Software Projekt Anwendungen von Algorithmen

Metaheuristiken II

Robert Lion Gottwald, Lars Parmakerli, Phil Schmidt

Freie Universität Berlin
Institut for Computer Science
robert.gottwald@fu-berlin.de
-Lars - fu-Adresse-
philschmidt@inf.fu-berlin.de
https://www.inf.fu-berlin.de/lehre/WS14/SWPAlg/index.html

Zusammenfassung Der folgende Text befasst sich mit der Arbeit der Gruppe Metaheuristiken II des Softwareprojekts Anwendungen von Algorithmen im Wintersemester 2014/15. Dabei werden wir uns mit zwei spezifischen Metaheuristiken befassen, namentlich Simulated Annealing und ein genetisch-inspiriertes Verfahren, sowie mit dem globalen Lösungsverfahren Branch-and-Bound und seinen Schwierigkeiten für die Anwendung auf unsere Probleme und schließlich zuletzt mit numerischen Verfahren für die nicht-lineare Optimierung, angewendet auf die gewählten Aufgabenstellungen des Stapelns und des Packens.

1 Metaheuristiken

Das Wort Metaheuristik setzt sich aus den beiden Wörtern 'Meta' und 'Heuristik' zusammen. Eine Heuristik ist ein Verfahren, welches für eine bestimmte Art von Problemen eine Lösung findet, die als ausreichend 'gut' erachtet wird. Das Wort 'Meta-' bedeutet in diesem Kontext 'auf einer Höheren Ebene'.

Eine Metaheuristik ist demnach möglichst effizientes Lösungsverfahren, welches sich auf einer höher abstahierten Ebene abspielt und somit kein Wissen über das unterliegende Problem benötigt. Die einzige bestehende Voraussetzung für das Anwenden einer Metaheuristik auf ein System, ist eine Funktion existiert, welche die Güte einer Lösung berechnen kann.

Metaheuristiken stellen nützliche Werkzeuge dar, um Optimierungsprobleme in komplexeren Systemen zu lösen, ohne für diese eine spezifische, kompliziertere Heuristik zu entwerfen, sind also sozusagen wiederverwendbare Lösungswege. Der Nachteil von Metaheuristiken liegt in der Effizienz in Laufzeit, sowie der meist niedrigeren Güte der erzielten Lösung.

In unserem Softwareprojekt bestehen Metaheuristikmodule aus generischen Klassen mit einem Typenparameter für die Anwendungsdomäne sowie mindestens einer Auswertungsfunktion. Einzelne Implementierungen werden in den Folgekapiteln beschrieben.

2 Simulated Annealing

2.1 Inspiration

Metaheuristiken liegen zumeist realen Prozessen zugrunde, welche ein bestimmtes Verhalten aufweisen. Simulated Annealing ist vom namensgebenden Annealing-Prozess aus der Schwerindustrie inspiriert.

Beim Annealing wird Metall stark erhitzt und über einen längeren Zeitraum langsam abgekühlt. Bei den hohen Temperaturen tritt eine starke Molekular-Bewegung im zu bearbeitenden Material auf. Dadurch, dass der Abkühlungsprozess über einen längeren Zeitraum gestreckt wird, reduziert sich die Teilchenbewegung auch nur allmählich. Der Abkühlungsprozess wird so durch etwaige Behandlungsmethoden gesteuert, dass die einzelnen Teilchen einen für einen bestimmten zu erzielenden Effekt, wie zum Beispiel eine stabile Statik, guten Zustand einnehmen.

Wie der Name uns sagt, ist die Herangehensweise beim Simulated Annealing nun so, dass wir einen derartigen physikalischen Prozess über einen abstrakten Wertebereich simulieren.

2.2 Simulation

Das zu behandelnde Metall wird in unserer Implementierung durch einen generischen Typ-Parameter namens 'State' realisiert. Bei der Anwendung auf die uns gegebene Problemstellung, wie zum Beispiel Stapeln, ist dies ein Feld aus Polygonen.

Die Molekularbewegung wird durch eine Umgebungsfunktion namentlich 'Mutator' umgesetzt. Eine Umgebungsfunktion bekommt einen 'State' als Input und gibt einen anderen 'State' zurück, der in der 'Nachbarschaft' des Eingabe-States befindet.

Die benötigte Gütefunktion, die jedem 'State' einen Double-Wert zuweist und für jede Metaheuristik notwendig ist, ist als Quantifizierung des durch das Annealing zu erzielenden Effekts zu interpretieren.

Die Unberechenbarkeit des Materials, beziehungsweise das chaotische Verhalten der Teilchen, lässt sich durch die Temperatur messen. In unserer Simulation wird eine CoolingSchedule-Klasse verwendet, die nach beliebig wählbaren Abkühlmustern, wie zum Beispiel logarithmisches oder lineares Abkühlen, nach jedem Abfragen einer T()-Funktion eine neue Temperatur zurückgibt.

2.3 Ablauf

Simulated Annealing ist ein iteratives Verfahren. Wir nutzen zwei Zeiger; einen, der auf das aktuell beste gefundene Ergebnis zeigt ('best'), und einen, der den aktuellen Ist-Zustand der Domäne speichert ('current'). In jeder Iteration wird ein neuer State in der Umgebung des aktuellen States mithilfe des Mutators generiert.

Anschließend wird die Objective Function dazu genutzt, den neuen State zu bewerten und mit den alten States zu vergleichen. Ist ein neues globales Optimum gefunden worden, wird dieses gespeichert (best).

Für die Bestimmung des neuen current-Zustandes wird eine zufällige Zahl zwischen 0 und der Maximaltemperatur generiert. Im Normalfall nimmt current den nach

Gütefunktion besseren State an, es sei denn es wird ein kleinerer Wert, als die aktuelle Temperatur gewürfelt. Dann nimmt current den schlechteren der beiden Werte an.

Dieser Prozess wird solange fortgeführt, bis entweder eine feste Iterationsanzahl erreicht ist, oder die Lösung 'gut'-genug ist. Unsere Implementierung terminiert einfach nach einer festen Schrittzahl.

3 Genetische Heuristik

3.1 Inspiration

Ähnlich wie beim Simulated Annealing gibt es für den folgenden genetischen Algorithmus Inspiration durch einen real-weltlichen Prozess. In diesem Fall ist es das biologische Konzept der Evolution.

Evolution wird ein biologischer Prozess genannt, der optimale Adaption von Lebensformen an einen bestimmten Lebensraum über mehrere Generationen erlaubt. Dieser Prozess lässt sich in verschiedene Iterationen (Generationen) aufteilen. Jede Iteration kennt verschiedene Phasen, die sie durchläuft, die den Evolutionsvorgang ausmachen:

Natürliche Selektion
: Das Ökosystem um eine Population herum übt einen so genanten Selektionsdruck auf die darin lebenden Organismen aus. Lebensformen, die nicht bestimmten Ansprüchen genügen, werden aus dem Genpool eliminiert.

Rekombination
: Durch sexuelle Fortpflanzung wird es zwei genetisch 'fit'-en Organismen ermöglicht, ihre Gene untereinander auszutauschen. In der nächsten Generation werden ihre Nachkommen eine zufällige Auswahl an Genen ihrer beiden Elternteile tragen.

Mutation
: Im Vorgang der Gen-Rekombination kann es vorkommen, dass Reproduktionsfehler auftreten. Die Folge davon ist, dass die Nachkommen Gen-Eigenschaften haben können, die nicht Teil des ursprünglichen Genpools der Elterngeneration waren. Diese Mutationen treten zufällig auf und haben ebenso willkürliche Auswirkungen.

Nachdem durch diesen Prozess eine neue Generation erschaffen wurde, durchläuft diese Generation erneut den selben Prozess. Dabei verbessert sich langfristig das Gen-Material in Bezug auf den Selektionsdruck des bestehenden Ökosystems. Der folgende Algorithmus versucht sich daran, ein derartiges Ökosystem mit verschiedenen Populationen zu simulieren, um einen ähnlichen Optimierungsprozess zu erzielen.

3.2 Simulation

Ein individuelles Mitglied einer Population (Individuum), wird in unserer Implementierung durch einen generischen Typ-Parameter namens 'State' realisiert. (Der im Bezug auf das biologische Konzept nicht ganz aussagekräftige Name 'State' ist an dieser Stelle der einheitlichen Gestaltung der Schnittstellen für die verschiedenen Metaheuristiken

geschuldet) Im Zusammenhang mit unserem Algorithmus steht ein solches Individuum für eine (potentiell befriedigende) Lösung des gegebenen Problems. Angewendet auf das Stapel-Problem wäre das beispielsweise eine bestimmte Anordnung von Polygonen.

Die natürliche Selektion wird in unserem Algorithmus mit Hilfe von Instanzen des generischen 'Survival'-Typs simuliert. Das entsprechende Interface spezifiziert die 'findSurvivors'-Methode, die, ausgehend von der konkreten Implementierung, aus der aktuellen Population diejenigen Individuen auswählt, die schließlich die neue Generation bilden. Ein naheliegender Ansatz für die Implementierung wäre beispielsweise ein Verfahren, das die n fittesten Individuen (also die n aktuell besten Lösungen für das Problem) auswählt. Eine weitere beliebte Möglichkeit stellen randomisierte Auswahlverfahren dar.

Die Berechnung der Güte eines konkreten Individuums erfolgt mit Hilfe einer Bewertungsfunktion, die über das Interface 'ObjectiveFunction' definiert werden kann. Für das Packen und Stapeln entspricht das beispielsweise der Berechnung der Fläche der konvexen Hülle.

Für die Rekombination steht in unserer Implementierung das 'CrossOver'-Interface zur Verfügung. Hier wird die 'cross'-Methode spezifiziert, die als Parameter Individuen aus der aktuellen Population entgegennimmt und neue zurückgibt. An dieser Stelle abstrahieren wir von dem biologischen Konzept, indem wir nicht nur Rekombinationen von zwei Elternteilen zulassen. Die Methode erhält als stattdessen eine beliebig lange Liste von Individuen und kann ebenso eine beliebig lange Liste von neu kombinierten Individuen zurückgeben.

Das Konzept der Mutation wird durch das 'Mutator'-Interface bereitgestellt. Dies spezifiziert die 'mutate'-Methode, welche aus einem als Parameter übergebenen Mitglied der Population ein neues (mutiertes) Individuum erzeugt. Als konkrete Implementierung für das Packen-Problem ist beispielsweise randomisiertes Verschieben und Drehen der einzelnen Polygone möglich oder ein Verschieben bis zum nächsten Schnittpunkt mit einem anderen Polygon.

3.3 Ablauf

Dem Konstruktor der 'GeneticAlgorithm'-Klasse werden zunächst folgende Parameter übergeben:

1. Die gewünschte Größe einer Population
2. Die gewünschte Anzahl an Iterationen (Eine Iteration entspricht der Herausbildung einer neuen Generation)
3. Die Anzahl der mutierten und rekombinierten Individuen für jede Iteration
4. Die 'Survival'-, 'CrossOver'-, 'ObjectiveFunction' und 'Mutator'-Objekte

Aus einem initial gegeben 'State' wird zunächst mit Hilfe von Mutationen die Ausgangspopulation erstellt. Das beste Individuum wird als aktuell beste Lösung gespeichert. Anschließend werden iterativ neue Generationen herausgebildet, wobei durch Mutation und Rekombination solange neue Individuen zu der aktuellen Population hinzugefügt werden, bis die spezifizierte Anzahl erreicht ist. Anschließend wird durch das

'Survival'-Objekt die neue Generation ausgewählt. Nach jeder Iteration wird erneut ermittelt ob ein besseres Individuum, als das aktuell beste erzeugt wurde. Nachdem die festgelegte Anzahl von Iterationen durchlaufen wurde wird das die beste Lösung zurückgegeben.

4 Branch and Bound

Branch and Bound ist ein Entwurfsmuster für Lösungsalgorithmen von Optimierungsproblemen. Unserer Gruppe wurde von Veranstalterseite empfohlen, uns mit diesem System zu befassen und die Möglichkeiten, einer Anwendung der Branch and Bound Methode zur Konstruktion eines Algorithmus für die Bearbeitung des Stapel- oder Packen-Problems, einzuschätzen.

In den folgenden beiden Abschnitten werden wir das generelle Entwurfsschema Branch and Bound erläutern und seine Tauglichkeit für die Behandlung der Problemstellungen des Projekts evaluieren.

Das Verfahren Branch and Bound basiert auf einer selektiven Suche durch bestimmte, gewählte Teilmengen der Menge aller möglichen Belegungen. Das Vorgehen besteht aus den zwei namensgebenden Schritten Branch und Bound.

Im Branch-Schritt wird ein Lösungsbaum generiert, in dem iterativ das Problem mit jedem Schritt in immer weiter vereinfachte Teilprobleme aufgeteilt wird. Zur Veranschaulichung: Eine der einfachsten Vereinfachungen wäre es, einen Parameter auf einen fixen Wert festzulegen, oder auf einen bestimmten Wertebereich einzugrenzen. Dadurch entsteht ein möglicherweise sehr großer Baum, der alle möglichen Parameterbelegungen hierarchisch ordnet.

Der Bound-Schritt entfernt nun Teilprobleme, deren mögliche Lösungen nicht ausreichend gut sind. Für jedes Teilproblem wird zu diesem Zwecke eine obere und eine untere Schranke berechnet. Dazu können beispielsweise Heuristiken oder spezifische Kennwerte des Problems herangezogen werden. Sollten nun bei einem Minimierungsproblem, wie unserem, Teilprobleme/Zweige mit einer unteren Schranke gefunden werden, die größer als die obere Schranke eines anderen Teilproblems, so brauchen wir in diesem Zweig nicht weiter nach einer Lösung suchen. Der Zweig wird dann entsprechend vom Baum abgetrennt und nicht weiter aufgefächert.

Tauglichkeit Branch and Bound lässt sich theoretisch auf jede Art von Problem anwenden, jedoch stellt sich stets die Frage, ob für eine Zerlegung Schranken gefunden werden können, die scharf genug sind, um die Suche auf eine Teilmenge einzuschränken, die klein genug ist. Findet sich das nicht, entartet Branch and Bound zu einer einfachen brute-force Suche durch den Lösungsraum nach einer guten Lösung.

Diese Schwierigkeit findet sich auch bei der Anwendung auf Packen und Stapeln wieder. Branches können zwar generiert werden in dem wir zum Beispiel beim Packen gemeinsame Kanten von Polygonen festlegen, oder die Polygone diskreten, räumlichen Abschnitten zuweisen, jedoch findet sich keine einfache Methode, sinnvolle Schranken für diese Teilprobleme zu berechnen. Natürlich kann beispielsweise der gesamte

Flächeninhalt der Polygone als untere Schranke gewählt werden, allerdings ändert sich diese nicht für verschiedene Teilprobleme. Demnach ist Branch and Bound für unseren Anwendungsfall tendenziell ungeeignet.

5 Numerische Verfahren

5.1 Nicht-lineare Optimierung ohne Nebenbedingungen

Stapeln als nicht-lineares Optimierungsproblem Das Stapelproblem lässt sich als Optimierungsproblem einer nicht-linearen Zielfunktion $f : \mathbb{R}^n \to \mathbb{R}$ ohne Nebenbedingungen formulieren. Der Eingabevektor der Zielfunktion enthält Translations-Parameter und Rotationswinkel für jedes Polygon. Für m zweidimensionale Polygone hat er also $3m$ Komponenten. Die Zielfunktion f ordnet diesem Vektor dann die Fläche der konvexen Hülle bzw. der Bounding-Box zu.

Newton-Verfahren Für derartige Optimierungsprobleme gibt es verschiedene mathematische Verfahren zur numerischen Berechnung lokaler Optima. Viele dieser Verfahren basieren auf dem Newton-Verfahren zum Finden von Nullstellen einer Funktion. Um das Newton-Verfahren zur Optimierung einzusetzen, wendet man es auf die erste Ableitung f' an. Denn an den Nullstellen der ersten Ableitung liegen auch die Extremwerte der Funktion. Allerdings benötigt das Newton-Verfahren auch noch die erste Ableitung der Funktion, auf die es angewendet wird, deshalb wird auch noch die zweite Ableitung f'' benötigt, wenn man das Newton-Verfahren zur Optimierung verwendet.

Das Verfahren berechnet die Lösung iterativ. Eine gegebene Lösung x_n wird in jedem Schritt mit der Formel

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

weiter der optimalen Lösung angenähert.

Bei der Minimierung der Funktion aus 5.1 ist die erste Ableitung eine vektorwertige Funktion, die einen Vektor mit partiellen Ableitungen, den Gradienten, zurückgibt $f' : \mathbb{R}^n \to \mathbb{R}^n$ und die zweite Ableitung gibt dann sogar eine $n \times n$-Matrix zurück, nämlich die partielle Ableitung nach jedem der n Eingabeparameter für jede der n Komponenten des Ausgabevektors der ersten Ableitung $f'' : \mathbb{R}^n \to \mathbb{R}^{n \times n}$. Diese Matrix wird Hesse-Matrix genannt.

Um das Newton Verfahren für vektorwertige Funktionen einzusetzen wird die Formel zu

(1) $x_{n+1} = x_n - H^{-1}g$

Dabei ist $H := f''(x_n)$ die Hesse-Matrix und $g := f'(x_n)$ der Gradient.

Anstatt aber die Inverse direkt zu berechnen, ist es günstiger mit Hilfe des lineare Gleichungssystemss

(2) $Hs_n = -g$

s_n zu berechnen und dann den nächsten Lösungskandidaten $x_{n+1} = x_n + s_n$.

Ableitungen Da wir in der Implementierung der Zielfunktion 5.1 die Ableitungen nicht gegeben haben, müssen wir sie numerisch approximieren. Mit der Finite-Differenzen-Methode benötigt die Berechnung der ersten Ableitung zwei Funktionsauswertungen pro Komponente, also insgesamt $2n$ Funktionsauswertungen. Würde man nun die zweite Ableitung f'' ebenfalls numerisch approximieren, bräuchte man $2n$ Auswertungen von f', also insgesamt $4n^2$ Auswertungen der ursprünglichen Funktion f. Das ist nicht nur sehr aufwendig, sondern auch problematisch für die numerische Stabilität des Verfahrens. Denn die Finite-Differenzen-Methode führt leicht zu Auslöschungen, die bei der Subtraktion fast gleich großer Gleitkommazahlen auftreten.

Außerdem ist die Ableitung der Zielfunktion für das Stapelproblem nicht glatt, denn wenn ein Polygon sich komplett innerhalb der konvexen Hülle befindet, ist die Ableitung bezüglich dessen Translations- und Rotationsparameter null. Sobald das Polygon jedoch so weit verschoben wird, dass es den Rand erreicht, wird die Ableitung positiv, hat also an dieser Stelle einen Knick und die zweite Ableitung ist somit nicht stetig. Beweise für die Konvergenzeigenschaften von Newton-artigen Verfahren setzen jedoch mindestens zweimal stetig differenzierbare Funktionen voraus.

In der Praxis hat sich jedoch gezeigt, dass es Newton-artige Verfahren zur Optimierung gibt, die gut für nicht-glatte Funktionen funktionieren [1] und ohne zweite Ableitungen auskommen.

Quasi-Newton Verfahren Sogenannte Quasi-Newton Verfahren, sind Newton-artige Verfahren, die die direkte Berechnung der zweiten Ableitung vermeiden. Stattdessen wird eine Approximation der Hesse-Matrix verwaltet, die in jeder Iteration des Algorithmus mithilfe der Informationen der ersten Ableitung weiter der echten Hesse-Matrix angenähert wird. Dafür existieren verschiedene Update-Formeln. Wir verwenden die BFGS Update-Formel, benannt nach ihren Erfindern **B**royden, **F**letcher, **G**oldfarb und **S**hanno. [2]

$$B_n + 1 = B_n + \frac{(y_n - B_n s_n)(y_n - B_n s_n)^T}{(y_n - B_n s_n)^T s_n}$$

wobei B_n die Approximation der Hesse-Matrix in Schritt n ist und

$$s_n := x_{n+1} - x_n = -B_n^{-1} * g \qquad\qquad y_k := f'(x_{n+1}) - f'(x_n)$$

Außerdem gibt es noch eine Variante der Formel um direkt die Inverse Hesse-Matrix zu approximieren. Das hat den Vorteil, dass der Quasi-Newton-Schritt wie in Gleichung (1) berechnet werden kann und daher kein lineares Gleichungssystem in jedem Schritt gelöst werden muss wie in Gleichung (2). Diese Variante verwenden wir auch in unserer Implementierung des BFGS-Verfahrens.

$$(3)\quad B_{n+1}^{-1} = (I - p_n s_n y_n^T) B_n^{-1} (I - p_n y_n s_n^T) + p_n s_n s_n^T \qquad\qquad p_n = \frac{1}{y_n^T s_n}$$

Line search Für die globale Optimierung mit (Quasi-)Newton-Verfahren ist es oft nicht sinnvoll immer den vollen Schritt s_n zu verwenden, sondern diesen zu skalieren. Die

Schrittweite α soll so gewählt werden, dass der Funktionswert möglichst klein wird, also die univariate Funktion

$$\Phi(\alpha) = f(x_n + \alpha s_n)$$

minimal wird.

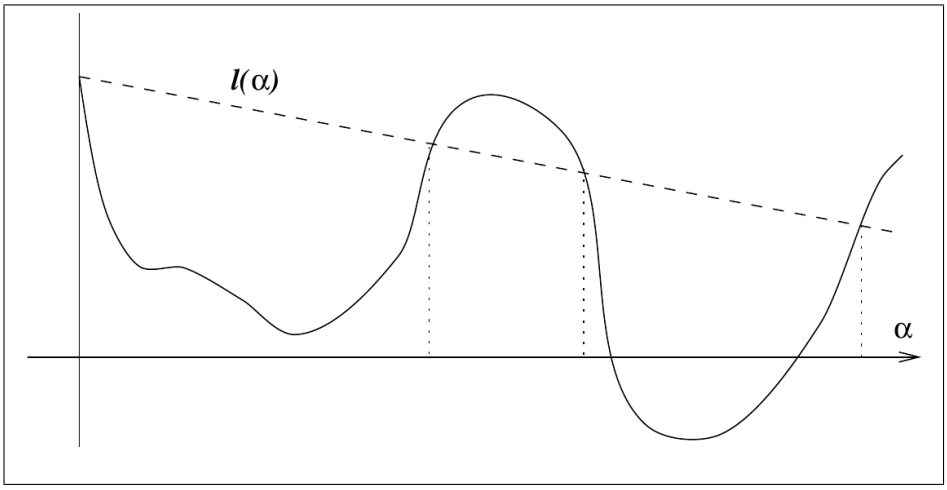


Abbildung 1: Armijo Bedingung [2]

Dazu ist es wichtig, dass die Richtung s_n eine Abstiegsrichtung ist, also der Funktionswert in dieser Richtung zunächst abnimmt. Das ist im Allgemeinen sehr aufwändig und es lohnt sich meistens nicht, diesen Aufwand in jeder Iteration zu betreiben. Es reicht in der Praxis bestimmte Anforderungen an die gewählte Schrittweite zu stellen, sodass die Konvergenz garantiert werden kann. Damit die Schrittweite nicht zu groß wird, verwendet man häufig die Armijo-Bedingung.

(4) $f(x_n + \alpha s_n) \leq f(x_n) + c_1 \alpha f'(x_n)^T s_n$

Dabei ist $f'(x_n)^T s_n = \Phi'(0)$ die Richtungsableitung von f in Richtung s_n, welche negativ ist wenn s_n eine Abstiegsrichtung ist. Die Bedingung erzwingt also, dass die Zielfunktion f in Richtung s_n proportional zur Steigung im Punkt x_n und proportional zur Schrittweite abnimmt. D.h. große Schritte werden nur akzeptiert, wenn die Funktion entsprechend stark abgenommen hat. Die Konstante $c_1 \in (0, 1)$ kann frei gewählt werden, wobei kleinere Werte die Bedingung abschwächen.

Die gestrichelte Linie in Abbildung 1 veranschaulicht die Armijo-Bedingung. Die Funktionswerte, die unter ihr liegen, erfüllen die Bedingung, die Funktionswerte darüber jedoch nicht. Diese Bedingung alleine reicht aber meistens nicht um eine Konvergenz zu garantieren, denn ein genügend kleiner Schritt erfüllt sie immer. Um zu kleine Schritte zu vermeiden, stellt man deswegen oft noch die Krümmungs-Bedingung.

(5) $f'(x_n + \alpha * s_n)^T s_n \geq c_2 f'(x_n)^T s_n$

Diese Bedingung erzwingt, dass die Funktion im gewählten Punkt nicht mehr so stark abnehmen darf wie im Punkt x_n. Wenn nämlich die Steigung in einem Punkt noch stark negativ ist, ist es naheliegend noch weiter in diese Richtung zu gehen. Die Konstante $c_2 \in (c_1, 1)$ kann, wie c_1, frei gewählt werden, wobei kleinere Werte die Bedingung verstärken. Wir verwenden die Werte $c_1 = 1e - 4$ und $c_2 = 0.9$, da sich durch Experimentieren gezeigt hat, dass strengere Werte die Gesamtlaufzeit eher negativ beeinflussen.

Algorithmus 1 : Line search [1]

Eingabe : Konstanten c_1, c_2, Zielfunktion f, aktuelle Lösung x, Suchrichtung s
Ausgabe : Eine Schrittweite α, die die Wolfe-Bedingungen erfüllt

$u \leftarrow 0$
$v \leftarrow \infty$
$\alpha \leftarrow 1$
while *true* do
  if *Bedingung (4) ist für* α *verletzt* **then**
    $v \leftarrow \alpha$
  end
  else if *Bedingung (5) ist für* α *verletzt* **then**
    $u \leftarrow \alpha$
  end
  else
    return α
  end
  if $v < \infty$ **then**
    $\alpha \leftarrow \frac{u+v}{2}$
  end
  else
    $\alpha \leftarrow 2\alpha$
  end
end

Die Bedingungen (4) und (5) zusammen nennt man die Wolfe-Bedingungen. Fügt man zur linken und rechten Seite der Bedingung (5) noch Beträge hinzu, dann spricht man von den starken Wolfe-Bedingungen. Wir verwenden jedoch die Variante ohne, denn die starken Wolfe-Bedingungen eignen sich nur für glatte Funktionen [1].

Ein recht einfacher und für nicht-glatte Funktionen geeigneter Algorithmus um einen Punkt zu finden, welcher die Bedingungen (4) und (5) erfüllt, ist **Algorithmus 1**. Beginnend mit der Schrittweite $\alpha = 1$ verdoppelt er diese solange, bis ein Punkt gefunden wurde, der die Armijo-Bedingung verletzt. Dieser Wert für die Schrittweite wird dann als obere Schranke v gesetzt. Daraufhin wird Bisektion verwendet, um im Intervall (u, v) einen Punkt zu finden, der beide Bedingungen, (4) und (5), erfüllt. Immer wenn die Bedingung (4) verletzt ist, wird in der linken Intervallhälfte weitergesucht und wenn Bedingung (5) verletzt ist, dann in der rechten Hälfte.

Algorithmus 2 : BFGS Algorithmus

Eingabe : Startwert x_0, Zielfunktion f, maximale Iterationszahl N
Ausgabe : Eine bessere bzw. lokal optimale Lösung x_{BFGS}

Wähle B_0^{-1} positiv Definit (z.B als Identitätsmatrix)
foreach $i \in [0, N)$ **do**
  Berechne $s_i = -B_i^{-1} f'(x_i)$.
  Wähle α mit **Algorithmus 1**
  if *Abbruchbedingung erreicht* **then**
    return $x_{\text{BFGS}} = x_i$
  end
  $x_{i+1} \leftarrow x_i + \alpha s_i$.
  Berechne B_{i+1}^{-1} nach Formel (3)
end
return $x_{\text{BFGS}} = x_{N-1}$

Die BFGS-Updateformel erhält die positive Definitheit der approximierten Hesse-Matrix, wenn der nächste gewählte Punkt die Wolfe-Bedingungen erfüllt. Ist die Hesse-Matrix positiv definit, liefert der Newton-Schritt (1) garantiert eine Abstiegsrichtung. **Algorithmus 2** zeigt den BFGS-Algorithmus, wie wir ihn implementiert haben. Der Algorithmus terminiert wenn die maximale Iterationszahl erreicht wurde, oder wenn die Abbruchbedingung, z.B. dass der Line search Algorithmus keinen geeigneten Schritt mehr findet, erfüllt ist.

5.2 Nicht-lineare Optimierung mit Nebenbedingungen

Packen als nicht-lineares Optimierungsproblem Zur Formulierung des Packproblems werden, im Gegensatz zu dem Stapelproblem, auch Nebenbedingungen benötigt. Und zwar, dass für jedes Paar von Polygonen i, j mit $i \neq j$ die Überlappungsfläche $c_{ij} = 0$ ist. Für die nicht-lineare Optimierung mit Nebenbedingungen gibt es verschiedene Ansätze.

Im Rahmen des Softwareprojekts kamen Penalty-, Barriere-, und augmentierte Lagrange-Verfahren in Frage, da diese nicht zu Komplex sind und trotzdem gute Ergebnisse möglich sind [2]. Diese Verfahren haben gemeinsam, dass sie die Nebenbedingungen behandeln, indem sie zusätzliche Terme in die Zielfunktion einfügen und dann eine Folge von nicht-linearen Optimierungsproblemen ohne Nebenbedingungen auf dieser modifizierten Zielfunktion lösen. Der Vorteil dabei ist, dass die Algorithmen aus Abschnitt 5.1 wiederverwendet werden können, um die Teilprobleme zu lösen.

Für das Packproblem haben wir Barriere-Verfahren als weniger gut geeignet eingestuft. Barriere-Verfahren arbeiten mit Bestrafungs-Termen, die schon groß werden, wenn die Lösung sich einer ungültigen Lösung annähert. So wird eine Suche durch den Raum der gültigen Lösungen ausgeführt. Im Packproblem sind jedoch gerade die Lösungen optimal, bei denen sich zwei Polygone berühren, also durch die Barriere-Terme schon unendlich stark bestraft werden.

Das Penalty-Verfahren funktioniert ebenfalls mit Bestrafungs-Termen in der Zielfunktion, welche aber erst ansteigen, wenn die Nebenbedingungen auch tatsächlich verletzt sind. Bei beiden Verfahren gibt es jedoch numerische Schwierigkeiten [2]. Das augmentierte Lagrange-Verfahren basiert auf dem Penalty-Verfahren, erweitert die Zielfunktion noch um einen zusätzlichen Term, der die numerische Stabilität des Verfahrens verbessert. Auch gibt es bereits Software, die dieses Verfahren implementiert und gut zu funktionieren scheint [3]. Vor diesem Hintergrund haben wir uns schließlich für das augmentierte Lagrange-Verfahren entschieden.

Augmentiertes Lagrange-Verfahren Das Verfahren fügt zwei Terme zur Zielfunktion hinzu, so dass sie die folgende Form hat.

$$\mathcal{L}(x, \lambda, \mu) := f(x) - \sum_{\{i,j\} \in \binom{P}{2}} \lambda_{ij} c_{ij}(x) + \frac{1}{2\mu} \sum_{\{i,j\} \in \binom{P}{2}} c_{ij}(x)^2 \tag{6}$$

Dabei bezeichnet P die Menge der Polygone und $c_{ij}(x)$ die Überlappungsfläche der Polygone i und j für eine Lösung x. Der Parameter μ skaliert den dritten Term der Zielfunktion, der die Verletzung der Nebenbedingungen quadratisch bestraft.

Zu jeder Nebenbedingung c_{ij} gibt es zugehörige Lagrange-Multiplizierer λ_{ij}. Diese skalieren im zweiten Term der Funktion die Verletzung der jeweils zugehörigen Nebenbedingung. Nur durch diesen Term unterscheidet sie sich von der Zielfunktion des Penalty-Verfahrens. Dieser Term bewirkt, dass μ im augmentierten Lagrange-Verfahren nicht so klein gewählt werden muss, um eine gültige Lösung zu erhalten. Das verringert die Wahrscheinlichkeit von numerischen Problemen, die entstehen können, wenn mit sehr kleinen Gleitkommazahlen gerechnet wird.

Die Idee ist nun in Iteration n die Zielfunktion $\mathcal{L}$ für ein festes μ_n und λ^n, den Vektor mit allen Lagrange-Multiplizierern, ohne Nebenbedingungen zu minimieren, z.B. mit dem BFGS-Algorithmus. Dann wird $\mu_{n+1} \leq \mu_n$ gewählt und λ^{n+1} mit der Formel

$$\lambda_{ij}^{n+1} = \lambda_{ij}^n - c_{ij}(x_n)/\mu_n \tag{7}$$

berechnet. In Iteration $n + 1$ wird dann erneut die Zielfunktion $\mathcal{L}$ minimiert, welche sich durch die neuen Werte für μ und den Vektor λ^{n+1} verändert hat. Dabei wird die Lösung der Iteration n als Ausgangspunkt benutzt.

Für gewöhnlich ist μ Anfangs groß. Verletzungen der Nebenbedingungen werden also kaum bestraft. Dann wird μ immer kleiner, wodurch die Bestrafung der Nebenbedingungen zunimmt. Gleichzeitig nähern sich durch die Formel (7) die Werte des Vektors λ den optimalen Lagrange-Multiplizierern an [2]. Das führt dazu, dass die Verletzung der Nebenbedingungen letzendlich immer kleiner wird, bis eine gültige Lösung im Rahmen der gewünschten Genauigkeit gefunden wurde. Bezogen auf die Polygone heißt das, dass diese Anfangs gestapelt werden, um dann schrittweise wieder auseinander gezogen zu werden.

Diesen Prozess kann man gut in Abbildung 2 beobachten. Dort sieht man eine Serie von Bildern ausgewählter Iterationen einer Ausführung des Algorithmus. Die Bilder wurden mithilfe der Option erstellt, die Animationen während der Ausführung des Algorithmus als GIF zu exportieren. Sie stellen also eine tatsächliche Ausführung des

Algorithmus in der Abgabeversion des Softwareprojekts dar. Iterationen, in denen der Fortschritt nicht gut sichtbar ist, wurden ausgelassen. Unter den Bildern ist die Fläche der konvexen Hülle unter der Bezeichnung f, wie in Abschnitt 5.1, angegeben, sowie die Summe der paarweisen Überlappungsflächen $\sum c_{ij}$.

Ableitungen Um die Ableitung der Zielfunktion (6) zu berechnen ist es wichtig, nicht die gesamte Funktion numerisch abzuleiten. Denn durch die Skalierung mit μ und λ ist so keine hinreichend genaue Ableitung der Nebenbedingungen möglich. Stattdessen verwenden wir die Formel [2]

$$\mathcal{L}(x, \lambda, \mu)' = f'(x) - \sum_{\{i,j\} \in \binom{P}{2}} (\lambda_{ij} - c_{ij}(x)/\mu) c'_{ij}(x)$$

Dabei berechnen wir $f'(x)$ numerisch, wie in Abschnitt 5.1, und $c'_{ij}(x)$ ebenfalls. Dafür nutzen wir die besondere Struktur von $c'_{ij}(x)$ aus: Es sind nur sechs Einträge nicht null, nämlich die Translations- und Rotationsparameter von Polygon i und j. Zusätzlich sind die Ableitungen nach den Translationsparametern von Polygon i die negierten Werte der Parameter von Polygon j, denn wenn die Überlappung durch Verschiebung in positiver x-Richtung von Polygon i kleiner wird, muss sie durch Verschiebung in negativer x-Richtung von Polygon j um denselben Wert kleiner werden. Dadurch müssen nur vier Einträge des Ableitungsvektors $c'_{ij}(x)$ pro Nebenbedingung tatsächlich berechnet werden.

Algorithmus 3 : Augmentierter Lagrange-Algorithmus

Eingabe : Startwert x_0, Zielfunktion f, Nebenbedingungen $c_i j$ maximale Iterationszahl N
Ausgabe : Eine gültig, bzw. nahezu gültige lokal optimale Packung x_{PACK}

$\mu = 10^6$
foreach c_{ij} **do**
  $\lambda_{ij} \leftarrow 0$
end
foreach $i \in [0, N)$ **do**
  Sei $g(x) := \mathcal{L}(x, \lambda, \mu)$
  $x_{i+1} \leftarrow BFGS(x_0 = x_i, f = g, N = 500)$
  if *Lösung gültig für gewünschte Genauigkeit* **then**
    return $x_{\text{PACK}} = x_{i+1}$
  end
  $\mu \leftarrow 0.5\mu$
  foreach c_{ij} **do**
    Update λ_{ij} nach Formel (7)
  end
end
return $x_{\text{PACK}} = x_{N-1}$

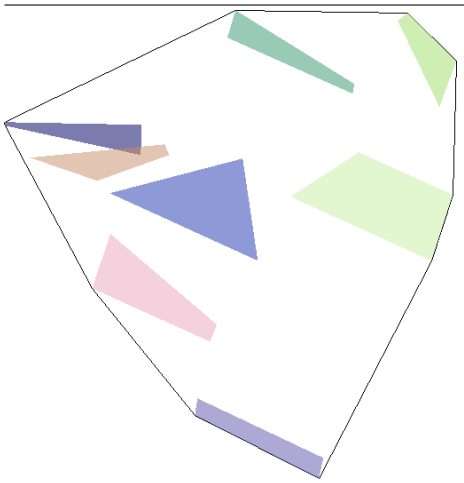
Algorithmus Wie wir das augmentierte Lagrange-Verfahren zum Lösen des Packproblems implementiert haben, ist in **Algorithmus 3** zu sehen. Unsere Implementierung funktioniert jedoch nicht immer zuverlässig. Das ist dem Umstand geschuldet, dass für eine der komplexesten Funktionen der Geometrie-Gruppen, nämlich die Überlappungsfläche zweier Polygone zu berechnen, nicht genug Zeit war diese ausgiebig zu testen. Diese Funktion ist jedoch essentiell für dieses Verfahren, denn wenn es während der Ausführung zu Auswertungsfehlern der Überlappungsfläche kommt, so können riesige Werte in den Ableitungen der Nebenbedingungen entstehen.

Das kann dann dazu führen, dass nicht nachvollziehbare Schritte gewählt werden oder das Programm in ungültigen Konfigurationen stecken bleibt. In einer Implementierung dieser Funktion wird z.B. die Überlappungsfläche 0 zurückgegeben, wenn ein Polygon in einem anderen komplett enthalten ist. Wenn der Algorithmus erfolgreich durchläuft, sind jedoch oft vielversprechende Ergebnisse zu beobachten. Um den Nachteil zu umgehen, dass die numerischen Verfahren nur lokale Optima finden können, wäre es auch denkbar, diese mit klassischen Metaheuristiken zu kombinieren.

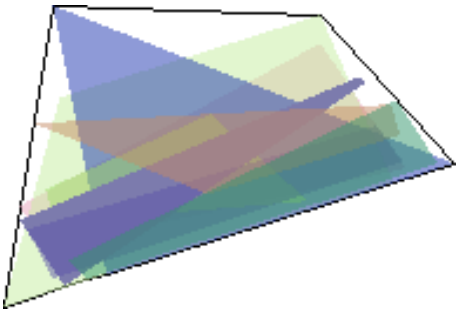
Literatur

1. A. S. Lewis and M. L. Overton, “Nonsmooth optimization via quasi-newton methods,” *Math. Program.*, vol. 141, no. 1-2, pp. 135–163, 2013.
2. J. Nocedal and S. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research. Springer, 1999.
3. A. R. Conn, N. I. M. Gould, and P. L. Toint, *Lancelot: A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, 1st ed. Springer Publishing Company, Incorporated, 2010.

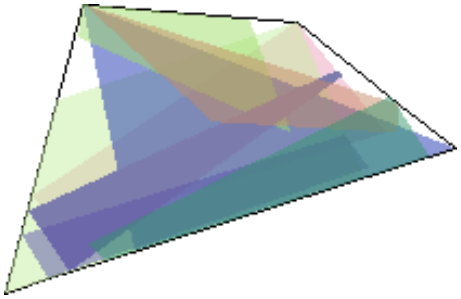
Abbildung 2 Packen mit dem augmentierten Lagrange-Verfahren



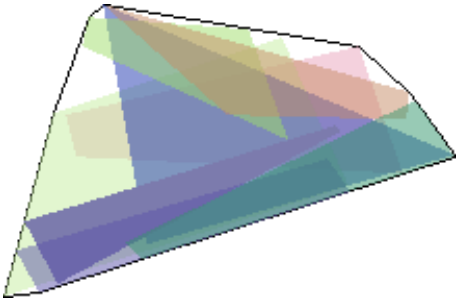
Startkonfiguration:
$f = 174675.7631$, $\sum c_{ij} = 158.7219$



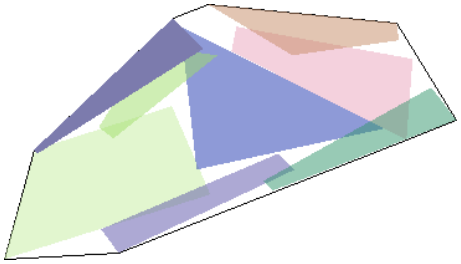
Iteration 1 ($\mu = 10^6$):
$f = 15084.4567$, $\sum c_{ij} = 64343.8222$



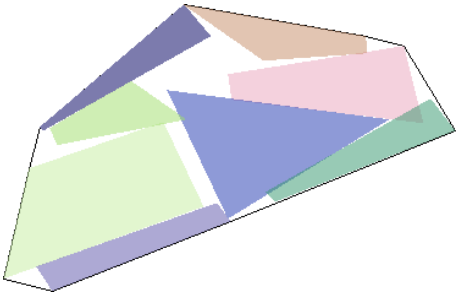
Iteration 6 ($\mu = 31250$):
$f = 15087.5107$, $\sum c_{ij} = 51559.7376$



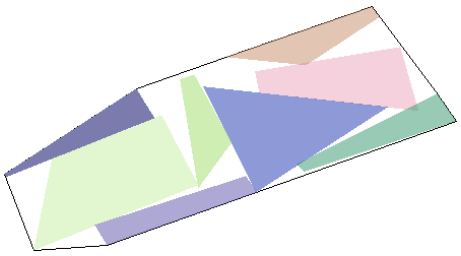
Iteration 10 ($\mu = 1953.125$):
$f = 17536.8763$, $\sum c_{ij} = 42119.6406$



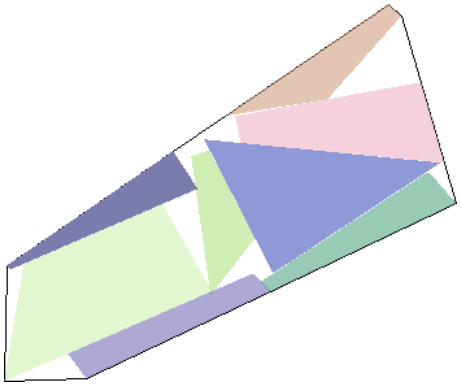
Iteration 11 ($\mu = 976.5625$):
$f = 43590.2132$, $\sum c_{ij} = 3488.8509$



Iteration 19 ($\mu = 3.8147$):
$f = 48315.87$, $\sum c_{ij} = 294.2552$



Iteration 20 ($\mu = 1.9073$):
$f = 52861.6379$, $\sum c_{ij} = 91.6183$



Iteration 21 ($\mu = 0.9537$):
$f = 47591.331$, $\sum c_{ij} = 14.0686$