[A10] DijkstraPQShortestPath

Aufgabe

Im Package A10_DijkstraPQShortestPath finden Sie einige Klassen zu Graphen. Nehmen Sie sich die
Zeit, diese Klassen durchzusehen, da sie als Grundgerüst auch bei der Klausur verwendet werden.

- **Hilfsklassen**
    - `Main`: Testklasse, die zum Aufrufen der anderen Klassen dient. Enthält die *main()*-Funktion zum Programmstart.
- **Klassen zum Speichern von Graphen**
    - `Graph`: Interface, das Funktionen zum Hinzufügen, Suchen und Entfernen von Kanten definiert. Die Parameter *u* und *v* der Funktionen sind die Nummern der Knoten (*u*: Start, *v*: Ziel). ***Achtung:*** In der Implementierung werden Knoten in aller Regel nur durch Integer-Zahlen repräsentiert, nicht durch eine eigene Klasse. Für Kanten gibt es die Klasse *WeightedEdge*.
    - `ListGraph`: Implementierung eines Graphen als ein Array von Listen (siehe Folien).
    - `ArrayGraph`: Implementierung eines Graphen als zweidimensionales Array (siehe Folien).
    - `WeightedEdge`: gewichtete Kante zwischen zwei Knoten.
- **Algorithmen**
    - `FindWay`: Abstrakte Basisklasse für alle Algorithmen, welche unter anderem bereits eine Hilfsfunktion enthält, um aus dem Vorgänger-Array eine Weg-Liste zu machen.
    - `DijkstraPQShortestPath`: Dijkstra-Algorithmus für lichte Graphen mit Priority-Queue (= Heap). Findet den kürzesten Weg zwischen zwei Punkten.
        - `Vertex`: Klasse für Knoten, die im Heap gespeichert werden. Neben der Knotennummer enthalten sie auch die aktuelle Priorität (bei Dijkstra: berechnete Entfernung des Knotens vom Startknoten; Variable *cost*)
        - `VertexHeap`: Heap für Knoten, der im Dijkstra-Algorithmus verwendet wird.
          Dijkstra verwendet vor allem die Funktion *setCost()*.

Implementieren Sie den Dijkstra-Algorithmus mit Heap (*DijkstraPQShortestPath*) anhand der Folien nach. Setzen Sie die beiden Arrays *pred[]* (Vorgänger) und *dist[]* (kumulierte Entfernung) entsprechend und verwenden Sie für den Heap die Klasse *VertexHeap*.

Lösung

Die Methode `findWay` ist der Ausgangspunkt für den *Algorithmus von Dijkstra*. Sie ruft die Initialisierung der Entfernungstabelle auf.

```
public List<Integer> findWay(int from, int to) {
  initPathSearch();
  if (!calculatePath(from, to)) {
    return null;
  }
  return createWay(from, to);
}
```

Die Methode `createWay` erstellt einen Weg aus der Tabelle der Vorgänger, die mit dem Feld `pred` implementiert ist.

```
protected ArrayList<Integer> createWay(int from, int to) {
  ArrayList<Integer> way = new ArrayList<Integer>();
  int i = to;
  while (i != from) {
    way.add(0, i);
    i = pred[i];
  }
  way.add(0, from);
  return way;
}
```

Die Methode `initPathSearch` erstellt die Entfernungstabelle. Sie ist mit dem Feld `dist` implementiert und hat am Anfang alle Werte auf die größtmögliche ganze Zahl vom Typ `Integer` gesetzt.

```
protected void initPathSearch() {
  int numv = graph.numVertices();
  dist = new int[numv];
  for (int i = 0; i < numv; i++) {
    dist[i] = Integer.MAX_VALUE;
  }
}
```

Der Algorithmus von Dijkstra wurde in der Methode `calculatePath` implementiert.

Zuerst werden die Knoten mit den ursprünglichen Entfernungen aus der initialisierten Entfernungstabelle als Prioritäten in die Priority-Queue eingefügt. Der Startknoten wird mit dem Wert 0 gespeichert. Anschließend werden Knoten aus der Priority-Queue rausgenommen bis sie leer ist.

Von jedem entfernten Knoten, werden die Nachbaren betrachtet. Die kumulierte Entfernung vom Startknoten wird mit dem Wert verglichen, der für jeden Nachbar in der Entfernungstabelle gespeichert wird. Falls die Entfernung vom Startknoten kleiner ist als die bis jetzt bekannte, dann wird die neue Entfernung in der Entfernungstabelle gespeichert und die Priorität in der Priority-Queue verringert. Somit werden immer diejenigen Knoten aus der Priority-Queue genommen, die am nächsten zum Startknoten sind. Zusätzlich wird auch der Vorgänger des Nachbarn in der Tabelle der Vorgänger aktualisiert, um den Weg am Ende erstellen zu können (`createWay`).

Abschließend wird überprüft, ob unser Endknoten erreicht wurde. Falls nicht wird der Wert `false` ausgegeben.

```
protected boolean calculatePath(int from, int to) {
  VertexHeap heap = new VertexHeap(graph.numVertices());
  dist[from] = 0;
  for (int i = 0; i < dist.length; i++) {
    Vertex v = new Vertex(i, dist[i]);
    heap.insert(v);
  }
  while (!heap.isEmpty()) {
    Vertex current = heap.remove();
    List<WeightedEdge> outgoingEdges = graph.getEdges(current.vertex);
    for (WeightedEdge outgoingEdge : outgoingEdges) {
      int distToHere = dist[current.vertex];
      int distToNext = outgoingEdge.weight;
      int distCumulative = distToHere + distToNext;
      int neighbor = outgoingEdge.to_vertex;
      if (distCumulative < dist[neighbor]) {
        dist[neighbor] = distCumulative;
        heap.setCost(neighbor, distCumulative);
        pred[neighbor] = current.vertex;
      }
    }
  }
  if (pred[to] == -1) {
    return false;
  }
  return true;
}
```

Laufzeit

Die Laufzeit dieser Implementierung des Algorithmus von Dijkstra ist *O((E + V) log V)*.

Die Priority-Queue wird in *O(V log V)* aufgefüllt, da jeder Knoten eingefügt werden muss, was im schlimmsten Fall eine Laufzeit von *O(log V)* ergibt.

Anschließend wird jeder Knoten in *O(1)* aus der Priority-Queue rausgenommen und dann werden alle Kanten aus diesem Knoten genommen, um zu seinen Nachbaren zu gelangen. Im schlimmsten Fall werden insgesamt alle Kanten im Graph verwendet. Wenn ein Nachbar erreicht wird, wird im schlimmsten Fall immer seine Priorität im Priority-Queue verringert. Dies erfolgt mit einer Laufzeit von *O(log V)*. Das bedeutet, dass nachdem alle Knoten aus dem Priority-Queue entfernt wurden, eine Laufzeit von *O(E log V)* erreicht wird.

Somit berechnet sich in Summe eine Laufzeit von *O((E + V) log)* für das Auffüllen der Priority-Queue und die Änderungen der Prioritäten der Knoten.