

Beispiellösung: Schwierigkeiten

Team-ID: ?????? 

Team-Name: Team-Name 

Bearbeiter/-innen dieser Aufgabe:
Vor- und Nachnamen 

Inhaltsverzeichnis

1 Lösungsidee	1
1.1 Ohne Konflikte	1
1.2 Mit Konflikten 	2
1.3 Laufzeitanalyse (optional)	2
2 Umsetzung	3
3 Werkzeuge	3
4 Beispiele	3
5 Quellcode	3

1 Lösungsidee

Für eine Teilmenge von Aufgaben soll eine Sortierung nach Schwierigkeit berechnet werden. Die Schwierigkeit ergibt sich aus mehreren Altklausuren: In jeder Altklausur ist eine Kette von Aufgaben mit aufsteigender Schwierigkeit vorgegeben.

1.1 Ohne Konflikte

Aus den Altklausuren wissen wir für manche Aufgaben, dass sie leichter sind als andere. Dabei gilt auch Transitivität: Wenn $A < B$ und $B < C$, dann folgt daraus auch $A < C$. Aus diesen Beziehungen bilden wir einen gerichteten Graphen. Die Aufgaben sind die Knoten des Graphen. Stehen zwei Aufgaben in einer Altklausur direkt hintereinander, fügen wir eine Kante von der leichteren zur schwereren Aufgabe ein. So werden die Beziehungen verschiedener Altklausuren miteinander verknüpft.

Damit können wir für jede Aufgabe X bestimmen, welche Aufgaben leichter als X sind. Das sind alle Aufgaben, von denen es einen Pfad zu X gibt. Diese Menge nennen wir S_X .

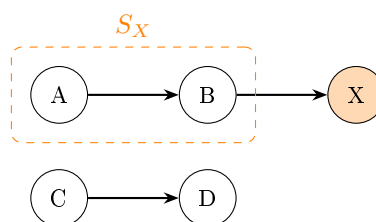


Abbildung 1: S_X enthält alle Aufgaben, von denen ein Pfad zu X führt, in diesem Fall A und B . C und D gehören nicht dazu, da von ihnen kein Pfad zu X existiert.

Beispiellösung: Schwierigkeiten

Die Mengen S_X für jede Aufgabe X bestimmen die gesuchte Reihenfolge: Aufgaben mit kleinerer Menge S_X müssen vor Aufgaben mit größerer Menge S_X stehen. Diese Beobachtung zeigt ein Beispiel mit Aufgaben X und Y . X hat eine leichtere Aufgabe und Y hat zwei. Es gibt drei Fälle:

1. X ist leichter als Y . Dann gehören X und alle Aufgaben aus S_X auch zu S_Y . X muss also vor Y stehen.
2. X ist schwieriger als Y , also Y ist leichter als X . Dann gehören Y und alle Aufgaben aus S_Y auch zu S_X . Demnach müsste S_X mindestens eine Aufgabe mehr enthalten als S_Y , dieser Fall kann also nicht eintreten.
3. X und Y sind nicht vergleichbar. Es gibt keinen Pfad zwischen ihnen. Dann ist die Reihenfolge egal.

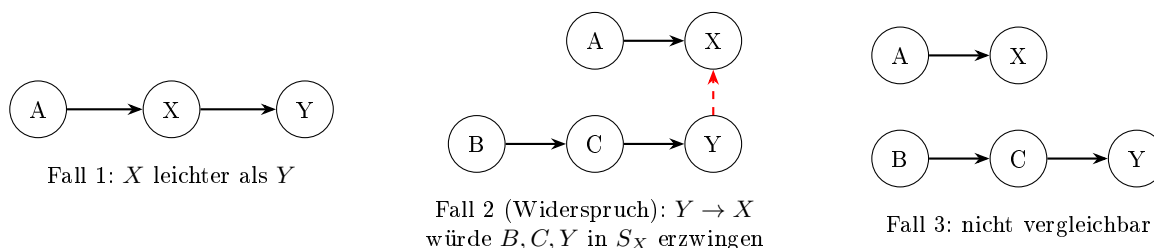


Abbildung 2: Die drei möglichen Fälle für zwei Aufgaben X und Y , wobei ein Element in S_X und zwei Elemente in S_Y .

Da Fall 2 unmöglich ist, gilt allgemein: Ist S_X kleiner als S_Y , steht X immer vor Y und nie umgekehrt. Übrig bleibt noch der Fall gleich großer Mengen. Wären X und Y vergleichbar, müsste eine der beiden Mengen mindestens ein Element größer sein als die andere (siehe Fall 2). Gleich große Mengen bedeuten also automatisch, dass kein Pfad zwischen X und Y existiert und auch hier ist die Reihenfolge egal. Es reicht also, die Aufgaben der neuen Klausur nach der Größe ihrer Menge S_X zu sortieren.

1.2 Mit Konflikten

Widersprechen sich die Altklausuren, entsteht im Graphen ein Zyklus: eine Kette von Kanten, die zum Ausgangsknoten zurückführt. Innerhalb eines Zyklus gibt es keine eindeutig leichteste oder schwerste Aufgabe mehr. Das nennen wir einen Konflikt. Unser Programm behandelt alle Aufgaben eines Zyklus als gleich schwer und kennzeichnet sie so in der Ausgabe.

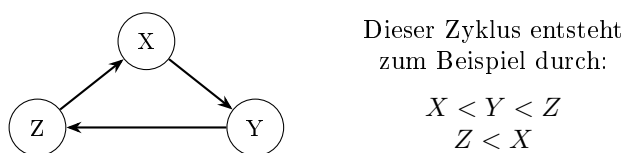


Abbildung 3: Ein Konflikt: Durch den Zyklus $X \rightarrow Y \rightarrow Z \rightarrow X$ können die Aufgaben X , Y und Z nicht geordnet werden und werden als gleichschwer angesehen.

Zwei Aufgaben X und Y stehen genau dann in Konflikt, wenn X in S_Y und gleichzeitig Y in S_X liegt, es also einen Pfad in beide Richtungen gibt. Alle anderen Aufgaben in S_X und S_Y sind dann identisch, da es für jede Aufgabe, für die es einen Pfad nach X gibt, auch einen nach Y gibt sowie auch umgekehrt. S_X und S_Y sind also gleich groß.

Der Algorithmus bleibt also gleich, nur bei gleich großen Mengen müssen wir jetzt zusätzlich unterscheiden: Stehen die betroffenen Aufgaben tatsächlich in Konflikt (Pfad in beide Richtungen), oder besteht einfach keine Beziehung zwischen ihnen (kein Pfad)?

1.3 Laufzeitanalyse (optional)

Um die Laufzeit des Programms einordnen zu können, betrachten wir die einzelnen Schritte des Algorithmus und wie oft sie jeweils durchlaufen werden.

Beispiellösung: Schwierigkeiten

Beim Einlesen wird über alle Zeilen der Eingabe iteriert. Für jedes $<$ in einer Zeile wird eine Kante in den Graphen eingefügt. Die Anzahl der Kanten und auch die Laufzeit des Einlesens hängt also von der Anzahl der $<$ in der Eingabe ab. Die Anzahl der Knoten ist dagegen durch die Anzahl der Aufgaben begrenzt. In unserem Fall sind das höchstens 26, da die Aufgaben mit Buchstaben bezeichnet werden.

Anschließend wird für jede Aufgabe eine Breitensuche durchgeführt, um herauszufinden, welche Aufgaben leichter sind. Eine einzelne Breitensuche schaut sich im schlechtesten Fall jeden Knoten und jede Kante einmal an. Da dies für jede Aufgabe wiederholt wird, addiert sich dieser Aufwand über alle Aufgaben hinweg.

Danach werden die q Aufgaben der neuen Klausur nach der Anzahl ihrer leichteren Aufgaben sortiert. Pythons Sortierfunktion braucht dafür $q \log(q)$ Schritte.

Zuletzt werden die Aufgaben bei Konflikten gruppiert. Dabei wird jede Aufgabe der neuen Klausur mit jeder anderen Aufgabe der neuen Klausur verglichen. Am Ende werden die entstandenen Gruppen nur noch ausgegeben.

Insgesamt ist der Aufwand der Breitensuchen am größten, da für jeden Knoten (also jede Aufgabe) jeder weitere Knoten und jede Kante besucht wird. Die anderen Schritte fallen im Vergleich zu den Breitensuchen kaum ins Gewicht.

2 Umsetzung

Die Lösungsidee wird in Python umgesetzt, Ein- und Ausgabe erfolgen über die Konsole.

Der Graph wird durch eine Adjazenzliste **adj** dargestellt. **adj** ist ein Dictionary, das für jede Aufgabe die Menge von direkten Nachfolgern speichert. Beim Einlesen wird für jede Relation $<$ eine passende Kante eingefügt.

Ein weiteres Dictionary **leichtereAufgaben** wird verwendet, um für jede Aufgabe X die Menge S_X zu speichern. Dafür wird für jede Aufgabe X durch Breitensuche bestimmt, welche Knoten von X aus erreichbar sind. Für jeden Knoten wird X in dessen Menge eingetragen.

Zur Sortierung übergeben wir **sorted** eine Funktion, die nach der Größe der **leichtereAufgaben**-Mengen sortiert. Anschließend wird für jede ausgewählte Aufgabe geprüft, ob eine andere ausgewählte Aufgabe gegenseitig in ihrer Menge vorkommt. Wenn das der Fall ist, werden beide als Konflikt markiert, indem sie für die Ausgabe in einer Menge gespeichert werden. Aufgaben ohne Konflikt werden als einziges Element in einer Menge gespeichert.

3 Werkzeuge

Wir haben **defaultdict** aus der **collections**-Bibliothek verwendet, um neue Dictionary-Einträge direkt mit der leeren Menge zu initialisieren. Für die Liste **aufgaben** war das Modul **string** der Standardbibliothek hilfreich, um die ersten n Buchstaben des Alphabets zu erzeugen.

4 Beispiele

Das Programm gibt die Sortierung der Aufgaben von leicht nach schwer aus. Dabei sind Aufgaben, zwischen denen es einen Konflikt gibt, in Klammern geschrieben.

```
schwierigkeiten0.txt  B E D F C
schwierigkeiten1.txt  A G C F D
schwierigkeiten2.txt  (A B D E) (F G)
schwierigkeiten3.txt  (M N) (H I L) (A B C D) J E K F G
schwierigkeiten4.txt  B I F N W
schwierigkeiten5.txt  H R Z Q (C S) K E N (M O) J L F V (A D G P T U X) B W I Y
```

5 Quellcode

```
1 from collections import defaultdict
2 import string
3
4 # Einlesen
5 eingabe = input().split("_")
```

Beispiellösung: Schwierigkeiten

```
anzahlKlausuren = int(eingabe[0])
7 anzahlAufgaben = int(eingabe[1])
anzahlNeueKlausur = int(eingabe[2])
9
# Adjazenzliste des Graphen erstellen
11 adj = defaultdict(set)
for i in range(anzahlKlausuren):
13     ordnung = input().split("_<_")
    for j in range(len(ordnung)-1):
15         # Kante zwischen benachbarten Aufgabe
        adj[ordnung[j]].add(ordnung[j+1])
17
# Aufgabe fuer die neue Klausur einlesen
19 neueKlausur = input().split("_")
# Liste mit allen Aufgaben
21 aufgaben = list(string.ascii_uppercase[:anzahlAufgaben])

23 # leichtere Aufgaben Mengen befuellen
leichtereAufgaben = defaultdict(set)
25 for aufgabe in aufgaben:
    # Breitensuche mit jeder Aufgabe als Startpunkt
27     besucht = {a: False for a in aufgaben}
    erreichbar = []
29     erreichbar.append(aufgabe)
    while len(erreichbar)>0:
31         schwereAufgabe = erreichbar.pop(0)
        besucht[schwereAufgabe] = True
33         if schwereAufgabe!=aufgabe:
            leichtereAufgaben[schwereAufgabe].add(aufgabe)
35         for nachbar in adj[schwereAufgabe]:
            if not besucht[nachbar]:
37                 erreichbar.append(nachbar)

39 # Sortierung der Aufgaben fuer die neue Klausur nach der Anzahl leichter Aufgaben
sortiert = sorted(neueKlausur, key=lambda a: len(leichtereAufgaben[a]))
41
# Ausgabe vorbereiten und Konflikte erkennen
43 ausgegeben = set()
ausgabe = []
45 for aufgabe in sortiert:
    # Aufgaben, welche bereits durch einen Konflikt ausgegeben wurden
47     if aufgabe in ausgegeben:
        continue
49
    gleicheSchwierigkeit = [aufgabe]
51     # alle Aufgaben welche einen Konflikt mit der aktuellen Aufgabe haben
    gleicheSchwierigkeit += [a for a in neueKlausur
53                          if a in leichtereAufgaben[aufgabe]
                          and aufgabe in leichtereAufgaben[a]]
55
    ausgegeben.update(gleicheSchwierigkeit)
57     # Konflikte durch Klammern kennzeichnen
    if len(gleicheSchwierigkeit)>1:
59         ausgabe.append("(" + "_".join(gleicheSchwierigkeit) + ")")
    else:
61         ausgabe.append(gleicheSchwierigkeit[0])

63 # Ausgabe
print("_".join(ausgabe))
```

a2.py