Aufgabe 3: Voll daneben

Team: Jan Niklas Groeneveld
Team-ID: 00828

22. November 2018

Inhaltsverzeichnis

1 Lösungsidee

Um das Ziel einer geringen Auszahlung, die Al Capone erreichen soll, zu erreichen, ist es wegen eines zu hohen Aufwandes nicht effizient, alle Möglichkeiten durchzurechnen, um die günstigste Anordnung durch Ausprobieren zu finden. Deshalb soll ein Approximationsverfahren angewandt werden, das auf eine einem genetischen Algorithmus ähnliche Weise sukzessiv geringere Auszahlungen erreicht. Aus der Erfahrung mit einem früheren Algorithmus, der aus Effizienzproblemen nicht zur Anwendung geeignet ist, ist bekannt, dass Al Capone immer auf einen Einsatz eines Spielers legen sollte, niemals in den Bereich zwischen den Einsätzen. Das erscheint auch in der Hinsicht logisch, dass so auf jeden Fall immer ein Einsatz vollständig neutralisiert wird. Das Wissen um diese Tatsache ermöglicht es, mit einer geringeren Zahl von auszuprobierenden Orten zu arbeiten.

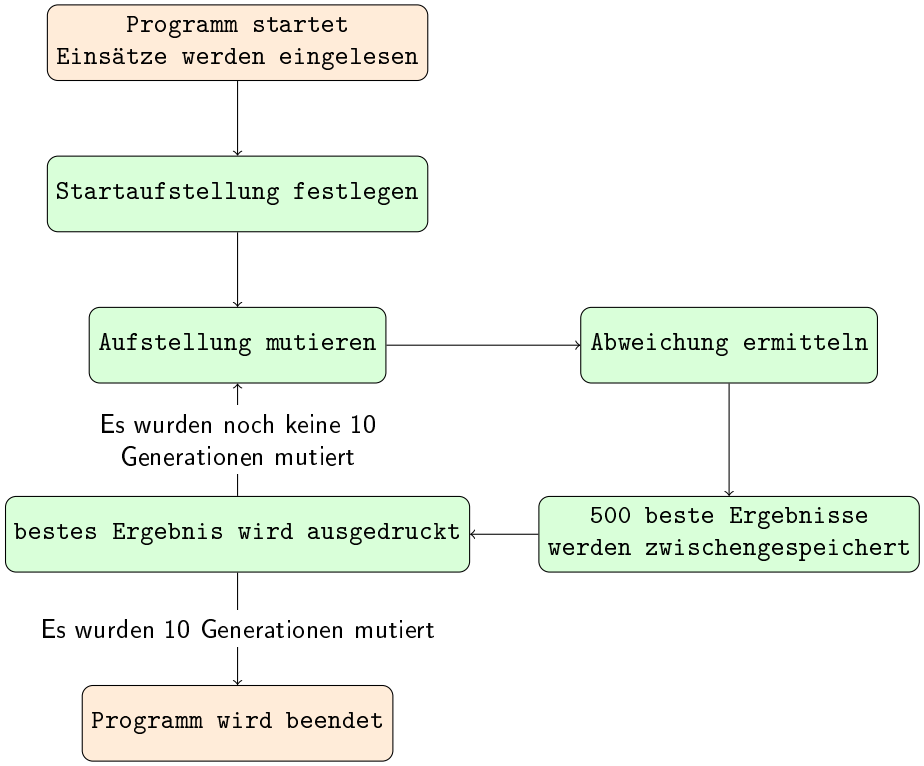
Als erstes werden die Werte aus der Datei eingelesen. Im Anschluss wird nach folgender Formel zuerst eine Ausgangsstellung gesucht, die bereits eine passable Näherung darstellt, von der ausgehend die genetische Optimierung gestartet wird:

$$p(i) = (i+1) \cdot \frac{n_e}{n_{Al}} \tag{1}$$

Dabei ist $p(i)$ die Nummer des Einsatzes, auf den in Abhängigkeit der Zählvariable i gesetzt wird, n_e die Anzahl der Einsätze, und n_{Al} die Zahl der Al Capone zur Verfügung stehenden Steine.

Anschließend wird der Wert der Abweichung ermittelt, indem zu jedem der Einsätze der am nächsten liegende Stein der derzeitigen Aufstellung gesucht wird und der Betrag der Differenz der Einsätze dieser Nummern an die Gesamtabweichung addiert wird. Dieser Wert wird der Stellung zugeordnet. Dann wird diese Stellung so oft mutiert, wie eine Variable dies definiert (in der Implementierung liegt dieser Wert bei 200): Für jeden der Steine ermittelt ein Zufallsgenerator eine Änderung der Position, die zwischen -3 und 3 liegt. Sollte die derzeitige Position des Steins addiert mit der Änderung null nicht unterschreiten und die Zahl der Einsätze nicht erreichen, bzw. überschreiten, ist dies die neue Position des Steines. Die die Abweichung ermittelnde Routine wird nun auf diese neue Anordnung angesetzt, sodass ihr ein neuer Wert der Abweichung zugeordnet wird. Aus dieser neuen Generation werden die besten Anordnungen gespeichert, damit sie in der nächsten Generation weitermutiert werden. Die beste Anordnung der Vorgeneration wird dabei immer auch einmal unverändert übergeben. In dieser Implementierung werden die besten 50 Anordnungen weiterverfolgt. Die Anordnungen mit höheren Abweichungen werden vom Speicher gelöscht. Die jeweils beste Anordnung wird als bestes Produkt der Generation ausgegeben. Nach

zehn Generationen wird das Programm beendet. Im Folgenden ist der Algorithmus in seinen Grundzügen graphisch dargestellt:



2 Umsetzung

Zuerst sollte eine für das Verständnis unerlässliche Begrifflichkeit geklärt werden: Wird in diesem Teil oder in den Kommentaren des Quellcodes von »Position« gesprochen, meint dies die Position eines Einsatzes im Array `einsaetze`. Das ist deswegen sehr nützlich, weil Al Capone immer nur auf Einsätze legt, nie in Zwischenräume, wie im Abschnitt »Grundidee« erklärt wurde. Über diesen Array ist jeder Position der Wert zwischen eins und tausend zugeordnet, auf den der Spieler gesetzt hat. Er wird konsequent als »Wert« bezeichnet.

Für das Speichern einer Anordnung ist die Struktur `Loesung` zuständig. Sie enthält den Array `alssteine`, der die Positionen der Einsätze speichert, auf denen einer von Als Steinen liegt. Darüber enthält die Struktur einen Wert für die Abweichung, der direkt zur notwendigen Auszahlung umgerechnet werden kann. Für diese Struktur ist ein Konstruktor implementiert, eine Methode, die die Startanordnung festlegt, sowie die Methode `equals`, die prüft, ob zwei Anordnungen gleich sind. Im späteren Verlauf dieses Textes wird auf die Methoden `gleicher`, `voriger`, `folgender` sowie `mutate` näher eingegangen.

Die Struktur `Speicher` enthält einen Array, in dem die Werte der Einsätze gespeichert sind, einen unsignierten 16-Bit Integer, in dem die Zahl der Einsätze gespeichert ist, sowie einen Array `speicherloesungen`, der in der Implementierung eine Länge 50 besitzt, und der Instanzen des Typs `Loesung` speichert. In diesem Array werden immer die besten 50 Anordnungen einer Generation gespeichert. Für diese Struktur ist ebenfalls ein Konstruktor implementiert, sowie die Methoden `clonespeicher()`, die einen Klon ihrer eigenen Instanz zurückgibt, `datei_auslesen()`, die die Textdatei, dessen Name als erstes Argument beim Programmstart übergeben wurde, ausliest und in den Array `einsaetze()` schreibt, `sorteinsaetze()`, die nach einem simplen Bubblesort-Verfahren die Werte der Einsätze in ihrem Array sortiert, und `printbest()`, die die beste Anordnung einer Generation auf die Kommandozeile ausdruckt. Über die anderen Methoden wird im Anschluss zu sprechen sein.

Die Funktion `main()` beginnt damit, dass ein Array der Länge zwei intialisiert wird, der zwei Instanzen des Typs `Speicher` enthält. Im folgenden Ablauf wird die Datei ausgelesen und die Werte in die erste der beiden Instanzen geschrieben. Diese werden sortiert und die Instanz wird geklont und an den zweiten Platz des Arrays eingefügt. Als beste bekannte Anordnung wird die durch `startloesung()` definierte eingeführt. Im Anschluss beginnt der eigentliche Algorithmus: Ziel ist es hier, abwechselnd die jeweils besten Anordnungen der einen Generation zu mutieren und die besten Ergebnisse in die jeweils andere Instanz des Typs `Speicher` zu übertragen. Dazu wird bei der Instanz, dessen Anordnungen mutiert werden

sollen, die Methode `new_generation()` aufgerufen und als Ziel die Referenz auf die jeweils andere Instanz übergeben. Die Methode überträgt als erstes einmal das beste Produkt der Vorgeneration unverändert über die Methode `insert()`, die weiter unten erläutert wird. Für jede Anordnung, die in diesem Array gespeichert ist, wird 500 mal der Methode `mutate()` ausgeführt, die zufällig jede Position der Steine um bis zu drei Positionen erhöht oder reduziert und die neue Anordnung zurückgibt. Mit der Methode `gesamtabweichung()` wird dann die Abweichung, also der zu zahlende Betrag, bestimmt. Dann wird mit dieser Anordnung als Übergabeparameter die Methode `insert()` aufgerufen. Sie fügt die neue Anordnung nur ein, wenn ihre Abweichung kleiner als die schlechteste gespeicherte Anordnung der Vorgeneration ist. Eine Zählschleife zählt so lange durch die Anordnungen, bis sie bei einer Anordnung ist, die eine höhere Abweichung als die der neuen Anordnung aufweist. Die Methode `nachruecken` verschiebt ab hier alle Anordnungen, sodass die schlechteste bisher gespeicherte gelöscht wird. Dadurch wird an der Stelle Platz geschaffen, an der die neue Anordnung einsortiert wird.

Die Methode `gesamtabweichung()` arbeitet folgendermaßen: Eine Schleife zählt durch den Array `einsaetze` und prüft als erstes mit der Methode `gleicher` der Struktur `Loesung`, ob dieser Einsatz genau von einem der Steine bedeckt wird, also keine Auszahlung fällig wird. Sollte das nicht der Fall sein, wird unter der Berücksichtigung gewisser Spezialfälle, auf die in den Kommentaren des Quellcodes Bezug genommen wird, mithilfe der Methoden `voriger()` und `folgender()` der Stein ermittelt, der diesem Einsatz am nächsten liegt, sodass der Betrag der Differenz der Werte an die Gesamtabweichung addiert werden kann.

Am Ende einer jeden neuen Generation wird durch die Methode `printbest()` immer die beste erreichte Anordnung ausgedruckt.

Zur Verwendung sollte hier angemerkt werden, dass bei der Benutzung des Programmes immer als erstes Argument die Datei mit den Einsätzen übergeben werden muss. Es zum Beispiel folgendermaßen über die Kommandozeile aufgerufen werden: `Voll_daneben_Jan_Groeneveld.exe beispiel1.txt`

3 Beispiele

Im folgenden werden die Beispiele der BWINF-Seite angefügt. Dabei ist zu beachten, dass die Ergebnisse bei jedem Durchlauf des Programmes variieren, da das Mutieren mithilfe von Zufallszahlen erfolgt. Daher dienen diese Ausdrucke nur als Exempla. Dies ist das erste Beispiel:

```
Datei beispiel1.txt wird ausgelesen.
Ich bin jetzt bei Generation 0:
85, 185, 275, 365, 445, 540, 635, 725, 815, 915,
'-> notwendige Auszahlung: 5200
Ich bin jetzt bei Generation 1:
75, 175, 275, 355, 445, 540, 635, 730, 815, 925,
'-> notwendige Auszahlung: 5090
Ich bin jetzt bei Generation 2:
65, 175, 270, 365, 455, 545, 630, 725, 820, 935,
'-> notwendige Auszahlung: 5015
Ich bin jetzt bei Generation 3:
55, 165, 260, 355, 450, 540, 630, 735, 840, 940,
'-> notwendige Auszahlung: 4970
Ich bin jetzt bei Generation 4:
55, 155, 255, 350, 445, 540, 645, 740, 835, 940,
'-> notwendige Auszahlung: 4955
Ich bin jetzt bei Generation 5:
55, 155, 250, 350, 445, 550, 650, 745, 840, 945,
'-> notwendige Auszahlung: 4950
Ich bin jetzt bei Generation 6:
55, 155, 250, 350, 445, 550, 650, 745, 840, 945,
'-> notwendige Auszahlung: 4950
Ich bin jetzt bei Generation 7:
55, 155, 250, 350, 445, 550, 650, 745, 840, 945,
'-> notwendige Auszahlung: 4950
Ich bin jetzt bei Generation 8:
55, 155, 250, 350, 445, 550, 650, 745, 840, 945,
'-> notwendige Auszahlung: 4950
Ich bin jetzt bei Generation 9:
55, 155, 250, 350, 445, 550, 650, 745, 840, 945,
'-> notwendige Auszahlung: 4950

Datei beispiel2.txt wird ausgelesen.
Ich bin jetzt bei Generation 0:
59, 167, 281, 368, 413, 526, 584, 736, 802, 919,
'-> notwendige Auszahlung: 2184
Ich bin jetzt bei Generation 1:
60, 172, 313, 368, 421, 537, 582, 736, 808, 925,
'-> notwendige Auszahlung: 2064
Ich bin jetzt bei Generation 2:
60, 170, 315, 370, 413, 530, 649, 763, 857, 929,
'-> notwendige Auszahlung: 2015
Ich bin jetzt bei Generation 3:
59, 170, 313, 368, 421, 540, 649, 777, 862, 925,
'-> notwendige Auszahlung: 1935
Ich bin jetzt bei Generation 4:
59, 170, 315, 368, 421, 539, 651, 782, 862, 926,
'-> notwendige Auszahlung: 1927
Ich bin jetzt bei Generation 5:
60, 172, 315, 368, 421, 539, 676, 777, 862, 929,
'-> notwendige Auszahlung: 1925
Ich bin jetzt bei Generation 6:
60, 172, 315, 368, 421, 539, 676, 777, 862, 929,
```

```
'-> notwendige Auszahlung: 1925
Ich bin jetzt bei Generation 7:
60, 172, 315, 368, 421, 539, 676, 777, 862, 929,
'-> notwendige Auszahlung: 1925
Ich bin jetzt bei Generation 8:
59, 172, 315, 368, 421, 539, 676, 782, 862, 929,
'-> notwendige Auszahlung: 1924
Ich bin jetzt bei Generation 9:
59, 172, 315, 368, 421, 539, 676, 782, 862, 929,
'-> notwendige Auszahlung: 1924

Datei beispiel3.txt wird ausgelesen.
Ich bin jetzt bei Generation 0:
120, 240, 340, 400, 520, 580, 640, 720, 780, 880,
'-> notwendige Auszahlung: 2420
Ich bin jetzt bei Generation 1:
120, 240, 340, 420, 520, 580, 640, 720, 800, 900,
'-> notwendige Auszahlung: 2280
Ich bin jetzt bei Generation 2:
100, 240, 340, 440, 520, 580, 660, 720, 820, 920,
'-> notwendige Auszahlung: 2180
Ich bin jetzt bei Generation 3:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 4:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 5:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 6:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 7:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 8:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
Ich bin jetzt bei Generation 9:
100, 240, 340, 440, 520, 580, 680, 720, 860, 960,
'-> notwendige Auszahlung: 2160
```

4 Quellcode

Die Funktion `main()`:

```
fn main() {
    let mut myspeicher: [Speicher; 2] = [Speicher::get_speicher(); 2]; // In einem Array werden zwei Instanzen vom Typ "Speicher" erstellt.
    myspeicher[0].datei_auslesen(); // Die Werte werden ausgelesen
    let numeinsatz = myspeicher[0].numeinsatz;
    myspeicher[0].sorteinsaetze(); // Die Werte werden sortiert
    myspeicher[1] = myspeicher[0].clonespeicher(); // Der Speicher wird geklont und im Array ein zweites Mal gespeichert
    let mut zwischen = myspeicher[0].beste_loesungen[0];
    zwischen.startloesung(numeinsatz); // Die Ausgangsanordnung wird erstellt
    myspeicher[0].gesamtabweichung(&mut zwischen); // Die Abweichung dieser Anordnung wird berechnet
    myspeicher[0].beste_loesungen[0] = zwischen; // Diese Anordnung wird als beste bisher bekannte Anordnung gespeichert
    let maxgen = 10;
    for i in 0..maxgen {
        println!("Ich bin jetzt bei Generation {}:", i); // Abwechselnd werden aus einem Speicher die Anordnungen mutiert und in dem anderen gespeichert
        let start = myspeicher[i % 2];
        start.new_generation(&mut myspeicher[1 - (i % 2)]);
        myspeicher[1 - (i % 2)].printbest(); // Die jeweils beste Anordnung der neuen Generation wird ausgedruckt
    }
}
```

```
#[derive(Clone, Copy)]
struct Loesung {
    // Diese Struktur stellt eine Anordnung dar. Sie enthält die Nummern der Einsätze, auf denen Al setzt, und die Abweichung.
    alssteine: [u16; MAXSTEINE],
    abweichung: u16,
}

impl Loesung {
    fn get_loesung() -> Loesung {
        // Der Konstruktor für die Instanzen des Typs "Loesung"
        Loesung {
            alssteine: [0; MAXSTEINE],
            abweichung: 0,
        }
    }

    fn startloesung(&mut self, numeinsatz: u16) {
        // Gemäß der in der Erklärung erklärten Formel wird hier die Startaufstellung definiert.
        for i in 0..MAXSTEINE {
            self.alssteine[i] = ((i + 1) * (numeinsatz as usize) / (MAXSTEINE + 1)) as u16;
        }
    }

    fn gleicher(&self, value: usize) -> bool {
        // Diese Funktion prüft, ob ein eingegebener Wert durch einen Stein dieser Anordnung direkt abgedeckt wird.
        let mut rueckgabe = false;
        for i in 0..MAXSTEINE {
            if self.alssteine[i] == value as u16 {
                rueckgabe = true;
                break;
            }
        }
        rueckgabe
    }

    fn voriger(&self, value: usize) -> usize {
        // Diese Funktion gibt den nächst kleineren Nachbarn zurück.
```

```
        let mut voriger = KONST;
        for i in 0..MAXSTEINE {
            if self.alssteine[i] > value as u16 {
                if i == 0 {
                    voriger = RETURN1 // Wenn bereits der von Als Steinen nach dem gesuchten Wert liegt, wird der Schlüssel 6789 in die Variable geschrieben. Dieser Fall wird dann gesondert behandelt.
                } else {
                    voriger = self.alssteine[i - 1]; // Andernfalls wird die Nummer des Vorgängers im Array in die Variable geschrieben.
                }
                break;
            } // Wenn die Schleife durchgelaufen ist, und noch immer 6789 in der Variable steht, ist der Vorgänger der letzte von Als Steinen. Dieser Fall wird dann gesondert behandelt.
        }
        if voriger == KONST {
            voriger = self.alssteine[MAXSTEINE - 1];
        }
        voriger as usize
    }

    fn folgender(&self, value: usize) -> usize {
        // Diese Funktion gibt den nächst größeren Nachbarn zurück.
        let mut nachfolgender = RETURN1;
        for i in 0..MAXSTEINE {
            if self.alssteine[i] > value as u16 {
                nachfolgender = self.alssteine[i]; // In diesem Fall wird die Nummer des Nachfolgers im Array in die Variable geschrieben
                break;
            }
        }
        nachfolgender as usize
    }

    fn mutate(&self, numeinsatz: u16) -> Loesung {
        // Diese Funktion gibt eine mutierte Anordnung zurück.
        let mut child = Loesung::get_loesung(); // Hier wird eine Instanz vom Typ "Loesung" erzeugt.
        let mut rng = rand::thread_rng(); // Hier wird eine Instanz des Zufallsgenerators erzeugt.
        for i in 0..MAXSTEINE {
            let randomnum = rng.gen_range(0, 30); // Hier wird eine Zufallszahl zwischen 0 und 30 erzeugt. Ein Zehntel dieser Zahl wird dann addiert oder subtrahiert
            let positionchange: i16 = {
                if randomnum % 2 == 0 {
                    randomnum / 10 // Gerade Zahlen führen zu einer Addition
                } else {
                    -randomnum / 10 // Ungerade Zahlen führen zu einer Subtraktion
                }
            };
            let position = positionchange + self.alssteine[i] as i16;
            if position >= 0 && position < numeinsatz as i16 {
                child.alssteine[i] = position as u16; // Wenn die Positionsänderung einen gültigen Wert aufweist, bekommt die Anordnung der nächsten Generation diesen Wert
            } else {
                child.alssteine[i] = self.alssteine[i]; // Andernfalls wird die Position der vorigen Anordnung übernommen.
            }
        }
        child
    }
}
```

```
#[derive(Clone, Copy)]
struct Speicher {
    // In dieser Struktur werden die Einsätze, die besten Lösungen und die Anzahl der Einsätze gespeichert.
    einsaetze: [u16; MAXEINSATZ],
    beste_loesungen: [Loesung; SPEICHERLOESUNGEN],
```

```
    numeinsatz: u16,
}

impl Speicher {
    fn get_speicher() -> Speicher { }

    fn clonespeicher(&self) -> Speicher { }

    fn datei_auslesen(&mut self) { }

    fn sorteinsaetze(&mut self) { }

    fn printbest(&self) { }

    fn gesamtabweichung(&self, versuch: &mut Loesung) {
        // Methode, die die Gesamtabweichung einer Anordnung bestimmt
        let mut gesamtabweichung = 0;
        for i in 0..(self.numeinsatz as usize) {
            // Diese Schleife zählt durch den Array "einsaetze"
            if versuch.gleicher(i) {
                continue; // Wenn die Position mit einem von Als Steinen besetzt ist, ist keine Abweichung zu addieren
            } else {
                let voriger = versuch.voriger(i); // Die Position des vorher liegenden von Als Steinen wird besimmt
                let folgender = versuch.folgender(i); // Die Position des nachfolgenden wird bestimmt
                if voriger == RETURN1 as usize {
                    gesamtabweichung += self.einsaetze[folgender] - self.einsaetze[i]; // Wenn es keinen von Als Steinen gibt, dessen Wert kleiner ist, wird die Differenz aus dem Wert des folgenden Steins und dem eigenen Einsatz addiert
                } else if folgender == RETURN1 as usize {
                    gesamtabweichung += self.einsaetze[i] - self.einsaetze[voriger]; // Wenn es keinen von Als Steinen gibt, dessen Wert größer ist, wird die Differenz aus dem eigenen Einsatz und dem Wert des vorigen Steins addiert
                } else {
                    let zum_vorigen = self.einsaetze[i] - self.einsaetze[voriger]; // Wenn es Vorgänger und Nachfolger gibt, wird der kleinere Abstand addiert
                    let zum_folgenden = self.einsaetze[folgender] - self.einsaetze[i];
                    if zum_vorigen < zum_folgenden {
                        gesamtabweichung += zum_vorigen;
                    } else {
                        gesamtabweichung += zum_folgenden;
                    }
                }
            }
        }
        versuch.abweichung = gesamtabweichung; // Die Abweichung wird in die Variable der Instanz geschrieben
    }

    fn nachruecken(&mut self, position: usize) {
        // Diese Methode lässt alle Anordnungen im Speicher ab dem Wert "position" eine Position nach hinten wandern
        let mut zaehlvariable: usize = SPEICHERLOESUNGEN - 1;
        loop {
            if zaehlvariable == position {
                break;
            }
            self.beste_loesungen[zaehlvariable] = self.beste_loesungen[zaehlvariable - 1];
            zaehlvariable -= 1;
        }
    }

    fn insert(&mut self, neueloesung: Loesung) {
        // Diese Methode fügt eine neue Anordnung in den Speicher ein
        if neueloesung.abweichung < self.beste_loesungen[SPEICHERLOESUNGEN - 1].abweichung
            || self.beste_loesungen[SPEICHERLOESUNGEN - 1].abweichung == 0
        { // Eine neue Anordnung wird nur eingefügt, wenn ihre Abweichung kleiner als die schlechteste gespeicherte Anordnung der Vorgeneration ist
            for i in 0..SPEICHERLOESUNGEN {
```

```
                if self.beste_loesungen[i].equals(&neueloesung) {
67                  break; // Wenn eine gleiche Anordnung bereits existiert, wird sofort
    abgebrochen
                }
69              if neueloesung.abweichung < self.beste_loesungen[i].abweichung
                    || self.beste_loesungen[i].abweichung == 0
71              {
                    self.nachruecken(i); // Wenn die Abweichung dieser neuen Anordnung
    eine geringere Abweichung aufweist als die an der Stelle i gespeicherte, werden die
    Anordnungen ab hier weitergerückt, und die neue eingefügt.
73                  self.beste_loesungen[i] = neueloesung;
                    break;
75              }
            }
77      }
    }
79
    fn new_generation(&self, ziel: &mut Speicher) {
81      // Diese Methode erstellt die neue Generation
        ziel.insert(self.beste_loesungen[0]); // Die beste Lösung der vorigen Generation
    wird auf jeden Fall unverändert übertragen.
83      for i in 0..SPEICHERLOESUNGEN {
            if self.beste_loesungen[i].abweichung == 0 {
85              break; // Dieser Fall ist nur am Anfang erfüllt, wenn der Speicher noch
    leer ist.
            }
87          for _j in 0..CHILDREN {
                let mut help = self.beste_loesungen[i].mutate(self.numeinsatz); // So oft
     wie gefordert, wird eine bestimmte Anordnung mutiert
89              self.gesamtabweichung(&mut help); // Für das Ergebnis dieser Mutation
    wird die Abweichung berechnet
                ziel.insert(help); // Die Mutation wird in den Speicher eingefügt
91          }
        }
93  }
}
```