Dykstra

Finn Jannsen, Philipp Schwarz

3. Juni 2019

Inhaltsverzeichnis

1 Einführung **1**

2 Implementation **1**
2.1 Aufbau 1
2.2 Algorithmus 1

3 Testen und Verifikation **3**
3.1 Verifizieren 3
3.2 Aufwandsanalyse 3

1 Einführung

Diese Dokumentation beschreibt die Implementation von Dykstra für einen gewichteten Graphen aus einer Adjazenzmatrix, als auch einer Adjazenzliste. In Abschnitt 2 wird darauf eingegangen, wie der Algorithmus realisiert wurde. Anschließend wird in Abschnitt 3.1 geprüft, ob die Implementation korrekt funktioniert und in Abschnitt 3.2 die Performance auf den beiden Graphen verglichen.

2 Implementation

Der Algorithmus wurde als Klasse realisiert, die ein Interface implementiert, welches es ermöglicht, einfach weitere Implementationen, sofern dies gewünscht wird, zu schreiben.

2.1 Aufbau

Um die Struktur zu verdeutlichen, ist unter 2.1 ein UML-Diagramm zu sehen, welches die Verbindungen der einzelnen Klassen und Interfaces beinhält. Die Adjazenzmatrix und Adjazenzliste sind hier jeweils GraphMatrix und GraphList.

Die Adjazenzmatrix beinhält einen 2-Dimensionalen Array zum Speichern des Graphen, während die Adjazenzliste eine 1-Dimensionale Liste von Nodes ist, welche jeweils auf ihre Kanten verweisen.

2.2 Algorithmus

Der eigentliche Algorithmus zum Berechnen der schnellsten Pfade mit Dijkstra ist unter 3.2 zu sehen. Bei diesem werden zuerst alle Nodes des Graphen initialisiert und anschließend

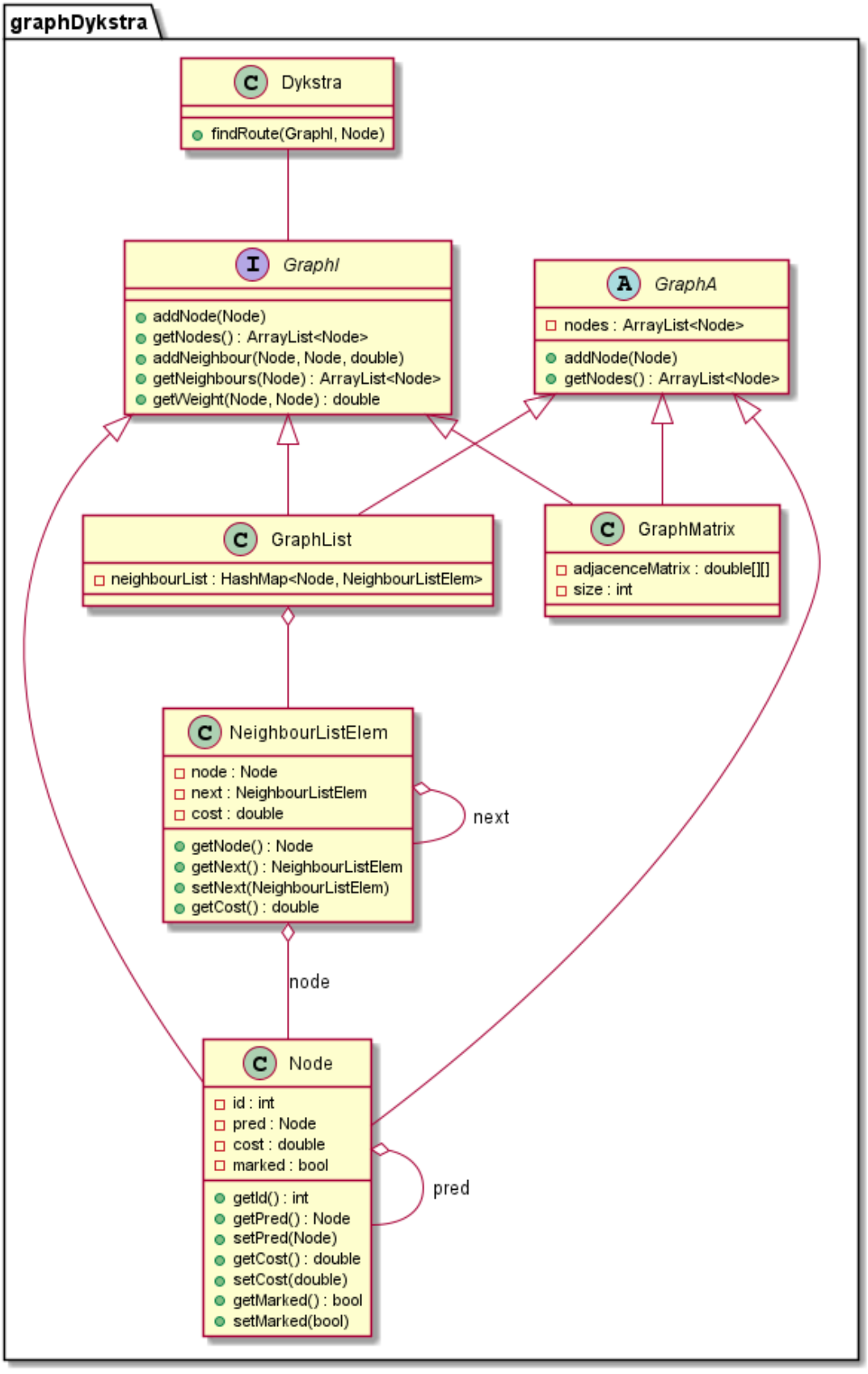

Abbildung 2.1: UML-Diagramm

alle Nachbarn des Startknoten in eine PriorityQueue, die nach den Kosten um zu dem Knoten zu kommen, priorisiert ist. Danach wird die PriorityQueue durchgearbeitet, jeder Nachbar des aktuellen Knoten hinzugefügt und bei jedem Nachbarn geprüft, ob der Weg über den aktuellen Knoten schneller ist, als bisher bekannt. Wenn dies der Fall ist, wird sich das neue Gewicht und der aktuelle Knoten im Nachbar gemerkt. In jedem Fall wird der aktuelle Knoten als abgearbeitet makiert. Am Ende, wenn die PriorityQueue leer ist, steht sicher, dass der kürzeste Weg vom Endknoten der ist, der herauskommt, wenn man den gespeicherten vorherigen Knoten folgt.

3 Testen und Verifikation

3.1 Verifizieren

Der Algorithmus wurde auf seine korrekte Funktionalität getestet. Hierzu zählt das Hinzufügen mittels randomisierter Sequenzen, die Anwendung von Dykstra und das Überprüfen des kürzesten Weges. Alle Tests wurden erfolgreich mit unterschiedlichen Eingabewerten absolviert.

3.2 Aufwandsanalyse

Für die Aufwandsanalyse wurden die beiden Graphen mit 10^n Werten gefüllt, wobei die Werte randomisiert und $n = 1, ..., 4$ war (Der Arbeisspeicher für 10^5 war ungenügend. Für Durschnittswerte wurden die Tests 30 mal durchgeführt. Beobachtet wurden die Rechenoperationen bei der Ausführung von Dykstra. Die Ergebnisse sind unter 3.1 ersichtlich. Anhand der Anzahl an Operationen lässt sich der Asymptotische Zeitaufwand annähernd wie folgt eingrenzen:

$$T(n) = \mathcal{O}(n * ln(n))$$

Auch kann man sehen, dass die Anzahl an Rechenoperationen bei der Adjazenzliste höher ist, als die der Adjazenzmatrix. Dafür braucht die Adjazenzmatrix jedoch aufgrund der Array-Allokierung wesentlich mehr Speicher. Unter anderem wurde festgestellt, dass sich zwischen 10^3 und 10^4 Nodes fast nur noch eine 10-er Potenz hinzukommt.

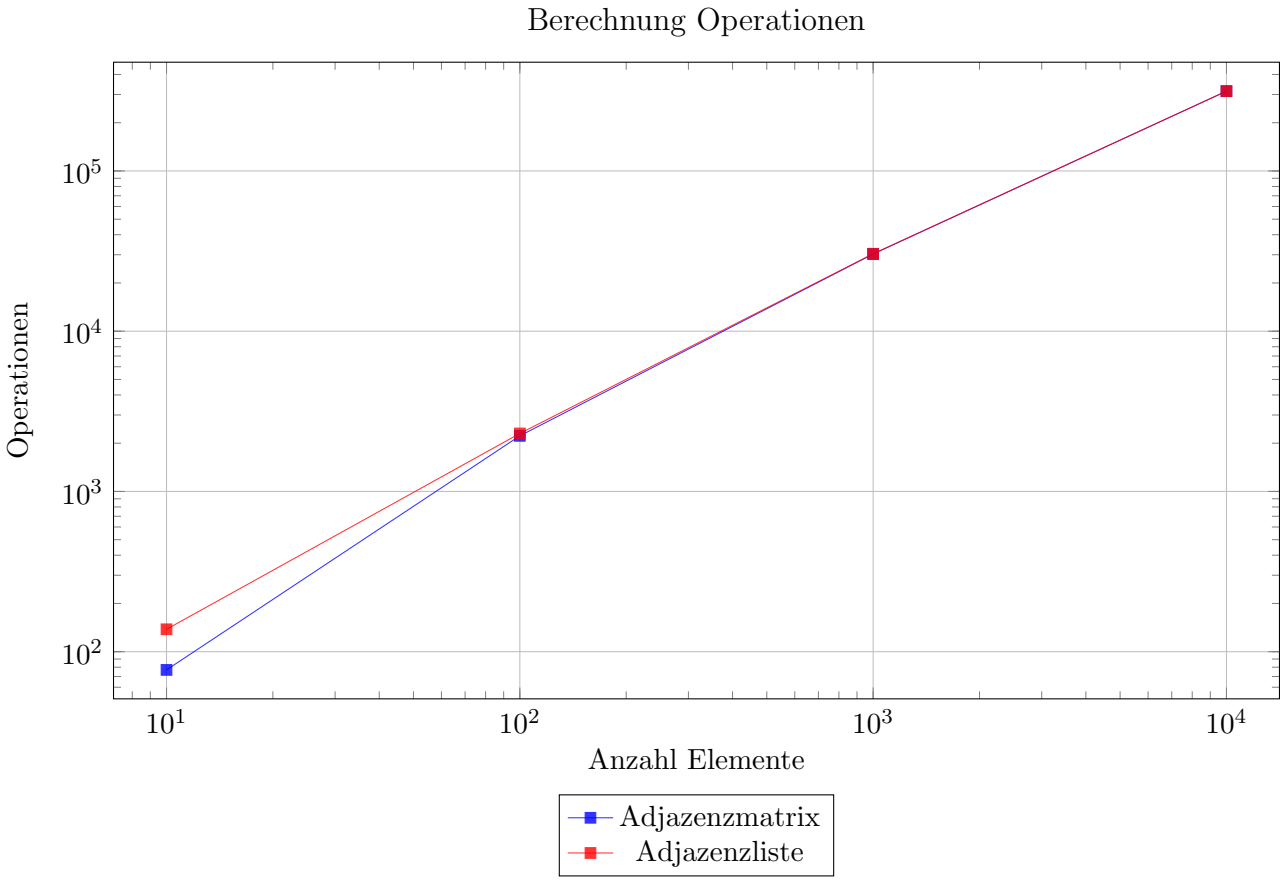

Abbildung 3.1: Quantitativer Vergleich von Adjazenzmatrix und Adjazenzliste

```
public static void findRoute(GraphI graph, Node start, Counter counter) {
    Queue<Node> rand = new PriorityQueue<>();

    // 1
    // init every node
    for(Node n : graph.getNodes()) {
        n.setPred(null);
        n.setCost(Double.MAX_VALUE);
        n.setMarked(false);
    }
    // init start node
    start.setPred(start);
    start.setCost(0.0);
    start.setMarked(true);

    // 2
    // fill rand with adjacence nodes of start and init the neighbour nodes of start
    for(Node n : graph.getNeighbours(start)) {
        if (counter != null) {
            counter.increaseCount();
            counter.increaseCount();
        }
        n.setCost(graph.getWeight(start, n));
        n.setPred(start);
        rand.add(n);
    }

    // 3
    // process every node starting with the lowest cost one
    // (automatically, since this is a prioriry queue, sorted after the cost
    while(!rand.isEmpty()) {
        if (counter != null) {
            counter.increaseCount();
        }
        Node v = rand.poll();
        v.setMarked(true);

        // process every node in rand that is not marked
        for(Node k : graph.getNeighbours(v)) {
            // 3.1
            if (!k.getMarked()) {
                // 3.2
                if (counter != null) {
                    counter.increaseCount();
                    counter.increaseCount();
                }
                double VKCost = v.getCost() + graph.getWeight(v, k);
                if(k.getCost() > VKCost) {
                    // check if the cost to go to this node via the previous node
                    // is faster than currently known
                    k.setCost(VKCost);
                    k.setPred(v);
                }
                // 3.3
                rand.add(k);
            }
        }
    }
}
```


Abbildung 3.2: Code-Ausschnitt Dykstra