Prof. Dr. Timm Grams
Hochschule Fulda
Fachbereich Elektrotechnik
Marquardstraße 35
36039 Fulda

Tel (0661) 78157
(0661) 9640-550
Fax (0661) 9640-559
mailto:Timm.Grams@et.hs-fulda.de
www.hs-fulda.de/~grams

Samstag, 30. Mai 2009

Java-Projekt: Three Colour Maps

Eine beispielhafte Anwendungen von Heuristiken bei der Optimierung

Inhalt

Einleitung

Ziel des Projekts ist weitestgehende Übertragung der KoopeEgo-Strategie (Grams, 2007) auf ein bekanntes Optimierungsproblem. Das Projekt ist eine Vorstudie zum Projekt „Beispielhafte Anwendungen der Evolutionsverfahren auf ökonomisch-technische Optimierungsprobleme“ gedacht.

Die Aufgabenstellung: Das Programm soll zu einer vorgegebenen Landkarte bestimmen, ob diese Karte mit drei (optional vier) Farben koloriert werden kann, so dass je zwei aneinander angrenzende Länder verschiedene Farben haben.

Diese Aufgabenstellung wird als *Satisfiability-Problem* formuliert und in ein Optimierungsproblem transformiert.

Die Optimierung geschieht mit einem *Evolutionsverfahren,* das aus den Rechenvorschriften des KoopEgo-Modells abgeleitet ist. Wie bei KoopEgo gibt es (a) eine Welt mit ortsfesten Individuen als Repräsentanten der Lösungsvorschläge, (b) die elementaren Evolutionsmechanismen Mutation und Selektion, und (c) die Berücksichtigung von Nachbarschaftsbeziehungen.

Ein paar Unterschiede gibt es auch: Zu den Evolutionsmechanismen tritt das Crossing-Over. Das zufällige Entstehen und Sterben, gesteuert über die Geburts- und die Sterbewahrscheinlichkeit, wird ersetzt durch den Mechanismus der *simulierten Abkühlung* (Simulated Annealing): Alle Felder der Welt sind mit Individuen belegt, es gibt also keinen Freiraum mehr für Geburten und es entstehen auch keine Leerfelder durch Tod. Dafür gibt es jetzt die direkte Ersetzung. Die Wahrscheinlichkeit, mit der ein bestehendes Individuum durch ein neues ersetzt wird, hängt von der Güte dieser Individuen und vom Parameter *Temperatur* ab. Je höher die Temperatur ist, desto eher haben auch schwache Individuen eine Überlebenschance. So kommt schöpferisches Chaos in eine erstarrte Welt.

Grundidee: In der schachbrettartigen Welt ist jedes Feld mit einem Individuum besetzt. Jedes dieser Individuen repräsentiert ein Färbungsmuster für die Landkarte. Das Färbungsmuster ist das Gen des Individuums.

Je *Spielzug* wird ein Platz der Welt rein zufällig als Zentrum bestimmt. Aus der Umgebung des Zentrums wiederum werden zwei Individuen gewählt. Das sind die Eltern eines neuen Individuums. Das Färbungsmuster des neuen Individuums ist Ergebnis einer Crossing-Over-Operation, angewandt auf die Muster der Eltern, und einer nachgeschalteten Mutationsoperation.

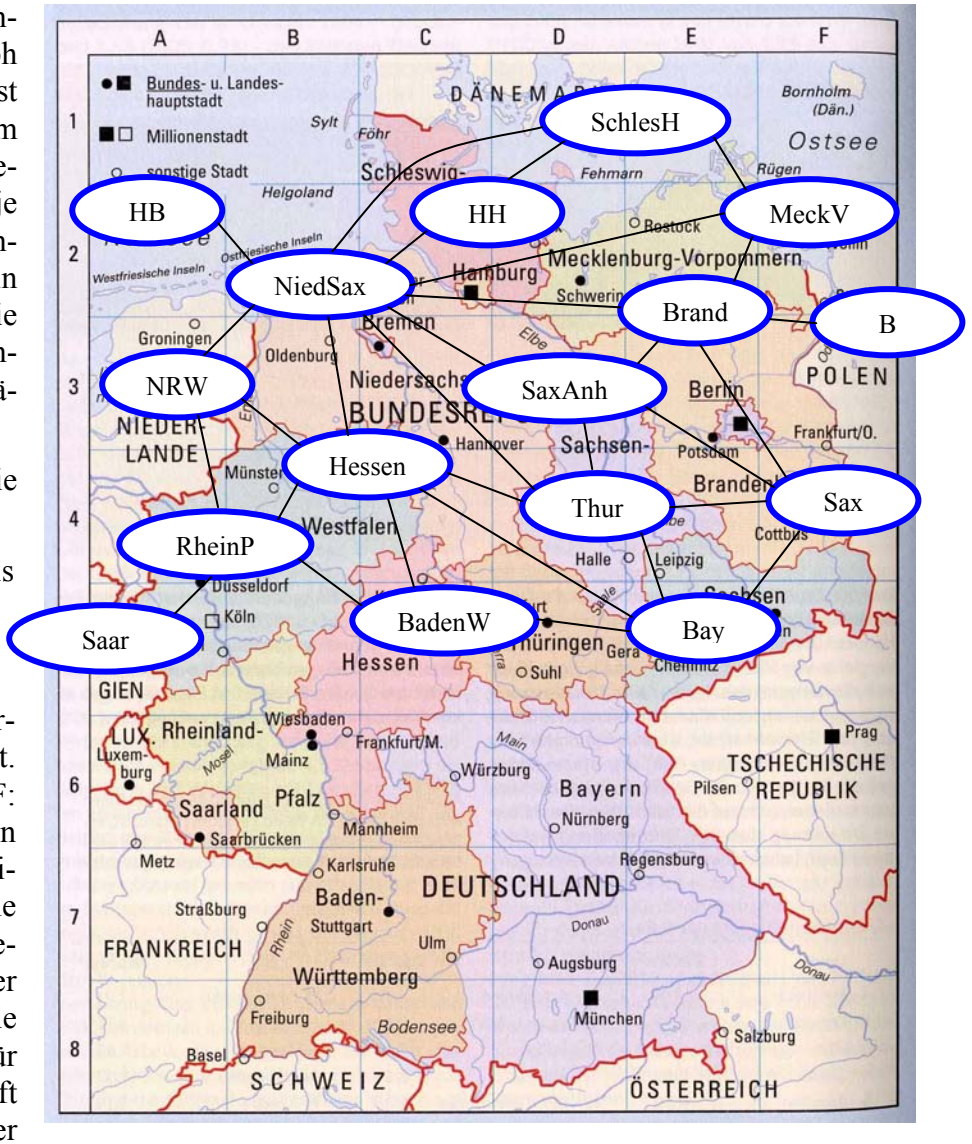
Ergibt sich dabei ein besserer Färbungsvorschlag als derjenige des zentralen Individuums, so wird letzteres durch das neue Individuum ersetzt. Bei der simulierten Abkühlung hängt die Wahrscheinlichkeit für eine Ersetzung vom Grad der Verbesserung ab.

Die hier vorgeschlagene Optimierungsmethode ist grundsätzlich auf alle Probleme anwendbar, bei denen die Anforderungen an eine Lösung in der Sprache der Logik formuliert werden können.

Pflichtenblatt

Eingabe: Datenstruktur für die Landkarte

Die Landkarte wird zunächst als gerichteter Graph aufgefasst: Jedes Land ist ein Knoten und zu jedem Nachbarland führt eine gerichtete Kante. Zwischen je zwei Ländern mit gemeinsamer Grenze gibt es ein Kantenpaar, sodass die Karte auch durch einen ungerichteten Graphen repräsentiert werden kann.



Als Beispiel dient uns die deutsche Länderkarte.

Das Netz wird mittels Textdatei beschrieben. Die Syntax der Eingabesprache wird mittels erweiterter Backus-Naur-Form (EBNF) definiert. Erläuterung der EBNF: Das Gleichheitszeichen wird gelesen als "ist definiert als", nichtterminale Symbole sind kursiv gesetzt. Die eckige Klammer steht für optionale Textteile und die geschweifte für optionale und beliebig oft wiederholbare Teile. Der senkrechte Strich trennt Alternativen. Terminale Symbole sind fett gedruckt.

Map = { *Node* }
Node = *Name* [*GraphicsInfo*] { *Edge* } **;**
GraphicsInfo = *Color xPos yPos*
Edge = *Name*
Color = **0** | **1** | **2** | **3**
xPos = *Integer*
yPos = *Integer*
Name = *Identifier*

Startsymbol ist das Symbol *Map*. Für die Symbole *Identifier* und *Integer* werden die Definitionen der Programmiersprache Java übernommen.

Die Eingabe des Netzes der Länderkarte ohne Grafikinformation sieht damit so aus:

```
B Brand;
BadenW RheinP Hessen Bayern;
Bayern BadenW Hessen Thur Sax;
Brand MeckV B Sax SaxAnh NiedSax;
```

```
HB NiedSax;
Hessen NiedSax Thur Bayern BadenW RheinP NRW;
HH NiedSax SchlesH;
MeckV Brand NiedSax SchlesH;
NiedSax HB SchlesH HH MeckV Brand SaxAnh Thur Hessen NRW;
NRW NiedSax Hessen RheinP;
RheinP NRW Hessen BadenW Saar;
Saar RheinP;
Sax Bayern Thur SaxAnh Brand;
SaxAnh Brand Sax Thur NiedSax;
SchlesH MeckV HH NiedSax;
Thur SaxAnh Sax Bayern Hessen NiedSax;
```

Bei fehlender Grafikinformation werden die Farben sämtlicher Knoten auf weiß gesetzt und die Positionen werden in einem Gitternetz der Reihe nach vergeben.

Eingabe: Steuerungsparameter

Für jeden Parameter ist ein Default-Wert voreingestellt. Durch die Parametereingabe kann diese Standardeinstellung verändert werden. Nach Drücken des Eingabe-Buttons werden die Verfahrensparameter und die Beschreibung der Landkarte von einer Hintergrunddatei geladen. Die Liste der Verfahrensparameter:

```
n  Größe der Welt (integer): ein Spielfeld aus n-mal-n Feldern.
k  Umgebungsparameter. Legt die k-Umgebung fest
h  Anzahl der Spielzüge bis zum Halt
o  Auswahl des Crossing-Over-Operators. Bedeutung der Werte
   0: Zufällige Mischung der Bits beider Eltern
   1: Kopie zusammenhängender Abschnitte. Eine zufällige Bruchstelle
p  Mutationswahrscheinlichkeit
q  Wahrscheinlichkeit für Crossing-Over
T  Temperatur (Simulated Annealing)
```

Die Grammatik für die Eingabe der Verfahrensparameter:

Parameter = { *ParameterNumber* }
ParameterNumber = *IntParam Integer* | *RealParam Float*
IntParam = **n** | **k** | **h** | **o**
RealParam = **p** | **q** | **T**

Das Format der gesamten Eingabedatei (Steuerungsparameter plus Landkarte) ist definiert durch:

InputText = *Parameter* **Map** *Map*

Anmerkung zur Schreibweise: Programmvariablen werden nicht kursiv geschrieben. Kommen diese Variablen in mathematischen Ausdrücken vor, dann wird zur Kursivschreibweise übergegangen.

Kommentare können am Anfang des Dokuments und hinter jedem Semikolon stehen. Kommentare beginnen mit der Zeichenfolge /* und enden mit der Zeichenfolge */. Wo ein Kommentar stehen kann, dürfen auch mehrere stehen – allerdings immer schön nacheinander, Schachtelung ist nicht erlaubt.

Verarbeitung

Das Färbungsproblem wird in ein Satisfiability-Problem überführt. Dazu werden für jedes Land (für jeden Knoten also) boolesche Variablen eingeführt, die die jeweilige Farbe bestimmen. Die Bedingungen für die Färbung werden als boolesche Ausdrücke in konjunktiver Normalform geschrieben (Dewdney, 1993, S. 230-236).

Satisfiability: Es wird eine Wertebelegung der Variablen gesucht derart, dass die booleschen Ausdrücke erfüllt sind. Zur Lösung des Satisfiability-Problems wird es zunächst in ein Optimierungsproblem übergeführt. Dazu muss eine Bewertungsfunktion eingeführt werden, die ihr Minimum genau dann annimmt, wenn das Satisfiability-Problem gelöst ist. Die Minimierung geschieht mit dem Verfahren des Simulated Annealing (Michalewicz, Fogel, 2000, S. 116 ff.) und – alternativ dazu – mit Evolutionsverfahren in Anlehnung an das Projekt KoopEgo.

Ausgabe

Die Färbung der Länder wird grafisch dargestellt. Bei Bedarf kann das Netz samt Grafikinformationen (also einschließlich der Farbinformation) in einer Textdatei gespeichert werden.

Entwurf

Modellierung der Individuen des Suchraums

Wir beschreiben die Färbungsmuster für das Netz (Pattern) mit Hilfe boolescher Variablen. Diese booleschen Variablen sind die Bits einer Integer-Variablen *v*, die wir uns hier als unbeschränkt lang vorstellen. Im Programm wird zu diesem Zweck ein Vektor von ausreichend vielen Byte-Variablen angelegt. $v[i]$ ist dann das Bit der Wertigkeit $2^{i \bmod 8}$ in der Byte-Variablen mit dem Index i div 8. (In Java werden mod-Operator und div-Operator geschrieben als „%“ bzw. „/“).

Für die Zuordnung der Variablen zu den Bits der Variablen *v* werden die alphabetisch sortierten Bundesländer durchnummeriert. Diese Nummerierung wird in unserem Beispiel durch das Array

```
String[] country = { "B", "BadenW", "Bayern", "Brand", "HB", "Hessen",
                     "HH", "MeckV", "NiedSax", "NRW", "RheinP", "Saar",
                     "Sax", "SaxAnh", "SchlesH", "Thur" };
```

definiert. Für die Farben machen wir die folgende Zuordnung:

```
String[] Color = { "red", "yellow", "blue", "green" };
```

Die Farbe Grün ist bei Dreifarbenkarten verboten. Aber wir nehmen sie hier aus zwei Gründen mit auf:

1. So lassen sich Vierfarbenkarten gleich mit abhandeln.
2. Die Zahl der Farben ist eine Zweierpotenz und die zusätzliche Farbe erforderte bei der Codierung keinen zusätzlichen Aufwand.

Die Indizierung des color-Arrays legt gleichzeitig die Codierung fest: Color[*i*] wird durch die letzten beiden Bits der Zahl *i* repräsentiert:

`"red"` ↔ 00
`"yellow"` ↔ 01
`"blue"` ↔ 10
`"green"` ↔ 11.

Wie bei Zahlen üblich, werden die Bits nach fallenden Wertigkeiten bzw. fallenden Indizes angeordnet.

Die Codierung von *v* wird folgendermaßen definiert: Seien *i* der Index des Landes und *jk* die beiden Bits seiner Farbe, dann ist $v[2i]=k$ und $v[2i+1]=j$. Wenn Niedersachsen (`NiedSax`) die Farbe Gelb hat, ist $v[16]= 1$ und $v[17]=0$.

Die Anzahl der booleschen Variablen ist in unserem Beispiel gleich 16*2 = 32. Die Anzahl aller möglichen Wertebelegungen ist etwa gleich 4 000 000 000. Wenn die Überprüfung einer

Wertebelegung 1 µs dauert, benötigt das Durchlaufen aller Möglichkeiten etwa eine Stunde. Die Lösungssuche lässt sich durchaus noch mit der Methode der vollständigen Enumeration bewältigen. Nimmt man die Anliegerstaaten Deutschlands noch hinzu, steigt die Anzahl der booleschen Variablen auf 25*2 = 50 und die Anzahl der möglichen Wertebelegungen vergrößert sich um etwa den Faktor 300 000. Wir landen also bei einem Zeitbedarf für die vollständige Enumeration in der Größenordnung von Jahrzehnten. Da nur eine Lösung von vielen möglichen gesucht ist, kann der Zeitaufwand für die Enumeration aber auch geringer ausfallen.

Bewertungsfunktion

Wir schreiben die Bedingung für die Färbung des Netzes in Klauselform. Eine Klausel ist eine disjunktive Verknüpfung von Variablen oder deren Negation, beispielsweise: $v[0] \vee \neg v[4] \vee v[17]$. Eine Bedingung in Klauselform ist eine Menge solcher Klauseln, die alle wahr sein müssen. Die konjunktive Verknüpfung sämtlicher Klauseln ergibt die konjunktive Normalform der Bedingung.

Jede Kante zwischen zwei Knoten liefert vier Klauseln, die ausdrücken, dass die Farben der Knoten verschieden sein müssen. Seien i und j zwei Knoten, die über eine Kante verbunden sind. Die Färbungsbedingung für diese Kante ist gegeben durch $(v[2i] \neq v[2j] \vee v[2i+1] \neq v[2j+1]$. Das entspricht den folgenden Klauseln:

$$v[2i] \vee v[2i+1] \vee v[2j] \vee v[2j+1]$$
$$v[2i] \vee \neg v[2i+1] \vee v[2j] \vee \neg v[2j+1]$$
$$\neg v[2i] \vee v[2i+1] \vee \neg v[2j] \vee v[2j+1]$$
$$\neg v[2i] \vee \neg v[2i+1] \vee \neg v[2j] \vee \neg v[2j+1]$$

Das Deutschlandnetz hat 16 Knoten und 29 Kanten. Für die Färbungsbedingung ergibt das 29*4 = 116 Klauseln. Beim Vierfarbennetz ist das auch alles. Beim Dreifarbennetz kommt je Knoten noch die Bedingung hinzu, dass die Farbe Grün auszuschließen ist. Das liefert beim Knoten mit dem Index i die Klausel $\neg v[2i] \vee \neg v[2i+1]$.

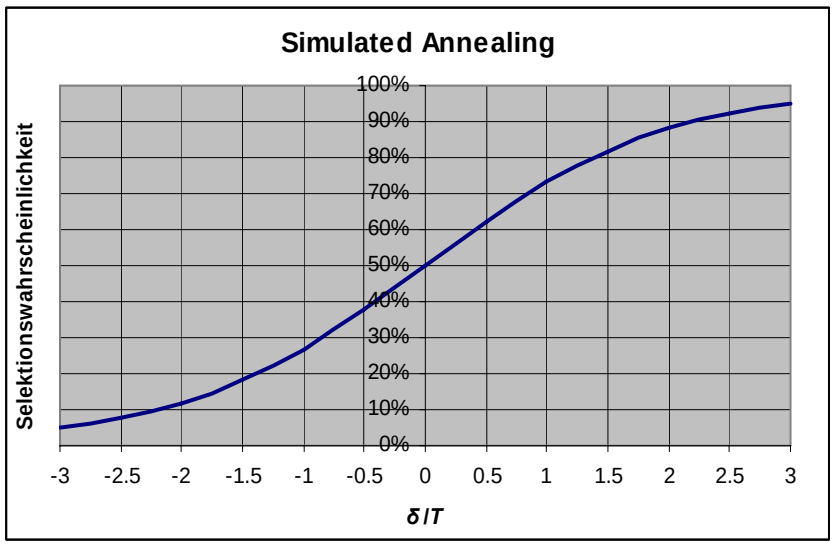
Für das Dreifarbennetz der deutschen Länder wären demnach insgesamt 29*4+16 = 132 Klauseln zu erfüllen.

Für jede Klausel führen wir zwei Bitmuster m (im Programm: mask) und n (im Programm: neg) ein. Die Organisation dieser Bitmuster ist dieselbe wie bei den Färbungsmustern. Mit $m[k]$ bezeichnen wir die Maske der k-ten Klausel. Sie hat genau dort eine 1 stehen, wo die Variablen dieser Klausel stehen. Und das Bitmuster $n[k]$ hat nur dort eine 1 stehen, wo in der Klausel eine negierte Variable steht. Der Java-Ausdruck $v\&m[k]\hat{}n[k]$ ist genau dann ungleich 0, wenn die Klausel erfüllt, also wahr ist. Die Bewertungsfunktion $f(v)$ gibt die Anzahl der *nicht* erfüllten Klauseln zurück.

Auswahloperator

Das Verfahren der simulierten Abkühlung (Simulated Annealing) ist eine Fortentwicklung bzw. Variante des Mutations-Selektionsverfahrens (Rechenberg, 1973).

So funktionieren die Mutations-Selektions-Verfahren: Man beginnt mit einem Startpunkt v_c für die Lösung. Das ist der Aufenthaltspunkt (Current Point), die momentan gültigen Lösung. Durch zufällige Variation wird ein Mutationspunkt v_n (New Point) ermittelt. Mit δ wird die Verbesserung der Zielgröße bezeichnet, die durch den Mutationspunkt erreicht wird: $\delta = f(v_c) - f(v_n)$. Ergibt sich eine Verbesserung der Zielgröße ($0 < \delta$), wird der Mutationspunkt zum neuen Aufenthaltspunkt, ansonsten wird der Mutationspunkt verworfen (Selektion). Nach mehreren vergeblichen Verbesserungsversuchen wird der Prozess abgebrochen.



Bei der simulierten Abkühlung wird die streng deterministische Selektion durch eine stochastische ersetzt. Auf diese Weise haben auch weniger gute Lösungen eine „Überlebenschance“ und es ergibt sich die Möglichkeit, einer Einengung der Suche auf ein schwaches lokales Optimum zu entgehen.

Die Selektionswahrscheinlichkeit für den Mutationspunkt wird auf den Wert $1/(1+e^{-\delta/T})$ gesetzt. Das heißt: Ist die Mutation neutral (δ=0), dann wird eine Ersetzung mit der Wahrscheinlichkeit ½ durchgeführt. Ergibt sich tatsächlich eine Verbesserung (0<δ), ist die Wahrscheinlichkeit für Ersetzung des Aufenthaltspunkts durch den Mutationspunkt größer als ½. Bei einer Verschlechterung (δ<0) wird die Ersetzung mit einer geringeren Wahrscheinlichkeit als ½ durchgeführt.

Das Verfahren wird über die Wahl des Parameters *T* („Temperatur“) gesteuert. Ist die Verbesserung so groß wie diese Temperatur, dann erhöht sich die Selektionswahrscheinlichkeit für den Mutationspunkt auf etwa 73%. Bei einer entsprechenden Verschlechterung sinkt die Selektionswahrscheinlichkeit auf etwa 27%. Genaueres zeigt die folgende Grafik:

Variationsoperatoren

Die *Mutation* des Bitmusters *v* geschieht derart, dass jedes Bit unabhängig von den anderen mit der Wahrscheinlichkeit *p* durch seine Negation ersetzt wird.

Beim *Crossing-Over* kann aus mehreren Verfahren ausgewählt werden. Das Verfahren mit der Kennzahl 0 geht so: Für jede Bitposition wird entschieden, ob das Bit des ersten Elters oder (mit derselben Wahrscheinlichkeit) das des zweiten Elters übernommen wird. Beim Verfahren mit der Kennzahl 1 wird eine Bruchstelle im Bitmuster zufällig gewählt. Bis zur Bruchstelle werden die Bits des ersten Elters und danach die des zweiten Elters übernommen. Dieses Bruchstellen-Crossing-Over ist äußerste zeiteffizient.

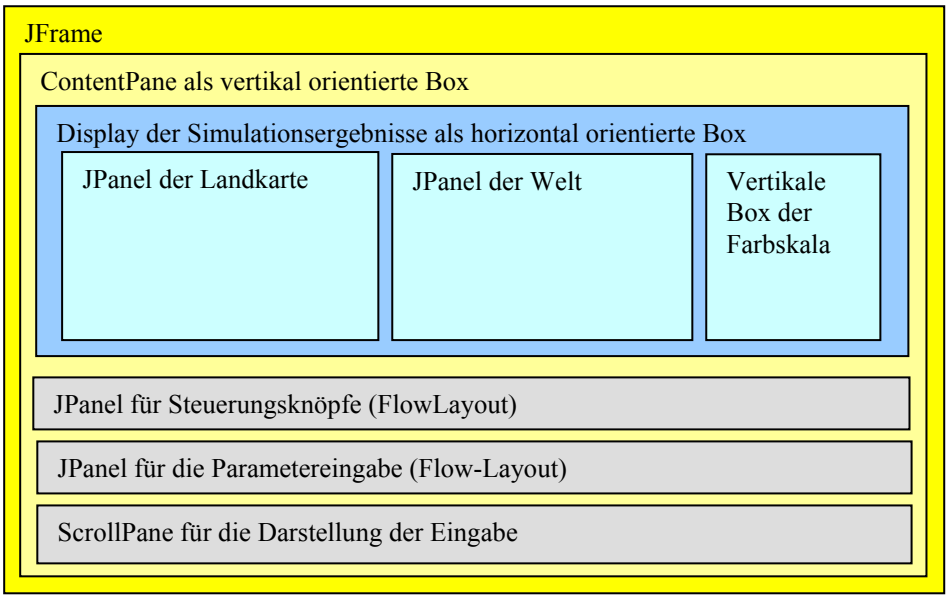
Interaktive Steuerung

Der Anfangswert für *v* ist eine Zufallsbelegung, bei der jedes Bit mit Halbe-halbe-Wahrscheinlichkeit gleich 0 oder 1 ist. Der Optimierungslauf wird interaktiv durch Vorgabe der Parameter *p* (Mutationswahrscheinlichkeit), *q* (Crossing-Over-Wahrscheinlichkeit) und *T* (Temperatur) gesteuert. Nach jedem Lauf aus *h* Spielzügen wird die Simulation automatisch angehalten. Der Wert der Bewertungsfunktion wird ausgegeben. Es besteht die Möglichkeit, die Parameter neu zu setzen und den nächsten Lauf zu starten.

Bedienoberfläche

Anders als bei KoopEgo wird das Steuerpult nicht separat angeordnet. Es gibt für die grafische Darstellung einen Frame und ansonsten nur Dialog-Elemente. Die Strukturierung des Layouts geschieht mittels Box-Klassen und BoxLayout. Boxen erlauben ein gut definiertes Layout und sie haben den Vorteil, dass sie sich bei Änderung der Frame-Dimensionen (z. B. beim Wechsel von der Normal- zur Vollbilddarstellung) automatisch an die eingelagerten Elemente vom Typ JPanel oder Box anpassen.

Bei Änderung der Frame Dimensionen wird eine Anpassung des Maßstabs für die Modelldarstellung (Landkarte und Welt) vorgenommen: Bei Vergrößerung des Frames werden die Quadrate für die Elemente der Welt größer und die Landkarte bekommt mehr Platz.



Zur Verwendung der Farben in den Panels für die Landkarte und die Welt: Im Panel der Landkarte steht die Farbe für die entsprechende Färbung des Landes auf der Landkarte. Im Panel der Welt gibt die Farbe eines Feldes die Güte des zugehörigen Lösungsvorschlags wieder.

Das Panel der Landkarte erhält einen einfachen Grafikeditor. Damit lassen sich die Knoten (Länder) verschieben, löschen und erzeugen. Außerdem können Verbindungen (Grenzen) erzeugt und gelöscht werden. Es empfiehlt sich, eine neue Landkarte nicht in Textform einzugeben, sondern mit dem Grafikeditor. Dazu lässt man einfach den *Map*-Teil der Eingabedatei leer.

Die Farbe eines Feldes der Welt gibt die Qualität des durch dieses Individuum repräsentierten Färbungsmusters wieder: Gute Färbungsmuster, also solche, die nur wenige Klauseln verletzen, werden blau dargestellt und schlechte bekommen eine rote Farbe. Mittlere Qualitäten erhalten passende Zwischenfarben des Regenbogens.

Realisierung

Der Programmtext ist im Java-Archiv ThreeColourMaps.jar enthalten[1]. Hier sollen nur einige der besonders erläuterungsbedürftigen Abschnitte besprochen werden.

Eingabe und Programmsteuerung

Die Eingabeparameter und die Netzdarstellung der Landkarte werden nach Drücken des Eingabe-Buttons aus einer Textdatei in ein Textfenster geladen. Der Text ist editierbar. Nach Drücken des Buttons „Netz bearbeiten“ erscheint in einem Fenster die Netzdarstellung der Landkarte.

Das Netz ist grafisch editierbar: Jeder der dargestellten Knoten kann mit gedrückter Maustaste verschoben werden. Bei Klick auf einen Knoten wird ein modaler Dialog geöffnet und es kann eine der folgenden Aktionen ausgewählt werden: Knoten löschen, Verbindung herstellen

[1] Es ist eines der Programme zum Thema Problemlösen und im Internet frei erhältlich.

und Verbindung löschen. Die letzten beiden Aktionen erfordern einen weiteren Klick auf den zweiten Knoten der Verbindung.

Mit dem Start-Button wird die Welt mit Individuen vorbelegt. Jedes dieser Individuen repräsentiert ein Färbungsmuster.

Drücken des Stop-and-Go-Buttons startet die Simulation oder setzt sie fort. Je Spielzug wird ein neuer Bewohner (Individuum, Arbeitsplan) der Welt erzeugt. Nach einem Lauf von *h* Spielzügen wird die Simulation angehalten und die grafische Darstellung des Zustands der Welt aktualisiert.

Der Weiter-Button startet eine Folge von Läufen, die sich mit dem Halt-Button unterbrechen lässt. Über den Parameter *h* lässt sich die Geschwindigkeit der Veränderungen der Welt steuern: Je kleiner *h,* desto langsamer geht es.

Der Ende-Button beendet die Simulation und schließt den Programmlauf.

Immer wenn die Simulation steht, können gewisse Parameter in den dafür vorgesehenen Textfenstern geändert werden. Eine Änderung wird nur wirksam, wenn sie mit dem Return-Knopf abgesendet wird.

Jeder Farbpunkt repräsentiert einen Bewohner der Welt (Färbungsmuster). Durch Mausklick auf einen solchen Farbpunkt wird ein Fenster mit allgemeinen Informationen über das Muster geöffnet. Gleichzeitig wird das Netz der Landkarte im ausgewählten Muster eingefärbt.

Sämtliche Informationen des dargestellten (aktuellen) Färbungsmusters werden auf Knopfdruck (Speichern-Button) in eine frei wählbare Datei gespeichert.

Mutationsoperator

Der Mutationsoperator wirkt auf das Bitmuster *v* der Färbung. Jedes Bit der Variablen *v* wird mit der Wahrscheinlichkeit *p* invertiert. Man könnte zu diesem Zweck mit der Integervariablen i die Bits der Reihe nach durchgehen und für jedes Bit entscheiden, ob es zu invertieren ist oder nicht. Etwa so:

```
for int i=0; i<v.length; i++) if(Math.random()<p) v[i]=1-v[i];
```

Bei 16 Ländern sind das 32 Aufrufe des Zufallszahlengenerators. Die Zahl der Aufrufe lässt sich deutlich vermindern, indem man nicht die Wahrscheinlichkeit der Invertierung direkt ermittelt, sondern stattdessen den Abstand der zu invertierenden Werte als Zufallsvariable nimmt. Die diskrete Verteilung dieser Abstände D ist gegeben durch $p_k = P(D=k) = (1-p)\cdot p^k$. Seien s_k die Partialsummen dieser Zahlen (Verteilungsfunktion von D): $s_0 = 0$, $s_1 = s_0 + p_1$, $s_2 = s_1 + p_2$, … Die Intervalle $[s_{k-1}, s_k)$ haben die Länge p_k und sie bilden eine Zerlegung des Intervalls [0, 1). Die Realisierung einer gleichverteilten Zufallsvariable (wie sie beispielsweise von Math.random() erzeugt wird) fällt mit der Wahrscheinlichkeit p_k in das Intervall $[s_{k-1}, s_k)$. Es ist also nur nötig, eine solche Realisierung r zu erzeugen und dann zu sehen, in welches der Intervalle die Zahl fällt. Das entsprechende k ist dann der gesuchte Abstand zum nächsten zu invertierenden Bit.

```
for (int i=0;;) {
for (double r= Math.random(), x=p, y=x; y<=r&&i<nPattern; x*=1-p,  y+=x, i++);
  if (i<nPattern) invertBit(i); else break;
  i++;
}
```

Crossing-Over-Operator

Für das Crossing-Over mit der Kennzahl 0 wird eine programmtechnisch einfache Lösung gewählt: Bit für Bit wird in einer Fifty-fifty-Entscheidung entweder das Bit vom ersten oder vom zweiten Elter übernommen. Bei der Kennzahl 1 wird eine Bruchstelle im Bitmuster zu-

fällig gewählt. Bis zu dieser Bruchstelle wird das Bitmuster des einen Elters und dahinter die des anderen Elters übernommen.

Die run-Methode der Pattern-Klasse: Spielzüge

Je Spielzug wird in der Welt ein Feld (x0, y0) – das aktuelle Zentrum – zufällig ausgewählt. Dann wird mit der Wahrscheinlichkeit *q* eine Crossing-Over-Operation ausgeführt. Dazu werden aus der *k*-Umgebung des Zentrums zwei Eltern rein zufällig ausgewählt. Diese Individuen können mit dem im Zentrum aber auch miteinander identisch sein. Mittels Crossing-Over-Operation wird das Muster für ein neues Individuum erzeugt. Alternativ zum Crossing-Over wird ein Klon des Individuums der Umgebung gebildet. Im Zuge der Objekterzeugung (Konstruktor) wird der Mutationsoperator auf das neue Muster angewendet.

```
if(rand.nextFloat()<q) newPattern= neighbour(x0, y0).combine(neighbour(x0, y0));
else newPattern= new Pattern(neighbour(x0, y0).pattern.clone());
```

Nach Maßgabe des Simulated Annealing wird das aktuelle Muster im Zentrum durch das neue Muster ersetzt:

```
if (rand.nextFloat()<1/(1+Math.exp((newPattern.f()-world[x0][y0].f())/t)))
world[x0][y0]= newPattern;
```

Bewertungsfunktion

Die Bewertungsfunktion wird durch den folgenden Java-Text realisiert:

```
void evaluate(){
  f=0;
  for(int i=0; i<(mask.length-(fourColors?nNodes:0)); i++) {
    f++;
    for(int j=0; j<pattern.length; j++)
      if(0!=(pattern[j]&mask[i][j]^neg[i][j])) {f--; break;}
  }
}
```

Die Anzahl der zu prüfenden Klauseln ist durch `mask.length` gegeben. Beim Vierfarben-Problem fällt das Verbot der Farbe Grün weg. Die Anzahl der Klauseln verringert sich dementsprechend um die Zahl der Knoten. (Sinnvollerweise überprüft man die optionalen Klauseln zum Schluss.)

Abnahme

Mit dem Programm lassen sich die fundamentalen evolutionären Optimierungsstrategien realisieren:

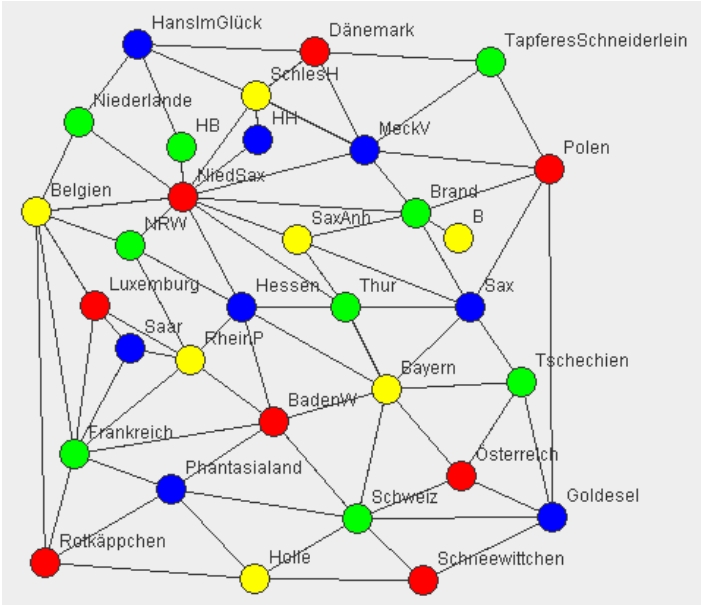
(a) das reine *Mutations-Selektionsverfahren* (n=1, k=0, 0<p, q=0, T=0),

(b) mehrere parallel durchgeführte Mutations-Selektionsverfahren (1<n, k=0, 0<p, q=0, T=0),

(c) ein Verfahren mit globalem *Crossing-Over* (1<n, k=n/2, p=0, 0<q, T=0),

(d) ein Verfahren mit lokalem Crossing-Over (1<n, 0<k<n/2, p=0, 0<q, T=0),

(e) die Kombination der Crossing-Over-Verfahren (c) und (d) mit dem Mutations-Selektions-Verfahren (b) und



(f) Kombination der Verfahren (a) bis (e) mit dem *Simulated Annealing* ($0<T$).

Zwei dieser Strategien liegen den folgenden Experimenten zugrunde.

Experimente

Die Experimente werden mit dem hier abgebildeten Netz durchgeführt. Es hat 32 Knoten, 152 Kanten und folglich 336 Klauseln. Die korrekte Färbung mit vier Farben wurde nach etwa 30 000 Spielzügen mit dem reinen *Mutations-Selektionsverfahren* (n=1, k=0, q=0, p=0.1, T=0) erreicht.

Die zugehörige Textdatei:

```
n  1 /*Spielfeldgroesse*/
k  0 /*Umgebungsparameter*/
h  1000 /*Anzahl der Spielzuege je Lauf*/
p  0.1 /*Mutationswahrscheinlichkeit*/
q  0.0 /*Crossing-Over-Wahrscheinlichkeit*/
T  1.4E-45 /*Temperatur*/
Map
B 1 300 152 Brand;
Brand 3 271 135 B MeckV Sax SaxAnh NiedSax Polen;
MeckV 2 236 92 Brand NiedSax SchlesH Polen Dänemark TapferesSchneiderlein;
Sax 2 308 199 Brand Bayern Thur Polen Tschechien SaxAnh;
SaxAnh 1 190 153 Brand Sax Thur NiedSax;
NiedSax 0 112 124 Brand MeckV SaxAnh HB SchlesH HH Thur Hessen NRW Belgien Niederlande;
Polen 0 362 105 Brand MeckV Sax Goldesel TapferesSchneiderlein;
SchlesH 1 162 55 MeckV NiedSax HH Dänemark HansImGlück;
Dänemark 0 202 25 MeckV SchlesH TapferesSchneiderlein HansImGlück;
TapferesSchneiderlein 3 322 32 MeckV Polen Dänemark;
Bayern 1 251 255 Sax BadenW Hessen Thur Tschechien Österreich Schweiz;
Thur 3 223 199 Sax SaxAnh NiedSax Bayern Hessen;
Tschechien 3 343 250 Sax Bayern Österreich Goldesel;
HB 3 111 90 NiedSax HansImGlück;
HH 2 163 85 NiedSax SchlesH;
Hessen 2 152 199 NiedSax Bayern Thur BadenW RheinP NRW;
NRW 3 76 157 NiedSax Hessen RheinP Belgien;
Belgien 1 12 134 NiedSax NRW Frankreich Luxemburg Niederlande Rotkäppchen;
Niederlande 3 41 73 NiedSax Belgien HansImGlück;
Goldesel 2 364 342 Polen Tschechien Österreich Schweiz Schneewittchen;
HansImGlück 2 81 20 SchlesH Dänemark HB Niederlande;
BadenW 0 174 277 Bayern Hessen RheinP Schweiz Frankreich Phantasialand;
Österreich 0 302 314 Bayern Tschechien Goldesel Schweiz;
Schweiz 3 231 343 Bayern Goldesel BadenW Österreich Phantasialand Schneewittchen Holle;
RheinP 1 117 235 Hessen NRW BadenW Saar Frankreich Luxemburg;
Frankreich 3 38 299 Belgien BadenW RheinP Saar Luxemburg Phantasialand Rotkäppchen;
Luxemburg 0 52 198 Belgien RheinP Frankreich Saar;
Rotkäppchen 0 18 373 Belgien Frankreich Phantasialand Holle;
Schneewittchen 0 276 385 Goldesel Schweiz Holle;
Phantasialand 2 104 323 BadenW Schweiz Frankreich Rotkäppchen Holle;
Holle 1 161 384 Schweiz Rotkäppchen Schneewittchen Phantasialand;
Saar 2 76 227 RheinP Frankreich Luxemburg;
```

Ein weiteres Experiment wurde mit den folgenden Parametern durchgeführt:

```
n  40 /*Spielfeldgroesse*/
k  3 /*Umgebungsparameter*/
h  1000 /*Anzahl der Spielzuege je Lauf*/
o  1 /*Crossing-Over: Bruchstelle*/
p  0.0 /*Mutationswahrscheinlichkeit*/
q  1.0 /*Crossing-Over-Wahrscheinlichkeit*/
T  1.4E-45 /*Temperatur*/
```

Auch bei diesem Experiment, bei dem eine zufällige Anfangsbelegung der 40-mal-40-Welt nur noch durch Crossing-Over verändert wird, kommt es schnell zu einer korrekten Lösung des Vierfarbenproblems. In kleineren Welten (20 mal 20) gerät man bei dieser Methode zuweilen in eine Sackgasse. Aber dann hilft eine vorübergehende Erhöhung der Mutationsrate (beispielsweise auf p=0.1). Und falls das noch nicht ausreicht, führt eine vorübergehende Temperaturerhöhung (beispielsweise $T = 2$) zum Verlassen der Sackgasse. Danach kann man wieder zur reinen Crossing-Over-Strategie zurückkehren. Das funktioniert nicht nur mit dem Bruchstellen-Crossing-Over, sondern auch mit der Zufallsmischung.

Erste Resultate und Folgerung

Das Färbungsproblem für Landkarten der hier betrachteten Größe (32 Knoten, 152 Kanten) ist mit dem reinen Mutations-Selektionsverfahren lösbar.

Das Bruchstellen-Crossing-Over ist außerordentlich zeiteffizient. Bereits ein reines Crossing-Over-Verfahren führt schnell zu einer optimalen Lösung. Aber man kann auch in Sackgassen geraten. Sackgassen erkennt man daran, das Verbesserungen ausbleiben.

Die Überwindung von Sackgassen gelingt, wenn man sich bei der Steuerung der Optimierung an dem weiter unten dargestellten Programmablaufplan orientiert.

Literatur

Allgemein

Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Mass. 1983. Algorithmen auf Graphen: Dijstra u. a.

Grams, Timm: KoopEgo. Links zur Programmdokumentation und zum Programmtext (2007)
http://www.hs-fulda.de/~grams/OekoSimSpiele/KoopEgoProgramm/KoopEgo.pdf
http://www.hs-fulda.de/~grams/OekoSimSpiele/KoopEgoProgramm/KoopEgo.jar

Grams, Timm: PageRank. Link zur Programmdokumentation (2007)
http://www.hs-fulda.de/~grams/Informatik/PageRank.jar

Holland, John H.: Genetische Algorithmen. Spektrum der Wissenschaft (1992) 9, 44-51

Michalewicz, Zbigniew; Fogel, David B.: How to Solve It: Modern Heuristics. Springer-Verlag, Berlin, Heidelberg 2000

Neumann, K.; Morlock, M.: Operations Research. Carl Hanser, München, Wien 1993

Rechenberg, Ingo: Evolutionsstrategie. Optimierung technischer System nach den Prinzipien der biologischen Evolution. Friedrich Frommann Verlag, Stuttgart-Bad Cannstatt 1973

Satisfiability

Appel, Kenneth; Haken, Wolfgang: Der Beweis des Vierfarbensatzes. Spektrum der Wissenschaft (Oktober 1978, Erstedition), 82-91

Dewdney, A. K.: The (New) Turing Omnibus. 66 Excursions in Computer Science. Freeman 1993

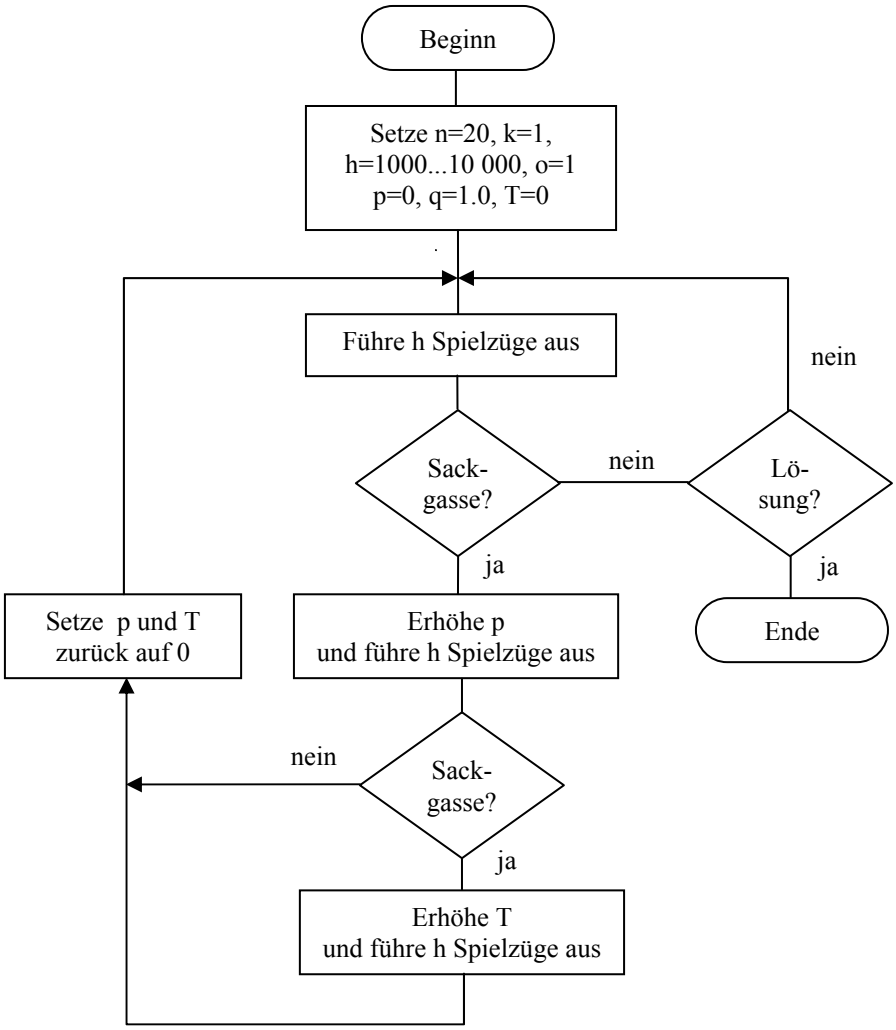
Programmierung

Culwin, Fintan: A Java GUI Programmer's Primer. Prentice Hall, Upper Saddle River, New Jersey 1998

Doberenz, Walter, Druckermüller, Uwe: Java. Programmierung interaktiver WWW-Seiten. Hanser, München, Wien 1998

Fischer, Paul: An Introduction to Graphical User Interfaces with Java Swing. Addison-Wesley 2005

Li, Liwu: Java. Data Structures and Programming. Springer, Berlin, Heidelberg 1998


Beginn
Setze n=20, k=1,
h=1000...10 000, o=1
p=0, q=1.0, T=0
Führe h Spielzüge aus
nein
Sack-
gasse?
nein
Lö-
sung?
ja
ja
Setze p und T
zurück auf 0
Erhöhe p
und führe h Spielzüge aus
Ende
nein
Sack-
gasse?
ja
Erhöhe T
und führe h Spielzüge aus