Aufgabe 3: Telepaartie

Richard Wohlbold
Einsendenummer: ???

5. September 2019

Inhaltsverzeichnis

1 Lösungsidee **1**

2 Umsetzung **2**

3 Beispiele **2**
3.1 Allgemein . . . 2
3.1.1 Parameterfehler . . . 2
3.2 Berechnung der Leerlauflänge . . . 3
3.2.1 1. Verteilung aus der Aufgabe . . . 3
3.2.2 2. Verteilung aus der Aufgabe . . . 3
3.2.3 Eigene Verteilung . . . 3
3.2.4 Keine Zahl als Parameter . . . 3
3.2.5 Negativer Parameter . . . 3
3.3 Berechnung $L(n)$ . . . 4
3.3.1 $L(10)$ . . . 4
3.3.2 $L(100)$ . . . 4
3.3.3 Negativer Parameter . . . 4
3.3.4 Keine Zahl als Parameter . . . 4

4 Quellcode (ausschnittsweise) **4**

1 Lösungsidee

Die Zustände und Transformationen der Biberverteilungen können als gerichteter Graph dargestellt werden. Dabei repräsentiert ein Knoten im Graph eine Biberverteilung. Eine Kante von Knoten K_1 zu Knoten K_2 existiert, falls die Biberverteilung des Knotens K_1 mit dem *Telepaarteur* in die Biberverteilung des Knotens K_2 umgewandelt werden kann. Die Anordnung der Füllmengen der Behälter ist egal, da sich die Anordnung (x, y, z) genauso lösen lässt wie die Anordnung (x, z, y). Somit repräsentiere ich die Füllmengen an einem Knoten als Menge von drei Zahlen. Enthält die Menge an einem Knoten den Wert 0, beträgt die Leerlauflänge 0. Ansonsten beträgt die Leerlauflänge die geringste Anzahl von Kanten zu einem Knoten, dessen Menge eine 0 enthält.

Ein Beispielgraph ist in Abbildung 1 zu sehen.

Ein Aufgabenmerkmal ist, dass der eingeführte Graph gerichtete Zyklen enthalten kann, d.h. man kann über Telepaartie zum gleichen oder einem vorherigen Zustand gelangen. Ein Beispiel dafür ist die Biberverteilung $2, 4, 4$, die erneut durch Umfüllen zweier Biber in den Behälter, in dem sich zwei Biber befinden, erreicht werden kann. Diese Eigenschaft muss bei der Lösung der Aufgabe berücksichtigt werden.

Um das Problem zu lösen und den kürzesten Weg zu einem Knoten zu finden, der eine 0 enthält, wende ich eine *Breadth-First-Search* an. Diese durchquert den Graphen gleichmäßig, indem stets der ununtersuchte Knoten untersucht wird, der den kürzesten Weg zum Wurzelknoten, also der Ausgangsverteilung, besitzt. Wird ein Knoten gefunden, der eine 0 enthält, ist der Prozess beendet, denn man kann sich sicher sein, dass es keinen kürzeren Weg zu einem Knoten gibt, der eine 0 enthält.

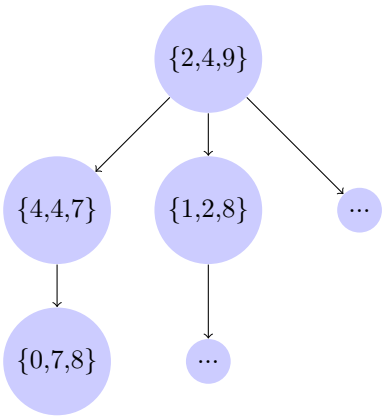

Abbildung 1: Beispiel aus der Aufgabe, als Graph repräsentiert

Breadth-First-Search wird mit einer *Queue*, also First-In-First-Out-Datenstruktur, umgesetzt. Wird ein Knoten untersucht, so wird überprüft, ob die Verteilung am zu untersuchenden Knoten eine 0 enthält. Ist dies der Fall, ist die Prozedur beendet. Ansonsten werden alle Knoten, zu denen Kanten vom untersuchten Knoten aus bestehen und die noch nicht untersucht wurden, ans Ende der Schlange gesetzt. Der erste zu untersuchende Knoten ist der Wurzelknoten (also die zu untersuchende Biberverteilung). Eine leere Menge wird anfangs initialisiert, der nach und nach alle untersuchten Knoten hinzugefügt werden.

2 Umsetzung

Das oben beschriebene Verfahren habe ich in Python 3 umgesetzt. Eine Biberverteilung repräsentiere ich als 3-Tupel, die aufsteigend sortiert sind. Zur Bestimmung der Leerlauflänge (LLL) habe ich die Funktion `lll` definiert, die als Argument die ursprüngliche Biberverteilung akzeptiert. Der Algorithmus verwendet die `Queue`-Datenstruktur aus der Python-Standardbibliothek, um die Breadth-First-Search umzusetzen. Diese enthält sowohl alle noch zu untersuchenden Biberverteilungen als auch die jeweilige Anzahl an Telepaartien, die benötigt wurden, um zur entsprechenden Verteilung zu gelangen. Zusätzlich wird ein `set` verwendet, das alle bereits untersuchten Verteilungen enthält, da es Elemente nur einmal enthalten kann und dabei gute Abrufzeiten bietet. In einer Schleife wird der Queue eine Verteilung und ihr Abstand zum Wurzelknoten entfernt und überprüft, ob das `set` sie schon enthält. Falls dies nicht der Fall ist, wird überprüft, ob die Verteilung eine 0 enthält, d.h. ob nur zwei Behälter gefüllt sind. In diesem Fall ist die Prozedur beendet. Falls dies nicht der Fall ist, werden alle Verteilungen, die sich mithilfe von Telepaartie aus der momentan untersuchten ergeben, der Queue hinzugefügt. Da die momentane Biberverteilung (x, y, z) sortiert ist $(x \leq y; y \leq z)$, gibt es nur drei Möglichkeiten:

1. $(2x, y - x, z)$
2. $(2x, y, z - x)$
3. $(x, 2y, z - y)$

Diese werden der Queue, inklusive des um 1 erhöhten Abstandes, angehangen.

Um die Funktion $L(n)$ zu bestimmen, iteriere ich durch alle möglichen Biberverteilungen mit n Elementen und füge sie sortiert einem `set` hinzu. Für jede dieser Verteilungen rufe ich die Funktion $lll(n)$ auf und speichere den maximal zurückgegebenen Wert in einer Variablen und gebe sie zum Schluss aus.

Das Programm wird entweder mit dem Parameter `l` oder `lll` aufgerufen. Falls die Funktion $L(n)$ ausgerechnet werden soll, muss der Parameter `l` mit der Zahl n als weiteren Parameter.

3 Beispiele

3.1 Allgemein

3.1.1 Parameterfehler

Werden zu wenige Parameter angegeben, gibt das Programm Informationen zur Benutzung aus:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py
Benutzung: a3-telepaartie/Implementierung/main.py <lll|l> <x,y,z|n>
[richard@planck ~/Programs/bwinf38]$
```

3.2 Berechnung der Leerlauflänge

3.2.1 1. Verteilung aus der Aufgabe

Die Biberverteilung $2, 4, 9$ stammt aus der Aufgabe. In der Aufgabe wird eine LLL von 2 für die Verteilung angegeben, die das Programm bestätigt:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py lll 2 4 9
2
[richard@planck ~/Programs/bwinf38]$
```

3.2.2 2. Verteilung aus der Aufgabe

. Die Biberverteilung $\{3, 5, 7\}$ stammt aus der Aufgabe In der Aufgabe wird angegeben, dass $LLL(\{3, 5, 7\}) > LLL(\{2, 4, 9\})$. Dies wird durch das Programm bestätigt:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py lll 3 5 7
3
[richard@planck ~/Programs/bwinf38]$
```

3.2.3 Eigene Verteilung

Als eigene Verteilung verwende ich $\{1248, 573, 12499\}$. Das Programm findet fast direkt die Antwort:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py lll 1248 573 12499
11
[richard@planck ~/Programs/bwinf38]$
```

3.2.4 Keine Zahl als Parameter

Wird für die Berechnung der Leerlauflänge keine Zahl für x, y oder z angegeben, gibt das Programm einen Fehler aus:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py lll 5 5 test
Benutzung: a3-telepaartie/Implementierung/main.py <lll|l> <x,y,z|n>
[richard@planck ~/Programs/bwinf38]$
```

3.2.5 Negativer Parameter

Wird für die Berechnung der Leerlauflänge eine negative Zahl für x, y oder z angegeben, gibt das Programm einen Fehler aus:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py lll -1 5 7
x, y oder z d\"urfen nicht negativ sein!
[richard@planck ~/Programs/bwinf38]$
```

3.3 Berechnung $L(n)$

3.3.1 $L(10)$

Laut meinem Programm ist $L(10) = 2$:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py l 10
2
[richard@planck ~/Programs/bwinf38]$
```

3.3.2 $L(100)$

Laut meinem Programm ist $L(100) = 7$:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py l 100
7
[richard@planck ~/Programs/bwinf38]$
```

3.3.3 Negativer Parameter

Wird bei der Berechnung von $L(n)$ ein negativer Parameter angegeben, gibt das Programm einen Fehler aus:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py l -1
n darf nicht negativ sein!
[richard@planck ~/Programs/bwinf38]$
```

3.3.4 Keine Zahl als Parameter

Wird bei der Berechnung von $L(n)$ ein keine Zahl als Parameter angegeben, gibt das Programm einen Fehler aus:

```
[richard@planck ~/Programs/bwinf38]$ python a3-telepaartie/Implementierung/main.py l test
Benutzung: a3-telepaartie/Implementierung/main.py <lll|l> <x,y,z|n>
[richard@planck ~/Programs/bwinf38]$
```

4 Quellcode (ausschnittsweise)

Als erstes werden einige Pakete importiert:

```
#!/usr/bin/python3
import sys
import math
from queue import Queue
```

imports.py

Danach wird die Funktion `lll` definiert:

```
def lll(v):
    """
    Breadth-First-Search durchführen, um LLL zu finden.
    """
    done = False
    v = tuple(sorted(v))

    q = Queue()
    q.put((v, 0))

    nodes = set()

    while True:
```

```
        v, steps = q.get()
        v = tuple(sorted(v))
        if v in nodes:
            continue
        nodes.add(v)
        # print("debug: processing", v)
        if 0 in v:
            break
        q.put(((2*v[0], v[1]-v[0], v[2]), steps+1))
        q.put(((2*v[0], v[1], v[2]-v[0]), steps+1))
        q.put(((v[0], 2*v[1], v[2]-v[1]), steps+1))

    return steps
```

lll.py

Danach wird die Funktion `l` definiert:

```
def l(n):
    solution = 0
    combs = set()
    for i in range(n+1):
        for j in range(n-i+1):
            combs.add(tuple(sorted((i,j,n-i-j))))

    for i,comb in enumerate(combs):
        solution = max(solution, lll(comb))

    return solution
```

l.py

Der Code, der die Parameter einliest, die Funktionen aufruft und das Ergebnis ausgibt, wurde hier aufgrund seiner geringen Relevanz für die Lösung der Aufgabe ausgelassen.