Aufgabe 2: Vollgeladen

Team-ID: 00921

Team: Ctrl + Intelligence

Bearbeiter/-innen dieser Aufgabe:
Ediz Köller

21. November 2020

Inhaltsverzeichnis

Lösungsidee

Jeden Tag kann man nur 6h also 360min fahren. Das entspricht einer Gesamtstrecke von 1800min. Da man nur maximal 5 Tage für den Weg brauchen darf, kann man nicht mehr als 4 Hotels besuchen, da die Ankunft am Ziel auch Teil dieser 5 Tage ist. Es ergibt sich außerdem eine minimale Strecke, die man am Tag zurücklegen muss, welche wie folgt zu beschrieben ist:

Dauer der Fahrt := Zeile 1 in Textdokument
Minimale Strecke := Dauer der Fahrt – (4*360)

Beträgt die Strecke, die zurückgelegt werden muss, 1800min, so muss man jede Fahrt 360min fahren, denn wenn man auch nur einmal 359min fährt, kann man maximal nur noch 1799min fahren.
Ein Punkt ist ein Hotel, der Startpunkt oder das Ziel.
Daher muss von einem zum anderen Punkt eine minimale Strecke zurückgelegt werden. Es macht auch Sinn, dass die Strecke von Punkt zu Punkt positiv ist, da ein zurückfahren zum einen im Sachkontext nicht Sinnvoll ist, aber zum anderen auch aufwändiger ist und eine unnötige Gelegenheit für eine niedrigere Bewertung eines Hotels darstellt. Somit gilt ein Punkt von einem anderen Punkt aus als erreichbar, wenn der Schritt, der gemacht wird, größer als 0 ist, minimal die minimale Strecke beträgt und maximal 360min entfernt ist.
Es ist ebenfalls sinnvoll anzumerken, dass eine Route, auf denen ausschließlich Hotels mit einer Bewertung von 1 besucht werden, gleichwertig mit einer Route ist, auf welcher nur ein Hotel mit einer Bewertung von 1, während die restlichen Hotels eine Bewertung von zum Beispiel 5 haben. Das mach die Auswahl der Route einfacher.

Es muss ermittelt werden, von wo ich welche Hotels erreichen kann und wann eine Route als abgeschlossen gilt. Ein Hotel gilt von einem Punkt aus als erreichbar, wenn deren Distanz maximal 360min beträgt. Ebenso gilt eine Route als abgeschlossen, wenn die Distanz zwischen Hotel und Ziel maximal 360min beträgt.

Mit diesen Bedingungen lässt sich nun ein Graph, oder genauer ein Baum, modellieren.
Bei diesem Vorgang muss man sich für jeden Punkt merken, welcher Punkt der Vorgänger ist.
Durch den rekursiven Charakter dieser Datenstruktur kann man den Aufbau wie folgt formulieren:
Eine Wurzel der Tiefe n des Baumes beinhaltet einen Punkt, festgelegt durch den Startpunkt oder ein Hotel der Eingabe. Eine Wurzel der Tiefe n ist der Vorgänger der Teilbäume der Tiefe n+1 dieser Wurzel. Eine Wurzel der Tiefe n+1 beinhaltet ein Hotel, welches vom Vorgänger als erreichbar gilt. Dabei wird n nicht größer als 4. Knoten ist genau dann ein Blatt, wenn er von der Tiefe n=4 ist, oder man von diesem Knoten aus den Zielpunkt erreichen kann.
Der Knoten bei n=0 entspricht dem Startpunkt, die anderen Tiefen enthalten somit Hotels.
Man muss sich dabei merken, welcher Knoten ein Blatt ist.

Ist der Baum aufgebaut, so kann man die Blätter untersuchen. Wenn ein Blatt einem Hotel entspricht, von welchem aus man das Ziel erreichen kann, wird dieser Pfad zusammen mit der niedrigsten Bewertung gespeichert. Das Ergebnis ist dann der Pfad, dessen niedrigste Bewertung am höchsten ist.

Umsetzung

Die Umsetzung erfolgte bei dieser Aufgabe mit Java. Um den wichtigsten Teil der Lösungsidee umzusetzen, braucht man die Datenstruktur Baum. Da ich allerdings keine Library für einen Baum gefunden habe, der zum Problem passt, habe ich diesen kurzerhand während einer Geschichtsstunde entwickelt. Es handelt sich hierbei um den von mir benannten „ArbitraryTree“, also ein generischer und dynamischer Baum, der keine Grenzen bezüglich des Grades seiner Knoten hat. Außerdem kennt ein Knoten seinen Vorgänger, falls er denn einen hat. Dazu habe ich aus den Abiturklassen den BinaryTree verändert und die Klasse List verwendet. Beim BinaryTree habe ich den linken und rechten Teilbaum durch eine Liste von Teilbäumen (successors) ersetzt und um den Vorgänger-Baum (predecessor) erweitert. Natürlich mussten in Zuge dessen die vorhandenen Methoden verändert und neue Methoden geschrieben werden.
Da es einige Informationen zu jedem Hotel gibt, zum Beispiel, ob es als Letztes oder Erstes geeignet ist, die Bewertung und die Entfernung, habe ich eine Klasse dazu erstellt.
Dann gibt es noch die Klasse „Datenanalyse“, welche den Zweck erfüllt, die Informationen aus der Textdatei auszulesen und zu speichern. Die Klasse „Routenplanung“, welche den „ArbitraryTree“ aufbaut und untersucht, benutzt für diesen zweck unter anderem die Methoden, welche die Klasse „Datenanalyse“ bereitstellt. Zum Beispiel weiß diese Klasse auch, welche Hotels auf ein beliebiges Hotel folgen können. Wenn eine Instanz von „Datenanalyse“ erzeugt wird, analysiert sie automatisch die Grundinformationen aus dem Textdokument „hotels.txt“, welches sich im Ordner des Projekts, bzw. im Ordner der jar-Datei befindet.

Im Prinzip sind alle Klassen dazu da, der Klasse „Routenplanung“ zu helfen, da sie die eigentliche Problemstellung löst.
Der Startpunkt der Reise ist in der Implementierung eine Instanz von „Hotel“, welche eine Distanz von 0min hat und eine Bewertung von Double.MAX_VALUE, damit der Startpunkt nicht die Routenplanung beeinflusst. Alle anderen Instanzen der Hotels wurden bereits von der „Datenanalyse“ erstellt. Der erste Knoten im ArbitraryTree ist somit der Startknoten und hat die Tiefe 0. Jetzt werden die möglichen Nachfolger ermittelt und in eine Liste von Knoten der Tiefe 1 eingefügt, wobei sie ebenfalls, wie in der Lösungsidee beschrieben, dem Baum hinzugefügt werden und der Vorgänger der Startknoten ist. Der Baum bis zur Tiefe 1 wurde ermittelt. Für die Tiefe 2 wird dann die Liste der Knote in Tiefe 1 durchlaufen, wobei für jeden Knoten geschaut wird,

welche möglicher Nachfolgerknoten sind, welche dann in die Liste von Knoten in Tiefe 2 und in den Baum mit entsprechendem Vorgänger eingefügt werden. Die Nachfolger eines Knotens werden aber nur ermittelt, wenn der Knoten, dessen Nachfolger man bestimmen will, nicht ein möglicher letzter Knoten sein kann. Kann ein Knoten der Letzte sein, wird er in eine Blätterliste eingefügt. Das wird für die restlichen Tiefen entsprechen wiederholt, aber eben nur bis zur Tiefe von einschließlich 4. Die Blätterliste wird in einen Array umgewandelt, damit man per Index die Inhaltsobjekte einfacher findet. Jedes Element in diesem Array entspricht nämlich einer Route. Das Finden der besten route ist relativ selbsterklärend und der Code ist einfacher zu verstehen als die Erklärung in Sätzen, weshalb ich mich entschieden habe, den Code bereits hier einzufügen, allerdings sollte man anmerken, dass der Vorgänger des Startknotens „null“ ist:

```
ArbitraryTree<Hotel> vorgänger;
int indexBesteBewertung = 0;
double besteBewertung = 0;
	for(int k = 0; k<arrBlätter.length; k++) {
		vorgänger = arrBlätter[k].getPredecessor();
		double bewertung = arrBlätter[k].getContent().getBewertung();
		while(vorgänger!=null) {
			if(vorgänger.getContent().getBewertung()<bewertung) bewertung =
			vorgänger.getContent().getBewertung();
			vorgänger = vorgänger.getPredecessor();
			}
		if(bewertung>besteBewertung) {
			besteBewertung = bewertung;
			indexBesteBewertung = k;
		}
	}
```

Der Knoten in dem Blätterarray mit Index k ist also der mit der besten Route. Da die Route aber nur Rückwärts abgerufen wird, da man von Vorgänger zu Vorgänger geht, verwende ich java.util.Stack, sodass ich die Reihenfolge der Knoten umkehren kann. Da, nachdem alle Hotels eingefügt wurden, ganz oben der Startknoten liegt, welcher aber eigentlich keinem Hotel entspricht, wird vor der Ausgabe des Ergebnisses bereits einmal das oberste Objekt des Stacks entfernt. Dann kann mit einer einfachen While-Schleife das finale Resultat generiert und ausgegeben werden.

Leider konnte ich erst am Freitag anfangen zu programmieren, da ich spät von diesem Wettbewerb erfahren habe. Und natürlich ist praktisch kein Programm von vorne hinein vollkommen. Ich habe bei diesem Programm, welches vom Algorithmus her funktioniert, das Problem, wenn es zu viele Möglichkeiten für nachfolgende Hotels gibt, weil zum Beispiel die gesamte Fahrzeit zu gering ist, oder die Dichte an Hotels quasi ein schwarzes Loch aus der Erde macht, wobei man dann ja eher ein anderes Problem hätte, aber gut, dass die Datenstruktur Array, die ich der Einfachheit des Programmierens zu Gunsten verwendet habe, so groß sind, dass mein Rechner an seine technischen Grenzen stößt. Woher weiß ich, dass es kein Fehler im Algorithmus ist? Ist die Eingabedatei zu groß, sind alle Kerne des Prozessors ausgelastet. Das bedeutet, dass parallelisierte Prozesse meinen Rechner quälen. Da ich keine Parallelisierung von Rechnungen vorgenommen, habe kann ich mir sicher sein, dass es Java-interne Prozesse sind. Das können hier eigentlich nur die Arrays sein. Da Arrays sehr rechen- und speicherineffizient sind, vor allem wenn sie sehr groß werden, stürzt das Programm dann mit einem Heap-Space-Error ab. Eigentlich ist die Idee mit einer minimalen Schrittweite, wie sie oben beschrieben ist, auch erst aus diesem Problem entsprungen. Es hat zwar

dafür gesorgt, dass eines der Beispiele mit 500 Hotels auch funktioniert, aber um alle Beispiele funktionieren zu lassen hat es nicht gereicht.

Beispiele

- Hotels1
  - Start --> Hotel bei 347.0min mit Bewertung 2.7 --> Hotel bei 687.0min mit Bewertung 4.4 --> Hotel bei 1007.0min mit Bewertung 2.8 --> Hotel bei 1360.0min mit Bewertung 2.8 --> Ziel
  - Schlechteste Bewertung: 2.7
- Hotels2
  - Start --> Hotel bei 341.0min mit Bewertung 2.3 --> Hotel bei 700.0min mit Bewertung 3.0 --> Hotel bei 1051.0min mit Bewertung 2.3 --> Hotel bei 1380.0min mit Bewertung 5.0 --> Ziel
  - Schlechteste Bewertung: 2.3

- Hotels3
  - Start --> Hotel bei 358.0min mit Bewertung 2.5 --> Hotel bei 717.0min mit Bewertung 0.3 --> Hotel bei 1075.0min mit Bewertung 0.8 --> Hotel bei 1433.0min mit Bewertung 1.7 --> Ziel
  - Schlechteste Bewertung: 0.3
- Hotels4
  - `Exception in thread "main" java.lang.OutOfMemoryError: Java heap space`
- Hotels5
  - `Exception in thread "main" java.lang.OutOfMemoryError: Java heap space`
- Eigenes Beispiel: Wie bei hotels5.txt, allerdings mit einer Strecke von 1756min
  - Start --> Hotel bei 356.0min mit Bewertung 4.8 --> Hotel bei 699.0min mit Bewertung 4.9 --> Hotel bei 1049.0min mit Bewertung 5.0 --> Hotel bei 1399.0min mit Bewertung 4.8 --> Ziel
  - Schlechteste Bewertung: 4.8

Wie man sieht, sind die Beispiele, die eine Kombination von vielen Hotels und einer eher kleineren Strecke haben, nicht lösbar für das Programm, da der Heap vollläuft, sodass das Programm abbrechen muss. Der Einschränkung positiven und einer minimalen Schrittweite ist also definitiv sinnvoll, da sie dafür sorgt, dass prinzipiell auch größere Eingaben verarbeitet werden können, sofern die Strecke passt. Interessant ist außerdem, dass sobald der Rechner die Aufgabe überhaupt lösen kann, er dies auch innerhalb von ein paar Sekunden tut.

Quellcode

Im Prinzip ist nur die Klasse „Routenplanung“ am wichtigsten, da sie alleine die Aufgabe wirklich löst.

```
package task;

import java.util.Stack;

import util.*;

public class Routenplanung {

	public static void main(String[] args) {
		List<Hotel> tempListHotels = new List<Hotel>();
```

```
		List<ArbitraryTree<Hotel>> tempListTrees = new List<ArbitraryTree<Hotel>>();

		//Der Einfachheit merken, welche Knoten in welcher Tiefe auftauchen:
		List<ArbitraryTree<Hotel>> depth1 = new List<ArbitraryTree<Hotel>>();
		List<ArbitraryTree<Hotel>> depth2 = new List<ArbitraryTree<Hotel>>();
		List<ArbitraryTree<Hotel>> depth3 = new List<ArbitraryTree<Hotel>>();
		List<ArbitraryTree<Hotel>> depth4 = new List<ArbitraryTree<Hotel>>();

		List<ArbitraryTree<Hotel>> blätter = new List<ArbitraryTree<Hotel>>();

		Datenanalyse d = new Datenanalyse();
		ArbitraryTree<Hotel> start = new ArbitraryTree<Hotel>(new Hotel(0, Dou-
ble.MAX_VALUE)); //Startwurzel des Baumes, Tiefe 0

		//Baum aufbauen, mögliche Starthotels als Array in Liste überführen und als Nach-
folger des Startknotens festlegen, maximale Tiefe von 4, da nur 4 hotels besucht werden können,
um innerhalb von 5 Tagen am Ziel zu sein.
		//Wichtig ist, dass die Hotels mehrmals im Baum auftauchen. Daher müssen nebenbei
die Blätter gespeichert werden, die rechtzeitig zum Ziel führen.
		//Tiefe 1:
		tempListHotels = d.convertArrToList(d.getFirstHotels());

		tempListHotels.toFirst();
		tempListTrees.toFirst();
		while(tempListHotels.hasAccess()) {
			ArbitraryTree<Hotel> tree = new ArbitraryTree<Hotel>(tempListHotels.getCon-
tent());
			tempListTrees.append(tree);
			depth1.append(tree);
			tempListHotels.next();
		}

		start.setSuccessors(tempListTrees);


		//Tiefe 2:
		depth1.toFirst();
		while(depth1.hasAccess()) {
			tempListTrees = null;
			tempListHotels = null;

			if(depth1.getContent().getContent().kannLetztesSein() == false) {
				tempListHotels = d.convertArrToList(d.getPossibleNextHo-
tels(depth1.getContent().getContent()));
				tempListHotels.toFirst();
				tempListTrees = new List<ArbitraryTree<Hotel>>();

				while(tempListHotels.hasAccess()) {
					ArbitraryTree<Hotel> tree = new ArbitraryTree<Hotel>(tempLis-
tHotels.getContent());
					tempListTrees.append(tree);
					depth2.append(tree);
					tempListHotels.next();
				}
				depth1.getContent().setSuccessors(tempListTrees);
			}else if(depth1.getContent().getContent().kannLetztesSein() == true){
				blätter.append(depth1.getContent());
			}

			depth1.next();
		}
		depth1.toFirst();
```

```
		while(!depth1.isEmpty()) {
			depth1.remove();
		}
		depth1 = null;


		//Tiefe 3:
		depth2.toFirst();
		while(depth2.hasAccess()) {
			tempListTrees = null;
			tempListHotels = null;

			if(depth2.getContent().getContent().kannLetztesSein() == false) {
				tempListHotels = d.convertArrToList(d.getPossibleNextHo-
tels(depth2.getContent().getContent()));
				tempListHotels.toFirst();
				tempListTrees = new List<ArbitraryTree<Hotel>>();

				while(tempListHotels.hasAccess()) {
					ArbitraryTree<Hotel> tree = new ArbitraryTree<Hotel>(tempLis-
tHotels.getContent());
					tempListTrees.append(tree);
					depth3.append(tree);
					tempListHotels.next();
				}
				depth2.getContent().setSuccessors(tempListTrees);
			}else if(depth2.getContent().getContent().kannLetztesSein() == false){
				blätter.append(depth2.getContent());
			}
			depth2.next();
		}
		depth2.toFirst();
		while(!depth2.isEmpty()) {
			depth2.remove();
		}
		depth2 = null;

		//Tiefe 4:
		depth3.toFirst();
		while(depth3.hasAccess()) {
			tempListTrees = null;
			tempListHotels = null;

			if(depth3.getContent().getContent().kannLetztesSein() == false) {
				tempListHotels = d.convertArrToList(d.getPossibleNextHo-
tels(depth3.getContent().getContent()));
				tempListHotels.toFirst();
				tempListTrees = new List<ArbitraryTree<Hotel>>();

				while(tempListHotels.hasAccess()) {
					ArbitraryTree<Hotel> tree = new ArbitraryTree<Hotel>(tempLis-
tHotels.getContent());
					tempListTrees.append(tree);
					depth4.append(tree);
					tempListHotels.next();
				}
				depth3.getContent().setSuccessors(tempListTrees);
			}else if(depth3.getContent().getContent().kannLetztesSein() == true){
				blätter.append(depth3.getContent());
			}
			depth3.next();
		}
		depth3.toFirst();
```

```
		while(!depth3.isEmpty()) {
			depth3.remove();
		}
		depth3 = null;

		//Durchsuchen von depth4 nach fertigen Routen. Sie sind fertig, wenn man vom Inhaltsobjekt des Knotens direkt zum Ziel kommt.
		depth4.toFirst();
		while(depth4.hasAccess()) {
			if(depth4.getContent().getContent().kannLetztesSein() == true) blätter.append(depth4.getContent());
			depth4.next();
		}
		depth4.toFirst();
		while(!depth4.isEmpty()) {
			depth4.remove();
		}
		depth4 = null;


		//Jetzt wird geschaut, welche Route wie gut ist. Um das Arbeiten zu vereinfachen, wird die Liste in einen Array überführt.
		blätter.toFirst();
		int i = 0;
		while(blätter.hasAccess()) {
			i++;
			blätter.next();
		}
		blätter.toFirst();

		int j = 0;

		ArbitraryTree<Hotel>[] arrBlätter = (ArbitraryTree<Hotel>[]) new ArbitraryTree[i];
		while(blätter.hasAccess()) {
			arrBlätter[j] = blätter.getContent();
			blätter.next();
			j++;
		}


		//Vom Blatt aus müssen nun die Vorgänger identifiziert werden, um die Qualität dieser Route zu bewerten.
		ArbitraryTree<Hotel> vorgänger;
		int indexBesteBewertung = 0;
		double besteBewertung = 0;
		for(int k = 0; k<arrBlätter.length; k++) {
			vorgänger = arrBlätter[k].getPredecessor();
			double bewertung = arrBlätter[k].getContent().getBewertung();
			while(vorgänger!=null) {
				if(vorgänger.getContent().getBewertung()<bewertung) bewertung = vorgänger.getContent().getBewertung();
				vorgänger = vorgänger.getPredecessor();
			}
			if(bewertung>besteBewertung) {
				besteBewertung = bewertung;
				indexBesteBewertung = k;
			}
		}


		//Nun wurde ein möglicher, nach der Aufgabenstellung, idealer Weg gefunden.
		//Jetzt wird mit Hilfe eines Stacks vom Blatt ausgehend die Route aus den Vorgängern ermittelt.
```

```
		//Der Stack korrigiert dabei die verkehrte Reihenfolge beim Auslesen.
		Stack<Hotel> stackHotel = new Stack<Hotel>();
		ArbitraryTree<Hotel> blatt = null;
		if(arrBlätter.length>0) blatt = arrBlätter[indexBesteBewertung];
		else {
			System.out.println("Es gibt keine funktionierende Route.");
			System.exit(0);
		}

		stackHotel.push(blatt.getContent());

		vorgänger = blatt.getPredecessor();
		while(vorgänger!=null) {
			stackHotel.push(vorgänger.getContent());
			vorgänger = vorgänger.getPredecessor();
		}

		//Startwurzel entfernen:
		stackHotel.pop();

		//Somit kann jetzt im finalen Schritt die Route ausgelesen werden, wobei der Stack
geleert wird.
		System.out.print("Start --> ");
		while(!stackHotel.isEmpty()) {
			System.out.print("Hotel bei " + stackHotel.peek().getEntfernung() + "min
mit Bewertung " + stackHotel.peek().getBewertung() + " --> ");
			stackHotel.pop();
		}
		System.out.println("Ziel");
		System.out.println("Schlechteste Bewertung: " + besteBewertung);
	}
}
```