

Aufgabe 2: Schwierigkeiten

Team-ID: 00178

Team-Name: Team-Name

Bearbeiter/-innen dieser Aufgabe:
Matthew Greiner

November 1, 2024

Contents

1	Lösungsidee / Ansatz	1
1.1	Annahme: Keine Konflikte in Klausuren	1
1.2	Annahme: Konflikte in Klausuren	2
2	Umsetzung	2
2.1	Annahme: Keine Konflikte in Klausuren	2
2.2	Annahme: Konflikte in Klausuren	3
3	Beispiele	3
3.1	Beispiele der BwInf-Webseite	3
3.2	Eigene Beispiele	3
4	Quellcode	3

1 Lösungsidee / Ansatz

Bei dieser Aufgabe ist es das Ziel, eine "gute" Anordnung der gegebenen Aufgaben zu finden. Diese Anordnung basiert auf den eingelesenen Klausuren und dessen Schwierigkeitsabstufungen. Mein Ansatz dieses Problem zu lösen, ist die Untergliederung in folgende Teilaufgaben:

1.1 Annahme: Keine Konflikte in Klausuren

1. Texteingabe lesen und Inhalt strukturieren

Mithilfe von einem `BufferedReader` Zeilen der Inputdatei lesen und in entsprechende Members der Klasse speichern.

2. Schwierigkeitsanordnung analysieren und Einordnung in passende Datenstruktur

Da die Aufgaben der Klausuren nur nach Schwierigkeit geordnet sind, gibt es nur eine "leichter als" Beziehung in den Aufgaben. In kurz: Eins kommt immer vor einem Anderem. Diese Eigenschaft kann man mithilfe von ungewichteten gerichteten Graphen darstellen, wobei eine Kante von A nach B zeigt, dass Aufgabe A leichter als B ist. Die Aufgaben der Klausuren könnte man auch in einem Baum darstellen, aber da ein Baum prinzipiell einfach ein ungewichteter gerichteter Graph ist, habe ich mich entschieden, dieses Teilproblem mit der Erstellung eines Graphen zu lösen.

3. Sortierung dieser Datenstruktur

Der erstellte Graph mit den Aufgaben als Knoten und den Beziehungen als Kanten, kann mit einer topologischen Sortierung sortiert werden. Dies funktioniert nur, wenn der Graph gerichtet und keine

Zyklen enthält, was unter der Annahme, dass es keine Konflikte in den Klausuren gibt, stimmt. Die topologische Sortierung ist eine Anordnung der Knoten, so dass alle Nachfolger eines Knotens, nach diesem Knoten vorkommen. Also könnte den Graphen so darstellen, dass die gerichteten Kanten nur nach rechts zeigen. Wenn der Graph topologisch sortiert ist, bedeutet das für unsere Aufgabe, dass die gewünschten Aufgaben, einfach von links nach rechts aus dem Graph ausgelesen werden muss, um eine "gute" Anordnung dieser Aufgaben zu erhalten. Diese sortierte Folge an Aufgaben zu erhalten ist somit das Ziel dieses Teilproblems.

4. **Sortierte Ausgabe der gewünschten Aufgaben** Da ich durch das vorgehende Teilproblem eine topologisch sortierte Liste mit den Aufgaben bekomme, müssen für das Ergebnis nur noch die gewünschten Aufgaben aus der Liste von links nach rechts ausgelesen werden.

1.2 Annahme: Konflikte in Klausuren

2 Umsetzung

2.1 Annahme: Keine Konflikte in Klausuren

Um diesen Ansatz umzusetzen, löse ich die von oben beschriebenen Teilprobleme:

1. Texteingabe lesen und Inhalt strukturieren

Mithilfe eines `BufferedReader` aus der `java.io` package, kann jede Zeile ausgelesen werden und als `String` verwendet werden. Für die ersten Zeile des Inputtextes teile ich diesen `String` an den Leerzeichen und speichere die resultierenden `Strings` als `Array`. Aus diesem `Array` lese ich die `Strings` aus und speichere sie als `Integer` in den entsprechenden Members der Klasse. Eine Aufgabe wird als `String` repräsentiert.

Die folgenden Zeilen, die jeweils eine Klausur darstellen sollen, bearbeite ich gleichermaßen, allerdings nur mit dem Unterschied, dass ich die Zeile an den kleiner Zeichen (<) teile. Die einzelnen `Strings`, die jeweils eine Aufgabe darstellen, speichere ich als Liste in ein "Klausur" Objekt. Diese Klausur Objekte sichere ich nun wieder als Liste, welche ein Member der Hauptklasse ist.

Die letzte Zeile speichere ich, separat von den anderen Klausuren, wieder als Klausur Objekt.

2. Schwierigkeitsanordnung analysieren und Einordnung in passende Datenstruktur

Um den gerichteten azyklischen Graph darzustellen, verwende ich eine Adjazenzliste, welche ich mit einer `HashMap` realisiere. In der `HashMap` ist jeder Knoten bzw. Aufgabe ein Schlüssel der Map. Folglich ist der zugehörige Wert in der Map eine Liste aus Aufgaben, welche die Nachfolger der Knoten in dem Graph darstellen.

3. Sortierung der Datenstruktur

Da es sich um einen azyklischen Graph handelt, ist es möglich, jeden Knoten zu erreichen, basierend auf die Anzahl an Eingangsknoten (Eingangsgrad des Knotens). Das geht indem man bei den Knoten mit Eingangsgrad 0 beginnt, und von dessen Nachbarn den Eingangsgrad um eins verkleinert. Dies führt führt man durch bis es keinen Knoten mit Eingangsgrad 0 mehr gibt. Dies geht nur, weil es keine Zyklen gibt. Auf diesem Prinzip basiert meine Implementierung des Sortieralgorithmus.

Zu Beginn erstelle ich eine `HashMap` in der jedem Knoten sein Eingangsgrad zugeordnet wird. Das geht indem man über die ganzen Knoten in dem Graph iteriert und für jedem Nachbar eines Knotens, den Eingangsgrad um eins inkrementiert. Dann können alle Knoten mit einem Eingangsgrad von 0 in eine `Linked-List` (wird als `FIFO Queue` verwendet) hinzugefügt werden, über die man iteriert, bis sie leer ist. Denn dann wurde der ganze Graph durchlaufen. Dies kann man mit einer `while`-Schleife machen und entnimmt der `Linked-List` jeweils den ersten Knoten und fügt diesen in eine Liste, die zum Schluss den geordneten Graphen darstellen soll. Von dem Knoten wird der Eingangsgrad in der `HashMap` um 1 verringert. Anschließend wird der Eingangsgrad von jedem Nachbar von diesem Knoten ebenfalls um eins verringert und der Nachbarknoten wird in die `FIFO Queue` hinzugefügt, falls dessen Eingangsgrad durch die Dekrementierung 0 geworden ist.

Wenn die `while`-Schleife zuende gelaufen ist, hat man die topologisch sortierte Knoten des Graphen in der Resultatsliste. Optional kann man folgern, dass der sortierte Graph Zyklen enthalten hat, falls die Endresultatsliste eine andere Anzahl an Elemente hat, wie Knoten im Graph. Dann könnte man dementsprechend handeln. Allerdings ist der Fall eines Zyklus mit der Annahme, dass der Graph keinen hat, abgedeckt.

4. **Sortierte Ausgabe der gewünschten Aufgaben** Da wir eine sortierte Folge der Knoten aus der vorherigen Teilaufgabe bekommen, muss nur noch über diese Liste von vorne iteriert werden und in eine Ergebnisliste hinten hinzugefügt werden, falls dieses Element mit einer der ursprünglich gewollten Klausurufgaben übereinstimmt. Danach muss nur noch die Ergebnis dem Nutzer gegeben werden, zum Beispiel mit einer Ausgabe auf die Konsole.

2.2 Annahme: Konflikte in Klausuren

3 Beispiele

3.1 Beispiele der BwInf-Webseite

- 0.
- 1.
- 2.
- 3.
- 4.
- 5.

3.2 Eigene Beispiele

4 Quellcode

Unwichtige Teile des Programms sollen hier nicht abgedruckt werden. Dieser Teil sollte nicht mehr als 2-3 Seiten umfassen, maximal 10.