Aufgabe 1: Störung

Team-ID: 00919

Team-Name: Object Grind

Bearbeiter/-innen dieser Aufgabe:
Ole Deifuß

November 21, 2022

Contents

1 Lösungsidee

1.1 Aufgabenstellung und Informationen zur Form

In der Aufgabe **Störung** geht es darum, ein Buch nach mit Lücken versehener Lückentexten (Eingabedateien) zu durchsuchen. Herausgefunden werden sollen alle Stellen im Buch, die ein Lückentext vervollständigen können. In Stellen im Buch müssen die im Lückentext vorgegebenen Worte genau an den richtigen Positionen stehen und in jede Lücke des Lückentextes darf exakt ein Wort hineinpassen.

Der Aufgabenstellung und der Form der vorgegebenen Lückentexte lässt sich Folgendes über den Aufbau der Lückentexte entnehmen:

- jede Lücke wird durch einen Unterstrich (_) dargestellt,
- alle Worte in den Lückentexten werden kleingeschrieben,
- Satz- oder Sonderzeichen werden in den Lückentexten nicht verwendet,
- daraus folgt, dass sich alle Lückentexte innerhalb eines geschlossenen Satzes beziehungsweise innerhalb eines durch Kommata abgetrennten Teilsatzes befinden.

1.2 Suchalgorithmus

Um die Stellen im Buch zu finden, bietet es sich an zunächst nach dem ersten Wort des Lückentextes zu suchen. Dafür wird Zeile für Zeile im Buch durchsucht. Beginnt ein Lückentext mit einer oder mehreren Lücken, soll trotzdem nach dem ersten Wort (das erste Wort bedeutet hier die erste Stelle, an der keine Lücke mehr steht) in der Eingabedatei gesucht werden. Um die Position zu ermitteln, soll allerdings im Buch dieselbe Anzahl an Worten von der Position des ersten Wortes zurückgegangen werden, wie der Lückentext Lücken am Anfang stehen hat. Alle dabei gefundenen Stellen wiederum sollen dann auch auf den Rest des Lückentextes überprüft werden, indem sie Wort für Wort mit dem Lückentext verglichen werden. Das erste Wort muss nicht verglichen werden, da es, falls der Lückentext mit einem Unterstrich beginnt, für jedes Wort stehen darf oder, falls nicht, bereits im ersten Schritt überprüft wurde. Unterstriche - im Kontext der Aufgabe also Lücken im Lückentext - dürfen dabei für jedes Wort stehen und beim Vergleichen wird somit das nächste Wort im Buch übersprungen. Trifft der Algorithmus an einer Stelle auf keine Übereinstimmung oder findet dieser einen Beginn eines neuen Absatzes durch eine leere Zeile, wird dieser abgebrochen und die nächste Position wird überprüft. Bei einem Satzzeichen kann der Suchvorgang ebenfalls abgebrochen werden, da die Lückentexte alle nur über einen Teilsatz verlaufen. Wurde der Algorithmus bis zum letzten Wort im Lückentext nicht abgebrochen, handelt es

sich bei der durchsuchten Stelle um eine mögliche Lösung des Lückentextes. Wurden so eine oder sogar mehrere Stellen gefunden, welche den Lückentext vervollständigen könnte, können die darin enthaltenen Lücken durch die jeweiligen Worte ersetzt werden. Diese Lösungen können dann mit der Position, da nach dieser in der Aufgabenstellung gefragt war, ausgegeben werden.

2 Umsetzung

2.1 Einlesen der Eingabedateien

Zunächst müssen sowohl das Buch als auch die Eingabedateien eingelesen werden.

Das Buch wird Zeile für Zeile eingelesen und als Liste gespeichert.

Für die Eingabedateien bieten sich ebenfalls Listen an, da sie Wort für Wort mit dem Buch, beziehungsweise Ausschnitten daraus, verglichen werden sollen. Speichert man diese also direkt einzeln als Worte ab, kann man mithilfe des Indexes auf bestimmte davon zugreifen.

2.2 Methoden

Die Methode `getPositions(...)` sucht für eine einzelne Eingabedatei die passenden Stellen im Buch, die mit demselben Wort beginnen, wie der Lückentext der Eingabedatei. Falls der Lückentext mit einer Lücke beginnt, muss die Methode mit einer **If-Bedingung** prüfen, ob sich an erster Stelle der gegebenen Liste ein Unterstrich befindet. Ist dies der Fall, wird ein Zähler erhöht, damit nach dem nächsten Wort des Lückentextes gesucht wird. Wurde dieses erste Wort im Lückentext gefunden, wird mit einer **For-Schleife** durch das Buch iteriert. Findet der Algorithmus dabei dieses Wort wieder, wird sowohl die Zeile (beziehungsweise die Zeile - 1, da eine Liste mit dem Index 0 beginnt) als auch der Index in der Zeile als Array gespeichert und an eine Liste angehängt, die am Ende der Methode zurückgegeben wird. mit einer **While-Schleife** wird dieser Suchprozess solange innerhalb einer Zeile ausgeführt, bis der zurückgegebene Index -1 ist. Das ist der Fall, wenn das Wort nicht mehr darin enthalten ist. Sollte der Lückentext mit einer Lücke beginnen, wird von der gespeicherten Position wieder mit einer **For-Schleife**, bevor sie an die Liste angehängt wird, noch die jeweilige Anzahl an Worten im Buch zurückgegangen. Das kann man durch die Leerzeichen im Buch gut zählen, es soll dementsprechend einfach die Position nach dem letzten Leerzeichen angegeben werden.

Die in der Methode `getPositions(...)` herausgefundenen Positionen werden dann der zweiten Methode `checkPositions(...)` als Parameter gegeben. Diese iteriert durch die Liste der Positionen und sucht sich erstmal den Text nach der jeweiligen Stelle raus. Hat diese Liste nicht mindestens die Anzahl an Worten wie der Lückentextes, was passieren kann, wenn die Stelle nahe des Endes der Zeile im Buch steht, werden mit einer **For-Schleife** noch so viele Zeilen daran angehängt, bis die Anzahl gleich oder sogar größer ist. Da der Algorithmus nicht auf Zeichensetzung achtet, kann es passieren, dass in so einem Text ein Satz oder sogar ein Absatz endet. Da aber nach jeder Zeile ein Leerzeichen eingefügt wird, damit nicht das letzte Wort der ersten Zeile und das erste Wort der zweiten Zeile ohne Abstand aneinander hängen, gibt es im Text nach einer leeren Zeile, also einem Absatz oder ähnlichem, zwei Leerzeichen direkt hintereinander. Findet der Algorithmus so eine Stelle, soll direkt die nächste Position untersucht werden, da Lückentexte niemals über ein Satzende hinweg verlaufen. Der dadurch zusammengesetzte Text wird dann auch in eine Liste aus Worten umgewandelt. Diese muss allerdings zunächst noch mit einer weiteren **For-Schleife** von Satz- und Sonderzeichen befreit werden, da diese nicht von den Worten bei trennen durch die Leerzeichen entfernt werden. Erst dann werden mit einer dritten **For-Schleife** die Liste aus dem Buch und die des Lückentextes miteinander verglichen. Darin prüft eine **If/Else-If/Else-Bedingung**, ob

1. das aktuelle Element des Buchs mit dem aktuellen Element der Liste des Lückentextes übereinstimmt,
2. das aktuelle Element der Liste des Lückentextes keine Lücke ist oder
3. keins der beiden obigen Ereignisse der Fall ist.

Ist das aktuelle Element beider Listen identisch, wird mit einer weiteren **If-Bedingung** überprüft, ob es sich dabei um das letzte Element der Liste des Lückentextes handelt. Ist das der Fall, wird die Position an die Liste angehängt, die am Ende der Methode zurückgegeben wird, falls aber nicht, wird weiter durch die beiden Listen iteriert. Wenn das Element in der Liste des Lückentextes keine Lücke ist, kann die **For-Schleife** abgebrochen werden, da die erste **If-Bedingung** nicht den Wert **true** hatte und somit die aktuellen Elemente nicht übereinstimmen. Hat keine der beiden obigen Bedingungen den Wert **true**

erhalten, handelt es sich bei dem aktuellen Element der Liste des Lückentextes um eine Lücke. Dabei muss auch wieder geprüft werden, ob es das letzte Element ist. Ist es dies, kann auch in dieser Situation die Position an die Liste der korrekten Positionen angehängt werden.

Da beide Methoden jeweils nur eine Eingabedatei prüfen können, müssen diese für jede Eingabedatei einmal ausgeführt werden, was in der **main**-Methode des Programms geschieht. Diese hat keine Relevanz für den Algorithmus, weshalb sie nicht unter `Quellcode` zu finden ist.

3 Beispiele

3.1 Sonderfälle im Überblick

Lässt man das Programm laufen, werden für jede der sieben Eingabedateien die passenden Lösungen ausgegeben. Dabei wird zunächst die Position im Buch angegeben, gefolgt von den Teilsätzen, welche den jeweiligen Lückentext vervollständigen.

Unter den Eingabedateien können folgende Sonderfälle auftreten:

Lückentexte können ...

1. ... mehrere Lücken direkt nacheinander haben,
2. ... mit einer Lücke beginnen,
3. ... mit einer Lücke enden.

Unter den vorgegebenen Eingabedateien befinden sich zwar keine, die mit einer Lücke beginnen, trotzdem könnten solche durch den Algorithmus vervollständigt werden.

Neben den drei Sonderfällen, die sich aus der Form der Eingabedateien ergeben, gibt es auch Sonderfälle, die zwar vom Algorithmus besonders behandelt werden müssen, bei denen dies allerdings nicht beim Betrachten der Eingabedateien auffällt.

Dazu zählen Fälle, bei denen ...

4. ... eine zu untersuchende Stelle im Buch am Ende einer Zeile steht,
5. ... das erste Wort eines Lückentextes mehrmals in einer Zeile vorkommt oder
6. ... der `zweite Sonderfall` aus der obigen Liste gegeben ist und zudem das erste Wort, nach dem der Algorithmus sucht, am Anfang einer Zeile steht.

Bevor die einzelnen Sonderfälle anhand von Beispielen erklärt werden, sollte angemerkt werden, dass in der Ausgabe des Programms nicht exakt der vervollständigte Lückentext steht, sondern jeweils der gesamte Teilsatz, in dem diese Stelle zu finden ist. Dies ist beabsichtigt, der ausgegebene Text wird durch eine weitere Methode, die nicht relevant für den Algorithmus ist und daher nicht unter `Quellcode` zu finden ist, zusammengestellt. Diese sucht den Abschnitt zwischen dem letzten und nächsten Satz- beziehungsweise Sonderzeichen heraus.

Die gesamte Ausgabe des Programms sieht folgendermaßen aus:

```
Störung Nr.0:
das _ mir _ _ _ vor
Zeile: 440
'Das kommt mir gar nicht richtig vor,'

Störung Nr.1:
ich muß _ clara _
Zeile: 425
'Ich muß in Clara verwandelt sein!'
Zeile: 441
'Ich muß doch Clara sein,'

Störung Nr.2:
fressen _ gern _
Zeile: 213
'Fressen Katzen gern Spatzen?'
Zeile: 214
'Fressen Katzen gern Spatzen?'
Zeile: 214
'Fressen Spatzen gern Katzen?'

Störung Nr.3:
das _ fing _
Zeile: 2319
'und das Spiel fing an.'
Zeile: 3301
'Das Publikum fing an,'

Störung Nr.4:
ein _ _ tag
Zeile: 2293
'es ist ein sehr schöner Tag!'

Störung Nr.5:
wollen _ so _ sein
Zeile: 2185
'Wollen Sie so gut sein,'

Störung Nr.6:
_ kleine thür _ _ verschlossen
Zeile: 465
'die kleine Thür war wieder verschlossen'
```

Ausgabe des Programms

In der `Ausgabe` wird für jede der sieben Eingabedateien zunächst die Nummer angegeben, damit man die Ergebnisse den einzelnen Dateien zuordnen kann. Die Nummer der Störung ist dabei dieselbe wie die, die im Namen der Datei vorkommt. Dann wird der in der Eingabedatei enthaltene Lückentext ausgegeben, so kann man diesen direkt mit den danach ausgegebenen Lösungsmöglichkeiten vergleichen. Diese werden mit Zeilenangabe (hier ist es tatsächlich die Zeile im Buch, nicht die der Liste) ausgegeben.

Index in Liste	Zeile	Index in Zeile
0	35	4
1	61	5
2	67	4
3	88	28
4	111	3
5	115	4
6	135	44
7	135	54
...	...	...
26	308	64
...	...	...
41	439	1
...	...	...

Table 1: Liste der Methode `getPositions(...)` für Störung Nr.0

3.2 Sonderfall 1

`Sonderfall 1` stellt keine Besonderheit in der Arbeitsweise des Algorithmus dar, sondern unterscheidet sich ausschließlich von den anderen Lückentexten durch dessen Form. Dafür kann man als Beispiel die erste Eingabedatei wählen, diese hat drei Lücken direkt hintereinander:

"**das _ mir _ _ _ vor**".

Zunächst wird für den Lückentext nach dem ersten Wort im Buch gesucht, welches hier "**das**" ist. Mit der dadurch erstellten Liste, aus der man in `Tabelle 1` einen Ausschnitt sehen kann, wird dann die zweite Methode `checkPositions(...)` aufgerufen. Diese prüft jedes Element der Liste einzeln. Nimmt man die erste Position (Zeile 36, da der Index mit 0 startet) und sucht diese im Buch, erhält man folgende Zeile:

"**Das Steuer hält ein Kindesarm**".

Diese wird Wort für Wort verglichen. An zweiter Stelle im Lückentext steht eine Lücke, daher darf dort jedes Wort eingesetzt werden. Das nächste Wort, welches dementsprechend überprüft werden muss, ist das Wort "**mir**" aus dem Lückentext. Im Ausschnitt aus dem Buch steht allerdings "**hält**". Der Suchalgorithmus wird für diese eine Stelle abgebrochen, da die beiden Worte nicht miteinander übereinstimmen.

Die erste Position, an der auch die zweite Methode fündig wird, ist in Zeile 440 (in der Liste würde 439 stehen). Dort steht folgender Satz:

"**»Das kommt mir gar nicht richtig vor,«**".

Bei diesem wird ebenso jedes Wort mit dem Lückentext abgeglichen. Da Lücken aber einfach übersprungen werden, weil sie für jedes erdenkliche Wort stehen können, kann man erkennen, dass mehrere Lücken hintereinander kein Problem für den Algorithmus darstellen.

3.3 Sonderfall 2

Da der Algorithmus immer nach dem ersten Wort des Lückentextes suchen soll, stellt `Sonderfall 2` das aufwendigste Problem dar. Nach einer Lücke - im Kontext der Eingabedateien also ein Unterstrich - kann schlecht im Buch gesucht werden. Also muss nach dem ersten Element der Liste des Lückentextes gesucht werden, das keine Lücke ist. Als Beispiel hierfür bietet sich keine der vorgegebenen Eingabedateien an, deshalb wurde eine weitere Eingabedatei dazu gefügt (stoerung6.txt). Der darin enthaltene Lückentext ist folgender:

"**_ kleine Thür _ _ verschlossen**".

Hier wird nach dem Wort "**kleine**" gesucht, da es das erste schon angegebene Wort im Lückentext ist. Würde man allerdings einfach dieses Wort dem Suchalgorithmus der ersten Methode `getPositions(...)` geben, würden auch in der zweiten Methode `checkPositions(...)` nur Stellen zurückgegeben werden, die mit dem Wort "**kleine**" beginnen. Es könnten im schlimmsten Fall sogar Positionen gefunden werden, die den Lückentext vervollständigen könnten, wäre die Lücke am Anfang nicht.

Um das zu verhindern, muss von der Position des ersten Wortes im Lückentext aus, also hier wieder "`kleine`", dieselbe Anzahl an Worten im Buch zurückgegangen werden, wie der Lückentext Lücken am Beginn stehen hat.

Steht das erste Wort nach den Lücken am Anfang des Lückentextes und am Anfang einer Zeile, ergibt sich ein weiterer Sonderfall, der im Abschnitt `Sonderfall 6` behandelt wird.

3.4 Sonderfall 3

Um den `Sonderfall 3` zu erklären, bietet sich Störung Nr.1 an. Diese endet mit einer Lücke. Bei einem solchen Fall passiert nicht viel anderes als bei anderen Lückentexten, doch am Ende der zweiten Methode `checkPositions(...)`, während jedes Wort geprüft wird, wird für das letzte Element der Else Teil der **If-Bedingung** ausgeführt, da es sich um eine Lücke handelt. Hier muss mit einer weiteren **If-Bedingung** geprüft werden, ob es sich dabei um das letzte Element der Liste des Lückentextes handelt. Ist es das, so wie es in dem Fall der Störung Nr.1 ist, muss die jeweilige Stelle auch an die Liste richtiger Positionen angehängt werden.

3.5 Sonderfall 4

Für den `Sonderfall 4` ist es egal, welchen Lückentext man als Beispiel wählt. Es kann bei allen passieren. Da bereits eine `Tabelle für Störung Nr.0` abgebildet wurde, kann man diese auch hier als Beispiel verwenden. In der Liste kommt beim Index 26 eine Stelle, an der das Wort "**das**" am Ende der Zeile 309 (in der Liste 308) steht:

"**glatt; und als sie sich ganz müde gearbeitet hatte, setzte sich das**"

Hier muss beim Prüfvorgang in der zweiten Methode `checkPositions(...)` auch die nächste Zeile an den Textausschnitt aus dem Buch angehängt werden. Das geschieht immer, wenn eine Liste aus Worten, die aus dem Textausschnitt erstellt wird, nicht mindestens die Länge der Liste des Lückentextes besitzt.

3.6 Sonderfall 5

Auch der `Sonderfall 5` lässt sich anhand der `Tabelle für Störung Nr.0` erklären. Dort stehen an Index 6 und 7 der Liste beide Male die Zeile 136 (135 in der Liste). Um beide Zeilen zu finden, darf der Suchalgorithmus der ersten Methode getPositions(...) nicht abgebrochen werden, sobald er eine Stelle in einer Zeile gefunden hat. Dafür wurde eine **While-Schleife** verwendet, die erst abbricht, wenn der von der Methode zurückgegebene Index -1 ist. Das passiert, falls das gesuchte Wort nicht in dem gegebenen Text enthalten ist.

3.7 Sonderfall 6

`Sonderfall 6` ist nur eine Erweiterung von `Sonderfall 2`. Allerdings macht dieser nochmal eine größeres Problem auf: wird in der Methode `getPositions(...)` nach einem Wort gesucht, das nicht das erste Element in der Liste des Lückentextes ist, muss die Methode, bevor sie die Position an die Liste möglicher Positionen anhängt, von dieser Position aus so viele Worte zurückgehen, wie der Lückentext Lücken am Anfang stehen hat. Die Anzahl der Lücken am Anfang des Lückentextes wird zu Beginn der Methode bereits gespeichert. Am Ende, vor dem Anhängen, wird anhand der Leerzeichen von der jeweiligen Stelle die korrekte Position ermittelt, an der der Lückentext beginnt. Ist dieser aber über zwei Zeilen im Buch verteilt, muss nicht nur in der Zeile, in der das erste Wort steht, zurückgegangen werden, sondern auch in der davor. Mit einer **For-Schleife** wird für jede Lücke zu Beginn des Lückentextes genau ein Wort im Buch zurückgegangen. In dieser **For-Schleife** wiederum wird geprüft, ob aktuelle Element am Anfang einer neuen Zeile steht. Tut es das, wird die vorherige Zeile ohne Offset in den bestehenden Text eingefügt. So kann das Buch auch nach Lückentexten mit Lücken am Anfang durchsucht werden, die zudem auch über mehr als eine Zeile verteilt stehen.

4 Quellcode

4.1 getPositions(...)

```java
/**
* Die Methode "getPositions" sucht für eine durch den Parameter gegebene Eingabedatei das erste Wort im Buch.
* Dabei wird jede Zeile des Buchs einzeln auf das Anfangswort durchsucht
* und bei einem Fund wird sowohl die Zeile als auch die "Spalte" (der Index in der Zeile, an der das Wort beginnt) als Array an die Liste von Positionen hinzugefügt.
* Falls an erster Stelle der Eingabedatei eine Lücke ("_") steht, wird nach dem zweiten Wort gesucht und dann die Position für das vorherige Wort bestimmt.
* Diese Liste wird am Ende der Methode zurückgegeben.
*
* @param stoerung Die Eingabedatei, nach der das Buch durchsucht werden soll
* @return Eine Liste von Positionen, an denen dasselbe Wort steht wie das erste Wort der Eingabedatei
*
*/
private static List<int[]> getPositions(List<String> stoerung) {
  List<int[]> positions = new ArrayList<>();
    int startPosition = 0;
    while (Objects.equals(stoerung.get(startPosition),"_")) {
      startPosition++;
    }
    for (int lineNumber = 0; lineNumber < book.size(); lineNumber++) {
        String lineText = book.get(lineNumber).toLowerCase();
        int index = lineText.indexOf(stoerung.get(startPosition));
        while (index >= 0) {
          int tempIndex = index;
            int tempLineNumber = lineNumber;
            for (int i = 0; i < startPosition; i++) {
                if (tempIndex - 1 <= 0) {
                    tempLineNumber--;
                    lineText = book.get(tempLineNumber).toLowerCase();
                    tempIndex = lineText.length() - 1;
                }
                tempIndex = lineText.lastIndexOf("␣", tempIndex - 2);
                tempIndex++;
            }
            positions.add(new int[]{tempLineNumber, tempIndex});
            index = lineText.indexOf(stoerung.get(startPosition), index + 1);
        }
    }
    return positions;
}
```

Methode getPositions

4.2 checkPositions(...)

```java
    /**
     * Die Methode "checkPositions" untersucht für jede durch den Parameter "positions" gegebene Position,
     * ob auch der Rest der Eingabedatei, die durch den Parameter "stoerung" gegeben ist,
     * durch die nach der Position folgenden Worte vervollständigt werden kann.
     * Dabei wird ersteinmal eine Liste aus Worten erstellt, das mit dem durch die Position gegebenen Wort startet.
     * Hat diese nicht mindestens die Länge der Eingabedatei, wird auch die nächste Zeile hinzugefügt.
     * Alle Elemente in der Liste werden danach erstmal von Sonderzeichen bereinigt, da diese auch nicht in der Eingabedatei vorgegeben sind.
     * Dann wird Wort für Wort mit denen der Eingabedatei verglichen, wobei Unterstriche (also Lücken in der Eingabedatei) nicht verglichen werden,
     * sondern jedes Wort akzeptieren.
     * Sobald an einer Stelle keine Übereinstellung gefunden wird, wird der Suchvorgang für die eine Position beendet.
     * Alle Stellen, die die Eingabedatei "lösen" können, werden an eine Liste angefügt, die nach Ende der Methode zurückgegeben wird.
     *
     * @param positions Eine Liste von Positionen, die überprüft werden soll
     * @param stoerung Die Eingabedatei, nach der die möglichen Positionen überprüft werden sollen
     * @return Eine Liste von Positionen, die die Eingabedatei "lösen" können
     */
    private static List<int[]> checkPositions(List<int[]> positions, List<String> stoerung) {
        List<int[]> rightPositions = new ArrayList<>();
        positionLoop:
        for (int[] position : positions) {
            StringBuilder text = new StringBuilder(book.get(position[0]).substring(position[1]).toLowerCase());
            int counter = 1;
            while (text.toString().split("␣").length < stoerung.size()) {
                if (position[0] + counter >= book.size()) {
                    break positionLoop;
                }
                text.append("␣").append(book.get(position[0] + counter).toLowerCase());
                counter++;
            }
            if (text.toString().contains("␣␣")) {
                // Zwei aufeinander folgende Leerzeichen nach dem Anfügen eines Leerzeichens an jede Zeile bedeuten, dass hier eine leere Zeile im Buch war.
                // Daher kann diese Stelle das Bruchstück, welches sich immer innerhalb eines Satzes befindet, nicht vervollständigen.
                continue;
            }
            String[] lines = text.toString().split("␣");
            for (int i = 1; i < lines.length; i++) {
                lines[i] = lines[i].replaceAll("[^[a-zA-Z0-9äöüß]]", "");
            }
            for (int index = 0; index < stoerung.size(); index++) {
                if (Objects.equals(stoerung.get(index), lines[index])) {
                    if (index == stoerung.size() - 1) {
                        rightPositions.add(position);
                    }
                } else if (!Objects.equals(stoerung.get(index), "_")) {
                    break;
                } else {
                    if (index == stoerung.size() - 1) {
                        rightPositions.add(position);
                    }
                }
            }
        }
        return rightPositions;
    }
```

Methode checkPositions