Aufgabe 1: Blumenbeet

Richard Wohlbold
Team-ID: 00487

30. Oktober 2019

Inhaltsverzeichnis

1 Lösungsidee

Mein Ansatz, um die optimale Anordnung der Blumen zu finden, liegt darin, alle möglichen Anordnungen auszuprobieren und die Anordnung mit der höchsten Punktzahl zu benutzen. Durch die nach oben begrenzte Anzahl an Farben und durch die feste Zahl an Blumenplätzen ist die Anzahl der möglichen Anordnungen für eine gegebene Farbmenge M begrenzt. Die Anordnungen der Blumen sind dahingehend Permutationen mit Wiederholung, dass die Reihenfolge der Blumen relevant ist (die Punktzahl kann sich bei Veränderung der Reihenfolge verändern) und dass eine Farbe mehrmals eingesetzt werden kann. Im Unterschied zu regulären Permutationen muss jedoch jede Farbe einmal eingesetzt werden.

Ich führe den Begriff der *Grundfarbenmenge* ein, die eine Menge von Farben bezeichnet, aus deren Permutation sich mögliche Anordnungen von Farben ergeben. Die Anzahl an Elementen in möglichen Grundfarbenmengen n wird durch den Kunden festgelegt und kann somit als gegeben angenommen werden.

1.1 Bestimmung der Grundfarbenmengen

Der erste Schritt meines Verfahrens ist das Bestimmen aller möglichen Grundfarbenmengen. Dabei gehe ich von den Regeln aus, indem ich eine Menge aller in den Regeln erwähnter Farben R bilde. Da die Erfüllung einer Regel stets eine positive Anzahl von Bonuspunkten mit sich bringt, sollten in einer Grundfarbenmenge so viele in den Regeln erwähnten Farben vorkommen wie möglich. Dies kann man sich folgendermaßen vorstellen: Wird eine Farbe von Blumen, die in den Regeln erwähnt wird, durch eine andere Farbe erstetzt, die nicht vorkommt, können keinesfalls Punkte zum Ergebnis hinzukommen, entweder bleibt das Ergebnis gleich oder nimmt ab.

Nun gibt es drei Möglichkeiten:

1. $\text{len}(R) = n$. In diesem Fall ist R die einzig sinnvolle Grundfarbenmenge.
2. $\text{len}(R) < n$. In diesem Fall werden R so lange Farben hinzugefügt, die nicht in den Regeln erwähnt werden, bis R n Elemente hat. Da keine der nachträglich hinzufügbaren Farben Bonuspunkte erwirken kann, ist R die einzige Grundfarbenmenge.
3. $\text{len}(R) > n$. In diesem Fall bildet jede Kombination ohne Wiederholung aus n Elementen aus den Farben in R eine Grundfarbenmenge. Somit gibt es $\binom{\text{len}(R)}{n}$ Grundfarbenmengen, wobei $0 \leq \text{len}(R) \leq 7$ und $1 \leq n \leq 7$. Somit gibt es maximal 35 Grundfarbenmengen ($\text{len}(R) = 7$, $n = 4$ oder $n = 3$).

1.2 Bestimmung aller Permutationen

Ausgehend von einer Grundfarbenmenge M und einer Anzahl an verschiedenen Farben n können alle Permutationen folgendermaßen bestimmt werden:

1. Man nimmt das kartesische Produkt von M mit sich selbst 9 Mal.
2. Alle Produkte, die nicht mindestens n verschiedene Farben enthalten, werden entfernt.

Dadurch werden alle Permutationen mit Wiederholung, bei denen jede Farbe mindestens einmal vorkommt gefunden. Diese können dann im Folgenden mithilfe der Regeln bewertet werden, sodass die beste Permutation gefunden wird.

Dieser Prozess wird für jede mögliche Grundfarbenmenge wiederholt. Die beste Permutation wird ausgewählt und stellt somit das Ergebnis des Verfahrens dar.

2 Umsetzung

Ich habe das oben beschriebene Verfahren in Python 3 mithilfe des Moduls `itertools` umgesetzt.

Als erstes liest das Skript die Eingabedatei ein und speichert die Regeln in einem `dict` und die Anzahl der Farben in einer Variable. Dabei ist der Schlüssel ein Tupel aus zwei Farbnamen und der Wert die Anzahl an Bonuspunkten, die man bei Erfüllung der Regel erhält. Um die Bewertungslogik einfacher und schneller zu gestalten, speichere ich jede Regel doppelt, nämlich mit Wert (A, B) und (B, A), seien A und B zwei Farben aus einer Regel.

Ich definiere eine Funktion namens `grundfarbenmengen_bestimmen`, die mithilfe der ausgelesenen Regeln und der Anzahl der Farben alle möglichen Grundfarbenmengen bestimmt und zurückgibt. Wie oben beschrieben bestimmt sie die Menge der in den Regeln genannten Farben und fügt solange nicht erscheinende Farben hinzu, bis die geforderte Anzahl an Farben enthalten oder überschritten ist. Als letztes gibt sie alle Kombinationen der Farben der Länge `anzahl_farben` zurück.

Außerdem definiere ich eine Funktion `aufstellung_bewerten`, die mithilfe der Regeln den Punktewert einer Aufstellung berechnet. Dabei repräsentiere ich eine Aufstellung als einen Tupel aus 9 Farben. Die Reihenfolge der Farben im Tupel entspricht der Reihenfolge der Farben in einer Aufstellung, wenn man die Farben nach rechts und nach unten abliest. Die Funktion prüft nun für jedes benachbarte Farbenpaar, ob eine Regel für diese Kombination existiert und addiert gegebenenfalls ihren Punktewert zu einer Summe hinzu und gibt diese zurück.

Als letztes definiere ich eine Funktion `ideale_anordnung`, die, ausgehend von einer Grundfarbenmenge, nach dem oben beschriebenen Prinzip, alle Permutationen der Länge 9 der Grundfarben bestimmt, in denen alle Farben mindestens einmal enthalten sind. Diese Permutationen werden jeweils mit `aufstellung_bewerten` direkt bewertet und es wird die beste gespeichert. Diese wird mit ihrer Punktezahl zurückgegeben.

n (Anzahl der geforderten Farben)	Laufzeit
1	0.048s
2	0.051s
3	0.138s
4	0.823s
5	3.795s
6	12.065s
7	31.271s

Abbildung 1: Laufzeit nach Anzahl der geforderten Farben ohne Regeln

$\text{len}(R)$	n	$\binom{\text{len}(R)}{n}$	Ungefähre Laufzeit	Reale Laufzeit
7	7	1	$1 * 31.271s = 31.271s$	$26.74s$ (siehe Beispiel 3.2.7)
7	6	7	$7 * 12.065s = 84.455s$	$72.49s$ (siehe Beispiel 3.2.6)
7	5	21	$21 * 3.795s = 79.659s$	$73.49s$ (siehe Beispiel 3.2.5)
7	4	35	$35 * 0.825s = 28.805s$	$23.57s$ (siehe Beispiel 3.2.4)

Abbildung 2: Ungefähre maximale Laufzeiten und reale Laufzeiten des Skripts ausgehend von n und $\text{len}(R)$. Reale Laufzeiten sind aufgrund der in der Realität nur einmal vorkommenden Vorbereitungszeit kürzer.

Das Skript ruft nun für jede durch `grundfarbenmengen_bestimmen` bestimmte Grundfarbenmenge `ideale_anordnung` auf und bestimmt die ideale Anordnung über alle Grundfarbenmengen hinweg.

Diese Anordnung wird schlussendlich mit ihrer Punktezahl ausgegeben.

2.1 Laufzeit

Die Laufzeit der Funktion `ideale_anordnung` hängt von der Anzahl an Farben, die vom Kunden gefordert werden, ab: Je mehr Farben gefordert sind, desto länger braucht das Skript, um zu laufen. Als Illustration habe ich Tests erstellt, die unterschiedliche Anzahlen von Farben fordern und alle keine Regeln erhalten. Somit gibt es in allen Beispielen genau eine Grundfarbenmenge, die `ideale_anordnung` gegeben wird. Die Laufzeiten werden in 1 dargestellt. Die Laufzeiten sind quasi konstant, egal wie viele Regeln gegeben werden.

Wie in Abschnitt 1 beschrieben, gibt es bei $\text{len}(R) > n$ $\binom{\text{len}(R)}{n}$ Grundfarbenmengen. Für jede Grundfarbenmenge wird `ideale_anordnung` einmal aufgerufen. Multipliziert man nun die in Abbildung 1 gegebenen Laufzeiten mit $\binom{\text{len}(R)}{n}$, erhält man ungefähre Laufzeiten des Skripts mit Beispielen mit diesen Parametern. Um die maximale Laufzeit zu ermitteln, sollte $\text{len}(R)$ maximal sein, damit der Binomialkoeffizient maximal wird und damit n erhöht werden kann. Dadurch ergeben sich ungefähre maximale Laufzeiten, die in Abbildung 2 genannt sind.

Somit ist die maximale Laufzeit des Skripts auf meinem Computer (Intel i5-5300U) ca. 1 Minute und 30 Sekunden, falls der Kunde Regeln mit allen 7 Farben angibt, aber nur 5 oder 6 fordert. Ein Praxistest mit einem solchen Beispiel bestätigt diesen Wert ungefähr. Alle anderen Anforderungen ergeben kürzere Laufzeiten (siehe Abschnitt 3)

3 Beispiele

3.1 Beispiele von der `bwinf`-Website

3.1.1 Beispiel 0

Die Eingabedatei:

```
7
2
rot blau 3
rot tuerkis 2
```

../../material/a1–Blumenbeet/beispieldaten/blumen.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen.txt
1. Reihe: orange
2. Reihe: gelb, rosa
3. Reihe: tuerkis, rot, blau
4. Reihe: blau, rot
5. Reihe: gruen
Diese Aufstellung erhält 14 Punkte.

real    0m27.566s
user    0m27.235s
sys     0m0.027s
```

3.1.2 Beispiel 1

Die Eingabedatei:

```
2
2
rot tuerkis 3
gruen rot 1
```

../../material/a1–Blumenbeet/beispieldaten/blumen1.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen1.txt
1. Reihe: rot
2. Reihe: tuerkis, tuerkis
3. Reihe: rot, rot, rot
4. Reihe: tuerkis, tuerkis
5. Reihe: rot
Diese Aufstellung erhält 36 Punkte.

real    0m0.051s
user    0m0.044s
sys     0m0.007s
```

3.1.3 Beispiel 2

Die Eingabedatei:

```
3
2
tuerkis rot 3
rot gruen 1
```

../../material/a1–Blumenbeet/beispieldaten/blumen2.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen2.txt
1. Reihe: gruen
2. Reihe: rot, rot
3. Reihe: tuerkis, tuerkis, tuerkis
4. Reihe: rot, rot
5. Reihe: tuerkis
Diese Aufstellung erhält 32 Punkte.

real    0m0.152s
user    0m0.128s
sys     0m0.023s
```

3.1.4 Beispiel 3

Die Eingabedatei:

```
7
2
rot tuerkis 3
gruen rot 1
```

../../material/a1–Blumenbeet/beispieldaten/blumen3.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen3.txt
1. Reihe: gelb
2. Reihe: orange, rot
3. Reihe: blau, tuerkis, tuerkis
4. Reihe: rosa, rot
5. Reihe: gruen
Diese Aufstellung erhält 13 Punkte.

real    0m25.768s
user    0m25.585s
sys     0m0.027s
```

3.1.5 Beispiel 4

Die Eingabedatei:

```
7
6
rot tuerkis 3
gruen rot 1
rot rosa 3
blau rosa 2
gelb orange 1
gruen orange 1
```

../../material/a1–Blumenbeet/beispieldaten/blumen4.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen4.txt
1. Reihe: blau
2. Reihe: rosa, rosa
3. Reihe: rot, rot, gruen
4. Reihe: tuerkis, orange
5. Reihe: gelb
Diese Aufstellung erhält 22 Punkte.

real    0m26.296s
user    0m26.191s
sys     0m0.014s
```

3.1.6 Beispiel 5

Die Eingabedatei:

```
7
7
rot tuerkis 3
rot gruen 1
rot rosa 3
blau rosa 2
gelb orange 1
gruen orange 1
blau tuerkis 2
```

../../material/a1–Blumenbeet/beispieldaten/blumen5.txt

Ausgabe:

```
$ time python main.py ../../material/a1-Blumenbeet/beispieldaten/blumen5.txt
1. Reihe: gruen
2. Reihe: rot, orange
3. Reihe: tuerkis, tuerkis, gelb
4. Reihe: rot, blau
5. Reihe: rosa
Diese Aufstellung erhält 22 Punkte.

real    0m28.637s
user    0m28.225s
sys     0m0.059s
```

3.2 Eigene Beispiele

3.2.1 Keine Eingabedatei

Ausgabe:

```
$ python main.py
Benutzung: main.py <Eingabedatei>
```

3.2.2 Keine Regeln, eine Farbe

Die Eingabedatei:

```
1
0
```

../Implementierung/322.txt

Die Ausgabe:

```
$ time python main.py 322.txt
1. Reihe: blau
2. Reihe: blau, blau
3. Reihe: blau, blau, blau
4. Reihe: blau, blau
5. Reihe: blau
Diese Aufstellung erhält 0 Punkte.

real    0m0.046s
user    0m0.042s
sys     0m0.003s
```

3.2.3 Keine Regeln, sieben Farben

Die Eingabedatei:

```
7
0
```

../Implementierung/323.txt

Die Ausgabe:

```
$ time python main.py 323.txt
1. Reihe: gruen
2. Reihe: gruen, gruen
3. Reihe: gelb, rosa, blau
4. Reihe: rot, tuerkis
5. Reihe: orange
Diese Aufstellung erhält 0 Punkte.

real    0m24.914s
user    0m24.848s
```

```
sys     0m0.007s
```

3.2.4 $n = 4, \mathbf{len}(R) = 7$

Die Eingabedatei:

```
4
4
blau gelb 1
gruen orange 1
rosa rot 1
tuerkis rot 1
```

../Implementierung/324.txt

Die Ausgabe:

```
$ time python main.py 324.txt
1. Reihe: orange
2. Reihe: rot, rot
3. Reihe: tuerkis, tuerkis, tuerkis
4. Reihe: rot, rot
5. Reihe: rosa
Diese Aufstellung erhält 10 Punkte.

real    0m23.750s
user    0m23.707s
sys     0m0.011s
```

3.2.5 $n = 5, \mathbf{len}(R) = 7$

Die Eingabedatei:

```
5
4
blau gelb 1
gruen orange 1
rosa rot 1
tuerkis rot 1
```

../Implementierung/325.txt

Die Ausgabe:

```
$ time python main.py 325.txt
1. Reihe: gelb
2. Reihe: rosa, rot
3. Reihe: rot, rosa, rosa
4. Reihe: tuerkis, rot
5. Reihe: orange
Diese Aufstellung erhält 9 Punkte.

real    1m13.490s
user    1m13.136s
sys     0m0.020s
```

3.2.6 $n = 6, \mathbf{len}(R) = 7$

Die Eingabedatei:

```
6
4
blau gelb 1
gruen orange 1
rosa rot 1
tuerkis rot 1
```

../Implementierung/326.txt

Die Ausgabe:

```
$ time python main.py 326.txt
1. Reihe: rosa
2. Reihe: rot, rot
3. Reihe: rosa, tuerkis, gruen
4. Reihe: rot, orange
5. Reihe: gelb
Diese Aufstellung erhält 8 Punkte.

real    1m12.491s
user    1m12.303s
sys     0m0.023s
```

3.2.7 $n = 7, \mathbf{len}(R) = 7$

Die Eingabedatei:

```
7
4
blau gelb 1
gruen orange 1
rosa rot 1
tuerkis rot 1
```

../Implementierung/327.txt

Die Ausgabe:

```
$ time python main.py 327.txt
1. Reihe: gruen
2. Reihe: rot, orange
3. Reihe: rosa, rosa, blau
4. Reihe: rot, gelb
5. Reihe: tuerkis
Diese Aufstellung erhält 7 Punkte.

real    0m26.736s
user    0m26.352s
sys     0m0.070s
```

4 Quellcode (ausschnittsweise)

Zuerst einige Imports und die Definition der möglichen Farben:

```python
import sys
import itertools
import math

FARBEN = [
    'blau',
    'gelb',
    'gruen',
    'orange',
    'rosa',
    'rot',
    'tuerkis'
]
```

imports.py

Definition der Funktion `grundfarbenmengen_bestimmen`:

```python
def grundfarbenmengen_bestimmen(anzahl_farben, regeln):
    """
    Bestimmt alle möglichen Mengen an Grundfarben, ausgehend von den gegebenen Regeln und
    ↪ der Anzahl der Farben in einer Grundfarbenmenge
    @param anzahl_farben: Anzahl an Grundfarben pro Menge
    """
```

```
    # Welche Farben werden in den Regeln erwähnt?
    farben = set()
    for regel in regeln:
        farben.add(regel[0])
        farben.add(regel[1])

    # Hinzufügen zusätzlicher Farben, um auf die geforderte Anzahl zu kommen
    for farbe in FARBEN:
        if len(farben) >= anzahl_farben:
            break
        farben.add(farbe)

    # Wenn mehr Farben in den Regeln erscheinen als gefordert sind, sind alle Kombinationen an Farben aus den Regeln Grundfarbenmengen
    return itertools.combinations(farben, anzahl_farben)
```

grundfarbenmengen_bestimmen.py

Definition der Funktion `aufstellung_bewerten`:

```
def aufstellung_bewerten(a, regeln):
    """
    Bewertet eine Aufstellung mit den gegebenen Regeln
    @param a: Aufstellung als Liste an Farben
    @param regeln: dict mit Tupeln zweier Farben als Schlüssel und der Punktzahl als Wert
    """
    score = 0
    score += regeln.get((a[0],a[1]),0)
    score += regeln.get((a[0],a[2]),0)
    score += regeln.get((a[1],a[2]),0)
    score += regeln.get((a[1],a[3]),0)
    score += regeln.get((a[1],a[4]),0)
    score += regeln.get((a[2],a[4]),0)
    score += regeln.get((a[2],a[5]),0)
    score += regeln.get((a[3],a[4]),0)
    score += regeln.get((a[4],a[5]),0)
    score += regeln.get((a[3],a[6]),0)
    score += regeln.get((a[4],a[6]),0)
    score += regeln.get((a[4],a[7]),0)
    score += regeln.get((a[5],a[7]),0)
    score += regeln.get((a[6],a[7]),0)
    score += regeln.get((a[6],a[8]),0)
    score += regeln.get((a[7],a[8]),0)
    return score
```

aufstellung_bewerten.py

Definition der Funktion `ideale_anordnung`:

```
def ideale_anordnung(grundfarben, anzahl_farben, regeln):
    """
    Bestimmt die beste Anordnung der gegebenen Grundfarben mit der gegebenen Anzahl an Farben und den gegebenen Regeln
    @param grundfarben: Die Grundfarben, deren Anordnung bestimmt werden soll (als Liste)
    @param anzahl_farben: Anzahl an verschiedenen Farben in der zu bestimmenden Anordnung
    @param regeln: dict mit Tupeln zweier Farben als Schlüssel und der Punktzahl als Wert
    """
    score = -math.inf
    comb = None
    for f in itertools.product(grundfarben, repeat=9):

        # Sind genürgend Farben in der Anordnung?
        if len(set(f)) < anzahl_farben:
            continue

        s = aufstellung_bewerten(f, regeln)
        if s > score:
            comb = f
            score = s
    return comb, score
```

ideale_anordnung.py

Auswerten der Eingabedatei und Aufruf aller Funktionen:

```python
if __name__ == '__main__':
    if len(sys.argv) < 2:
        print("Benutzung: " + sys.argv[0] + " <Eingabedatei>", file=sys.stderr)
        exit(1)

    try:
        with open(sys.argv[1]) as f:
            zeilen = f.read().split("\n")
    except FileNotFoundError:
        print("Eingabedatei nicht gefunden!", file=sys.stderr)
        exit(1)


    # Anzahl der Farben und Regeln aus der Eingabedatei lesen
    anzahl_farben = int(zeilen[0])
    regeln = {}
    for zeile in zeilen[2:]:
        zeile = list(filter(None, zeile.split(" ")))
        if len(zeile) != 0:
            regeln[(zeile[0], zeile[1])] = int(zeile[2])
            regeln[(zeile[1], zeile[0])] = int(zeile[2])

    score = -math.inf
    comb = None

    # Für jede Grundfarbenmenge die ideale Anordnung bestimmen und die beste verwenden
    for farben in grundfarbenmengen_bestimmen(anzahl_farben, regeln):
        c, s = ideale_anordnung(farben, anzahl_farben, regeln)
        if s > score:
            score = s
            comb = c

    print("1. Reihe:", comb[0])
    print("2. Reihe:", ', '.join(comb[1:3]))
    print("3. Reihe:", ', '.join(comb[3:6]))
    print("4. Reihe:", ', '.join(comb[6:8]))
    print("5. Reihe:", comb[8])
    print("Diese Aufstellung erhält", score, "Punkte.")
```

main.py