Aufgabe 2: Vollgeladen

Teilnahme-ID: 62344

Manuel Frohn

21.11.2021

Inhalt

Lösungsansatz

In diesem Abschnitt werden wir uns mit meinem grundlegenden Ansatz zur Lösung des gegebenen Problems beschäftigen. Zu Beginn ist es erst einmal wichtig zu verstehen, wie ich das Problem modelliere. In meiner Modellierung stellen die Hotels einen Graphen dar, bei dem sich Kanten zwischen Hotels aufspannen, wenn man vom einem das andere erreichen kann, diese also maximal sechs Stunden also 360 Minuten vom anderen Hotel entfernt ist. Die Knoten (Hotels) halten hierbei sowohl deren Position als auch Wertung als Information.

Jetzt ist es aber aus Laufzeitgründen nicht möglich, beziehungsweise erstrebenswert den vollständigen Graphen zu konstruieren. Deswegen ziehe ich zur Lösung des Problems einen vom Dijkstra-Algorithmus inspirierten Algorithmus ran. Mein Algorithmus hat also eine Liste aus Knoten, zu denen es bereits einen bekannten Pfad gibt. Aus diesen wählt er dann den Besten aus und fügt alle Knoten, die vom Betrachteten aus erreichbar sind, wieder zur Liste der zur Betrachtenden Knoten hinzu und legt diesen, insofern dies nicht die Anzahl der Knoten, die durchlaufen werden müssen, um den neuen Knoten zu erreichen verlängert als dessen Vorgängerknoten fest. Diesen Prozess wiederholt er so lange bis einen Pfad gefunden wurde, der zum Ziel führt. Welcher Knoten der Beste ist wird klassisch durch den Vergleich der „Fitness“ der Knoten gemacht. Diese wird bestimmt, insofern sich, der aus dem Betrachteten Knoten und dem zu diesem führenden Pfad bildende Pfad innerhalb der angegebenen maximalen Reisezeit bewegt, da die Fitness sonst 0, also „nicht möglich“ wäre, durch die Ermittlung der niedrigsten Wertung, die im, wie vorher beschriebenen Pfad vorkommt.

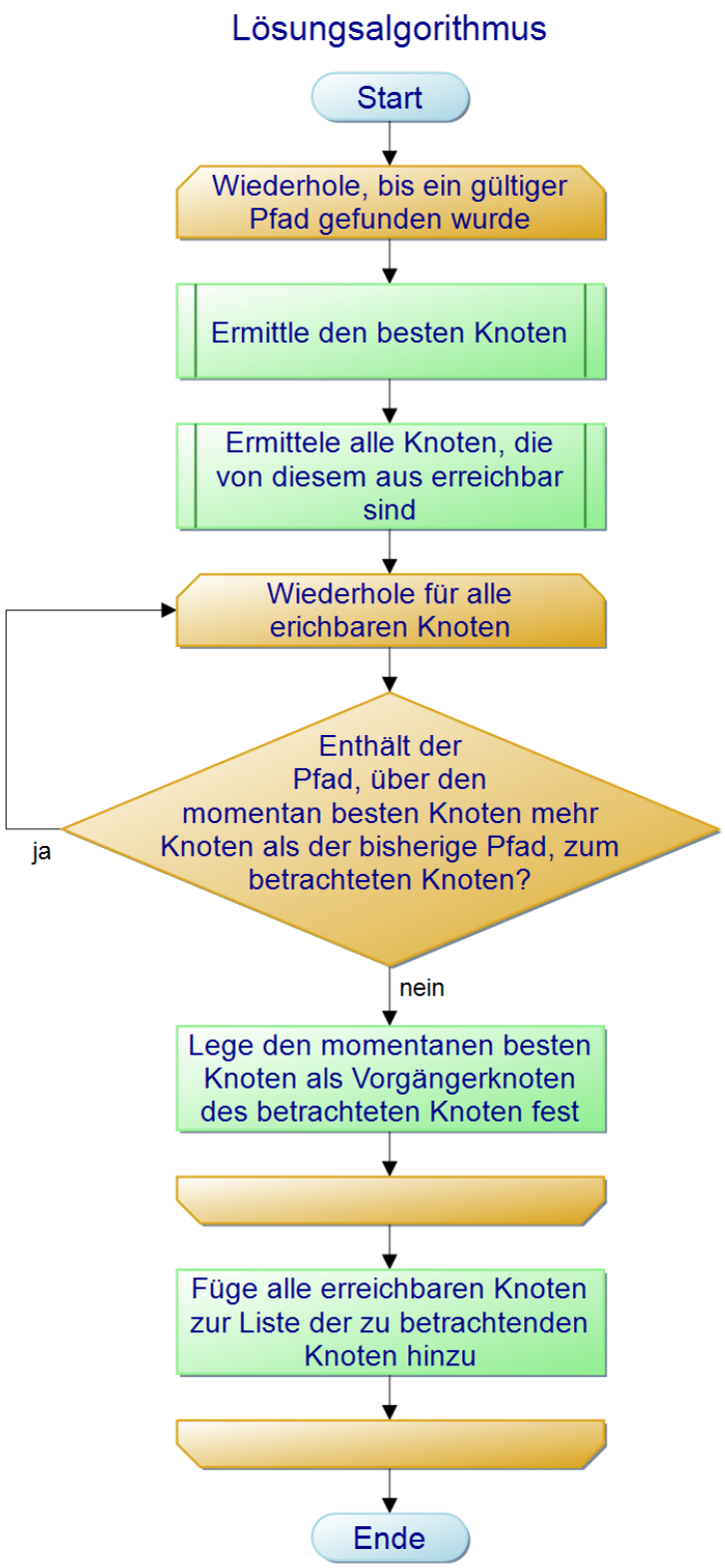

Abbildung 1: Lösungsalgorithmus

Umsetzung

Kommen wir nun zur Umsetzung. Zunächst zur Klassenstruktur. Ich verwende hier lediglich die *Node*-Klasse, welche die Daten des Hotels (Position und Wertung), sowie die Node, die dieser voraus geht trägt. Der Rest der Umsetzung geschieht in der *solveMap(File mapData)*-Methode, welche Teil der GUI-Klasse ist.

Innerhalb dieser Methode werden zuerst die Hoteldaten eingelesen. Hierbei werden alle Hotels in einer Liste, als Node-Objekt gespeichert. Für den End und den Startpunkt werden auch Nodes angelegt. Die Endnode wird danach zu der Liste der anderen Nodes hinzugefügt. Dan werden noch zwei weitere Listen angelegt: die openNodes-Liste für die Nodes die zu betrachten sind und die closedNodes-Liste für die Nodes die bereits betrachtet wurden. Der openNodes-Liste wird dann zu Beginn die startNode hinzugefügt. Danach wird folgendes so lange ausgeführt, bis die endNode einen Vorgänger hat, dieser also ungleich null ist.Zuerst wird geprüft, ob die openNodes-Liste leer ist, wenn ja wird zurückgegeben, dass die Hotelkarte nicht lösbar ist. Danach wird ermittelt, welche der Nodes aus der openNodes-Liste die beste ist, also betrachtet werden soll. Hierzu wird deren Wertung, welche durch die *rateNode(Node node, Node startNode, int pathLenght)*-Methode bestimmt wird zu Rate gezogen. Die Node mit der höchsten Wertung wird zur bestNode. Sollten mehrere Nodes dieselbe Wertung erhalten, so wird jene ausgewählt, die sich am nächsten am Ziel befindet. Danach wird die Node aus der openNodes-Liste entfernt und zur closedNodes-Liste hinzugefügt. Danach wird mithilfe der getReachebleNodes(Node subjectNode, ArrayList<Node> nodes)-Methode eine Liste aller Nodes erstellt, die von der bestNode aus erreichbar sind. Für alle diese wird dann folgendes gemacht: Sollte die Node noch keinen Vorgänger Knoten haben, so wird die bestNode als dieser festgelegt, sonst wird zunächst geprüft, ob die bestNode als Vorgänger der betrachteten Node gleichwertig oder besser wäre als der momentane Vorgänger. Ist dem so, dann wird die bestNode als neuer Vorgänger festgelegt, und wenn die betrachtete Node bereits Teil der closedNodes-Liste ist, dann wird sie aus dieser wieder entfernt und der Liste der openNodes wieder hinzugefügt. Danach wird noch geprüft, ob der Abstand der betrachteten Node zum Vorgänger ihres Vorgängers kleiner oder gleich 360. Ist dem so, dann wird der Vorgänger des Vorgängers direkt als Vorgänger der betrachteten Node festgelegt. Zuletzt wir noch geprüft, ob die Node weder Teil der openNodes-Liste noch der closedNodes-Liste ist. Ist dem so, wird sie zur openNodes-Liste hinzugefügt.

Lösungen

```
Starte bei 0
Gehe zum Hotel, bei 347 mit der Wertung 2.7
Gehe zum Hotel, bei 687 mit der Wertung 4.4
Gehe zum Hotel, bei 1007 mit der Wertung 2.8
Gehe zum Hotel, bei 1360 mit der Wertung 2.8
Gehe zum Endpunkt bei 1680
Das geringst gewertetste Hotel auf dem Pfad hat eine Wertung von 2.7
```

Abbildung 2: Lösung Beispiel 1

```
Starte bei 0
Gehe zum Hotel, bei 341 mit der Wertung 2.3
Gehe zum Hotel, bei 700 mit der Wertung 3.0
Gehe zum Hotel, bei 1053 mit der Wertung 4.8
Gehe zum Hotel, bei 1380 mit der Wertung 5.0
Gehe zum Endpunkt bei 1737
Das geringst gewertetste Hotel auf dem Pfad hat eine Wertung von 2.3
```

Abbildung 3: Lösung Beispiel 2

```
Starte bei 0
Gehe zum Hotel, bei 360 mit der Wertung 1.0
Gehe zum Hotel, bei 717 mit der Wertung 0.3
Gehe zum Hotel, bei 1075 mit der Wertung 0.8
Gehe zum Hotel, bei 1433 mit der Wertung 1.7
Gehe zum Endpunkt bei 1793
Das geringst gewertetste Hotel auf dem Pfad hat eine Wertung von 0.3
```

Abbildung 4: Lösung Beispiel 3

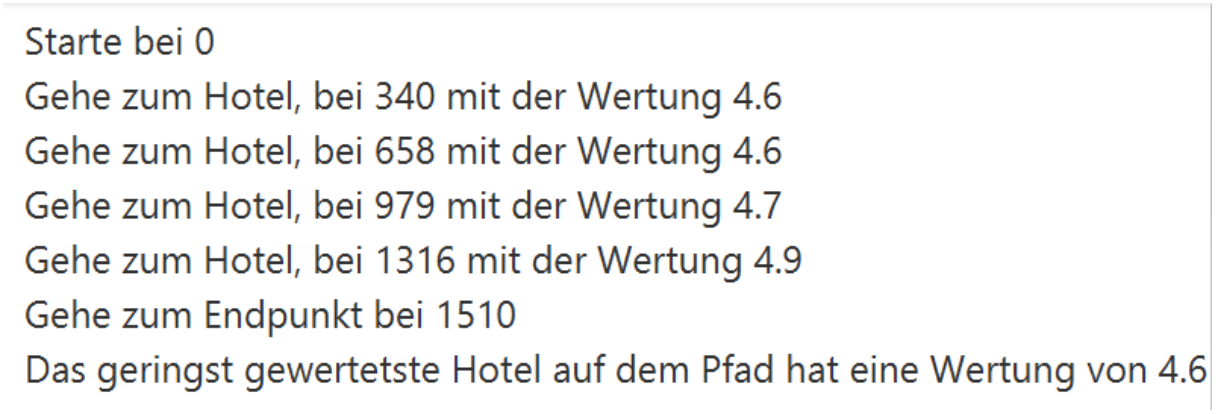

Abbildung 5: Lösung Beispiel 4

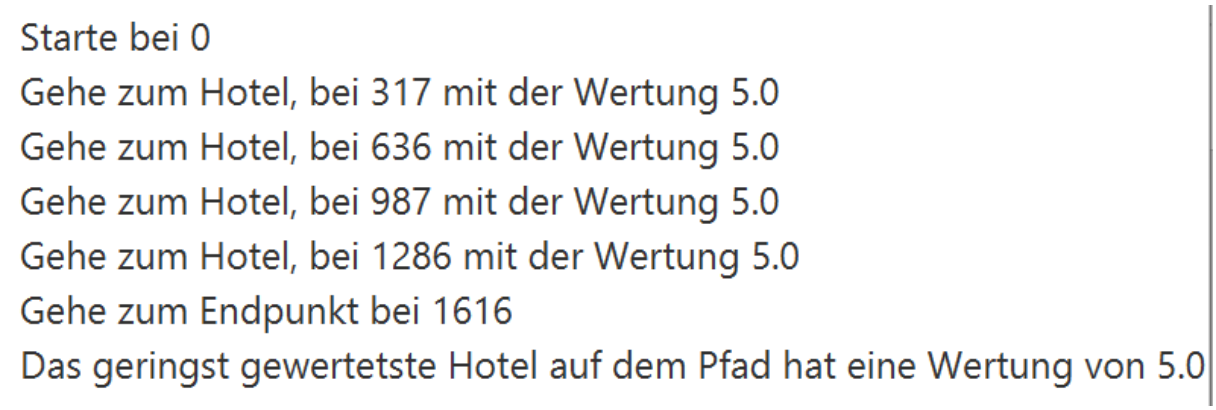

Abbildung 6: Lösung Beispiel 5

Code-Ausschnitt

```java
public String solveMap(File mapData) throws IOException
{
    ArrayList<Node> nodes = new ArrayList<>();
    Node startNode;
    Node endNode;


    /*
     * Load Data
     */
    FileReader fileReader = new FileReader(mapData);
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    int hotelCount = Integer.parseInt(bufferedReader.readLine());
    int wayLenght = Integer.parseInt(bufferedReader.readLine());

    startNode = new Node(0);

    for(int i = 0; i < hotelCount; i++)
    {
        String hotelLine = bufferedReader.readLine();

        String[] hotelLineParts = hotelLine.split(" ");

        int position = Integer.parseInt(hotelLineParts[0]);
        double rating = Double.parseDouble(hotelLineParts[1]);

        nodes.add(new Node(position, rating));
    }

    bufferedReader.close();

    endNode = new Node(wayLenght);
    nodes.add(endNode);

    /*
     * Find best path
     */
    ArrayList<Node> openNodes = new ArrayList<Node>();
    ArrayList<Node> closedNodes = new ArrayList<Node>();

    openNodes.add(startNode);

    while(endNode.getPreviousNode() == null)
    {
        if(openNodes.size() == 0) return "Karte nicht lößbar";

        Node bestNode = null;

        /*
         * Find best Node
         */
```

Abbildung 7: solveMap-Methode Teil 1

```
        for(Node node : openNodes)
        {
            if(bestNode == null)
            {
                bestNode = node;
            }
            else if(rateNode(bestNode, startNode, 1) < rateNode(node, startNode, 1))
            {
                bestNode = node;
            }
            else if(rateNode(bestNode, startNode, 1) == rateNode(node, startNode, 1))
            {
                if(bestNode.getPosition() < node.getPosition())
                {
                    bestNode = node;
                }
            }
        }

        closedNodes.add(bestNode);
        openNodes.remove(bestNode);

        ArrayList<Node> nodesInReach = getReachebleNodes(bestNode, nodes);

        for(Node node : nodesInReach)
        {
            if(node.getPreviousNode() == null)
            {
                node.setPreviousNode(bestNode);
            }
            else
            {
                if(getPathLength(node, startNode) >= (getPathLength(bestNode, startNode) + 1))
                {
                    node.setPreviousNode(bestNode);
                    if(closedNodes.contains(node))
                    {
                        closedNodes.remove(node);
                        openNodes.add(node);
                    }
                }
            }

            if(node.getPreviousNode() != null && node.getPreviousNode().getPreviousNode() != null)
            {
                if(node.getPosition() - node.getPreviousNode().getPreviousNode().getPosition() <= 360)
                {
                    node.setPreviousNode(node.getPreviousNode().getPreviousNode());
                }
            }

            if(!closedNodes.contains(node) && !openNodes.contains(node))
            {
                openNodes.add(node);
            }
        }
    }

    return constructSolution(endNode, startNode, endNode);
}
```

Abbildung 8: solveMap-Methode Teil 2