skaliert werden. Beide Varianten der Eingabe können natürlich auch parallel genutzt werden.
Durch einen Klick mit der rechten Maustaste auf einen bestehenden Punkt im Vorschaufenster

wird dieser Punkt wieder aus dem Polygon entfernt.
Mit dem Buttons „Accept“ wird das aktuell im Vorschau-Fenster gezeichnete Polygon akzeptiert und gespeichert. Daraufhin können die Algorithmen zur Lösung gestartet werden.
Durch drücken des Buttons „Edit“ kann ein akzeptiertes Polygon erneut angepasst werden.
„Delete“ löscht das komplette Polygon, damit ein neues gezeichnet oder eingetippt werden kann.

Die Feldgröße wird beim Programmstart auf 800*800 Pixel vordefiniert. Der Nutzer kann dies jederzeit, durch eine Eingabe in die dafür vorgesehene Maske, ändern.

4.2 Grundlagen Funktionen

Im folgenden wird die Umsetzung einiger wichtiger Grundlagen-Funktionen genauer betrachtet.

4.2.1 Geometrischer Schwerpunk

Um den geometrischen Schwerpunkt als Position und Drehangelpunkt nutzen zu können, muss er zunächst berechnet werden.
Mit Hilfe der gauß'schen Dreiecksformel kann zunächst der Flächeninhalt des Polygons bestimmt werden.

$$A = \frac{1}{2} \sum_{i=0}^{N-1} (x_i y_{i+1} - x_{i+1} y_i)$$

Den geometrischen Schwerpunkt erhält man dann mit den folgenden Formeln.

$$x_s = \frac{1}{6A} \sum_{i=0}^{N-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$
$$y_s = \frac{1}{6A} \sum_{i=0}^{N-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

4.2.2 Rotation eines Polygons

Zum Verschieben des Polygons und der Spiegelung muss wohl nicht viel gesagt werden. Die Drehung hingegen ist keine solch triviale Sache und verdient einen genaueren Blick.
Genau wie beim Verschieben und Spiegeln, wird auch hier jeder Punk für sich betrachtet. Als Rotationszentrum wird, wie oben bereits erwähnt, der geometrische Schwerpunkt verwendet, sodass sich jeder Punkt auf einem Kreisbogen um diesen bewegt.
Betrachten wir also die Drehung eines einzelnen Punktes:
Zunächst gilt es, den Ausgangswinkel zu bestimmen, in dem der Punkt zum Drehmittelpunkt steht, sowie die Distanz der beiden Punkte. Diese Distanz entspricht dem Radius des Kreises, auf dem sich der Punkt beim Rotieren bewegt.
Der Radius „rad“ lässt sich mit Hilfe des Satzes des Pythagoras berechnen:

$$\text{rad} = \sqrt{\Delta x^2 + \Delta y^2}$$

Δx und Δy sind jeweils die Differenzen zwischen den x- und y-Werten des Drehmittelpunktes und des zu drehenden Punktes.
Um mit Hilfe eines Winkels die Position eines Punktes zu berechnen, wird der Sinus oder der Kosinus verwendet. Wollen wir umgekehrt aus der gegebenen Position den Startwinkel „ωS“ erhalten, brauchen wir die Umkehrfunktion, in diesem Fall den Arkuskosinus:

$$\omega S = \text{acos} (\Delta x / \text{rad}) * 180 / \pi$$

Der Arkuskosinus liefert Ergebnisse aus dem Bereich 0 bis π, was einem Winkel von 0° bis 180° entspricht. Der Faktor $180 / \pi$ projiziert das erhaltene Ergebnis auf den entsprechenden Winkel. Da wir nur Ergebnisse zwischen 0° und 180° erhalten, können wir nicht entscheiden ob ein Punkt oberhalb des Drehangelpunkts oder unterhalb davon liegt. Liegt er nämlich unterhalb,muss das Ergebnis negiert werden. Auf diese Weise können wir alle möglichen Winkel zwischen 0° und 360° oder gleichbedeutend zwischen -180° und 180° unterscheiden:

$$\omega S = - \omega S$$

Dabei entspricht ein Winkel von 330° = -30° , 300° = -60°, 270° = -90°, usw.
Um den Winkel des neuen Punktes ωZ zu erhalten, müssen wir den eigentlichen Drehwinkel „ψ“ nun aufaddieren:

$$\omega Z = \omega S + \psi$$

Mit Hilfe dieses Winkels und dem Kosinus lässt sich nun der rotierte Punkt berechnen:

$$x = \text{rad} * \cos (\omega Z * \pi / 180) + x_0$$
$$y = \text{rad} * \sin (\omega Z * \pi / 180) + y_0$$

wobei (x_0,y_0) die Position des Drehmittelpunkts ist.

4.2.3 Schnitt zweier Segmente

Die Lage eines Punktes zu einem Geraden kann mit der Determinante einer 3x3-Matrix errechnet werden.
Folgende Matrix gibt die Lage von Punkt p zur Geraden durch die Punkte q und r:

$$D = \begin{pmatrix} p(x) & q(x) & r(x) \\ p(y) & q(y) & r(y) \\ 1 & 1 & 1 \end{pmatrix}$$

Ist die Determinante positiv, liegt der Punkt links von der Geraden. Ist sie negativ, liegt der Punkt rechts. Ist die Determinante gleich Null liegt der Punkt auf der Geraden.
Wir berechnen dementsprechend für jedes der beiden Segmente die beiden Determinanten und vergleichen sie. Haben beide Determinanten-Paare verschiedene Vorzeichen, schneiden sich die beiden Segmente, hat eins der Paare oder haben beide Paare die jeweils gleichen Vorzeichen, schneiden sich die Segmente nicht.
Für den Sonderfall, dass eine oder mehrere Determinanten gleich Null sind, sind 3 Fälle zu unterscheiden:
Fall 1 : Eine der Determinanten ist gleich Null.

Fall 2 : Zwei der Determinanten sind gleich Null
Fall 3 : Alle vier Determinanten sind gleich Null
Tritt der erste Fall ein, bedeutet dies, dass einer der Punkte auf der Geraden liegt, die vom anderen Segment aufgespannt wird. In diesem Fall gilt es zu testen, ob der Punkt zwischen den beiden Endpunkten des Segments liegt. Falls ja, schneiden sich die Segmente, sonst nicht.
Der zweite Fall kann nur bedeuten, dass die beiden Punkte, deren Determinante gleich Null ist, an der gleichen Stelle liegen und es ist klar. Da wir davon ausgehen können, dass es kein Segment gibt, dessen beide Endpunkte an derselben Stelle liegen, bedeutet dieser Fall, dass sich die Segmente schneiden.
Im dritten Fall liegen alle 4 Punkte auf einer Geraden. Die Segmente schneiden sich in diesem Fall genau dann, wenn einer der vier Punkte zwischen den Endpunkten des jeweils anderen Segments liegt.
Der Fall, dass nur eine Determinante ungleich Null ist, kann nicht eintreten.

4.3 Independent Set Algorithmen

4.3.1 Datenstrukturen

Für die Independent Set Algorithmen muss zunächst eine Datenstruktur für einen Graphen aufgebaut werden. Um dies zu realisieren benötigen wir eine Möglichkeit, Knoten und ungerichteten Kanten zu speichern.

Für die Knoten wurde eine Datenstruktur „pointList“ erstellt. In dieser Struktur sollten alle relevanten Informationen zu dem Polygon, für welches der Knoten steht, erfasst werden.
Zunächst die Position des Polygons auf dem Feld, welche in „xStp“ und „yStp“ vermerkt wird. „drStp“ gibt an um welchen Winkel das Polygon gedreht wurde und in „reflected“ wird gespeichert, ob das Polygon gespiegelt wurde oder nicht.
Um nicht bei jedem Schritt des Algorithmus die komplette Datenstruktur, die hinter einem Knoten steht löschen zu müssen, wird in „exist“ vermerkt, ob der Knoten noch beachtet werden muss oder kein Teil des Graphen mehr darstellt. Auf diese Weise können auch all diejenigen Knoten, bei denen sich die Position des Polygons im Feld befindet, das Polygon aber gleichzeitig den Rand schneidet, in den Array aufgenommen werden. „anzEdge“ enthält die Anzahl der adjazenten Knoten und an „edge“ wird eine Liste mit Kanten zu den adjazenten Knoten angehängt.
Zur Laufzeit wird ein Array, von der Größe „count“ (siehe Kapilte 4.3.2), aus Zeigern auf „pointList“-Elemente erstellt. Die globale Variable „pL“ ermöglicht den Zugriff auf diesen Array aus unterschiedlichen Funktionen.

```
struct pointList{
    int xStp;
    int yStp;
    int drStp;
    bool reflected;
    bool exist;
    int anzEdge;
    int index;
    edgeList *edge;
};
pointList **pL;
```

Eine Kante wird mit Hilfe von zwei zueinander gehörenden Elementen vom Typ „edgeList“

dargestellt. Jeweils eins von beiden für jeden Endpunkte der Kante. Die beiden Elemente sind über den Zeiger „partner“ mit dem jeweils anderen verknüpft. „knoten“ verweist auf den Knoten an dem die Kante angehängt ist und „next“ auf die jeweils nächste Kante des Knotens. Auch hier wird die eine Kante beim Entfernen aus dem Graphen nicht gelöscht, sondern die Existenz der Kante wird in „exist“ vermerkt.

```
struct edgeList{
    edgeList *partner;
    edgeList *next;
    pointList *knoten;
    bool exist;
};
```

4.3.2 Aufbau der Datenstrukturen

Bevor diese Datenstrukturen erstellt werden können, müssen die Parameter des Algorithmus festlegen werden.
Der Nutzer kann die Variablen „xStep“, „yStep“ und „numRot“ nach seinen Wünschen angeben. „xStep“ und „yStep“ geben dabei die Distanz zwischen zwei möglichen Positionen in die jeweilige Richtung an. „numRot“ gibt die Anzahl der unterschiedlichen Drehungen des Polygons wieder.
„numX“ und „numY“ werden aus den Werten „xStep“ und „yStep“ berechnet. In ihnen werden die Anzahl der unterschiedlichen Positionen auf dem Feld gespeichert, also wird es insgesamt numX*numY viele Positionen auf dem Feld geben,
„degree“ wird aus „numRot“ berechnet und gibt den Winkel an, um den sich zwei unterschiedliche Drehungen mindestens voneinander unterscheiden.
„count“ ist die Anzahl aller möglichen Lagen des Polygons. Es berechnet sich aus der Anzahl der Positionen und der Anzahl der Drehungen. Also numX*numY*numRot. Dies ist dementsprechend auch die Größe des Arrays „pL“, in dem die Knoten gespeichert werden.

```
int numX;

int numY;

int numRot;

double xStep;

double yStep;

double degree;

long count;
```

Nachdem die Parameter gesetzt wurden, kann der Nutzer die Datenstrukturen aufbauen lassen. Zunächst wird der Array für die Knoten erstellt und befüllt. Alle Knoten, deren Polygone den Rand des Feldes schneiden, werden mit erstellt, aber als nicht Existent markiert.
Wurden alle Knoten erstellt und auf den Schnitt mit dem Rand getestet, müssen als nächstes die Kanten berechnet und erstellt werden. Für jedes Knotenpaar werden zwei Instanz von „edgeList“ erstellt, an die Knoten angehängt und miteinander verbunden, falls die beiden zugehörigen Polygone sich schneiden. Wie bereits erwähnt, wird um Zeit zu sparen, für eine einzelne Position des Polygons ein Muster berechnet und dieses Muster danach auf alle anderen Positionen übertragen.

4.3.2.1 Erstellen des Musters zur Optimierung des Rechenaufwandes

Zum Erstellen des Musters betrachten wir das Polygon „poly“ an der Stelle (0.0). Das Muster besteht aus einem vierdimensionalen Array aus bool'schen Werten. Existiert eine Kante steht „true“ im Array, sonst „false“
Die erste Dimension enthält die Drehungen des Polygons „poly“, sowie die Drehungen des gespiegelten Polygons an dieser Position. Bei drei unterschiedlichen Drehwinkeln hätte diese Dimension eine Größe 6.
Die zweite Dimension enthält alle möglichen Positionen eines Polygons „movepoly“ in y-Richtung, bei denen ein Schnitt mit dem Polygon am Ursprung möglich ist. Es ist zu beachten, dass nur Polygone, die in der Liste der Knoten weiter hinten stehen, gewertet werden dürfen. Sollte eine Kante zu einem Knoten existieren, der weiter vorne steht, so wurde diese Kante bereits vom Partnerknoten erstellt. Auf diese Weise wird vermieden, dass es zu Mehrfachkanten kommt. Da die Knoten zunächst von rechts nach links und dann von oben nach unten sortiert sind, müssen in y-Richtung demnach nur diejenigen Positionen beachtet werden, die auf gleicher Höhe oder unterhalb liegen. Die Anzahl dieser Positionen wird in „anzY“ gespeichert.
Die dritte Dimension enthält alle möglichen Positionen in x-Richtung. Hier müssen sowohl die Positionen rechts als auch die Positionen links betrachtet werden, da alle Polygone, die unterhalb von „poly“ liegen, im Knotenarray weiter hinten stehen, also auch alle Polygone, die links unterhalb liegen. Die Anzahl dieser Positionen wird in „anzX“ gespeichert.
Die vierte Dimension hat wiederum die Größe der ersten und enthält die Lagen des Polygons „movepoly“ an der Stelle (x,y) welche durch die Dimensionen 2 und 3 definiert wird.

```
bool ****muster;

int anzX;

int anzY;
```

In Abbildung 3 wird ein Muster beispielhaft veranschaulicht.
Für dismax = 20 und yStep = xStep = 3 ergeben sich anzY = 7 und anzX = 13. drStep wurde in diesem Fall = 3 gewählt, woraus sich sechs verschiedene Lagen des Polygons ergeben. Die erste Dimension wurde hier auseinander gezogen, somit zeigt jeder Block die adjazenten Knoten für eine Lage des Polygons „poly“ an einer festen Position. Die vierte Dimensionen, wird farblich gekennzeichnet. Ein grünes Rechteck steht für fünf oder sechs adjazente Knoten, ein rotes Rechteck für drei oder vier, ein blaues Rechteck für einen oder zwei und ein schwarzes Rechteck für keinen adjazenten Knoten an der entsprechenden Position.

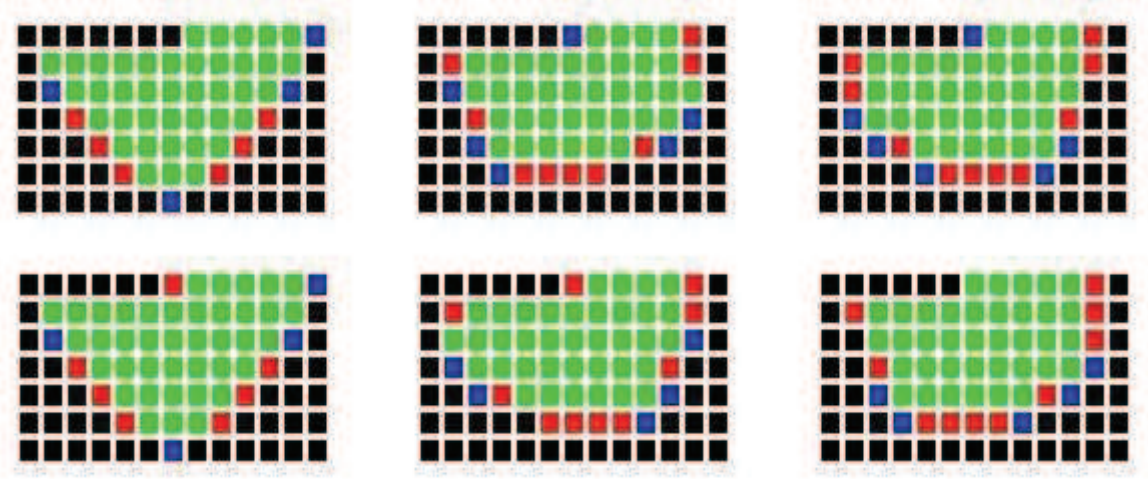


Abb3.: Darstellung eines Musters, aufgefächert nach der ersten Dimension

Schön zu sehen ist hierbei auch, dass für diejenigen Stellen, die im Knotenarray weiter vorn stehen, keine Kanten erstellt werden. An der Stelle (0/0), also in der obersten Reihe in der Mitte, wird bei jedem Block eine Kante mehr erstellt, damit die Knoten der Polygone, die an der selben Position liegen, auch keine Mehrfachkanten zwischen sich haben.

Um das Muster aufzubauen, wird nun ein Polygon „movepoly“ für jeden Eintrag im Muster an die richtige Stelle geschoben und gedreht. Dabei wird von oben nach unten und jeweils von rechts und links in Richtung Mitte gearbeitet. Überschneiden sich alle Polygone an einer Position (x,y) mit „poly“, dann werden sich alle Polygone auf der gleichen y-Ebene zwischen x und 0 ebenfalls mit „poly“ schneiden und müssen nicht mehr extra getestet werden. Auf diese Weise können erneut einige Schnitttests gespart werden.

Nachdem das Muster erstellt wurde, kann nun für jeden Knoten im Knotenarray das Muster durchlaufen werden. Je nachdem, wie das entsprechende Polygon gedreht wurde, wird eine der in Abbildung 3 gezeigten „Scheiben“ ausgewählt. Für jeden Eintrag wird dann ermittelt, ob das entsprechende Polygon außerhalb des Feldes liegt oder den Rand schneidet. Ist dies nicht der Fall, wird eine Kante erstellt. Danach kann das Muster wieder zerstört werden.

4.3.2.2 Alternativer Umgang mit den Kanten

Wie bereits gezeigt wurde, kann die Anzahl der Kanten eines Graphen sehr groß sein. Das Erstellen dieser Kanten wird dadurch eine Menge Zeit in Anspruch nehmen. Um dies zu umgehen wurde, ein alternatives Vorgehen entwickelt.
Das Prinzip des Musters bleibt dasselbe, allerdings werden anstatt der bool'schen Werte Integerwerte gespeichert und die Einschränkung auf die Polygone, die im Knotenarray weiter hinten stehen wird aufgehoben.

```
int ****muster;
```

In Abbildung 4 wird ein solches Muster dargestellt.

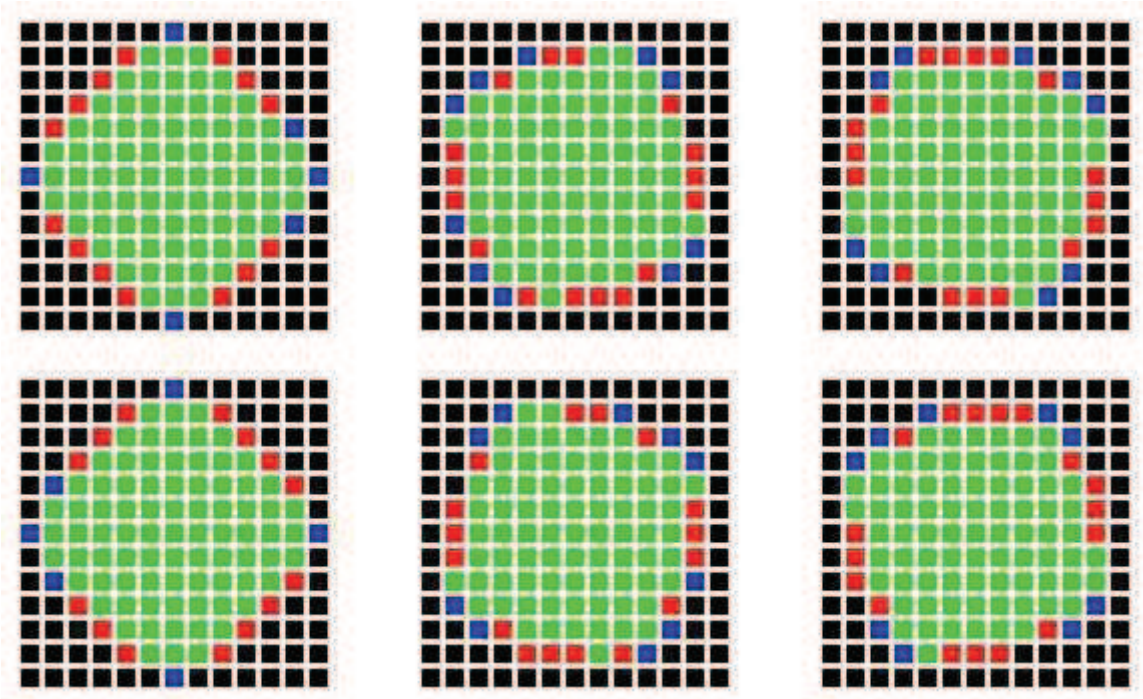


Abb.4: Darstellung eines Musters mit Integerwerten, aufgefächert nach der ersten Dimension

In jedem Eintrag des Musters steht nun eine Differenz, und zwar genau die Differenz zwischen den Indizes der Knoten von „poly“ und „movpoly“. Da die Knoten nach festem Muster erstellt wurden, kann von der Lage der Polygone auch auf den Index des entsprechenden Knotens im Knotenarray geschlossen werden. Somit wird im Muster nicht nur vermerkt, ob sich zwei Polygone schneiden, sondern auch, wie groß der Abstand zum jeweils anderen im Knotenarray ist. Schneiden sich zwei Polygone nicht, wird eine Null ins Muster geschrieben.
Somit können nun von jedem möglichen Knoten durch einen Blick in das Muster alle adjazenten Knoten im Knotenarray gefunden werden, indem der eigene Index und die Differenz aufaddiert werden und das Erstellen der Kanten wird überflüssig.
Allerdings wissen wir nun nicht, welcher Knoten wie viele Kanten besitzt, was für die Independent Set Algorithmen unabdingbar ist. Also muss auch hier für jeden Knoten das Muster durchlaufen werden und die existierenden Kanten müssen gezählt werden.
Liegt ein Polygon weit genug vom Rand entfernt, werden alle möglichen adjazenten Knoten existieren. Für diesem Fall kann ein „fullMuster“ berechnet werden, welches für jede Lage von „poly“ die maximal möglich Zahl an Kanten berechnet und speichert. So muss das Muster für alle Knoten in der Mitte nur einmal durchgegangen werden.

```
int fullMuster[2*numRot];
```

Nachdem nun alle Datenstrukturen erstellt wurden, können die eigentlichen Algorithmen gestartet werden. Je nach Art und Weise wie das Muster erstellt wurde, unterscheiden sich die Algorithmen leicht voneinander.

4.3.3 Algorithmen mit Kanten

Bei den Algorithmen mit Kanten ist das Muster nicht mehr weiter von Belang. Die Kanten der Knoten sind direkt über die Liste, die in „edges“ angehängt wurde, erreichbar.

Die Idee des Pruning-Algorithmus wurde auf zwei Arten umgesetzt. Wie bereits beschrieben, geht es bei der Greedy-Strategie stets darum, den Knoten mit den meisten Kanten zu finden. Zunächst wurde hierfür bei jedem Schritt schlicht der komplette Knotenarray durchlaufen und das Element mit den meisten Knoten gewählt und entfernt. Da im Normalfall über 90% der Knoten im Laufe des Algorithmus entfernt werden lässt sich die Laufzeit mit $O(n^2)$ abschätzen. Mit dem Ziel, dies etwas zu verbessern, wurde im Folgenden noch eine Variante programmiert, bei der zunächst der Knotenarray nach der Anzahl der Nachbarknoten sortiert wird. Nun kann immer das letzte existierende Element des Arrays gewählt werden. Wurde ein Element entfernt, werden gleichzeitig alle dazugehörigen Kanten zerstört und die adjazenten Knoten haben dementsprechend eine Kante weniger. Deshalb müssen einige dieser Knoten im Array neu einsortiert werden, also jeweils solange nach links geschoben werden, bis das nächste Element gleich viele oder weniger Kanten hat. Da diese Neusortierung allerdings zu viel Zeit braucht, ist die Performance dieses Algorithmus sogar schlechter als die ursprüngliche Variante.

Der Cherry-Pick-Algorithmus wurde nur in der Sort-Variante umgesetzt. Im Unterschied zum Pruning Ansatz wird der sortierte Array hier allerdings von vorn nach hinten durchlaufen und es wird in jedem Schritt nicht nur ein Knoten entfernt, sondern alle Nachbarknoten des gewählten Knotens. Für alle Nachbarn werden auch hier logischerweise die Kanten mit entfernt und für die Nachbarn dieser Nachbarn, also die Nachbarn zweiten Grades, muss dementsprechend die Anzahl der Kanten reduziert werden.
Da hier in jedem Schritt mehrere Knoten auf einmal gelöscht und dementsprechend auch wesentlich mehr Kanten wegfalle, lohnt es sich nach jedem Schritt den kompletten Array neu zu sortieren.

4.3.4 Algorithmen ohne Kanten

Wurden keine Kanten erstellt muss etwas anders vorgegangen werden. Grundsätzlich gilt, dass der Knotenarray nicht umsortiert werden darf, da wie bereits beschrieben, die Kanten über die jeweilige Index-Differenz ihrer beiden Knoten definiert wurde. Sollte die Reihenfolge des Arrays durcheinander kommen, wären das erstellte Muster und die darin enthaltenen Information nutzlos.

Für den Pruning-Algorithmus muss also wieder auf den naiven Ansatz zurückgegriffen werden. Durchlaufe in jedem Schritt den kompletten Array, finde den Knoten mit den meisten Kanten und entferne ihn. Beim Entfernen muss hier nun das Muster zu Rate gezogen werden. Für jeden Eintrag ungleich Null wird überprüft ob der mögliche Partnerknoten existiert, und falls dem so ist, wird die Anzahl seiner Kanten verringert. Zu jeder Zeit wird logischerweise die Anzahl der existierenden Partnerknoten und die Anzahl der Kanten übereinstimmen, sonst wäre irgendwo ein Fehler unterlaufen.

Dasselbe System wird nun auch beim Cherry-Pick-Algorithmus verwendet. Der komplette Array wird durchsucht und das Element mit den wenigsten Nachbarknoten wird gewählt. Mit Hilfe des Musters werden die existierenden Nachbarn gefunden. Diese werden gelöscht, nachdem die Anzahl der Kanten von deren Nachbarn, die ihrerseits wieder mit Hilfe des Musters berechnet wurden, verringert wurde.

4.4 Ausgabe

Zum Erstellen der Ausgabe wird der Knotenarray durchlaufen, und für jeden Knoten der noch existiert, wird das entsprechende Polygon an die outputview Klasse übergeben. Dort werden alle diese Polygone zusammen in eine Szene gezeichnet und in einem extra Fenster ausgegeben.
Zusätzlich zu den Polygonen werden noch einige weitere, nützliche Informationen zur Lösung in das Ausgabefenster geschrieben.
Zunächst natürlich die benötigte Zeit. Diese berechnet sich aus der Zeit, die zur Erstellung des Graphen benötigt wurde und der Laufzeit des eigentlichen Algorithmus.
Des weiteren ist die Information über die Anzahl, der auf dem Feld positionieren Polygone, wohl die Wichtigste.
Und schließlich wird die Fläche, die von allen Polygonen abgedeckt wird und die ungenutzte Fläche, als zusätzliche interessante Information übergeben

5 Analyse der Algorithmen

Zunächst wird der Pruning-Ansatz mit dem des Cherry-Pick-Algorithmus verglichen.
Der Letztere mag zwar etwas komplizierter erscheinen, da auch die Nachbarn zweiten Grades in jedem Schritt involviert sind. Im Endeffekt ist dies aber der weitaus schnellere Algorithmus.
Wenn wir annehmen, dass beide ungefähr ein gleich gutes Ergebnis erzielen, werden beide Algorithmen am Ende gleich viele Knoten und Kanten entfernt haben. Ob dies nun bei Nachbarn ersten oder zweiten Grades geschieht, ist irrelevant. Hierbei kommt es demnach zu keinem Unterschied.

Der Vorteil des Cherry-Pick-Algorithmus entsteht dadurch, dass weitaus mehr Knoten aus dem Graphen entfernt werden als zurückbleiben. Beim Pruning-Ansatz wird für jeden Knoten, der entfernt wird, einmal der komplette Array durchsucht. Beim Cherry-Pick Algorithmus muss der Array für jeden Knoten, der nicht entfernt wird, durchsucht werden. Bei einem ungefähren Verhältnis von ungefähr 90 zu 10 zwischen entfernten und nicht entfernten Knoten, kostet die Suche dementsprechend das neunfache an Zeit.

Die Güte der Ergebnisse scheint im Normalfall beim Cherry-Pick-Algorithmus ebenfalls besser zu sein. Ohne dies genauer untersucht zu haben vermute ich, dass dies daran liegt, dass die Anzahl der Kanten mit einer asymptotischen Funktion beschrieben werden könnte. Dass heißt, es gibt wenige Knoten am Rand, die wenige Nachbarn haben und in der Mitte jede Menge Knoten mit ziemlich gleich vielen Nachbarn.
Der Pruning-Ansatz wird somit mehr oder weniger willkürlich Knoten aus der Mitte herausnehmen und der Graph wird gleichmäßig abgebaut, ohne dass durch die Wahl eines bestimmten Knotens ein wesentlicher Vorteil gegenüber der Wahl eines anderen Knotens entstehen würde.
Beim Cherry-Pick-Algorithmus hingegen, der auf der Seite mit dem großen Gefälle zwischen den einzelnen Knoten arbeitet, ist die Wahl des Knotens viel stärker begrenzt. Es gibt wenige Knoten, die sich eignen, gewählt zu werden, die aber auf ein wesentlich besseres Ergebnis hoffen lassen. Da außerdem vom Rand aus Schritt für Schritt nach innen gearbeitet wird lässt sich vermuten, dass dieses Gefälle immer bestehen bleibt und es jederzeit eine mehr oder weniger eindeutige Wahl gibt.

Im Vergleich zwischen den Algorithmen mit Erstellen der Kanten und denen ohne Erstellen der Kanten, liegen die Performance-Vorteile klar bei den Algorithmen ohne Kanten. Obwohl bei jedem Schritt auf das Muster zurückgegriffen werden muss, dauert dies, wenn überhaupt, nur unwesentlich länger als das durchlaufen der Kantenliste.
Der eigentliche Unterschied besteht jedoch darin, dass durch das Erstellen der Datenstrukturen ein nicht zu vernachlässigender Zeitaufwand entsteht, den man sich sparen kann. Je größer der Graph wird, desto deutlicher schlägt sich dies nieder.
Bei großen Graphen kann zusätzlich auch noch die Größe des Arbeitsspeichers zum Problem werden.

Zusammenfassend lässt sich klar sagen, dass der Cherry-Pick Algorithmus ohne Erstellen der Kanten der schnellste Ansatz zur Lösung ist.

6 Zusätzliche Klassen

Für die Eingabe und Vorschau des Polygons, sowie für die Ausgabe des Ergebnisses wurden zwei extra Klassen erstellt.

Die Klasse myscene verwaltet die Eingabe des Polygons im Vorschaufenster. Entsprechende der angegebenen Skalierung wird der Szenen-Hintergrund so gestaltet, dass der Nutzer einen Punkt so präzise wie möglich setzten kann und die Größe des erstellten Polygons gut ablesbar ist. Zu diesem Zweck wird eine Art Rastergitter über das Feld gelegt und beschriftet.
Durch Moustracking und MouseEvents, kann von der Szene erkannt werden wohin der Nutzer geklickt hat. Diese Position wird daraufhin in einem Event an die Hauptklasse zurückgegeben. Für jede Maustaste wird ein anderes Event erzeugt, sodass unterschieden werden kann, ob ein Punkt an der Positionen erstellt oder gelöscht werden soll.

Die Klasse outputview wurde kreiert, um die Ergebnisse der Berechnungen darstellen zu können. Zunächst wird eine Szene erstellt. Ähnlich wie bei myscene wird in den Hintergrund ein Rastergitter gezeichnet und beschriftet.
Mit Hilfe der Funktionen addText und addPolygon können Polygone und Text übergeben und in die Szene eingefügt werden.
Als ein weiteres Feature wurde eine Zoom-Möglichkeit für das Fenster implementiert. Über ein QwheelEvent kann der Nutzer hinein und heraus zoomen um sich das Ergebnis genauer zu betrachten

7 Graphische Oberfläche

Zum Schluss wird noch ein kurzer Überblick über die graphische Oberfläche gegeben, welche aus drei unterschiedlichen Fenstern besteht.

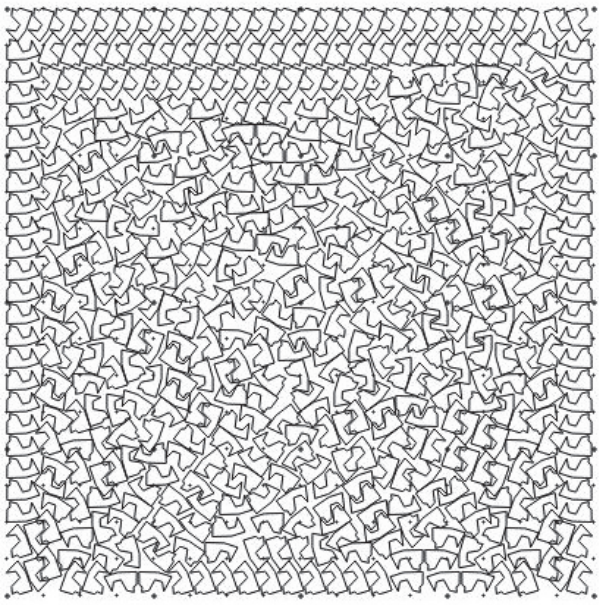
Beim Start des Programms landet man im Startfenster. Dieses dient in erster Linie dem Erstellen der Probleminstanz. Es enthält oben rechts ein Vorschaufenster, in welchem ein Polygon per Mausklick erstellt und geändert werden kann, darunter befinden sich die Buttons zur Verwaltung des Polygons. In der Eingabemaske oben links kann ein Punkt des Polygons und dessen Index explizit angeben werden. Mittig links kann der Nutzer die Größe der Grundfläche, auf der die Polygone angeordnet werden sollen, bestimmen.
Sobald ein Polygon erstellt und akzeptiert wurde, kann unten links gewählt werden, welche Art von Algorithmus zur Lösung der Probleminstanz genutzt werden soll. Je nach Wahl des Algorithmus landet man in einem der beiden anderen Fenster. In beiden bleibt das Vorschaufenster sichtbar.

Das Fenster für den naiven Algorithmus besteht nur aus einem Button, welcher bei Nutzung den Algorithmus startet und eine Ausgabe erzeugt.

Das Fenster für die Interessant Set Algorithmen bietet dem Nutzer weitere Auswahlmöglichkeiten. Zunächst kann er oben die Parameter für den Aufbau des Konfliktgraphen angeben. Darunter wird ihm angezeigt, wie groß der Graph bei den angegebenen Werten ungefähr werden wird.
Sind die Parameter gesetzt, muss als nächstes gewählt werden, wie die Datenstrukturen für die Algorithmen aufgebaut werden, also ob ein Graph mit Kanten erstellt wird oder ob die Informationen zu den Kanten nur im Muster gespeichert werden sollen.
Durch Drücken des Buttons „Erstellen“ werden die entsprechenden Datenstrukturen aufgebaut, woraufhin einer der eigentlichen Algorithmen gestartet werden kann.
Es kann vorkommen, dass manche Schritte der Algorithmen mehrere Minuten Zeit in Anspruch nehmen. Damit klar ist, dass sich das Programm nicht aufgehängt hat, und grob abgeschätzt werden kann wie weit der Algorithmus schon fortgeschritten ist, werden die aktuellen und die bereits beendeten Arbeitsschritte angezeigt. Zusätzlich läuft bei besonders zeitaufwändigen Arbeitsschritte ein prozentualer Zähler mit.

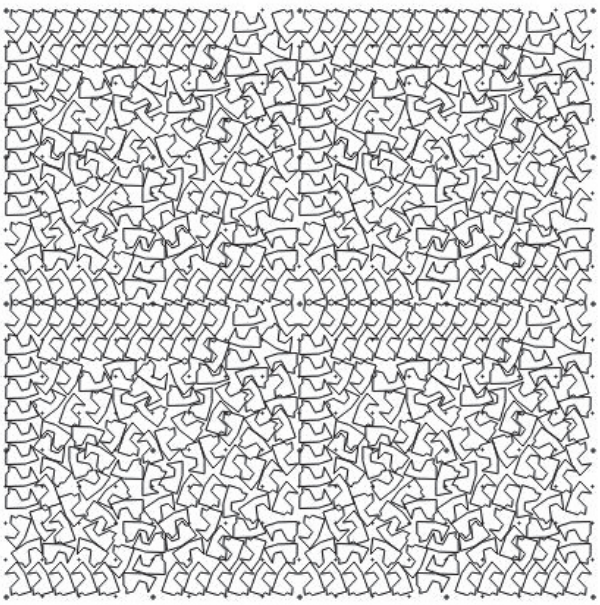
8 Beispiele

Im folgenden noch drei Beispiele mit zufälligen Polygonen.



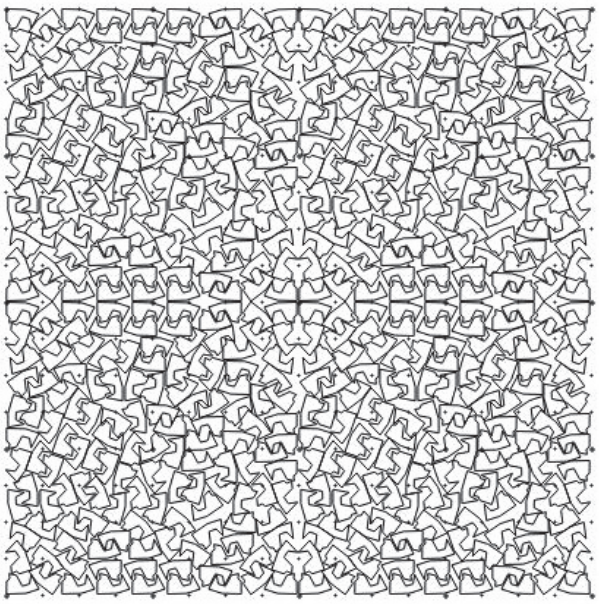
Rechenzeit : 10.463 sek.
Anzahl Polygone : 509
Genutzte Fläche : 93.956
Verschnittfläche : 66.043

Cherry-Pick-Alg ohne Unterteilung



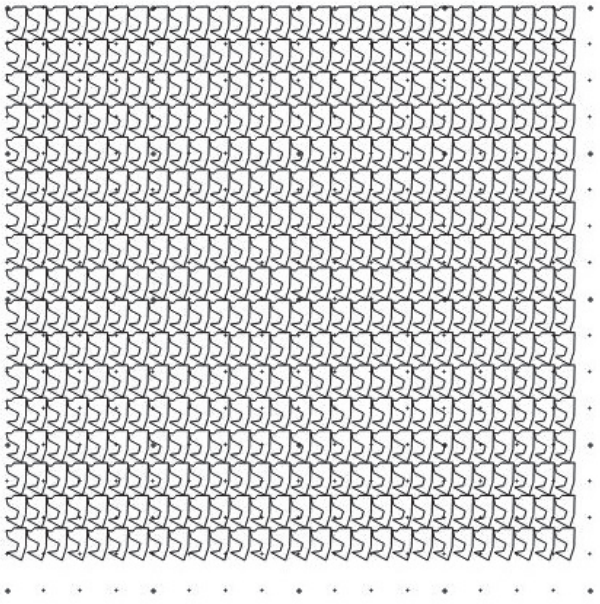
Rechenzeit : 4.164 sek.
Anzahl Polygone : 492
Genutzte Fläche : 90.818
Verschnittfläche : 69.181

Cherry-Pick-Alg mit Unterteilung



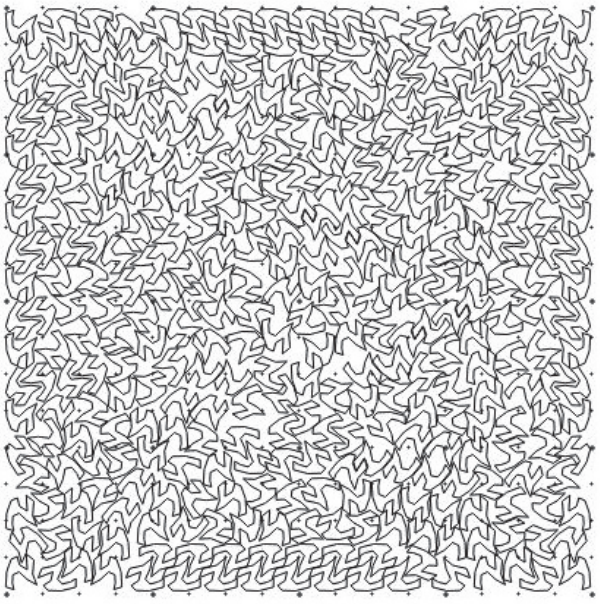
Rechenzeit : 1 min 0 sek.
Anzahl Polygone : 540
Genutzte Fläche : 99.678
Verschnittfläche : 60.321

Cherry-Pick-Alg mit Unterteilung und mehr Positionen



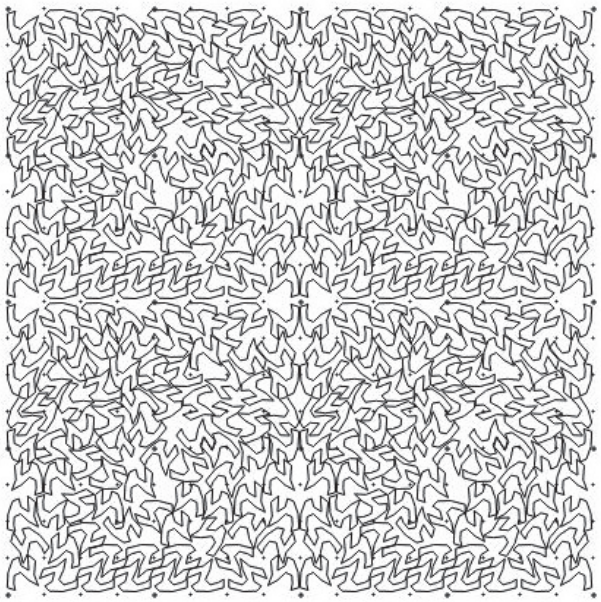
Rechenzeit : 0.078 sek.
Anzahl Polygone : 476
Genutzte Fläche : 87.864
Verschnittfläche : 72.135

Naiver-Algorithmus



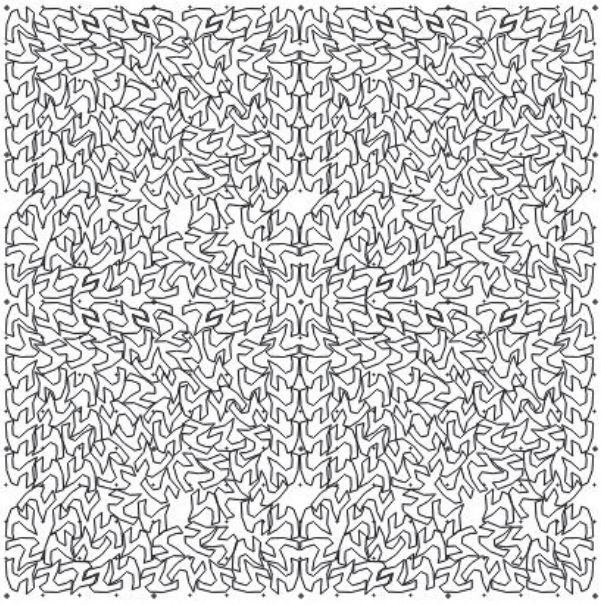
Rechenzeit : 12.68 sek. Genutzte Fläche : 79.315
Anzahl Polygone : 349 Verschnittfläche : 80.684

Cherry-Pick-Alg ohne Unterteilung



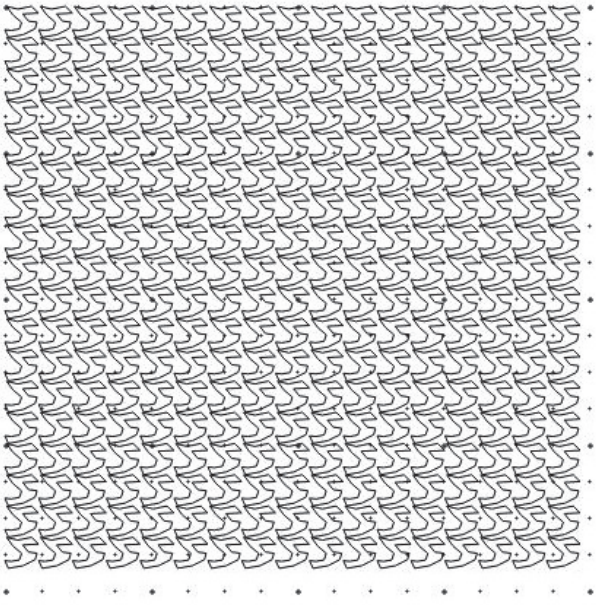
Rechenzeit : 6.107 sek. Genutzte Fläche : 76.361
Anzahl Polygone : 336 Verschnittfläche : 83.638

Cherry-Pick-Alg mit Unterteilung



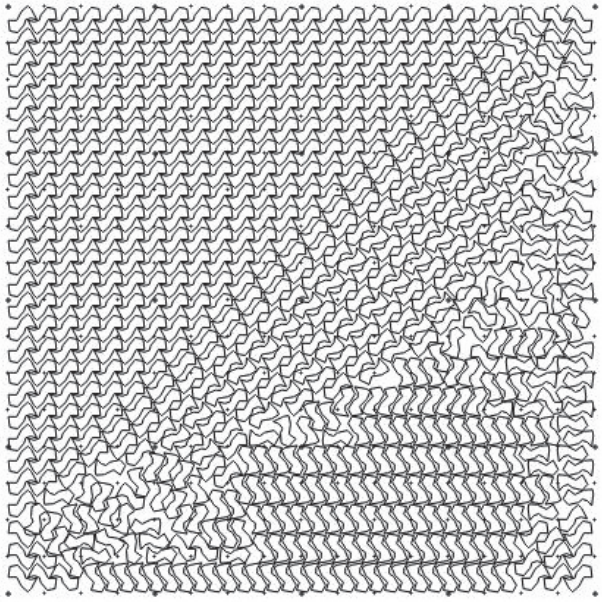
Rechenzeit : 1 min 19 sek. Genutzte Fläche : 81.815
Anzahl Polygone : 360 Verschnittfläche : 78.184

Cherry-Pick-Alg mit Unterteilung und mehr Positionen



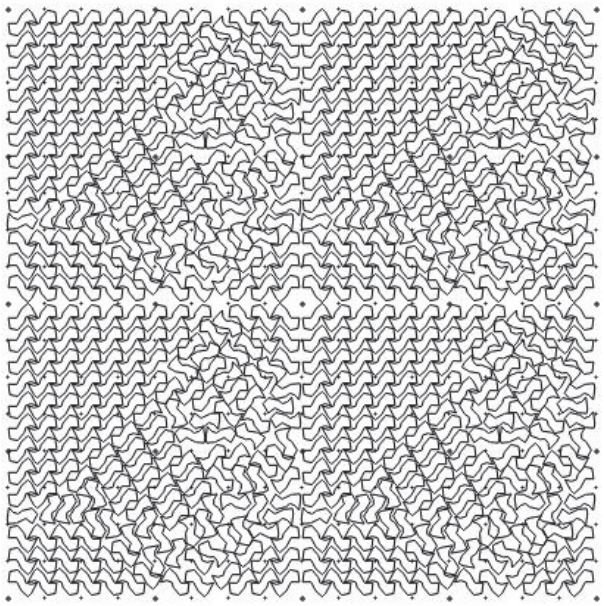
Rechenzeit : 0.085 sek. Genutzte Fläche : 69.543
Anzahl Polygone : 306 Verschnittfläche : 90.456

Naiver-Algorithmus



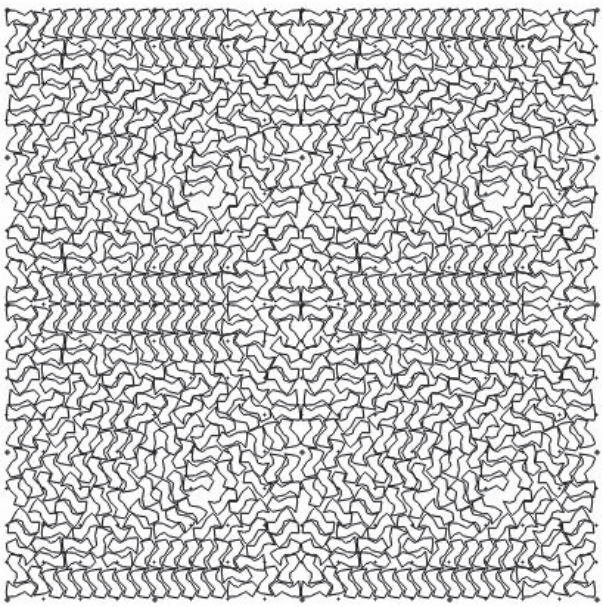
Rechenzeit : 9.516 sek.
Genutzte Fläche : 100.404
Anzahl Polygone : 699
Verschnittfläche : 59.595

Cherry-Pick-Alg ohne Unterteilung



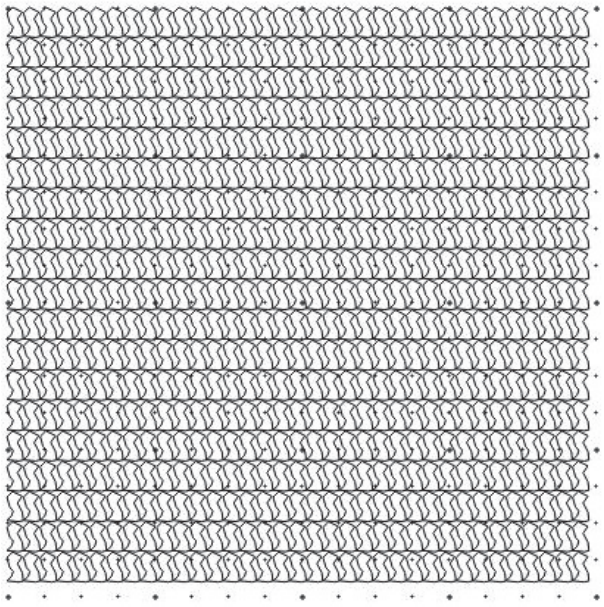
Rechenzeit : 4.298 sek.
Genutzte Fläche : 93.653
Anzahl Polygone : 652
Verschnittfläche : 66.346

Cherry-Pick-Alg mit Unterteilung



Rechenzeit : 41.788 sek.
Genutzte Fläche : 103.995
Anzahl Polygone : 724
Verschnittfläche : 56.004

Cherry-Pick-Alg mit Unterteilung und mehr Positionen



Rechenzeit : 0.08 sek.
Genutzte Fläche : 95.520
Anzahl Polygone : 665
Verschnittfläche : 64.479

Naiver-Algorithmus

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift: 

Stuttgart, 26.09.2014