Markus von Rimscha

Algorithmen kompakt und verständlich

Lösungsstrategien am Computer

Lösungen

Im Folgenden finden Sie einige Antworten auf die Fragen an den Kapitelenden.

www.springer.com/978-3-658-18610-4

Iterative Algorithmen

1. Wir haben mit Hilfe der Systemzeit gemessen, wie lange Bubble-Sort in verschiedenen Fällen rechnet.
   Wie aussagekräftig sind diese Zahlen?
   Welche Ergebnisse erhalten wir auf einem anderen Computer?
   Wie könnten wir die Geschwindigkeit von Bubble-Sort rechnerunabhängig messen?

Die *Messung* der Systemzeit ist sehr problematisch:

- Wie genau ist die Systemuhr?
- Hat während der gemessenen Zeit nur unser Programm gearbeitet, oder hat das Betriebssystem und/oder andere Programme ebenfalls Rechenzeit beansprucht?

Auf anderen Rechnern mit anderen CPUs werden wir vermutlich andere Resultate erhalten.

Besser wäre es, die Geschwindigkeit von Bubble-Sort zu messen, indem wir die Anzahl der durchgeführten Zahlenvergleiche – und damit verbunden ggf. Tausch-Operationen – *zählen*. Diese verändert sich nicht beim Wechsel auf einen anderen Rechner und ist damit ein gutes Mass für die benötigte Rechenzeit von Bubble-Sort.

2. Wie lange rechnet Bubble-Sort im günstigsten Fall?
   Wie lange im ungünstigsten Fall?
   Wie lange im Durchschnitt?

Im günstigsten Fall ist das Feld bereits sortiert. Bubble-Sort durchläuft die `while`-Schleife nur einmal, es werden n Vergleiche durchgeführt.

Im ungünstigsten Fall – wenn also die Zahlen z.B. in der falschen Reihenfolge sortiert sind – wird die `while`-Schleife $n-1$ mal durchlaufen, es werden also $n \cdot (n-1)$ Vergleiche durchgeführt.

Im Durchschnitt wird die `while`-Schleife $n/2$ mal durchlaufen, es werden also $n^2/2$ Vergleiche durchgeführt.

3. Wie verhält sich das Maze-Running-Verfahren, wenn Start und Ziel vertauscht werden?

Der Weg, den Maze-Running findet, ist nicht eindeutig. Deswegen wird ein Tausch von Start und Ziel i.d.R. zu einem anderen Ergebnis führen.

Da Maze-Running aber immer den kürzesten Weg findet, wird die Länge des Weges durch einen Tausch von Start und Ziel nicht beeinflusst.

4. Kann das Maze-Running-Verfahren auch eingesetzt werden, wenn wir uns frei bewegen können und nicht an ein konkretes Raster gebunden sind?

Maze-Running geht davon aus, dass es ein „nächstes" Feld gibt, also einen „Nachbarn".

Folglich ist ein vorgegebenes Raster nötig, und wir können Maze-Running nicht ohne Weiteres einsetzen, wenn wir uns frei bewegen können.

5. Zeichne ein Bild nach folgender Vorschrift: Der Punkt (x_p, y_p) durchläuft die Ebene im Bereich (-2,-2)..(2,2). Jedem Punkt wird dabei als Helligkeitswert die Anzahl der Iterationen zugewiesen, die nötig sind, bis ausgehend vom Punkt (0,0) die Formel $(x_{neu}, y_{neu}) = (x_{alt}^2 - y_{alt}^2 - x_p, 2 \cdot x_{alt} \cdot y_{alt} - y_p)$ entweder $x_{neu}^2 + y_{neu}^2 \geq 4$ liefert oder eine Maximalzahl an Durchläufen erreicht ist. Es entsteht das von **B. Mandelbrot**[1] entdeckte Apfelmännchen.

```
void apfel()
{
  for (int yy = 0; yy < GROESSE; yy++)
  {
    for (int xx = 0; xx < GROESSE; xx++)
    {
      double xPara = -2.0 + (4.0 / GROESSE) * ((double)xx);
      double yPara = -2.0 + (4.0 / GROESSE) * ((double)yy);
      double x = 0.0;
      double y = 0.0;
      int wert = 0;

      while (wert < MAXIMALE_TIEFE && Math.sqrt(x * x + y * y) < 2.0)
      {
        wert++;
        double xAlt = x;
        double yAlt = y;
        x = xAlt * xAlt - yAlt * yAlt - xPara;
        y = 2 * xAlt * yAlt - yPara;
      }

      farbe[xx][GROESSE-yy-1] = wert;
    }
  }
}
```

Code 1: Apfelmännchen

1 B. Mandelbrot, „Les objets fractals", Flammarion, 1975

Rekursive Algorithmen

1. Warum kann es bei der rekursiven Lösung des Türme-von-Hanoi-Spiels nicht vorkommen, dass im Spielverlauf eine größere Scheibe auf einer kleineren liegt?

In der rekursiven Methode werden zwei Schritte ausgeführt: Das Verschieben eines Stapels der Höhe n-1 und das Verschieben einer einzigen Scheibe. Der Stapel wird hier als Ganzes betrachtet, er soll ja gerade mit dem rekursiven Aufruf verschoben werden. Das Verschieben des gesamten Stapels und der einzelnen Scheibe verletzen aber jeweils die Spielregel nicht. Da der Stapel rekursiv nach dem gleichen Schema verschoben wird, das die Spielregel nicht verletzt, bleibt auch insgesamt die Ordnung der Scheiben erhalten.

2. Programmiere einen Computerspieler für das Spiel „4 gewinnt".

Hier kann der allgemeine Spielalgorithmus direkt übernommen werden. Es sind lediglich die entsprechenden Spielregeln zu programmieren.

3. Lässt sich ein Computerspieler für das Spiel „Minesweeper" programmieren?
   Ist der allgemeine Spielalgorithmus geeignet?
   Wo liegen die Unterschiede zu Schach oder 4 Gewinnt?

Bei Minesweeper handelt es sich nicht um ein Spiel zwischen zwei Gegnern sondern um ein reines Logikrätsel. Der allgemeine Spielalgorithmus ist hier also nicht geeignet. Wir können zwar anhand des Spielfeldes berechnen, welche Felder aufzudecken sind, es gibt aber keine gegnerische Reaktion, die zu berücksichtigen wäre.

Wenn wir diese Logikregeln implementieren müssen wir beachten, dass Minesweeper nicht immer lösbar ist. Im Spiel können Situationen entstehen, in denen nur noch geraten werden kann.

4. Lässt sich ein Computerspieler für das Spiel „Skat" programmieren? Wo liegen die Unterschiede zu den bisher betrachteten Spielen?

Der allgemeine Spielalgorithmus ist hier nur sehr bedingt geeignet, wir müssen einige Besonderheiten beachten:

- Es gibt hier *zwei* Mitspieler, von denen aber ggf. nur *einer* unser Gegner ist.
- Es gibt kein „Spielbrett" mit einer „Stellung". Wir kennen lediglich unsere eigenen und die bereits gespielten Karten, also müssen wir beim Spiel der nächsten Karte überlegen, welche Verteilung der Karten am *wahrscheinlichsten* ist. Durch geeignetes Spiel kann unser Gegner uns hier aber ein falsches Bild vortäuschen.

5. Lässt sich ein Computerspieler für das Roulette-Spiel programmieren?
   Wo liegen die Unterschiede zu den bisher betrachteten Spielen?
   Wie sind die Gewinnchancen?

Roulette ist ein reines Glücksspiel bei dem es keine erfolgversprechende Gewinnstrategie gibt, die anhand bereits gefallener Zahlen o.a. berechnet werden könnte. Wir können den Spielalgorithmus also nicht anwenden.

Es gibt allerdings eine Gewinnstrategie:

Setze einen Euro auf rot. Wenn rot fällt, erhalten wir zwei Euro, haben also einen Euro gewonnen. Wenn schwarz oder Null fällt, verlieren wir den Euro. In diesem Fall setzen wir das Spiel mit doppeltem Einsatz fort, setzen also zwei Euro auf rot. Wenn nun rot fällt haben wir einen Euro gewonnen (wir erhalten vier Euro und haben insgesamt drei Euro eingesetzt). Wenn wieder schwarz oder Null fällt verdoppeln wir den Einsatz wieder...

Wir spielen so lange, bis wir zum ersten Mal gewinnen. Unser Gewinn beträgt dann genau den ersten Spieleinsatz, hier also einen Euro.

Diese Gewinnstrategie ist leider rein theoretisch und funktioniert in der Realität nicht:

- Erstens haben wir nicht genug Geld um beliebig lange den Spieleinsatz zu verdoppeln. Falls wir beispielsweise zehn Mal hintereinander Pech gehabt haben müssen wir beim elften Spiel schon 1024 Euro setzen um einen Euro zu gewinnen. Falls wir 20 Mal Pech hatten müssen wir schon eine Million setzen.
- Zweitens wird kaum eine Spielbank Spiele ohne Limit zulassen.

Im Ergebnis gibt es also keine praktisch umsetzbare Gewinnstrategie für Roulette und die Zahl Null, die weder schwarz noch rot ist, sorgt dafür, dass wir auf Dauer Geld verlieren.

6. Ein einfaches Verfahren zum Füllen von beliebigen Flächen in Grafikprogrammen ist der Flood-Fill-Algorithmus: Hier werden rekursiv jeweils alle Nachbarn eines Bildpunkts gefärbt.
   Programmiere den Flood-Fill-Algorithmus.
   Ist dieses Verfahren Stacküberlauf-gefährdet?
   Ebenso wie Maze-Running füllt Flood-Fill Flächen beliebiger Form. Wo liegt der Unterschied zwischen diesen beiden Verfahren?

```
void fuellen(int x, int y, int level)
{
  if (hindernis(x,y)) return;
  if (feld[x][y] > 0) return;
  feld[x][y] = level;
  if (y>0) fuellen(x,y-1,level+1);
  if (x<breite-1) fuellen(x+1,y,level+1);
  if (y<hoehe-1) fuellen(x,y+1,level+1);
  if (x>0) fuellen(x-1,y,level+1);
}
```

Code 2: Flood-Fill

Wenn wir vergleichen, wie Flood-Fill und das Maze-Running-Verfahren einen Bereich füllen erkennen wir, das Maze-Running gleichmäßige Wellen erzeugt während Flood-Fill sich immer tiefer in die Rekursion „hineinwühlt". Flood-Fill führt damit leicht zu Stacküberläufen.

Am folgenden Beispiel sehen wir, welche Werte Maze-Running vergibt und welche Rekursionstiefe Flood-Fill erreicht:

Maze-Running

	1	2	3	4
1	S	1	2	
2		2	3	4
3		3	4	5
4	5	4	5	6

Flood-Fill

	2	3	4	5
18	1	14	5	
17		13	6	7
16		12	11	8
15	14	13	10	9

Abbildung 1: Maze-Running und Flood-Fill

7. Wie kann eine einfache Methode aussehen, die alle möglichen Permutationen eines Strings ermittelt, also alle möglichen Anordnungen der Buchstaben in diesem String?
   Alle Permutationen des Wortes „Eis“ sind „Eis“, „Esi“, „iEs“, „isE“, „sEi“ und „siE“. Ggf. auftretende Dubletten müssen dabei nicht berücksichtigt werden.
   Wie viele Permutationen eines Strings mit n Zeichen gibt es?

Wir bilden eine Liste der permutierten Strings, indem wir als Kandidat für den ersten Buchstaben alle Buchstaben im Original-String durchgehen. Der Rest der permutierten Strings besteht dann aus den rekursiv ermittelten Permutationen der verbleibenden Buchstaben:

```
ArrayList<String> perm(String text)
{
  int laenge = text.length();
  if (laenge == 1)
  {
    ArrayList<String> ergebnis = new ArrayList<String>();
    ergebnis.add(text);
    return ergebnis;
  }

  ArrayList<String> permutationen = new ArrayList<String>();
  for (int i=0 ; i<laenge ; i++)
  {
    char buchstabe = text.charAt(i);
    StringBuilder restText = new StringBuilder();

    for (int j=0 ; j<laenge ; j++)
    {
      if (i==j) continue;
      restText.append(text.charAt(j));
    }

    ArrayList<String> restPermutationen = perm(restText.toString());
    for (String restPermutation : restPermutationen)
    {
      permutationen.add(buchstabe + restPermutation);
    }
  }

  return permutationen;
}
```

Code 3: Permutationen eines Strings

Insgesamt gibt es n! Permutationen eines Strings mit n Zeichen.

www.springer.com/978-3-658-18610-4

8. Wenn eine Methode 4 Parameter vom Typ `int` hat (eine `int`-Variable benötigt 4 Byte), wie groß ist dann der Speicherbedarf einer Baumrekursion auf dem Stack, die eine Breite von 2 Aufrufen und eine Tiefe von 10 Aufrufen hat?
   Als Speicherbedarf wird hier vereinfacht nur der Speicherbedarf für die Methoden-Parameter gerechnet.
   Wie viele Methoden-Aufrufe werden getätigt?
   Wie groß ist der Speicherbedarf auf dem Stack bei einer linearen Rekursion mit einer Tiefe von 1000 Aufrufen? Wie viele Methoden-Aufrufe werden hier getätigt?

Jeder Methoden-Aufruf benötigt 4*4=16 Bytes auf dem Stack (vereinfacht gerechnet).
Die Baum-Rekursion benötigt 1+2+4+8+16+32+64+128+256+512+1024 = 2047 = 2^{11}-1 Aufrufe und belegt dabei 10*16 = 160 Byte, da für den Speicherbedarf nur die *Tiefe* der Rekursion relevant ist.
Die lineare Rekursion tätigt 1000 Aufrufe und benötigt dabei 1000*16 = 16000 Byte.

9. Ein wesentlicher Algorithmus bei der Berechnung fotorealistischer Grafiken ist das Raytracing, der auf **A. Appel**, **R. A. Goldstein** und **R. Nagel** zurückgeht[2]. Dabei wird die Natur auf den Kopf gestellt: Es treffen keine Lichtstrahlen mehr auf das Auge, sondern das Auge schickt Sehstrahlen in die Welt, die beispielsweise in einem Spiegel reflektiert oder an einer Glasfläche gebrochen werden. Auch die Kombination aus beidem kommt vor.
   Wie sieht der Raytracing-Algorithmus prinzipiell aus?
   Es soll nicht darum gehen, Gegenstände im Raum korrekt zu beleuchten etc. und auf diese Weise tatsächlich ein fotorealistisches Bild zu erzeugen.
   Relevant ist nur die rekursive Natur eines Verfahrens, das durch die Berechnung von Brechung und Spiegelung ermittelt, welche Objekte im Raum für den Betrachter sichtbar sind.

2 A. Appel, „Some Techniques for Shading Machine Renderings of Solids", Spring Joint Computer Conference, Arlington, Thompson Books, 1968, S. 37-45
R. A. Goldstein, R. Nagel, „3D Visual Simulation", Simulation, Vol. 16, Nr. 1, Jan. 1971, S. 25-31

Ein stark vereinfachter Raytracing-Algorithmus bei dem es lediglich um die rekursive Natur der Verfahrens geht, nicht um eine korrekte Beleuchtung, Schatten usw.:

```
for (int x=0 ; x<breite ; x++)
{
  for (int y=0 ; y<hoehe ; y++)
  {
    Strahl strahl = positionAuge.gibStrahlInRichtung(x, y);
    Farbe farbe = new Farbe(0.0);
    raytrace(strahl, 1.0, 0, farbe);
    zeichnePunkt(x, y, farbe);
  }
}

void raytrace(Strahl strahl, double anteil, int tiefe, Farbe farbe)
{
  if (anteil < GRENZE_ANTEIL || tiefe > GRENZE_TIEFE) return;

  Punkt schnittpunkt = gibNächstenSchnittpunkt(bildObjekte);
  if (schnittpunkt == null)
  {
    farbe.anteilHinzufügen(anteil * hintergrundfarbe);
  }
  else
  {
    Oberfläche fläche = bildObjekte.gibOberfläche(schnittpunkt);
    farbe.anteilHinzufügen(anteil * fläche.farbe * hintergrundlicht);
    double gebrochenerAnteil =
      fläche.gibBrechungsAnteil(strahl, schnittpunkt);
    Strahl gebrochenerStrahl =
      fläche.gibGebrochenenStrahl(strahl. schnittpunkt);

    double reflexionsAnteil =
      fläche.gibReflexionsAnteil(strahl, schnittpunkt);
    Strahl reflektierterStrahl =
      fläche.gibReflektiertenStrahl(strahl, schnittpunkt);

    raytrace(
      gebrochenerStrahl, anteil*gebrochenerAnteil, tiefe+1, farbe);
    raytrace(
      reflektierterStrahl,anteil*reflektierterAnteil,tiefe+1,farbe);
  }
}
```

Code 4: Raytracing

Dynamische Algorithmen

1. Kann der Spielalgorithmus durch dynamische Programmierung beschleunigt werden?

Es ist möglich, dass die gleiche Spielstellung durch unterschiedliche Züge und Reaktionen entsteht. Also kann das Verfahren beschleunigt werden, wenn Stellungen gespeichert werden und dadurch jede Spielstellung nur einmal berechnet wird.

2. Kann das Türme-von-Hanoi-Spiel durch dynamische Programmierung beschleunigt werden?

Das Türme-von-Hanoi-Problem lässt sich in 2^n-1 Schritten lösen, genau so viele Methodenaufrufe benötigen wir. Die gleiche Situation kann sich im Spielverlauf nicht ergeben, denn wenn wir zwei mal die gleiche Stellung erhalten würden, würden wir offenbar unendlich lange spielen, weil wir immer wieder zu dieser Stellung kämen. Also lässt sich das Türme-von-Hanoi-Spiel nicht auf diese Weise beschleunigen.

www.springer.com/978-3-658-18610-4

Heuristische Algorithmen

1. Welches Problem haben wir, wenn wir Menschen mit Bucket-Sort nach ihrer Körpergröße sortieren möchten? Was ist zu tun?

Die Körpergröße der Menschen ist nicht gleichmäßig verteilt. Es gibt beispielsweise wesentlich weniger Menschen, die 2.20m-2.30m groß sind als Menschen, die 1.70m-1.80m groß sind. Wir müssen dies berücksichtigen, in dem wir die Größe nicht gleichmäßig auf die Eimer verteilen sondern diese Verteilung berücksichtigen. Mathematisch können wir dazu die kumulierte Verteilungsfunktion der Werte invertieren.

2. Wo wenden wir im Alltag heuristische Verfahren an?

Ständig, überall. In welcher Reihenfolge besuchen wir die Geschäfte, wenn wir eine bestimmte CD suchen? In der Reihenfolge, in der wir erfahrungsgemäß davon ausgehen, dass eine gute Auswahl vorhanden und die gesuchte CD vorrätig ist. Usw.

In der Tat dürfte es wesentlich schwieriger sein einen Bereich aus dem Alltag zu finden, in dem wir *ohne* Erfahrungswerte auskommen.

Meistens kombinieren wir heuristische mit probabilistischen Verfahren.

3. Eine häufige Anwendung von Heuristiken besteht darin, Daten bzw. Arbeitsschritte gem. ihrer bisherigen Häufigkeit oder Wichtigkeit zu sortieren. Auf diese Weise wird oft eine geschickte Reihenfolge im Zugriff gefunden. Wenn die Heuristik versagt – was immer passieren kann – funktioniert das Verfahren noch immer. Zwar wird dann nicht die bestmögliche Effizienz erreicht, es werden aber noch immer korrekte Resultate geliefert.
   Welche Bilder einer großen Bilddatenbank legen wir in einen schnellen Datenbankpuffer, der nur eine sehr begrenzte Kapazität hat?
   In welcher Reihenfolge sollten wir beim Abheben von Geld am Automat die Schritte „Gewünschten Betrag eingeben" und „PIN eingeben" durchführen wenn wir davon ausgehen, dass 90% aller abgelehnten Abhebungen wegen mangelhafter Kontodeckung abgelehnt werden, nur 10% wegen falsch eingegebener PIN.

Vermutlich werden Bilder, die bisher sehr beliebt waren, auch künftig häufig angesehen. Also werden wir die beliebtesten Bilder in den Puffer legen.
Wir prüfen zuerst die Bedingung von der wir am ehesten einen Abbruch erwarten, also fragen wir zuerst den gewünschten Betrag ab.

Zufallsgesteuerte Algorithmen

1. Muss der Parameter T beim Metropolis-Algorithmus während der Suche konstant bleiben?

   Können wir das Verfahren verbessern, wenn wir T im Lauf der Zeit anpassen? Wie gehen wir bei der Anpassung vor? Sollte sich eine solche Anpassung an der konkreten Anwendung orientieren?

Der Metropolis-Algorithmus verlangt nicht, dass T konstant bleibt, wir können unterschiedliche Strategien wählen um T anzupassen.

Diese sollten sich immer an der konkreten Anwendung orientieren.

Eine mögliche Strategie könnte folgendermaßen aussehen:

- Wir beginnen mit einem großen Wert für T.
- Im Laufe der Berechnung senken wir T schrittweise.
  Auf diese Wiese werden wir zu Beginn recht zufällig suchen und später immer strenger eine Verbesserung der Werte fordern, d.h. nur noch lokal nach einem Optimum suchen.
- Wenn wir bei einem sehr kleinen T angelangt sind und ein lokales Minimum gefunden haben, können wir diese bisher ggf. beste Lösung vermerken, T wieder vergrößern und von vorne beginnen.

2. Sollten wir als Zufallsgenerator einen „echten“ Zufallsgeneratur verwenden, der beispielsweise anhand der Mausbewegungen Zufallsdaten sammelt oder sollten wir „Pseudo-Zufallsgeneratoren“ einsetzen, die beispielsweise ausgehend von einem Startwert Zahlenfolgen liefern, die zwar zufällig aussehen, aber tatsächlich jederzeit reproduziert werden können?

Das hängt stark von der jeweiligen Anwendung ab. Bei Verfahren wie dem Simulated Annealing ist es uns vermutlich wichtig, reproduzierbare Ergebnisse zu bekommen, wir werden also Pseudo-Zufall wählen. Wir können noch immer unterschiedliche „Zufalls“-Zahlenfolgen bekommen, wenn wir mit unterschiedlichen Startwerten beginnen.

Wenn wir aber beispielsweise mit Datenverschlüsselung zu tun haben, wäre es fatal, wenn man nur den Startwert korrekt erraten muss um die gesamte Zufallsfolge zu erhalten. Hier ist Reproduzierbarkeit meist von Nachteil, wir benötigen „echten“ Zufall. Aus diesem Grund erhält man von Verschlüsselungsprogrammen häufig Aufforderungen wie „Bewegen Sie die Maus eine Zeit lang zufällig im Fenster und klicken Sie dann auf 'Weiter' … “.

3. Zeichne ein Bild nach folgender Vorschrift: Ausgehend vom Punkt (0,0) wird der nächste Punkt nach einer der folgenden Formeln berechnet, die zufällig anhand der angegebenen Wahrscheinlichkeiten ausgewählt werden.

$$(x_{neu}, y_{neu}) = (0.85 \cdot x_{alt} + 0.04 \cdot y_{alt}, -0.04 \cdot x_{alt} + 0.85 \cdot y_{alt} + 0.16) \quad , \quad 85\%$$

$$(x_{neu}, y_{neu}) = (0.20 \cdot x_{alt} - 0.26 \cdot y_{alt}, 0.23 \cdot x_{alt} + 0.22 \cdot y_{alt} + 0.16) \quad , \quad 7\%$$

$$(x_{neu}, y_{neu}) = (-0.15 \cdot x_{alt} + 0.28 \cdot y_{alt}, 0.26 \cdot x_{alt} + 0.24 \cdot y_{alt} + 0.04) \quad , \quad 7\%$$

$$(x_{neu}, y_{neu}) = (0.00, 0.16 \cdot y_{alt}) \quad , \quad 1\%$$

Es entsteht ein Farn nach **M. Barnsley**[3].

```
void farn()
{
  double x = 0;
  double y = 0;

  for (int i = 0; i < DURCHLAEUFE; i++)
  {
    double xAlt = x;
    double yAlt = y;
    double z = zufall.nextDouble();

    if (z < 0.85)
    {
      x =  0.85 * xAlt + 0.04 * yAlt + 0.00;
      y = -0.04 * xAlt + 0.85 * yAlt + 0.16;
    }
    else if (z < 0.92)
    {
      x =  0.20 * xAlt - 0.26 * yAlt + 0.00;
      y =  0.23 * xAlt + 0.22 * yAlt + 0.16;
    }
    else if (z < 0.99)
    {
      x = -0.15 * xAlt + 0.28 * yAlt + 0.00;
      y =  0.26 * xAlt + 0.24 * yAlt + 0.04;
    }
    else
    {
      x =  0.00 * xAlt + 0.00 * yAlt + 0.00;
      y =  0.00 * xAlt + 0.16 * yAlt + 0.00;
    }
    farn[(int) (GROESSE / 2 + x * (GROESSE – 1))]
        [(int) (GROESSE - 1 – y * (GROESSE - 1))] = true;
  }
}
```

Code 5: Barnsley-Farn

3 M. Barnsley, „Fractals Everywhere”, Academic Press, 1988

Genetische Algorithmen

1. Warum liefert sich der genetische Algorithmus bei unterschiedlichen Mutationswahrscheinlichkeiten p unterschiedlich gute Resultate?
   Warum führen große Werte für p schnell zu guten Resultaten?
   Warum führen kleine Werte für p zwar langsamer zu guten Ergebnissen, langfristig aber scheinbar zu insgesamt besseren Werten?

Größere Werte für p führen dazu, dass die Ergebnisse stärker von der reinen Kombination der alten Gene abweichen. Solange diese alten Lösungen also noch nicht besonders gut sind, ist eine stärkere Abweichung hier von Vorteil und führt ggf. schneller zu einer Verbesserung.

Wenn aber bereits viele Generationen durchlaufen sind, und die Lösungen bereits sehr gut sind, führt ein größes p dazu, dass starke Abweichungen von einer bereits sehr guten Lösung entstehen, was zu schlechteren Resultaten führen kann.

Im konkreten Beispiel sorgt $p=1\%$ bei $n=200$ dafür, dass das Optimum quasi nicht gefunden werden kann. Hierzu müssten alle 200 Werte richtig belegt sein, was kaum möglich ist, wenn statistisch jeweils 2 der Werte zufällig belegt sind. Daher kommen wir nur auf ca. 98,5%. Bei $p=0{,}1\%$ jedoch ändert sich statistisch weniger als ein Wert, d.h. gute Lösungen bleiben „stabiler". Hier haben wir also die Chance, alle 200 Werte korrekt zu belegen, und die Lösung nicht gleich durch Mutation wieder zu zerstören.

2. Ist ein genetischer Algorithmus geeignet um die Primfaktorzerlegung einer Zahl zu bestimmen?

Bei der Primfaktorzerlegung geht es darum, zwei ganzzahlige Faktoren zu finden deren Produkt *exakt* die vorgegebene Zahl ergibt. Eine gute Näherung interessiert hier nicht, also kann der genetische Algorithmus nicht ohne Weiteres eingesetzt werden.

Probabilistische Algorithmen

1. Wo wenden wir im Alltag Monte-Carlo-Verfahren an?

Ständig, überall. Vermutlich wird es genau dann während der nächsten Stunden regnen, wenn jetzt schon dunkle Wolken am Himmel sind. Usw.

2. Wo wenden wir im Alltag Las-Vegas-Verfahren an?

Ständig, überall. Wenn wir dringend telefonieren müssen und jemand anderen mit dem Mobiltelefon sprechen sehen, dann wissen wir sicher, dass er ein Telefon hat, das wir uns evtl. ausleihen können. Wenn nicht telefoniert wird, könnte er trotzdem ein Telefon dabei haben, oder auch nicht. Usw.

3. Wie kann ein einfaches probabilistisches Verfahren aussehen, das zwei lange Strings auf Gleichheit prüft?
   Betrachte dazu folgende Test-Strings:

   ```
   Baum, Himmel, Fluss
   Fensterscheibe, Fensterladen, Fenstermacher
   Schreiner, Schraubenzieher, Schrift, Stein
   ```

   Um was für ein Verfahren handelt es sich?
   Lässt sich die Fehlerwahrscheinlichkeit des Verfahrens in Richtung 0 senken?
   Wie verhält sich der Aufwand in diesem Fall?

Wir vergleichen nur die ersten k Buchstaben. Es handelt sich wieder um eine Kombination aus Las-Vegas- und Monte-Carlo-Verfahren:
Wenn wir einen Unterschied feststellen, sind die Strings sicher nicht gleich. Wenn wir keinen Unterschied feststellen, sind die Strings wahrscheinlich gleich. Die Wahrscheinlichkeit ist um so größer, je größer k ist.
Wir können die Fehlerwahrscheinlichkeit auf 0 reduzieren, indem wir für k die Länge des Strings wählen. In diesem Fall ist das Verfahren ein einfacher Stringvergleich mit entsprechendem Aufwand.

1. Ist es erforderlich, Zugriffe auf gemeinsame Daten zu synchronisieren, selbst wenn nur ein einziger der beteiligten Threads überhaupt Schreibzugriffe durchführt, während alle anderen Threads die Daten ausschließlich lesen?

Ja. Das wird am Beispiel einer Überweisung klar, die letztlich *zwei* Buchungen durchführt: Die Belastung beim Auftraggeber und die Gutschrift beim Empfänger. Wenn nun *zwischen* diesen beiden Buchungen ein anderer Thread die Kontostände liest, ist Geld verloren, denn es wurde bereits dem Auftraggeber belastet aber noch nicht dem Empäfnger gutgeschrieben.

2. Wie lange darf eine Berechnung maximal dauern, wenn wir sie ohne Thread-Pool direkt im Grafik-Thread ausführen möchten?

Wenn wir unterstellen, dass die grafischer Oberfläche mindestens 10mal pro Sekunde aktualisiert werden muss, um eine flüssige und ruckfreie Reaktion zu zeigen, darf also eine Berechnung maximal 100ms benötigen, unter der Annahme, dass die Aktualisierung der grafischen Oberfläche selbst sehr schnell ist.

3. Ist es sinnvoll, in einer Anwendung „kleine“ Aufgaben direkt im Grafik-Thread zu erledigen und nur „große“ Aufgaben in einen Thread-Pool zu verlagern?

Prinzipiell ist das möglich, allerdings sehr gefährlich: Die Aufgaben im Thread-Pool werden in einer völlig anderen Reihenfolge bearbeitet als die Aufgaben, die direkt in der grafischen Oberfläche gerechnet werden. Im Ergebnis ist folgende Situation durchaus wahrscheinlich: An der Oberfläche wird zuerst Aktion A ausgelöst (im Thread-Pool), danach Aktion B (direkt im Grafik-Thread). Tatsächlich wird aber zuerst Aktion B ausgeführt, weil der Thread-Pool noch beschäftigt ist und Aktion A erst später ausführt. Also ist in der Regel von diesem Vorgehen abzuraten.

4. Wie könnte eine *einfache* Strategie aussehen, mit deren Hilfe wir sehr wichtigen Code – wie beispielsweise die Buchungsfunktionen für ein Konto – auf korrekte Synchronisation von Datenzugriffen und Deadlock-Situationen *testen* können?

   Ist der Ansatz auch für große und komplexe Software praktikabel?

Das Kernproblem ist, dass die Thread-Wechsel nicht deterministisch sind (siehe „Deterministisches Multi-Threading“). Wir können aber künstlich dafür sorgen, dass die Thread-Wechsel deterministisch sind, indem (automatisiert) an *allen möglichen Stellen* der relevanten Methoden Synchronisations-Anweisungen eingebaut werden, die eine bestimmte Abarbeitungs-Reihenfolge erzwingen – im trivialsten Fall könnte man das mit Hilfe eines `Sleep` programmieren. Auf diese Weise haben wir manuell für deterministische Thread-Wechsel gesorgt – mindestens im Falle der `Sleep`-Lösung ist dies jedoch sehr langsam (wie lange muss der `Sleep` sein?). Dann können (wieder automatisiert) alle möglichen oder zumindest sehr viele Thread-Wechsel-Szenarien durchgetestet werden. Bei großen Anwendungen wird das sehr aufwändig, die `Sleep`-Variante – die ohnehin nur als Trivial-Lösung dem Verständnis der Idee dient – ist hier natürlich völlig unbrauchbar.

5. Nehmen wir an, durch Bedienhandlungen an der grafischen Oberfläche wird alle 500ms ein neuer Auftrag gestartet, der in den Thread-Pool gelegt wird. Nehmen wir an, diese Aufgaben benötigen Rechenzeiten zwischen 300ms und 700ms.

   Über wie viele Arbeits-Threads müsste unser Thread-Pool verfügen, so dass wir *sicher* sein

können, dass *alle* Aufgaben parallel bearbeitet werden. Die Situation „A wartet auf B, B wartet auf C, ...“ soll also *garantiert* nicht zu einem Deadlock führen, egal wie lang diese Kette ist.

Lediglich das unvermeidbare Deadlock der Situation „A wartet auf B, B wartet auf C, C wartet auf A“ soll auftreten können.

Unendlich viele! Im *Durchschnitt* brauchen unsere Aufträge zwar scheinbar 500ms, wir können also berechnen, wie lange die Warteschlange der anstehenden Aufgaben *im Durchschnitt* sein wird, das hilft uns aber nicht weiter: Wenn wir wirklich *sicher* sein wollen, müssen wir davon ausgehen, dass auch viele Aufträge hintereinander eintreffen können, die alle 700ms benötigen. Die Warteschlange kann also beliebig lang werden, dementsprechend müsste unser Thread-Pool über unendlich viele Arbeits-Threads verfügen.
Natürlich ist dieser Fall recht unwahrscheinlich, es lässt sich berechnen, wie groß die Wahrscheinlichkeit ist, dass 10 Arbeits-Threads genügen, oder 100, oder 1000. Wenn wir das Programm aber lange genug laufen lassen, ist es nur eine Frage der Zeit, bis einmal mehr Aufträge als Threas vorhanden sind und wir wieder in die Deadlock-Situation geraten können.

6. Bei Anwendungen mit einer grafischen Oberfläche haben wir oft die Anforderung, eine große Aufgabe zu erledigen – etwa das Drucken einer großen Datei oder die Berechnung einer großen Tabelle. Dies soll natürlich komfortabel mit Fortschrittsbalken geschehen.

   Bevor diese Berechnungen fertig sind, soll der Anwender aber nach Möglichkeit nicht weitere Funktionen anstossen – nicht zuletzt, weil die dafür nötigen Daten evtl. noch gar nicht existieren.

   Wie können wir verhindern, dass der Anwender neue Befehle anstösst, wenn wir andererseits mühsam einen Thread-Pool eingesetzt haben, um den Grafik-Thread möglichst ungestört arbeiten zu lassen?

Offenbar können wir die Arbeit nicht in den Grafik-Thread verlegen. Wir müssen also die grafische Oberfläche explizit sperren. Dies tun wir *nicht*, indem wir den Grafik-Thread beschäftigen. Statt dessen können wir z.B. Buttons inaktiv schalten oder die ganze Oberfläche deaktivieren und den Mauscursor entsprechend anzeigen:

```
getGlassPane().setVisible(true);
setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
```

Datenstrukturen

1. Welche Datenstruktur(en) werden benötigt, um einer alphabetisch sortierten Liste von Strings beliebige Objekte zuzuordnen?

Wir können diese Funktion mit einer `HashMap` und einem `TreeSet` selbst programmieren oder eine `TreeMap` einsetzen.
Beides kann sinnvoll sein:
Während die Verwendung der `TreeMap` die einfachste Lösung ist müssen wir beachten, dass z.B. `containsKey` hier Aufwand $O(\log(n))$ erzeugt.
Wenn wir eine `HashMap` und ein `TreeSet` parallel betreiben, müssen wir bei Änderungen zwar beide Datenstrukturen aktualisieren, bei Abfragen können wir aber die jeweils besser geeignete Lösung wählen, `containsKey` in der `HashMap` lässt sich mit Aufwand $O(1)$ lösen.

2. Warum verfügt die `ArrayList` über einen weiteren Konstruktor, der als Parameter die `initialCapacity` übergeben bekommt, wenn doch der Sinn der `ArrayList` gerade darin besteht, dynamisch wachsen und schrumpfen zu können?

Die `ArrayList` *kann* dynamisch wachsen. Aus Performance-Gründen wächst oder schrumpft sie aber – je nach konkreter Implementierung der Klasse `ArrayList` – nicht immer um jeweils nur ein einziges Element sondern um Blöcke aus mehreren Elementen. Dementsprechend ist es ggf. sinnvoll, der `ArrayList` von Anfang an mitzuteilen, für wie viele Elemente sie größenordnungsmäßig gedacht ist.

3. Warum verfügt die Klasse `TreeSet` nicht über eine Methode `get(int i)`, die das i-te Element in der Reihenfolge der Sortierung liefert, so wie sie beispielsweise von der Klasse `ArrayList` angeboten wird?
   Warum fordert das Interface `Set`, das von der Klasse `TreeSet` implementiert wird, keine `get(int i)`-Methode?

Es gibt keinen zwingenden Grund, warum `TreeSet` diese Methode nicht anbietet. Da die Elemente im `TreeSet` sortiert sind, liesse sich ein `get(int i)`-Zugriff natürlich implementieren. Technisch ist es wohl dadurch begründet, dass das Interface `SortedSet`, das von der Klasse `TreeSet` implementiert wird, keine `get`-Methode kennt.
Ein Grund hierfür könnte darin liegen, dass der `get`-Zugriff eher für Datenstrukturen wie Listen oder Felder gedacht ist, die die Ausführung von `get` mit O(1) Aufwand ermöglichen. Im `TreeSet` wäre der Aufwand von `get` O(log(n)).
Eine `get`-Methode im `Set`-Interface wäre unsinnig, denn ein allgemeines `Set` ist *unsortiert*, es wäre also überhaupt nicht klar, welches das i-te Element sein soll.

4. Um Objekte einer Klasse in einem `TreeSet` sortiert abzulegen, muss die entsprechende Klasse das Interface `Comparable` implementieren, das die Methoden `compare` und `equals` fordert.
   Was bedeutet es in diesem Zusammenhang, dass die Implementierung der `compare`-Methode „consistent“ mit der Implementierung der `equals`-Methode sein muss?

„Konsistent“ bedeutet in diesem Fall, dass `compare` *genau dann* den Wert 0 liefert, wenn `equals` den Wert `true` liefert.

5. Zur Berechnung der k-Schritt-Adjazenz-Matrix benötigen wir häufig die Matrix-Multiplikation.
   Existieren schnellere Verfahren zur Multiplikation binärer Matrizen, als die Schulmethode, die $O(n^3)$ Aufwand erzeugt?

Es gibt schnellere Algorithmen zur Matrix-Multiplikation als die Schulmethode mit Aufwand $O(n^3)$, z.B. die Matrix-Multiplikation nach Strassen mit Aufwand $O(n^{2.81})$. Es existieren noch schnellere Algorithmen, die allerdings einen hohen konstanten Faktor beinhalten, also nur bei sehr großen n lohnend sind.

Da die Matrix komplett verarbeitet werden muss, ist $O(n^2)$ eine untere Schranke für den nötigen Aufwand.

Diese Verfahren operieren jedoch beispielsweise auf den reellen Zahlen, die im mathematischen Sinn ein Ring sind.

Binäre Werte sind jedoch kein Ring, es fehlt die Subtraktion. Deswegen können die anderen Verfahren nicht ohne Weiteres angewendet werden.

Wir können solche schnelleren Verfahren also nur direkt anwenden, indem wir die verwendeten mathematischen Operationen (Addition, Multiplikation usw.) umdefinieren.

Alternativ können wir aber jederzeit eine binäre true/false-Matrix als reelle Matrix interpretieren: false=0.0, true=1.0. Dann können wir die normale (oder eine schnellere) Matrix-Multiplikation in den reellen Zahlen durchführen und danach die reelle Matrix wieder in eine binäre Matrix verwandeln, in dem jeder Wert ungleich 0.0 als true interpretiert wird.

Achtung: Dieser Trick funktioniert jedoch nur, wenn wir die „Addition“ der Binärwerte als OR interpretieren, nicht als XOR. Auch müssen wir uns hier mit Rundungsfehlern etc. beschäftigen.

Maschinelles Lernen

1. Wir sind bisher davon ausgegangen, dass Kriterien nur über endlich viele Ausprägungen verfügen. Wie gehen wir vor, wenn wir z.B. ein Kriterium „Körpergröße" haben, das beliebige `double`-Werte zulässt?

Wir bilden Bereiche und erzeugen damit wieder endlich viele Möglichkeiten, z.B. Körpergröße <150cm, 150-165cm, 165-185cm, 185-200cm, >200cm

2. Welcher Entscheidungsbaum für unser Freizeitverhalten ergibt sich anhand folgender Beispiele:

Freunde?	Wetter?	Kinofilm?	Wochenende?	Entscheidung
ja	Bedeckt	ja	ja	Kino
ja	Bedeckt	ja	nein	Kino
ja	Bedeckt	nein	ja	Club
ja	Bedeckt	nein	nein	Café
nein	Regen	nein	ja	Daheim
nein	Regen	nein	nein	Daheim
nein	Bedeckt	ja	ja	Kino
nein	Bedeckt	ja	nein	Kino

Diese Beispiele tauchen auch in der bereits besprochenen Tabelle auf, sie lassen sich also mit dem ermittelten Entscheidungsbaum beschreiben.
Wenn wir aber die Berechnung neu starten, ergibt sich ein völlig anderer Baum.
Nun ist das Kino-Kriterium das stärkste, das Wetter – bisher das stärkste Kriterium – taucht überhaupt nicht mehr auf:

```
if (Film)
{
  Kino();
}
else
{
  if (Freunde)
  {
    if (Wochenende)
    {
      Club();
    }
    else
    {
      Café();
    }
  }
  else
  {
    Daheim();
  }
}
```

Code 6: Entscheidungsbaum

Grundsätzlich muss der gefundene Entscheidungsbaum nicht eindeutig sein, es kann mehrere gleich starke Kriterien geben, von denen eines als stärkstes Kriterium ausgewählt wird.

Schwarmintelligenz

1. In der Realität wird es Städte geben, zwischen denen keine direkte Wegverbindung existiert. Wie kann das beim Ameisenalgorithmus geschickt berücksichtigt werden?
   Das grundsätzliche Vorgehen soll dabei so wenig wie möglich verändert werden.

Wir ändern das Verfahren nicht, sondern tragen als Entfernung zwischen unverbundenen Städten eine sehr große Zahl M ein. Da eine kurze Rundreise gesucht wird, wird also jede Lösung schnell verworfen werden, die Wege der Länge M enthält.

2. Wie können Einbahnstraßen berücksichtigt werden?
   Das grundsätzliche Vorgehen soll dabei wieder so wenig wie möglich verändert werden.

Wir tragen in der befahrbaren Richtung die entsprechende Entfernung ein, in der Gegenrichtung M.

3. Ist es von Vorteil oder von Nachteil, wenn zusätzlich zur kollektiven Intelligenz auch das einzelne Individuum intelligent ist, über ein eigenes Bewusstsein verfügt usw.?

Auf den ersten Blick würden wir wohl sagen: „Intelligenz schadet nie".

Also ist es sicherlich von Vorteil, wenn auch die Individuen intelligent sind. Das kann selbstverständlich richtig sein, muss aber nicht unbedingt zu einem Vorteil führen:

In dem Maße, in dem die Individuen intelligent sind, werden sie auch die Situation aus ihrer persönlichen Sicht bewerten und egoistische Interessen verfolgen. Das Wesen der Schwarmintelligenz besteht aber darin, dass das einzelne Individuum wenig zählt. Das Individuum muss seine eigenen Interessen bedingungslos den Interessen der Gemeinschaft unterordnen – oder eben über so wenig Intelligenz verfügen, dass es überhaupt nicht im Stande ist, das Konzept „eigener Interessen" zu begreifen.

Wenn wir etwa beim Ameisenalgorithmus die Ameisen durch Menschen ersetzen würden, würde sich ein Mensch wohl weigern, eine Route mit geringer Markierung zu betreten. Er hätte Angst, die nächste Stadt könnte so weit entfernt sein, dass er auf dem Weg dorthin verhungert.

Neuronale Netze

1. Ist es vernünftig, als Trainingsbeispiele jeweils nur die Bilder zu verwenden, die später erkannt werden sollen? Wie sollten Trainingsbeispiele gewählt werden?

Es kann vorteilhaft sein, mehrere *ähnliche* Trainingsbeispiele zu haben.

Dies ist ein erster, sehr einfacher Weg um Overfitting zu vermeiden.

2. Zu welchem Unterschied im Verhalten führt es, wenn wir nicht jedes Neuron mit jedem Neuron der Vorgängerschicht verbinden sondern nur einzelne Verbindungen zulassen?

Wenn jede Verbindung prinzipiell zulässig ist, ist durch ein Gewicht $g_i=0$ auch jede Variante abgedeckt, in der bestimmte Verbindungen gar nicht existieren.

Der Unterschied besteht darin, dass wir mehr Gewichte haben, die im Zuge des Trainings ermittelt werden müssen. Das Wegfallen einer Verbindung muss also ebenfalls erst erlernt werden.

http://www.springer.com/978-3-658-18610-4

Algorithmen kompakt und verständlich
Lösungsstrategien am Computer
von Rimscha, M.
2017, X, 177 S. 168 Abb., 4 Abb. in Farbe., Softcover
ISBN: 978-3-658-18610-4