Aufgabe 4: Urlaubsfahrt

Team-ID: 00619

Team-Name: ncrypted

Bearbeiter/-innen dieser Aufgabe:
Oliver Schirmer

22. November 2019

Inhaltsverzeichnis

Lösungsidee

Jede Tankstelle, als auch der Beginn und das Ende der Strecke, werden als Knoten eines Out-Trees betrachtet, bei dem der Start der Strecke die Wurzel darstellt. Die Kanten des Baumes werden dann folgendermaßen gebildet:

Es wird in aufsteigender Reihenfolge über jeden Knoten iteriert und für jeden weiter entfernt liegenden Knoten (Knoten mit größerem Index) geprüft, ob dieser mit einem vollen Tank (beim Start mit der anfänglichen Tankfüllung) erreichbar ist. Ist dies der Fall, werden die Kosten berechnet, um diesen Knoten gerade so zu erreichen, der Tank also beim Ankommen leer ist. Diese werden dann als Kantengewicht zwischen den beiden Knoten in den Baum festgelegt.

Zudem wird für jeden Knoten die minimale Anzahl an Tankvorgängen, um diesen zu erreichen, zwischengespeichert, also die minimale Ebene, auf der der Knoten im Baum gefunden wird. Wird bei zukünftigen Überprüfungen eine Kante gefunden, die eine höhere Ebene hat, wird diese verworfen – so wird sichergestellt, dass nur die minimale Anzahl an Tankvorgängen durchgeführt wird. Durch Memoization werden also viele Abzweigungen beim Baum gar nicht erst überprüft und die Performance wird optimiert. Wird eine Kante gefunden, die der aktuellen minimalen Höhe des Zielknotens entspricht, wird der Startknoten als Parent für den Zielknoten gespeichert. Wird eine Kante mit niedrigerer Höhe gefunden, werden alle bisherigen Parents gelöscht und der aktuelle Startknoten zum Zielknoten hinzugefügt.

Daraufhin folgend wird beim Zielknoten (also dem Streckenende) eine Tiefensuche durchgeführt, die dann alle möglichen Pfade überprüft, um zum Start zu kommen. Beim Start wird der Preis des aktuellen Pfades mit dem aktuellen minimalen Preis abgeglichen und gegebenenfalls der aktuelle Pfad als neuer bester Pfad gespeichert. Bei der Ausgabe des besten Pfads müssen zudem die Kosten abgezogen werden, die durch die anfängliche Tankfüllung kompensiert werden. Da beim ersten und möglicherweise bei folgenden Tankvorgängen noch Restbenzin im Tank ist, muss dieser nicht angerechnet werden. Dadurch besteht die Möglichkeit, dass der Preis nochmals sinkt.

Umsetzung

Der Baum wird in einer Adjazenzmatrix gespeichert und für jede nicht existente Kante das Gewicht auf -1 gesetzt. Die Gewicht werden dabei als Double gespeichert, da die Preise auch gebrochene Zahlenwerte annehmen können. Es wird keine Adjazenzliste gewählt, da dies den Algorithmus verkomplizieren würde und lediglich eine Speicheroptimierung erzielt werden würde. Die Parents hingegen werden als Adjazenzliste gespeichert. Im Allgemeinen wurden ArrayLists Arrays vorgezogen, da das Handling mit diesen vereinfacht wird und Dynamik ermöglicht, falls diese benötigt wird. Trotz dessen werden zu Beginn die nötigen Kapazitäten, wie sie ein Array brauchen würde reserviert. Es werden finden also keine ressourcenbelastenden Reallocations statt.

Da Start und Ziel der Strecke als extra Knoten betrachtet werden, müssen diese ebenfalls der Distanz- und Preisliste der Tankstellen hinzugefügt werden und die Anzahl der Stationen muss um zwei erhöht werden. Die Preise an beiden Stationen sind 0 ct/l, da die Tankfüllung zu Beginn bereits vorhanden ist und nichts extra kostet und beim Zielknoten sowieso nicht mehr getankt werden muss.

Es wird einmalig zu Beginn des Algorithmus berechnet, wie groß die Reichweite mit der anfänglichen Tankfüllung und mit einer vollen Füllung ist. Dadurch kann sehr leicht ohne weitere Berechnungen überprüft werden, ob eine Tankstelle von einer anderen aus erreichbar ist oder nicht. Zudem wird die minimale Höhe des Startknotens auf 0 und die aller anderen Knoten auf unendlich (Integer Maximum) gesetzt.

Wird bei der Tiefensuche kein Pfad gefunden, also ist keine Reise möglich aufgrund der zu geringen Tankgröße, wird dies ebenfalls mit „Keine Reise möglich!“ quittiert.

Beispiele

Beispiel	Ausgabe
1	Reichweite (Start): 275,00 km Reichweite (Voll): 687,50 km Tankvorgänge: 2 100: 10,00 l -> 14,50 € 400: 48,00 l -> 67,20 € Preis: 81,70 €
2	Reichweite (Start): 1137,50 km Reichweite (Voll): 1141,67 km Tankvorgänge: 9 1118: 188,28 l -> 216,52 € 1922: 201,60 l -> 235,87 € 2762: 257,04 l -> 305,88 € 3833: 249,84 l -> 294,81 € 4874: 221,28 l -> 263,32 € 5796: 267,84 l -> 313,37 € 6912: 244,08 l -> 319,74 € 7929: 253,92 l -> 292,01 € 8987: 243,12 l -> 294,18 € Preis: 2535,71 €
3	Reichweite (Start): 533,33 km Reichweite (Voll): 533,33 km Tankvorgänge: 1 465: 80,00 l -> 99,20 € Preis: 99,20 €
4	Reichweite (Start): 370,00 km Reichweite (Voll): 400,00 km Tankvorgänge: 2 344: 71,10 l -> 87,45 € 607: 117,90 l -> 153,27 €

	Preis: 240,72 €
5	Reichweite (Start): 200,00 km Reichweite (Voll): 1161,90 km Tankvorgänge: 9 107: 209,58 l -> 243,11 € 1198: 240,66 l -> 276,76 € 2344: 233,10 l -> 272,73 € 3454: 234,15 l -> 271,61 € 4569: 202,02 l -> 236,36 € 5531: 243,81 l -> 319,39 € 6692: 232,26 l -> 311,23 € 7798: 240,66 l -> 279,17 € 8944: 221,76 l -> 290,51 € Preis: 2500,87 €
6	Reichweite (Start): 275,00 km Reichweite (Voll): 687,50 km Keine Reise möglich!

Die Beispiele 1-5 wurden der BwInf-Website entnommen. Beispiel 6 ist eine Abwandlung von Beispiel 1, wobei die Tankstelle bei 100 Kilometern entfernt wurde, also eine Strecke die nicht befahrbar ist, gegeben ist.

Quellcode

```
package de.ncrypted.urlaubsfahrt;

public class Urlaubsfahrt {

    // consumption in l per 100 km - double to max later
calculations easier
    private static double consumption;
    // tank size in l
    private static int tankSize;
    // initial tank level in l
    private static int tankLevel;
    // length of the way
    private static int wayLength;
    // amount of gas stations
    private static int stations;
    // gas station distances in km
    private static List<Integer> dists = new ArrayList<>();
    // gas station costs in cents/l (between 100 and 199)
    private static List<Integer> prices = new ArrayList<>();

    private static ArrayList<ArrayList<Double>> graph;
    private static ArrayList<ArrayList<Integer>> parents;

    private static double minPrice = Integer.MAX_VALUE;
    private static List<Integer> minPath;

    public static void main(String[] args) {
        // read input file and print out information ...


        // execute algorithm
        System.out.println("---ALGORITHMUS---");
        executeAlgorithm();
    }
```

```
    private static void calcPathPrice(int curNode, double
curPrice, List<Integer> path) {
        path.add(curNode);
        if (curNode == 0) {
            if (curPrice < minPrice) {
                minPrice = curPrice;
                minPath = path;
            }
            return;
        }
        for (int parent : parents.get(curNode)) {
            calcPathPrice(parent, curPrice +
graph.get(parent).get(curNode), new ArrayList<>(path));
        }
    }

    private static void executeAlgorithm() {
        double initialReach = tankLevel / consumption * 100;
        double maxReach = tankSize / consumption * 100;
        System.out.printf("Reichweite (Start): %.2f km\n",
initialReach);
        System.out.printf("Reichweite (Voll): %.2f km\n",
maxReach);

        graph = new ArrayList<>(stations);
        ArrayList<Integer> minDepth = new ArrayList<>(stations);
        parents = new ArrayList<>(stations);
        for (int i = 0; i < stations; i++) {
            graph.add(new ArrayList<>());
            for (int j = 0; j < stations; j++) {
                double price = (i == j ? 0 : -1);
                graph.get(i).add(price);
            }
            minDepth.add(Integer.MAX_VALUE);
            parents.add(new ArrayList<>());
        }
        minDepth.set(0, 0);

        for (int i = 0; i < stations; i++) {
            int curDepth = minDepth.get(i);
            int curDist = dists.get(i);
            double curReach = (i == 0 ? initialReach : maxReach);
            for (int j = i + 1; j < stations; j++) {
                int posDist = dists.get(j);
                int z = 1;
                if (posDist > curDist + curReach) {
                    break;
                }
                if (curDepth + 1 < minDepth.get(j)) {
                    minDepth.set(j, curDepth + 1);
                    parents.get(j).clear();
                    parents.get(j).add(i);
                } else if (curDepth + 1 == minDepth.get(j)) {
                    parents.get(j).add(i);
                } else {
                    // edge won`t be needed later on -> no
calculations required
                    continue;
                }
                double neededFuel = ((posDist - curDist) / 100D) *
consumption;
```

```
                double price = neededFuel * prices.get(i);
                graph.get(i).set(j, price);
            }
        }

        calcPathPrice(stations - 1, 0, new ArrayList<>(stations -
1));
        if(minPath == null) {
            System.out.println("Keine Reise möglich!");
            return;
        }
        minPrice = 0;
        System.out.printf("Tankvorgänge: %d\n", minPath.size() -
2);
        Collections.reverse(minPath);
        double leftFuel = tankLevel - (dists.get(minPath.get(1)) /
100D) * consumption;
        for (int i = 1; i < minPath.size() - 1; i++) {
            int node = minPath.get(i);
            double price = graph.get(node).get(minPath.get(i +
1));
            double fuel = price / prices.get(node);
            if (leftFuel > fuel) {
                fuel = 0;
                price = 0;
                leftFuel -= fuel;
            } else if(leftFuel != 0) {
                fuel -= leftFuel;
                price = fuel * prices.get(node);
                leftFuel = 0;
            }
            System.out.printf(dists.get(node) + ": %.2f l" + " ->
%.2f €\n", fuel, price / 100);
            minPrice += price;
        }
        System.out.printf("Preis: %.2f €\n", minPrice / 100);
    }

    private static void readFileData(String fileName) {
        …
    }
}
```