Lösung Aufgabe 1 “Blumenbeet”

Lösungsidee

Wir verwenden eine **optimierte Brute-Force** um alle möglichen Blumenbeete zu generieren, in denen genau j verschiedene Blumenfarben vorkommen. Wir generieren die Beete, in dem wir der Reihe nach Blumen platzieren, und jeweils die dadurch neu erworbenen Punkte hinzuaddieren. Wenn wir feststellen, dass wir das Maximum an verwendbaren Farben (=j) schon ausgeschöpft haben, dann verwenden wir nur noch Farben wieder; Wenn wir feststellen, dass wir weit unter dem Maximum liegen, und noch genau so viele Plätze frei sind, wie uns Farben fehlen, dann muss ab dann jedes Mal eine bisher noch nicht verwendete Farbe eingesetzt werden. Dies sind maximal 7^9=40.353.607 (rund $4*10^9$) Kombinationen - recht wenig.

Umsetzung

Bibliotheken

- `java.util.ArrayList`: Erweiterbarer Zwischenspeicher fürs Generieren einer Liste der Nachbarn eines Punktes (deren Länge erst nach dem Generieren bekannt ist)
- `java.util.HashMap`: Speichert konstante Zuordnung Farbe -> Nummer
- `java.io.File`, `java.io.FileReader`, `java.io.BufferedReader`: Nötig zum Lesen der Aufgabenstellungs-Datei
- `java.util.Scanner`: Nützlich zum eventuellen Einlesen einer Nutzereingabe (wenn keine Argumente gegeben sind)

Klassen

`Beet`

Repräsentiert ein Beet.

Das Blumenbeet wird als 3x3-Array dargestellt, gewissermaßen “auf die Seite gekippt”:

```
  1
 2 3
4 5 6
 7 8
  9
```

wird zu

```
1 3 6
2 5 8
4 7 9
```

Die Nachbarschaften sehen dann wie folgt aus:

```
1-3-6
|/|/|
```

```
2-5-8
|/|/|
4-7-9
```

Dieses 3x3-Array wird als eindimensionales 9-er Array dargestellt.

Es werden Byte-Arrays benutzt, da Bytes schon ausreichen, um 7 Zahlen abzubilden. Die verwendeten Farben werden als Boolean-Array gespeichert - pro Farbe ein Platz, ob die Farbe verwendet wurde. Außerdem speichert das Beet noch, wie viele verschiedene Farben es enthält. Kommt eine Blume hinzu werden Array und Farbanzahl geupdated. Weiter wird eine globale Variable gespeichert, welche Nachbarpositionen eine Position einer Blume auf dem Blumenbeet jeweils hat.

`Blumenbeet`

Die Main-Klasse des Programms. Enthält die Main-Methode (wo die Argumente eingelesen werden) und eine Methode um die Aufgabenstellung zu laden. Enthält & führt auch die rekursiv implementierte optimierte Brute-Force aus. Diese enthält eine for-Schleife, die alle möglichen Farben durchgeht, jeweils das Beet kopiert & die Farbe setzt und dann wieder die Brute-Force mit dem neuen Beet ausführt. Wird festgestellt, dass das Maximum an zu setzenden Farben erreicht wurde, so werden nur noch Farben "wiederverwertet" - also nur noch Farben, die schon als "verwendet" gespeichert sind (im Boolean-Array). Wird aber festgestellt, dass einem noch so viele Farben fehlen wie noch Plätze im Beet frei sind, dann werden fürs weitere Generieren des Beetes nur neue Farben, die vorher noch nicht verwendet wurden, eingesetzt. Die Anzahl der zu verwendenden Farben wird in globalen Variablen gespeichert. Ebenso wie eine konstante Zuordnung Farbe -> Nummer (`HashMap<String, Integer>`).

Verwendung

Lösen einer Aufgabe: `java -jar Blumenbeet.jar <pfad_zur_datei>`

Beispiele

Farben: Wie auf der Materialwebsite angegeben ("`blau 1, gelb 2, gruen 3, orange 4, rosa 5, rot 6 und tuerkis 7`")

`blumen.txt`

```
Punkte: 12
Hochbeet:
  5
 6 1
1 6 7
 2 4
  3
```

`blumen1.txt`

```
Punkte: 36
Hochbeet:
  6
 7 7
6 6 6
 7 7
  6
```

blumen2.txt

```
Punkte: 30
Hochbeet:
  7
 6 6
1 7 7
 6 6
  7
```

blumen3.txt

```
Punkte: 12
Hochbeet:
  5
 4 7
1 6 6
 2 7
  3
```

blumen4.txt

```
Punkte: 12
Hochbeet:
  5
 4 7
1 6 6
 2 7
  3
```

blumen5.txt

```
Punkte: 12
Hochbeet:
  5
 4 7
1 6 6
 2 7
  3
```

Quellcode

Beet.java

```java
package appguru;

import java.util.ArrayList;

/**
 *
 * @author lars
 */
public class Beet {
    public static final byte[][] NACHBARN_PRO_PUNKT;
    public static Beet BESTES_BEET = new Beet();

    static {
        BESTES_BEET.punkte = Byte.MIN_VALUE;
        NACHBARN_PRO_PUNKT = new byte[9][]; // Nachbarn eines Punktes mit gegebenem Index
        for (byte x = 0; x < 3; x++) {
            for (byte y = 0; y < 3; y++) {
                // alle Punkte (x|y) durchgehen
                byte p1 = posToIndex(x, y); // Index berechnen
                ArrayList<Byte> nachbarn = new ArrayList(2); // Nachbarn (min. 2)
                for (byte x2 = 0; x2 < 3; x2++) {
                    for (byte y2 = 0; y2 < 3; y2++) {
                        // alle Punkte (x2|y2) durchgehen, Entfernungen berechnen
                        byte xd = (byte) (x - x2);
                        byte yd = (byte) (y - y2);
                        // Diagonaler Nachbar genau dann wenn x-Entfernung und y-Entfernung beide
                            1 oder -1 sind
                        boolean diagonal_neighbor = (xd == 1 && yd == 1) || (xd == -1 && yd ==
                            -1);
                        // Horizontaler Nachbar genau dann wenn eine von beiden Entfernungen 0
                            und die Andere 1 oder -1 ist
                        boolean other_neighbor = (xd == 0 && (yd == 1 || yd == -1)) || ((xd == 1
                            || xd == -1) && yd == 0);
                        // Wenn (x2|y2) Nachbar von (x|y) ist
                        if (diagonal_neighbor || other_neighbor) {
                            nachbarn.add(posToIndex(x2, y2)); // füge zu Nachbarn hinzu
                        }
                    }
                }
                // Konvertiere zu Array...
                byte[] nachbar_array = new byte[nachbarn.size()];
                for (byte i = 0; i < nachbarn.size(); i++) {
                    nachbar_array[i] = nachbarn.get(i);
                }
```

```
                NACHBARN_PRO_PUNKT[p1] = nachbar_array; // ...und speichere als Nachbarn von (x|y) ab
            }
        }
    }

    // Wandelt eine (x|y)-Koordinate in einen Index um
    public static byte posToIndex(byte x, byte y) {
        return (byte) (x * 3 + y);
    }

    // pti = posToIndex, abkürzende Schreibweise, casted außerdem (nützlich für toString)
    private static byte pti(int a, int b) {
        return posToIndex((byte) a, (byte) b);
    }

    public byte punkte = 0;
    public int anzahl_verwendet = 0;
    public byte[] beet = new byte[9];
    public boolean[] verwendet = new boolean[7];

    // Füge Blume an Stelle index hinzu
    public void fuegeBlumeHinzu(int index, byte blume) {
        beet[index] = blume; // Setze Blume
        blume -= 1;
        // Wurde diese Sorte Blume noch nicht verwendet?
        if (!verwendet[blume]) {
            // Speichere als verwendet
            verwendet[blume] = true;
            anzahl_verwendet++;
        }
    }

    public Beet() {}

    // Kopiert ein Beet
    public Beet(Beet beet) {
        this.anzahl_verwendet = beet.anzahl_verwendet;
        this.punkte = beet.punkte;
        this.beet = new byte[9];
        System.arraycopy(beet.beet, 0, this.beet, 0, beet.beet.length); // arraycopy um nicht Referenz zu übernehmen
        this.verwendet = new boolean[7];
        System.arraycopy(beet.verwendet, 0, this.verwendet, 0, beet.verwendet.length); // arraycopy um nicht Referenz zu übernehmen
    }

    @Override
```

```
    public String toString() {
        // Formattiert ein Beet
        return String.format("Punkte: "+punkte+"\nHochbeet:\n  %d  \n %d %d \n%d %d %d\n %d %d \n
              %d  ",
                beet[pti(2, 0)],
                beet[pti(1, 0)], beet[pti(2, 1)],
                beet[pti(0, 0)], beet[pti(1, 1)], beet[pti(2, 2)],
                beet[pti(0, 1)], beet[pti(1, 2)],
                beet[pti(0, 2)]
        );
    }
}
```

Blumenbeet.java

```
package appguru;

import static appguru.Beet.BESTES_BEET;
import static appguru.Beet.NACHBARN_PRO_PUNKT;
import java.util.Scanner;
import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.util.HashMap;

/**
 *
 * @author lars
 */
public class Blumenbeet {

    public static byte[][] PUNKTE = new byte[8][8];

    // Setzt die Punkte p für nebenanderliegende Farben a und p
    public static void setzePunkte(byte a, byte b, byte p) {
        PUNKTE[a][b] = PUNKTE[b][a] = p;
    }

    // Gibt die Punkte für Farben a und b nebeneinander zurück
    public static int punkteFuer(byte farbe_a, byte farbe_b) {
        return PUNKTE[farbe_a][farbe_b];
    }

    public static final HashMap<String, Byte> FARBEN = new HashMap();
    public static byte ANZAHL_FARBEN = 0; // Anzahl Farben, die das Beet haben soll

    static {
        // Sieben Farben: blau, gelb, grün, orange, rosa, rot und türkis - werden Zahlen von 1-7
```

```
            zugeordnet
        FARBEN.put("blau", (byte) 1);
        FARBEN.put("gelb", (byte) 2);
        FARBEN.put("gruen", (byte) 3);
        FARBEN.put("orange", (byte) 4);
        FARBEN.put("rosa", (byte) 5);
        FARBEN.put("rot", (byte) 6);
        FARBEN.put("tuerkis", (byte) 7);
    }

    // Liest die Aufgabenstellung
    public static void ladeAufgabe(File datei) throws Exception {
        BufferedReader leser = new BufferedReader(new FileReader(datei));
        ANZAHL_FARBEN = Byte.parseByte(leser.readLine());
        int i=Integer.parseInt(leser.readLine());
        String zeile;
        int n=0;
        while (n < 1 && (zeile = leser.readLine()) != null) {
            String[] teile = zeile.split(" ");
            setzePunkte(FARBEN.get(teile[0]), FARBEN.get(teile[1]), Byte.parseByte(teile[2]));
            n++;
        }
    }

    // Rekursive Brute-Force, beet = letztes Beet, n = wieviele Blumen schon gesetzt worden sind
    public static void bruteForce(Beet beet, int n) {
        // Alle Blumen gesetzt, werte aus
        if (n == 9) {
            // Besser als das beste bisherige Beet?
            if (beet.punkte > BESTES_BEET.punkte) {
                // Neuen Rekordhalter setzen
                BESTES_BEET = beet;
            }
            return;
        }
        int zu_setzende_blumen = 9 - n;
        // Maximum an verwendbaren Farben erreicht?
        boolean maximum_erreicht = Blumenbeet.ANZAHL_FARBEN == beet.anzahl_verwendet;
        // Zu wenig Farben bisher verwendet, ab jetzt stets neue
        boolean minimum_erreicht = Blumenbeet.ANZAHL_FARBEN - beet.anzahl_verwendet ==
            zu_setzende_blumen;
        // Gehe alle sieben Möglichkeiten durch, eine Blume hinzuzufügen
        for (byte j = 1; j <= 7; j++) {
            // Wenn wir das Maximum erreicht haben, dürfen wir nur Farben wiederverwerten
            // Wenn wir das Minimum erreicht haben, dürfen wir nur bisher unbenutzte Farben
                verwenden, um zum Schluss noch genug Verschiedene zu haben
            if ((maximum_erreicht && !beet.verwendet[j - 1]) || (minimum_erreicht &&
                beet.verwendet[j - 1])) {
```

```
                continue;
            }
            Beet neu = new Beet(beet); // Beet kopieren
            neu.fuegeBlumeHinzu(n, j); // Füge Blume hinzu
            // Dadurch neu bekommenene Punkte berechnen
            for (byte nachbar : NACHBARN_PRO_PUNKT[n]) {
                neu.punkte += punkteFuer(neu.beet[(byte) n], neu.beet[nachbar]);
            }
            // Alle Möglichkeiten für nächste Blume durchgehen
            bruteForce(neu, n + 1);
        }
    }

    public static void main(String[] args) {
        String pfad_zur_datei;
        if (args.length == 1) {
            // ein Argument gegeben = Pfad zur Datei
            pfad_zur_datei = args[0];
        } else {
            // Frage nach Pfad zur Datei
            System.out.println("Pfad zur Aufgabenstellung: ");
            Scanner eingabe=new Scanner(System.in);
            pfad_zur_datei=eingabe.nextLine();
            eingabe.close();
        }
        try {
            ladeAufgabe(new File(pfad_zur_datei)); // lade Aufgabe
        } catch (Exception e) {
            // Falls die Aufgabe nicht gelesen werden konnte
            System.out.println("Datei nicht lesbar oder falsch formattiert");
            return;
        }
        bruteForce(new Beet(), 0); // Führe Brute-Force aus
        System.out.println(Beet.BESTES_BEET); // Gebe bestes Beet aus
    }

}
```