Ameisenalgorithmus zur Lösung komplexer Optimierungsprobleme

Paul Theodor Pyl
Konrad Ludwig Moritz Rudolph

Optimierungsprobleme

Optimierungsprobleme sind klassische Probleme in der Informatik, obwohl man jahrelang als Programmierer arbeiten kann, ohne jemals von ihnen gehört zu haben. Sobald man sie aber einmal studiert hat, begegnen sie einem plötzlich überall. Hinter vielen Alltagsproblemen verbirgt sich eine Optimierung. Tatsächlich zählt die Optimierung neben der Statistik zu den praxisnächsten Problemen der „theoretischen" Mathematik. Das äußert sich darin, dass man als Mathematiker mit Optimierung eine einträgliche Verdienstmöglichkeit hat. Entsprechende Projekte führen gerade in letzter Zeit oft zu Firmengründungen und großen staatlichen Förderungen. Es gibt also Grund genug, um sich mit der Materie auseinanderzusetzen.

Ein Beispiel aus dem Alltag ist das öffentliche Verkehrsnetz. Niemand wartet beim Umsteigen gerne auf die nächste Bahn oder den nächsten Bus. Es liegt also im Interesse der Unternehmen, die Umsteigezeiten ihrer Kunden so kurz wie möglich zu halten. Dieser Sachverhalt lässt sich elegant als Optimierungsproblem ausdrücken und mit mathematischen Methoden lösen. Ganz einfach ist das allerdings nicht. Die öffentlichen Verkehrsbetriebe haben vor kurzem große Summen investiert, um ihr Netz vom DFG-Forschungszentrum Matheon (auch bekannt für seinen mathematischen Adventskalender) optimieren zu lassen. Dabei mussten zigtausende verschiedener Parameter unter einen Hut gebracht werden. Und genau das zeichnet Optimierungsprobleme im Alltag aus: eine sehr große Anzahl von Parametern und Beschränkungen, die es einzuhalten gilt. Am Beispiel der Verkehrsbetriebe müssen für jedes Verkehrsmittel und jede Umsteigemöglichkeit Annahmen formuliert werden, und zwar mittels mathematischer Formeln, denn der Optimierungsalgorithmus weiß nicht, was eine U-Bahn ist, und kennt keine Wartezeit. Was für den Algorithmus benötigt wird, ist eine Aussage der Art

$$\min \sum_{i \in V} \sum_{j \in H} x_{ij} - \sum_{k \in H} x_{ik}$$

wobei H z. B. für die Haltestellen stehen könnte und V für die möglichen Verkehrsmittel. In Wahrheit sind die Formulierungen allerdings meist komplexer.

Wie dem auch sei: All dies ist nur eine Art, Probleme zu formulieren. Man minimiert oder maximiert einen mathematischen Ausdruck unter Beachtung bestimmter Bedingungen. Zur Lösung der Aufgabenstellung braucht man komplexe Verfahren. Das Problem mit diesen Verfahren ist ihre exponentiell ansteigende Rechenzeit. Das heißt, dass das Programm z. B. doppelt solange braucht, sobald man auch nur eine einzige Variable hinzufügt. Wenn man also mit 50 Variablen arbeitet, dann benötigt das Programm 2^{50}-mal solange wie es für eine Variable brauchen würde. Bräuchte man für eine Variable eine Millisekunde, dann wären das für 50 Variablen über 36.600 Jahre!

Durch kluge Optimierungen lässt sich diese Rechnung beschleunigen, allerdings nur um konstante Faktoren. Und selbst wenn man den benutzten Algorithmus um das Hundertfache beschleunigte, rechnete man immer noch 360 Jahre an dem Problem. Ein ganz anderer Ansatz zur Optimierung besteht darin, bei der Berechnung der Lösung einige der vorher formulierten Beschränkungen aufzuheben. Dadurch wird die Lösung des Problems sehr einfach – mit einem Nachteil: sie ist falsch, da sie nicht alle Einschränkungen beachtet. Man kann diese Lösung jedoch als Ansatz verwenden, um eine weitere Lösung zu errechnen, welche die eben noch

gelockerten Beschränkungen einhält. Dieser Ansatz nennt sich *Schnittebenenverfahren* und ist für einige Anwendungen sehr effizient.

Ein weiterer Ansatz namens *Lagrange-Multiplikatorenregel* (Englisch: „Lagrangian relaxation“) arbeitet ebenfalls damit, „schwere“ Beschränkungen zu lockern. In diesem Fall führt man weitere Faktoren in seine Optimierung ein, welche dazu dienen, eine Verletzung der gelockerten Beschränkungen zu „bestrafen“: Eine Lösung, die diese Beschränkungen verletzt, wird zwar akzeptiert, jedoch heruntergestuft. Auf diese Weise erhält man zwar keine gültige Lösung aber eine untere Schranke für sein Minimierungsproblem (d. h. man weiß, wie schlecht die Lösung im ungünstigsten Fall aussieht, und geht davon aus, dass dieser Wert bereits nahe bei der echten Lösung liegt).

Heuristiken und Metaheuristiken

Zurück zu den Optimierungsproblemen. Statt eine exakte Lösung zu suchen, verwendet man wegen der schlechten Laufzeit häufig Heuristiken. Heuristiken finden keine optimalen Lösungen, aber man hofft, in vertretbarer Zeit eine Lösung zu finden, die „gut genug“ ist (d. h. möglichst nahe an die optimale Lösung herankommt). Leider weiß man bei der Verwendung von Heuristiken im allgemeinen nicht, wie gut die Lösung denn nun ist und wieviel besser es theoretisch ginge. Doch immerhin haben sie einen Vorteil: Sie sind viel, viel schneller als ein exaktes Verfahren.

Heuristiken verfolgen dabei sehr verschiedene Ansätze. Aber letztendlich läuft es immer auf dasselbe heraus: Wir fangen mit einer – mehr oder weniger erratenen – Lösung an und versuchen, diese zu verbessern. Die anfängliche Lösung ist meistens miserabel (oft bietet es sich an, mit der schlechtesten möglichen Lösung anzufangen, weil diese einfach zu finden ist). Dann versucht man, diese Lösung mit „intelligentem Raten“ zu verbessern – das heißt, man benutzt Informationen, die man bisher gesammelt hat, geht dabei allerdings nicht deterministisch vor sondern zufällig. Auf diese Weise hofft man, sich der Lösung anzunähern. Das ganze ist durchaus vergleichbar mit der menschlichen Entscheidungsfindung: auch wir rechnen schließlich nicht alle möglichen Wege durch, wenn wir die kürzeste Strecke von Kiel nach München suchen, sondern wir fahren mit dem Finger über die Landkarte und denken uns „wird schon passen“.

Heuristiken haben allerdings eine sehr große Schwäche: Da sie, wie bereits erwähnt, nicht beurteilen können, wie gut die optimale Lösung ist, haben sie auch keine Ahnung, wie gut ihr momentaner Rateversuch ist. Es könnte sein, dass der aktuelle Rateversuch bereits sehr nahe an der optimalen Lösung dran ist (oder sogar bereits optimal ist). Es könnte aber auch sein, dass die aktuelle Lösung inakzeptabel schlecht ist. Die Beurteilung der Güte einer Lösung ist allerdings wichtig, denn sonst weiß der Algorithmus nicht, wann er anhalten soll.

Um dieses Problem zu umgehen, besitzen alle Heuristiken einen Mechanismus, der sich die nahe Umgebung der aktuellen Lösung anschaut. Die „Umgebung“ einer Lösung ist eine Menge von ähnlichen Lösungen. Was „ähnlich“ in diesem Fall bedeutet, ist von Aufgabe zu Aufgabe verschieden. Wenn man eine kürzeste Autostrecke sucht, könnte Ähnlichkeit z. B. bedeuten, dass man eine einzige Autobahn auf dem Weg durch eine andere austauscht. Doch auch dieses Verfahren hat eine offensichtliche Beschränkung: Nur, weil es in der aktuellen Umgebung keine bessere Lösung gibt, heißt dies ja nicht, dass es anderswo nicht bessere gibt: Vielleicht ist die aktuelle Route zwar besser als alle ähnlichen Routen, aber man hätte vielleicht nicht gleich zu Beginn über Riga fahren sollen sondern doch lieber in Richtung Hamburg.

Zuerst die schlechte Nachricht: Es gibt keine sichere Möglichkeit, solche Fehler zu verhindern. Vielleicht ist das auch der Grund dafür, dass Microsofts Routenplaner uns so viele Lacher beschert hat. Zwar ist für den menschlichen Betrachter offensichtlich, dass der Umweg über

Riga eine schlechte Idee war. Doch andere Fehler lassen sich nicht so leicht finden, auch nicht für einen Menschen.

Aus diesem Grund geht man noch einen Schritt weiter und erweitert die verwendete Heuristik um eine *Metaheuristik*. Metaheuristiken enthalten Strategien, die verhindern sollen, dass sich der Algorithmus verfrüht mit einer schlechten Lösung zufriedengibt. An diesem Punkt bedarf es einiger mathematischer Grundlagen.

Wie bereits oben gesehen, optimieren wir stets eine mathematische Funktion. Leider hat diese Funktion extrem viele Parameter und befindet sich dementsprechend in einem hochdimensionalen Raum. Für die Veranschaulichung reicht allerdings eine zweidimensionale Funktion. Die Menge der möglichen Lösungen (durch die Funktionsgerade dargestellt) wird hierbei als *Lösungsraum* bezeichnet und als Kurve mit Tälern und Hügeln dargestellt. Die Lösung zu finden, bedeutet ein globales Minimum (bzw. Maximum, wenn wir maximieren) der Funktion zu finden.

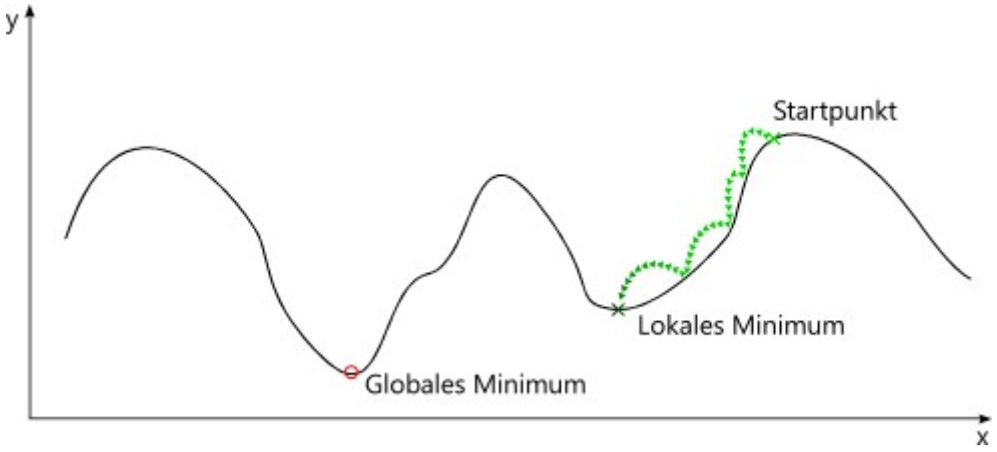


Abbildung 1: Optimierungsgradient in einem eindimensionalen Lösungsraum

Wenn die Funktion an allen Punkten differenzierbar wäre (d. h. man die Ableitung bilden kann), dann wäre das auch oft trivial. Wir könnten die Ableitung bilden und mit null gleichsetzen. Dann bekämen wir als Lösung direkt die Menge der Minima und Maxima und müssten nur noch diese (hoffentlich recht kleine) Menge durchsuchen. Leider ist es aber nicht so einfach, denn in der Regel ist die Optimierungsfunktion *nicht* differenzierbar, und zudem kann die Menge der lokalen Minima und Maxima unendlich groß sein. Im obigen Beispiel startet der Algorithmus oben rechts und arbeitet sich nach unten vor. Sobald er hingegen das lokale Minimum erreicht hat, findet er keine Lösung in der Umgebung, die besser als die aktuelle ist. Für den menschlichen Betrachter (der „von der Seite“ schaut) ist offensichtlich, dass der Algorithmus nur über den nächsten Hügel gehen müsste, um eine bessere Lösung zu finden. Aber der Algorithmus weiß nichts davon – und selbst wenn, es könnte ja noch ganz woanders (50.000 Einheiten nach rechts zum Beispiel) eine andere Lösung geben. Nur: wie weit sucht man? Wann hört man auf? Immerhin geht die Funktion unendlich lang in alle Richtungen weiter.

Metaheuristiken können dieses Problem auch nicht lösen. Aber Metaheuristiken verhindern recht effektiv, dass man zu früh aufhört zu suchen. Wie sie dies tun, ist je nach Art der Metaheuristik sehr unterschiedlich. Gemeinsam haben sie nur, dass es sich stets um eine sehr abstrakte algorithmische Beschreibung handelt, die auf beliebige Probleme angewendet werden kann. In dieser Hinsicht ähneln sie den Entwurfsmustern aus der Objektorientierung: Auch hier hat man lediglich einen Bauplan vorliegen, den man jedes Mal in die Tat umsetzen muss.

Ameisenalgorithmus

Metaheuristiken lassen sich in verschiedene Klassen unterteilen. Beispielsweise gibt es die Klasse der Algorithmen, die mit *Populationen* arbeiten. Auf die obige Grafik bezogen heißt das, dass man nicht nur an *einem* Ort anfängt, und dann auf *einem* Pfad das Gelände erkundet, sondern man sendet verschiedene Akteure aus, die dann größtenteils unabhängig voneinander nach Lösungen suchen. Untereinander kommunizieren diese Akteure nur minimal, um sich gegenseitig über Fortschritte zu informieren. Im Fall des *Ameisenalgorithmus* (auf Englisch „ant colony optimization", kurz „ACO") sind diese Akteure Ameisen. Der Algorithmus ist bewusst der Natur nachempfunden, da man hier eine interessante Entdeckung gemacht hat.

Ameisen schaffen es, ein ihnen unbekanntes Territorium sehr effizient für die Futtersuche zu erkunden, und das, obwohl Ameisen so gut wie blind sind und scheinbar ziellos durch die Gegend irren. Der Schlüssel zum Erfolg liegt in der Kommunikation mittels hormoneller Lockstoffe, Pheromone, die jede Ameise kontinuierlich aussendet. Eine Ameise, die auf ihrer Suche zufällig auf eine Futterquelle stößt, wird den gekommenen Weg zum Bau zurückgehen, um Nahrung zurückzubringen. Dabei orientiert sie sich an ihrer eigenen auf dem Hinweg gelegten Pheromonspur. Je näher die Ameise dem Bau kommt, desto stärker wird die Ansammlung von Pheromon, die Ameise findet also leicht zurück. Besser noch: Andere Ameisen, die das Gelände ebenfalls zufällig abgrasen, werden mit einiger Wahrscheinlichkeit auf die frisch gelegte Pheromonspur der Ameise eins stoßen und ihr folgen (nicht umsonst spricht man von „Lockstoff"). Dadurch finden auch andere Ameisen die Futterquelle und tragen ihren Teil der Beute zum Ameisenhügel zurück. Je mehr Ameisen dies tun, desto stärker werden die Routen zur Nahrung ausgeprägt, bis schließlich immer mehr Ameisen der Route folgen – und zwar nicht irgendeiner Route, sondern der kürzesten, weil der hinterlassene Lockstoff nämlich nach einiger Zeit verfliegt und daher nur auf den vielbenutzten Strecken zurückbleibt.

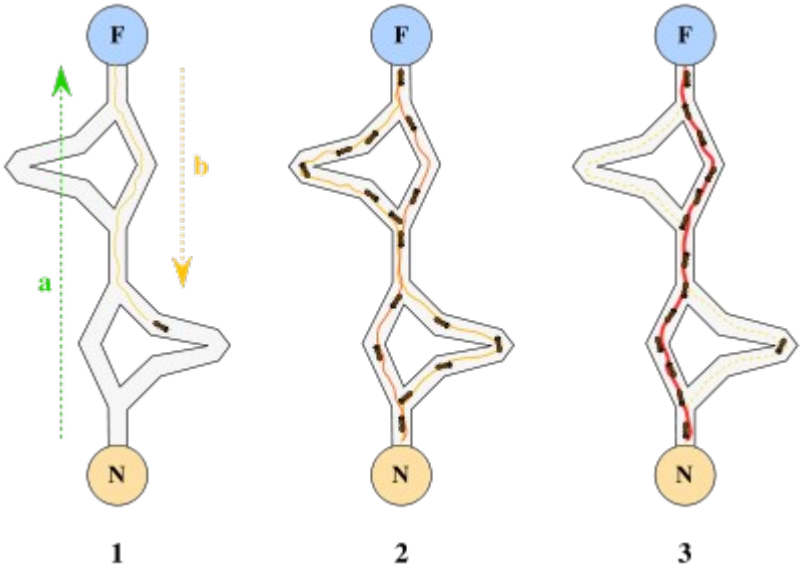


Abbildung 2: Wegsuche der Ameisen. *1:* Die erste Ameise findet einen Weg vom Nest („N") zum Futter („F") und hinterlässt eine Pheromonspur. *2:* Andere Ameisen folgen der Spur und prägen mehrere Wege aus. *3:* Durch Verblassen ungünstiger Pheromonspuren nehmen nahezu alle Ameisen den kürzesten Weg.

Das Ganze lässt sich mathematisch modellieren, indem man mehrere Akteure (Ameisen) instanziert und sie das Problem lösen lässt. Dabei lösen sie Teilprobleme und bauen daraus eine komplette Lösung zusammen. Die jeweiligen Teillösungen werden aus einer Menge möglicher Teillösungen gezogen. Dabei wird zufällig vorgegangen, allerdings ist jede Teillösung mit einem Pheromonniveau markiert. Je höher das jeweilige Niveau, desto höher die Wahrscheinlichkeit, dass sich eine Ameise für eine bestimmte Teillösung entscheidet:

$$P(c_r \mid s_a[c_l]) = \begin{cases} \dfrac{\eta_r^{\alpha} \cdot \tau_r^{\beta}}{\sum_{c_u \in J(s_a[c_l])} \eta_u^{\alpha} \cdot \tau_u^{\beta}} & \text{wenn } c_r \in J(s_a[c_l]), \\ 0 & \text{sonst.} \end{cases}$$

Die Formel sieht auf den ersten Blick sehr kompliziert aus, das ist sie aber eigentlich gar nicht. *P* bezeichnet eine Wahrscheinlichkeit. Der Term in Klammern dahinter bedeutet lediglich die Wahrscheinlichkeit, dass sich eine Ameise (s_a), die im Moment die Teillösung c_l hat, sich im nächsten Schritt für die Teillösung c_r entscheidet. Mit anderen Worten, wir berechnen die (bedingte) Wahrscheinlichkeit dafür, dass sich die Ameise für eine gewisse Teillösung entscheidet, ausgehend von der aktuellen Teillösung. Am Beispiel der Futtersuche könnte eine Teillösung einen Pfad-Abschnitt darstellen. *J* ist dabei die Menge der wählbaren Teillösungen. Wenn sich eine Ameise also an einer Kreuzungsstelle befände, dann enthielte diese Menge den Weg nach vorne, nach links und nach rechts, nicht aber nach hinten, denn da kommt sie ja gerade her. Wir berechnen jetzt für jeden der möglichen Wege die Wahrscheinlichkeit, als nächstes gewählt zu werden.

Der Term dahinter besagt: für jede Teillösung außerhalb der Menge der möglichen Teillösungen ist die Wahrscheinlichkeit, ausgewählt zu werden, gleich 0. Dadurch sind unmögliche Lösungen nicht wählbar. Ansonsten entspricht die Wahrscheinlichkeit dem Bruch. Über dem Bruchstrich werden zwei Faktoren miteinander multipliziert: Eta (η) und Tau (τ). Der Index *r* deutet an, dass sich die beiden Werte auf die potentielle Teillösung beziehen. Bei τ handelt es sich um den Pheromonwert der jeweiligen Teillösung. η ist ein Parameter, der je nach Problemstellung variiert, denn Metaheuristiken beschreiben, wie erwähnt, ein *allgemeines* Verfahren, welches problemspezifisch implementiert werden muss, und das wird unter anderem durch diesen Parameter erreicht. Ähnliches gilt für die Parameter α und β, mit denen η und τ potenziert werden. Im Nahrungssuche-Beispiel könnte η beispielsweise für die Länge eines Teilpfads stehen. Da wir lange Pfade natürlich vermeiden wollen, müsste α in diesem Fall negativ sein; dadurch würde die Wahrscheinlichkeit für einen Teilpfad kleiner werden, je länger dieser Teilpfad ist.

Kommen wir nun zum Teil unter dem Bruchstrich. Hier steht fast dasselbe wie oben, nur dass wir diesmal nicht eine einzige Teillösung herausgreifen sondern die Summe über alle Teillösungs-Faktoren errechnen. Dadurch erhalten wir mit dem Bruch für jede Teillösung die *relative* Wahrscheinlichkeit, als nächstes gewählt zu werden.

Damit hängt die Wahrscheinlichkeit, dass eine Teillösung ausgewählt wird, also von einem speziellen Gütekriterium und ihrem Pheromonniveau ab. Am Ende eines Durchgangs des Algorithmus werden diese Niveaus nun aktualisiert: Zuerst einmal findet die Verdunstung statt: jedes Pheromonniveau verringert sich um einen festen prozentualen Anteil. Dann bekommen alle Ameisen, die eine gültige Lösung des Problems gefunden haben, die Möglichkeit, Pheromon auf ihren Teillösungen zu hinterlassen. Der neue Pheromonwert für jede Komponente *j* berechnet sich also wie folgt:

$$\tau_j = (1-\rho)\tau_j + \sum_{a \in A} \Delta\tau_j^{s_a}$$

Hierbei ist ρ der konstante Verdunstungsfaktor zwischen 0 und 1. *A* ist die Menge aller Ameisen, die erfolgreich eine Lösung erarbeitet haben, und das Delta-Tau (= die Pheromon-„Belohnung“) berechnet sich wie folgt:

$$\Delta\tau_j^{s_a} = \begin{cases} F(s_a) & \text{wenn } c_j \text{ eine Komponente von } s_a \text{ ist,} \\ 0 & \text{sonst.} \end{cases}$$

Mit anderen Worten: jede Teillösung wird genau dann mit einem Pheromonanstieg honoriert, wenn sie Teil der Lösung für die aktuelle Ameise ist. Oder anders gesagt: nur die von der Ameise besuchten Teillösungen bekommen Pheromon ab. Wieviel das ist, berechnet die Funktion *F*, die wieder abhängig von der Problemstellung ist. In unserem Beispiel wäre es sinnvoll, kurze Strecken zu honorieren, d. h. für Ameisen, die eine kurze Strecke genommen haben, wäre der Wert groß, für andere wäre er klein.

Nun können wir einen Pseudocode für den Ameisenalgorithmus aufschreiben, und er ist denkbar einfach:

1. Versuche, das Problem durch *n* Ameisen lösen zu lassen
2. Führe die Pheromon-Aktualisierung durch
3. Wenn die Terminationsbedingungen noch nicht erfüllt sind, gehe zu Schritt 1

Nur: Was sind die Terminationsbedingungen? Leider erben Metaheuristiken dieses Problem von den Heuristiken: Es gibt keine in Stein gemeißelte Methode, um festzustellen, wie gut unsere Lösung ist. Welchen Vorteil haben Metaheuristiken dann? Nur diesen einen: Heuristiken geben sich mit der ersten besten Lösung zufrieden (lokales Minumum). Metaheuristiken hingegen bewegen sich sehr viel dynamischer durch den Lösungsraum und können dadurch lokale Minima überwinden.

Letztendlich sind Metaheuristiken eine sehr fragile Angelegenheit, und ihre Nützlichkeit hängt stark von den gewählten Parametern und Terminationsbedingungen ab. Zu allem Überfluss scheint es nicht einmal gute Methoden zu geben, um all diese Parameter allgemeingültig festzulegen. Eine oft verwendete Abbruchbedingung ist, zu prüfen, ob und wie sehr sich die beste bisher gefundene Lösung verändert: Wenn z. B. in den letzten 20 Iterationen keine Verbesserung erzielt worden ist, wird abgebrochen. Doch allgemein hilft hier nur viel Herumprobieren, bis ein gut funktionierender Parametersatz gefunden wurde. Dies ist auch der Grund, aus dem viele theoretische Informatiker eine gewisse Abneigung gegen Metaheuristiken hegen: die theoretische Basis ist recht dünn und vieles ist der Willkür überlassen.

Doch in der Praxis funktionieren diese Methoden oft sehr gut und finden auch dort Lösungen, wo der „klassische“ Ansatz versagt.

Problem des Handlungsreisenden

Ein Paradebeispiel dafür ist das *Problem des Handlungsreisenden* (auf Englisch „Travelling Salesperson Problem“, kurz „TSP“). Das Problem fasziniert die Mathematiker seit langem, da es eine sehr einfache Beschreibung besitzt und sich trotzdem hartnäckig allen Lösungsversuchen widersetzt. Das Problem gehört zu den klassischen NP-vollständigen Problemen, seine Laufzeit ist exponentiell.

TSP beschreibt das Problem, eine kürzeste Tour durch verschiedene Städte zu finden: Ein Handlungsreisender möchte sein Produkt in jeder größeren Stadt in Deutschland vorstellen. Dafür möchte er jede Stadt allerdings nur ein einziges Mal besuchen, und am Ende möchte er wieder am Anfang ankommen. Außerdem sucht er natürlich den kürzesten Weg. Das Problem hört sich erst einmal sehr vertraut an: Auch die Ameisen auf der Futterjagd suchen einen kürzesten Weg, und dieses Problem ist (auch mit klassischen Algorithmen) sehr effizient und elegant lösbar, beispielsweise mit dem Algorithmus von Dijkstra. Es wäre daher naheliegend, dass TSP sich genauso einfach lösen lässt, aber überraschenderweise ist dies nicht der Fall. Für kleine Pro-

bleminstanzen (5 Städte, oder ähnliches) ist es noch einfach, eine Lösung zu finden. Doch mit jeder weiteren Stadt, die hinzukommt, verdoppelt sich die Laufzeit des Algorithmus. Bereits ab 20 Städten ist das Problem auch auf modernen Computern unlösbar, da der PC jahrelang rechnen müsste.

Inzwischen hat man es zwar geschafft, mit geballter Rechenpower und ausgeklügelten Verfahren (lineare Programmierung mit Branch-and-Cut, eine Abwandlung des kurz erwähnten Schnittebenenverfahrens) auch große Probleminstanzen (im Moment in den 30.000ern) zu lösen. Doch für den Alltag sind diese Ansätze nur bedingt praktikabel, und man darf nicht vergessen, dass es sich um individuelle Lösungen handelt: Nur, weil ein spezieller TSP mit 33.810 Knoten gelöst wurde, heißt das nicht, dass die Lösung für ein ähnliches Problem mit genauso vielen oder weniger Knoten bekannt ist.

Warum steckt man solch einen enormen Aufwand in ein solches Problem? Nun, die Lösungen sind sehr nützlich. Beispielsweise haben Luftfahrtgesellschaften damit zu kämpfen, die Touren für ihre Flugzeuge zu optimieren, und ein vielleicht noch näherliegendes Beispiel sind Kurierdienste. Die Kosten für große Touren sind hier enorm und Kostenersparnisse durch optimale Touren liegen in astronomischen Höhen. Aber es gibt auch eine ganze Reihe indirekter Nutzungen, die teilweise durch geringe Änderungen der Problembeschreibung (oder einfach nur durch kluges Um-die-Ecke-Denken) resultieren. So hing beispielsweise das humane Genomprojekt (welches im Jahr 2000 die Sequenzierung der menschlichen DNS vollbrachte) stark davon ab, dass man TSP einigermaßen effizient lösen konnte. Für Informatiker wohl noch relevanter ist der Nutzen beim Design integrierter Schaltkreise.

Interessanterweise ist es relativ einfach, fast-ideale Lösungen für TSP zu finden. So gibt es zum Beispiel ein effizientes Verfahren, welches eine Lösung findet, die garantiert schlimmstenfalls doppelt so lang wie die optimale Tour ist (mittels eines minimal aufspannenden Baumes). Eine solche Lösung nennt sich 2-Approximierung. Leider ist eine solche Lösung in der Praxis meist viel zu schlecht.

Um den Kreis jetzt zu schließen

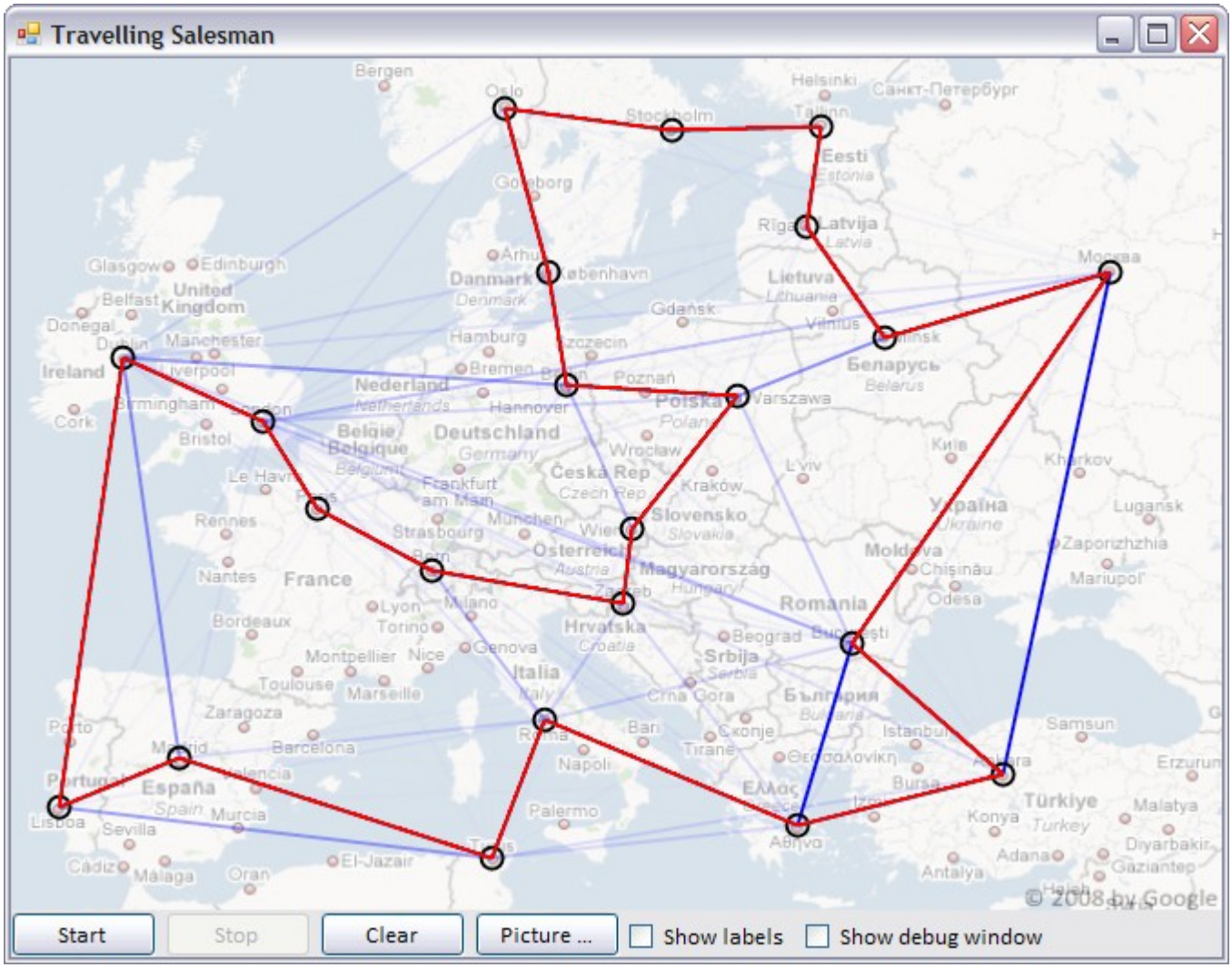
Die obige Problembeschreibung von den Ameisen, die Futter suchen, lässt sich fast 1:1 auf TSP übertragen und recht gut effizient lösen, obwohl dieses Problem doch eigentlich einige relevante Unterschiede aufweist. Es muss lediglich getestet werden, ob die Ameise auch tatsächlich eine Tour ausgeführt hat (d. h. jede Station wurde genau einmal besucht). Wir dürfen in der Euphorie jedoch eine Sache nicht vergessen: Der Ameisenalgorithmus gibt uns zwar oft sehr gute Lösungen, allerdings sind diese für schwere Probleminstanzen selten optimal, und was noch schwerer wiegt: Wir haben keinen Hinweis darauf, *wie* gut die Tour ist. Daher hinkt diese Methode in gewisser Hinsicht sogar der 2-Approximierung hinterher, obwohl die Ergebnisse oft wesentlich besser sind und sich nur minimal von der optimalen Lösung unterscheiden. Aber wir haben keinerlei Garantie, und dies ist in der Praxis oft vollkommen inakzeptabel.

Zur Implementierung

Üblicherweise würde man den Ameisenalgorithmus so implementieren, dass die Pheromonwerte der Teillösungen und die Ameisen in einer Matrix gespeichert werden. Dadurch kann man die Berechnungen sehr effizient implementieren. In unserer Implementierung haben wir uns entschieden, das ganze dadurch anschaulicher zu machen, dass die einzelnen Komponenten durch Objekte repräsentiert werden, auch wenn dadurch Effizienz verschenkt wird.

Die Oberfläche ist recht simpel: Ein Linksklick fügt eine Stadt hinzu, ein Rechtsklick entfernt sie. Hinzugefügte Städte werden automatisch durch alle möglichen Kanten miteinander verbunden. Wenn man die Optimierung startet, dann werden diese Kanten proportional zu ihrem Pheromonwert gefärbt: dunklere Kanten bedeuten höhere Pheromonwerte. Die grüne Tour stellt die derzeit kürzeste Tour dar. Sobald der Algorithmus terminiert hat, wird diese Tour rot dargestellt.

Im Debugfenster werden aktuelle Werte der Berechnung eingeblendet, deren Bedeutung sich dem Quelltext entnehmen lässt. Wenn man an der „rohen“ Geschwindigkeit des Algorithmus interessiert ist, dann kann man die „AddHandler“-Anweisung in der MainForm auskommentieren. Die daraus resultierende Beschleunigung ist nicht zu vernachlässigen.



Quellen

- DFG-Forschungszentrum Matheon: http://www.matheon.de/
- Adventskalender der Matheon: http://www.mathekalender.de/
- Matheon-Projekt „Service Design in Public Transport“: http://www.zib.de/Optimization/Projects/Traffic/Matheon-B15/Matheon-B15long2.en.html

- Marco Dorigo (2007) *Ant Colony Optimization*, Scolarpedia, 2(3):1461: http://www.scholarpedia.org/article/Ant_colony_optimization
- „Homepage“ des TSP: http://www.tsp.gatech.edu/
- Lineare Programmierung: V. Chvátal, *Linear Programming*, Freeman 1983
- Algorithmen: T. H. Cormen *et al.*, *Introduction To Algorithms*, 2nd ed., MIT Press 2001

Abbildungsverzeichnis

- Abbildung 2: http://fr.wikipedia.org/wiki/Image:Aco_branches.svg
- Abbildung 3: http://de.wikipedia.org/w/index.php?title=Bild:TSP_Deutschland_3.PNG&oldid=30740640