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 nicht-linearen Optimierungsverfahren unter Zuhilfenahme numerischer Verfahren bezogen auf die gewählten Aufgabenstellungen de 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 Optimierungsproleme 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 Metal 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 gesteuert durch etwaige Behandlungsmethoden, 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 "fiten 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 Reproduktions-Fehler 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

4 Branch and Bound

5 Nicht-lineare Optimierung mit numerischen Verfahren des Stapelproblems

5.1 Formulierung als nicht-lineares Optimierungsproblem

Das Stapelproblem lässt sich als Optimierungsproblem einer nicht-linearen Zielfunktion ohne Nebenbedingungen formulieren. Die Zielfunktion $f : \mathbb{R}^n \to \mathbb{R}$ erhält die Translations-Vektoren und die Rotationswinkel für jedes der gegebenen Polygone als Eingabe und ordnet diesen die Fläche der konvexen Hülle, bzw. der Bounding-Box, zu. Im zweidimensionalen sind das also drei Parameter für jedes Polygon.

5.2 Lösungs-Verfahren

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. Wenn die Lösung dann zu einer Nullstelle konvergiert ist, ist die erste Ableitung null und damit ein Extrempunkt erreicht. 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 den Iterations-Schritt $s_n := x_{n+1} - x_n$ durch lösen des linearen Gleichungssystems

(2) $H s_n = -g$

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

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 setzten jedoch 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.

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

Die BFGS-Updateformel hat zusätzlich noch die Eigenschaft

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

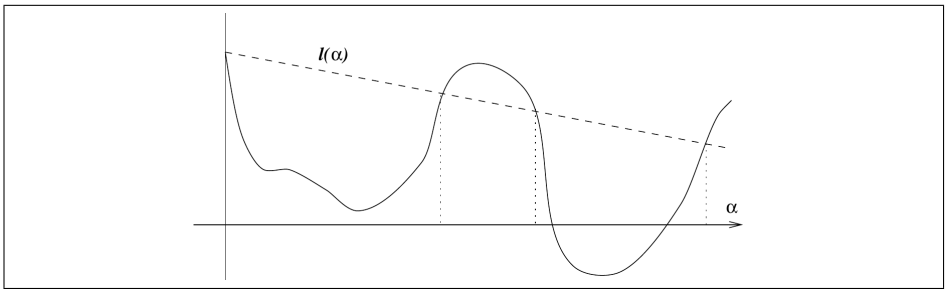

Abbildung 1: Armijo Bedingung [2]

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, wird häufig die Armijo-Bedingung verwendet.

(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. 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-Bedinung. Die Funktionswerte, die unter ihr liegen, erfüllen die Bedingung, die Funktionswerte darüber jedoch nicht. Die Armijo-Bedingung alleine reicht aber nicht um eine Kovergenz zu garantieren, denn ein genügend kleiner Schritt erfüllt die Bedingung immer. Dazu eignet sich 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 abnimmt wie im Punkt x_n. Wenn nämlich die Steigung in einem Punkt noch stark negativ ist, macht es Sinn 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

Eingabe Startwert x_0, Zielfunktion f, maximale Iterationszahl N, Konstanten c_1, c_2.
Ausgabe Eine bessere bzw. lokal optimale Lösung x_{BFGS}.

1. Wähle $u = 0$, $v = \infty$ und $\alpha = 1$
2. Wiederhole:
   (a) Wenn Bedingung (4) für α verletzt ist, setze $v = \alpha$
   (b) Ansonsten wenn Bedingung (5) für α verletzt ist, setzte $u = \alpha$.
   (c) Ansonsten gebe α zurück.
   (d) Wenn $v < \infty$ setze $\alpha = \frac{u+v}{2}$
   (e) Ansonsten setze $\alpha = 2\alpha$

Abbildung 2: Line search Algorithmus [1]

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.

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 Berträge, 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 in Abbildung (2) beschrieben. 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.

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

1. Wähle B_0^{-1} positiv Definit (z.B als Identitätsmatrix).
2. Für jedes $i \in [0, N)$:
   (a) Berechne $s_i = -B_i^{-1} f'(x_i)$.
   (b) Wähle α mit dem Algorithmus in Abbildung 2.
   (c) Wenn Abbruchbedingung erreicht gebe $x_{\text{BFGS}} = x_i$ zurück.
   (d) Setzte $x_{i+1} = x_i + \alpha s_i$.
   (e) Berechne B_{i+1}^{-1} nach Formel (3).
3. Gebe $x_{\text{BFGS}} = x_{N-1}$ zurück.

Abbildung 3: BFGS Algorithmus

BFGS-Algorithmus 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. Der BFGS-Algorithmus, wie wir ihn implementiert haben, funktioniert wie in Abbildung 3 beschrieben. 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.

6 Nicht-lineare Optimierung mit numerischen Verfahren des Packproblems

Das Packproblem benötigt im Gegensatz zu dem Stapelproblem noch Nebenbedingungen. 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 unter Nebenbedingung gibt es verschiedene Ansätze.

Im Rahmen des Softwareprojekts kamen Penalty-, Barriere-, and 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 der Zielfunktion hinzufügen und dann eine Folge von nicht-linearen Optimierungsproblemen ohne Nebenbedingunen auf dieser Zielfunktion lösen. Der Vorteil dabei ist, dass die Algorithmen aus Abschnitt 5.2 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 unendlich 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 jedoch erst ansteigen, wenn die Nebenbedingung auch tatsächlich verletzt ist. Bei beiden Verfahren gibt es jedoch numerische Schwierigkeiten [2]. Das augmentierte Lagrange-Verfahren basiert auf dem Penalty-Verfahren erweitert die Zielfunktion jedoch noch um zusätzliche Terme, die die numerische Stabilität des Verfahrens verbessern. Vor diesem Hintergrund haben wir uns schließlich für die Implementierung des augmentierten Lagrange-Verfahrens entschieden.

6.1 Augmentiertes Lagrange-Verfahren

TODO

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.