Aufgabe 3: Abbiegen

Team-ID: 00619

Team-Name: ncrypted

Bearbeiter/-innen dieser Aufgabe:
Oliver Schirmer

10. April 2020

Inhaltsverzeichnis

Lösungsidee

Grundlegend werden die Kreuzungen, die in Form von Koordinaten gegeben sind, als Knoten und die Straßen zwischen ihnen als Kanten betrachtet, wodurch ein Graph gegeben ist. Insbesondere ist dieser Graph planar, da das Überschneiden zweier Straßen zu einer weiteren Kreuzung führen würde und somit sichergestellt ist, dass es sich bei jedem Graphen um eine planare Einbettung handelt. Zur Berechnung der maximal möglichen Weglänge wird anfangs der kürzeste Weg benötigt. Zwischen den einzelnen Knoten lässt sich mit Hilfe der x- und y-Differenz und des Satz des Pythagoras ein Kantengewicht berechnen. Dadurch erhält man einen ungerichteten gewichteten Graphen mit einem Single-Source Shortest Path (SSSP) Problem. Dieses lässt sich in diesem Fall am schnellsten mit dem Dijkstra-Algorithmus lösen. Hat man die kürzeste Weglänge berechnet, lässt sich daraus auch die maximal gültige Weglänge berechnen, indem die eingegebene Prozentzahl der Wegverlängerung durch eine Verhältnisgleichung einberechnet wird. Daraufhin werden mittels rekursiver Tiefensuche alle vom Startknoten aus möglichen Pfade auf Gültigkeit überprüft. Um alle möglichen Pfade zu überprüfen, kann jedoch keine reguläre Tiefensuche genutzt werden. Diese muss bei jedem aktuell betrachteten Knoten eine komplett neue Tiefensuche starten, die lediglich Knoten betrachtet, die nicht bereits auf dem bis dahin genutzten Pfad liegen (statt wie üblich alle bereits einmal besuchten Knoten auszuschließen). Bei einem normalen, maximal dichten Graphen hätte dies eine Laufzeit von $O(\sum_{i=1}^{|V|} i+\frac{i*(i-1)}{2})=O(\frac{|V|^3+3|V|^2+2|V|}{6})$, also $O(|V|^3)$ zufolge. Da hier ein planarer Graph vorliegt, hat der Graph laut dem eulerschen Polyedersatz $|E|\leq 3|V|-6$ Kanten und somit eine Laufzeit von $O(\sum_{i=1}^{|V|} i+3i-6)=O(2|V|^2-4|V|)$ also $O(|V|^2)$. Zusätzlich kann die Anzahl der überprüften Pfade reduziert werden, indem bei jedem rekursiven Aufruf geprüft wird, ob der aktuelle Pfad beim Zielknoten angekommen ist und dieser Pfad gespeichert, falls die Anzahl der Abbiegungen und die Weglänge geringer als die des bisherigen besten Pfads sind, dieser Pfad also besser ist als der bisherige. Zudem wird bei jedem rekursiven Aufruf geprüft, ob die aktuelle Weglänge die zuvor berechnete maximale Weglänge überschreitet, die Anzahl der Abbiegungen größer oder bei selber Anzahl, die Weglänge größer als beim bisherigen besten Pfad ist. Trifft einer der drei Fälle zu, wird die Tiefensuche

abgebrochen, wodurch die Berechnung an diesem Punkt unnötiger Pfade vermieden wird. Die Anzahl der Abbiegungen jedes Pfads wird durch Winkelberechnung zwischen je zwei aufeinanderfolgenden Knoten (zweidimensionalen Koordinaten) ermittelt. D.h. während eines rekursiven Aufrufs, wird der Winkel zum vorherigen Knoten berechnet und bei Abweichung vom vorherigen Winkel die Anzahl der Abbiegungen inkrementiert.

Umsetzung

Beim Einlesen werden alle Kanten in einer Adjazenzliste gespeichert, wobei jede Kante zweimal, jeweils für beide Richtungen, gespeichert wird. Dadurch wird aus dem ungerichteten Graphen ein gerichteter zyklischer, worauf sich der Dijkstra-Algorithmus praktisch anwenden lässt. Da die Anzahl verschiedener Knoten nicht von Anfang an gegeben ist, findet sich schon beim Einlesen die erste Schwierigkeit: das dynamische Verwalten aller Knoten ohne Mehrfachspeicherung einer bestimmten Koordinate. Da jeder Knoten eindeutig durch seine Koordinaten gegeben ist, kann der Eingabestring eines jeden Knotens als Schlüssel betrachtet werden. D.h. neben einer ArrayList mit PointInts (Objekte, die die x- und y-Koordinate als Integer speichern) gibt es eine HashMap, die die Koordinaten-Strings zu den entsprechenden Listenindexes mappt. Dadurch erreichen wir konstante Laufzeit, da die Abfrage, ob ein Knoten bereits eingelesen wurde, als auch das Einfügen eines neuen Knotens in $O(1)$ möglich ist.

Beim Dijkstra-Algorithmus wird neben der Distanz auch der Vorgänger, durch den der kürzeste Weg zum jeweiligen Knoten erreicht wird, gespeichert. Dadurch lässt sich später der kürzeste Pfad rekonstruieren. Zur Verwaltung der Knoten wird eine Vorrangwarteschlange verwendet, wodurch wir eine Laufzeit von $O(|V| * \log(|V|))$ statt $O(|V|^2)$ erhalten.

Bei der Tiefensuche muss prinzipiell nur der aktuell betrachtete Pfad gespeichert werden. Dies wird mit Hilfe eines Stacks realisiert, da dieser das Hinzufügen, als auch das Entfernen des zuletzt hinzugefügten Knoten, in konstanter Laufzeit erlaubt. Zusätzlich wird ein Boolean-Array verwendet, um in konstanter Zeit überprüfen zu können, ob ein Knoten schon im aktuellen Pfad enthalten ist, was zur Implementierung des „Fast-Failure“ Systems benötigt wird. Als letztes ist ein zweidimensionales Double-Array zu nennen, welches lediglich als Cache für bereits berechnete Winkel dient. Normalerweise würde man den Winkel mit Hilfe des Arcustangens des Anstiegs berechnen. Bei näherer Betrachtung fällt allerdings auf, dass der Betrag des Anstiegs (Quotient aus x- und y-Differenz) schon ausreicht, um Abbiegungen erkennen zu können. Dadurch wird wiederum viel Laufzeit eingespart, da die Berechnung des Arcustangens relativ rechenaufwendig ist.

Alles in allem setzt sich die Laufzeit aus dem Dijkstra-Algorithmus und der DFS zusammen, wobei die Laufzeit der DFS schlechter ist und der gesamte Algorithmus eine Laufzeit von $O(|V|^2)$ aufweist.

Beispiele

Die folgenden Beispiele sind der BwInf-Webseite entnommen:

Beispiel 1: Abbiegen0			
10%	15%	30%	100%
---DIJKSTRA--- Länge: 5.8284 Maximale Weglänge: 6.4112 (+10.0%) Pfad: 0 2 5 7 9 1 ---TIEFENSUCHE--- Abbiegevorgänge: 3 Weglänge: 5.8284	---DIJKSTRA--- Länge: 5.8284 Maximale Weglänge: 6.7026 (+15.0%) Pfad: 0 2 5 7 9 1 ---TIEFENSUCHE--- Abbiegevorgänge: 2 Weglänge: 6.4142	---DIJKSTRA--- Länge: 5.8284 Maximale Weglänge: 7.5769 (+30.0%) Pfad: 0 2 5 7 9 1 ---TIEFENSUCHE--- Abbiegevorgänge: 1 Weglänge: 7.0 (+20.1%)	---DIJKSTRA--- Länge: 5.8284 Maximale Weglänge: 11.6568 (+100.0%) Pfad: 0 2 5 7 9 1 ---TIEFENSUCHE--- Abbiegevorgänge: 1 Weglänge: 7.0 (+20.1%)

(+0.0%) Pfad: 0 2 5 7 9 1 siehe Abb. 0.1*	(+10.05%) Pfad: 0 2 3 6 8 9 1 siehe Abb. 0.2*	Pfad: 0 2 3 4 6 8 9 1 siehe Abb. 0.3*	Pfad: 0 2 3 4 6 8 9 1 siehe Abb. 0.4*

Beispiel 2: Abbiegen1			
---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 18.8346 (+10.0%) Pfad: 0 10 2 3 18 34 28 13 49 50 29 51 53 1 ---TIEFENSUCHE--- Abbiegevorgänge: 5 Weglänge: 18.7147 (+9.29%) Pfad: 0 69 9 26 3 18 34 28 13 17 54 30 16 15 53 1 siehe Abb. 1.1*	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 19.6907 (+15.0%) Pfad: 0 10 2 3 18 34 28 13 49 50 29 51 53 1 ---TIEFENSUCHE--- Abbiegevorgänge: 5 Weglänge: 18.7147 (+9.29%) Pfad: 0 69 9 26 3 18 34 28 13 17 54 30 16 15 53 1 siehe Abb. 1.1	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 22.2591 (+30.0%) Pfad: 0 10 2 3 18 34 28 13 49 50 29 51 53 1 ---TIEFENSUCHE--- Abbiegevorgänge: 5 Weglänge: 18.7147 (+9.29%) Pfad: 0 69 9 26 3 18 34 28 13 17 54 30 16 15 53 1 siehe Abb. 1.1	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 34.2448 (+100.0%) Pfad: 0 10 2 3 18 34 28 13 49 50 29 51 53 1 ---TIEFENSUCHE--- Abbiegevorgänge: 5 Weglänge: 18.7147 (+9.29%) Pfad: 0 69 9 26 3 18 34 28 13 17 54 30 16 15 53 1 siehe Abb. 1.1

Beispiel 3: Abbiegen2			
---DIJKSTRA--- Länge: 10.8863 Maximale Weglänge: 11.9749 (+10.0%) Pfad: 0 23 2 3 18 14 43 56 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 11.0644 (+1.63%) Pfad: 0 23 22 3 18 27 31 43 56 1 siehe Abb. 2.1*	---DIJKSTRA--- Länge: 10.8863 Maximale Weglänge: 12.5193 (+15.0%) Pfad: 0 23 2 3 18 14 43 56 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 11.0644 (+1.63%) Pfad: 0 23 22 3 18 27 31 43 56 1 siehe Abb. 2.1	---DIJKSTRA--- Länge: 10.8863 Maximale Weglänge: 14.1522 (+30.0%) Pfad: 0 23 2 3 18 14 43 56 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 11.0644 (+1.63%) Pfad: 0 23 22 3 18 27 31 43 56 1 siehe Abb. 2.1	---DIJKSTRA--- Länge: 10.8863 Maximale Weglänge: 21.7726 (+100.0%) Pfad: 0 23 2 3 18 14 43 56 1 ---TIEFENSUCHE--- Abbiegevorgänge: 3 Weglänge: 15.9442 (+46.46%) Pfad: 0 9 77 60 47 32 36 39 74 51 56 1 siehe Abb. 2.2*

Beispiel 4: Abbiegen3			
---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 18.8346 (+10.0%) Pfad: 0 9 2 3 17 32 25 12 18 43 26 49 51 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 17.8863 (+4.46%) Pfad: 0 56 8 23 3 17 32 25 13 16 52 27 15 14 51 1 siehe Abb. 3.1*	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 19.6907 (+15.0%) Pfad: 0 9 2 3 17 32 25 12 18 43 26 49 51 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 17.8863 (+4.46%) Pfad: 0 56 8 23 3 17 32 25 13 16 52 27 15 14 51 1 siehe Abb. 3.1	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 22.2591 (+30.0%) Pfad: 0 9 2 3 17 32 25 12 18 43 26 49 51 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 17.8863 (+4.46%) Pfad: 0 56 8 23 3 17 32 25 13 16 52 27 15 14 51 1 siehe Abb. 3.1	---DIJKSTRA--- Länge: 17.1224 Maximale Weglänge: 34.2448 (+100.0%) Pfad: 0 9 2 3 17 32 25 12 18 43 26 49 51 1 ---TIEFENSUCHE--- Abbiegevorgänge: 4 Weglänge: 17.8863 (+4.46%) Pfad: 0 56 8 23 3 17 32 25 13 16 52 27 15 14 51 1 siehe Abb. 3.1

In Beispiel 1 wird sehr deutlich, welche Auswirkung die Veränderung des Parameters der maximalen Verlängerung hat, da bei den ersten drei Varianten jeweils ein anderer Weg mit weniger Abbiegungen gewählt

werden kann. Dies ist in den Beispielen 2-4 anders, da sich dort keine Veränderung zeigt. Lediglich bei Beispiel 3 bei der Verlängerung um maximal 100% ergibt sich ein Weg mit weniger Abbiegungen.

Beispiel 5: Abbiegen4
---DIJKSTRA--- Länge: 2.0 Maximale Weglänge: 2.2 (+10.0%) Pfad: 0 8 1 ---TIEFENSUCHE--- Abbiegevorgänge: 0 Weglänge: 2.0 (+0.0%) Pfad: 0 8 1 siehe Abb. 4.1*

Obiges Beispiel ist ein selbst erstelltes Beispiel und verdeutlicht den Edge-Case, der bei der Lösungsidee schon beschrieben wurde. Hier ist ein dichter Graph gegeben, bei dem erst der letzte betrachtete Pfad der optimale und sogar einzig mögliche ist. Dies liegt hauptsächlich an der Reihenfolge der Kanten, wobei die Kante von (0,0) zu (0,1) als letzte der Adjazenzliste hinzugefügt wird. Wie bereits gesagt könnte dieses Problem durch Festlegen des zuvor berechneten Dijkstra-Pfads als Grundlage umgangen werden, wobei auch hier einige unnötige Pfade betrachtet werden.

* die Abbildungen befinden sich im Ordner „Abbiegen_Abb“

Quellcode

```
public class Abbiegen {

    // General
    public static double extension = -1;
    public static int startNode;
    public static int endNode;
    public static List<PointInt> nodes = new ArrayList<>();
    private static Map<String, Integer> coordsToNodeIdx = new
HashMap<>();
    // node id -> node id, length
    public static List<List<Pair<Integer, Double>>> streets = new
ArrayList<>();
    // Dijkstra
    public static List<Integer> shortestPath;
    public static double shortestPathLen;
    public static double maxPathLen;
    // DFS
    public static int minTurns = Integer.MAX_VALUE;
    public static double minTurnDist = Double.MAX_VALUE;
    public static ArrayList<Integer> turnPath;

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

        // execute algorithm
        executeAlgorithm();

    private static void executeAlgorithm() {
        // Dijkstra
        Pair<Double, List<Integer>> shortestPathPair =
Dijkstra.dijkstra(nodes, streets, 0, 1);
        shortestPathLen = shortestPathPair.getKey();
        shortestPath = shortestPathPair.getValue();
        maxPathLen = shortestPathLen * (100 + extension) / 100;
        // DFS
        DFS.setup(streets);
```

```
            DFS.dfs(streets, 0, 1, 0, 0, -1);
        }

        private static void readFileData(String fileName) {
            // read street count, start node, end node …
            // when handling a street:
            int xDif = Math.abs(nodes.get(u).getX() -
    nodes.get(v).getX());
            int yDif = Math.abs(nodes.get(u).getY() -
    nodes.get(v).getY());
            double len = Math.sqrt(Math.pow(xDif, 2.0D) +
    Math.pow(yDif, 2.0D));
            streets.get(u).add(new Pair<>(v, len));
            streets.get(v).add(new Pair<>(u, len));
        }

        private static int getNode(String coords) {
            if (coordsToNodeIdx.get(coords) != null) {
                return coordsToNodeIdx.get(coords);
            }
            String[] commaSplit = coords.split(",");
            int x = asInt(commaSplit[0].substring(1), 0);
            int y = asInt(commaSplit[1].substring(0,
    commaSplit[1].length() - 1), 0);
            nodes.add(new PointInt(x, y));
            coordsToNodeIdx.put(coords, nodes.size() - 1);
            return nodes.size() - 1;
        }
    }

    public class Dijkstra {

        public static Pair<Double, List<Integer>>
    dijkstra(List<PointInt> nodes, List<List<Pair<Integer, Double>>>
    graph, int start, int end) {
            List<Double> dist = new ArrayList<>(nodes.size());
            List<Integer> pred = new ArrayList<>(nodes.size());
            for (int i = 0; i < nodes.size(); i++) {
                dist.add(Integer.MAX_VALUE + 0.0);
                pred.add(-1);
            }
            dist.set(start, 0.0D);
            pred.set(start, -1);

            PriorityQueue<Pair<Double, Integer>> pq = new
    PriorityQueue<>((p1, p2) -> p1.getKey() > p2.getKey() ? 1 : -1);
            pq.add(new Pair<>(0.0D, start));
            while (!pq.isEmpty()) {
                int u = pq.peek().getValue();
                double uDist = pq.poll().getKey();
                if (dist.get(u) != uDist) {
                    continue;
                }
                for (Pair<Integer, Double> edge : graph.get(u)) {
                    int v = edge.getKey();
                    double w = edge.getValue();
                    if (uDist + w < dist.get(v)) {
                        dist.set(v, uDist + w);
                        pred.set(v, u);
                        pq.add(new Pair<>(uDist + w, v));
                    }
```

```
                }
            }
            List<Integer> path = constructPath(pred, start, end);
            return new Pair<>(dist.get(end), path);
        }

        private static List<Integer> constructPath(List<Integer> pred,
    int start, int end) {
            List<Integer> path = new ArrayList<>();
            int cur = end;
            while (cur != start) {
                path.add(cur);
                cur = pred.get(cur);
            }
            path.add(start);
            Collections.reverse(path);
            return path;
        }
    }

    public class DFS {

        private static double[][] angles;
        private static Stack<Integer> path;
        private static boolean[] inStack;

        public static void setup(List<List<Pair<Integer, Double>>>
    graph) {
            int n = graph.size();
            angles = new double[n][n];
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    angles[i][j] = 90;
                }
            }
            path = new Stack<>();
            inStack = new boolean[n];
        }

        public static void dfs(List<List<Pair<Integer, Double>>>
    graph, int curNode, int targetNode, int turns, double wayLen,
    double lastAngle) {
            path.push(curNode);
            inStack[curNode] = true;
            if (curNode == targetNode) {
                Abbiegen.minTurns = turns;
                Abbiegen.minTurnDist = wayLen;
                Abbiegen.turnPath = new ArrayList<>(path);
                path.pop();
                inStack[curNode] = false;
                return;
            }
            for (Pair<Integer, Double> edge : graph.get(curNode)) {
                int nextNode = edge.getKey();
                double edgeLen = edge.getValue();
                if (inStack[nextNode]) {
                    continue;
                }
                double newAngle = angles[curNode][nextNode];
                if (newAngle == 90) {
```

```
                angles[curNode][nextNode] =
Abbiegen.nodes.get(curNode).getAngleTo(Abbiegen.nodes.get(nextNode
));
                newAngle = angles[curNode][nextNode];
            }
            int newTurns = (newAngle == lastAngle || lastAngle ==
-1) ? turns : turns + 1;
            double newWayLen = wayLen + edgeLen;
            if (newTurns > Abbiegen.minTurns || newWayLen >
Abbiegen.maxPathLen ||
                    (newTurns == Abbiegen.minTurns && newWayLen >=
Abbiegen.minTurnDist)) {
                continue;
            }
            dfs(graph, nextNode, targetNode, newTurns, wayLen +
edgeLen, newAngle);
        }
        path.pop();
        inStack[curNode] = false;
    }
}
```