AD Praktikum: Aufgabe 09, Graph and Dykstra

Sönke Peters
Karl-Fabian Witte

6. Juni 2017

Abstract

Die Datenstruktur eines gewichteter Graphen wird auf Adjazenzmatrix und -liste implementiert und der Algorithmus von Dykstra wird auf diesen mit einer Komplexitätsuntersuchung ausgewertet.

1 Aufgabenstellung

Es sollen zwei Implementationen von dem abstrakten Datentyp Graph realisiert werden. Die Komplexität des Dykstras Algorithmus ist auf beiden Graphenimplementationen zu messen. Die für den Algorithmus wichtigen zusatzinformationen sind nicht im Graphen selbst gespeichert. Zudem sollen zufallsgeneriertere Graphen erstellt werden, welche für die Messung der Komplexität verwedet werden.

1.1 Grapheninterface und Implementationsarten

1.1.1 Adjazenzmatrix

Eine Adjazenzmatrix eines Graphen ist eine Matrix, die speichert, welche Knoten des Graphen durch eine Kante verbunden sind. Sie besitzt für jeden Knoten eine Zeile und eine Spalte, woraus sich für n Knoten eine n×n-Matrix ergibt. Ein Eintrag in der i-ten Zeile und j-ten Spalte gibt hierbei an, ob eine Kante von dem i-ten zu dem j-ten Knoten führt. Steht an dieser Stelle eine -1, ist keine Kante vorhanden. Steht an dieser Stelle eine 0, ist keine Kante vorhanden, da der Knoten sonst eine Kante auf sich selbst führen würde, was wir ausschließen. Ist der Wert größer als 0 so gibt er die Kosten für diesen Weg an.

1.1.2 Adjazenzliste

Eine Adjazenzliste eines Graphen ist eine Liste, die speichert, welche Knoten des Graphen durch eine Kante verbunden sind. Jeder Knoten hat seine eigene Liste, die seine Nachbarn beinhaltet. In unserem Fall speichert die Liste außerdem noch die Kosten zu den Nachbarn.

1.2 Graphengenerator

Im Graphengenerator gibt es zwei Methoden. Die Methode $publicstaticint[][]genNonDirectionMatrix(intSize)$ erstellt aus einer vorgegebenen Knotenanzahl einen Graphen in Matrix Darstellung. Die Methode $publicstaticList<NodeEdgeList>genListGraph(int[][]matrix, List<Node<?>>nodes)$ erstellt aus einer Adjazensmatrix und einer Liste aller Knoten im Graph, die Adjazenslistendarstellung des Graphen. Es wird somit um einen Graphen in beiden Darstellungen zu erhalten, erst die Methode $genNonDirectionMatrix$ aufgerufen und mit ihrem Rückgabewert kann die Methode $genListGraph$ aufgerufen werden.

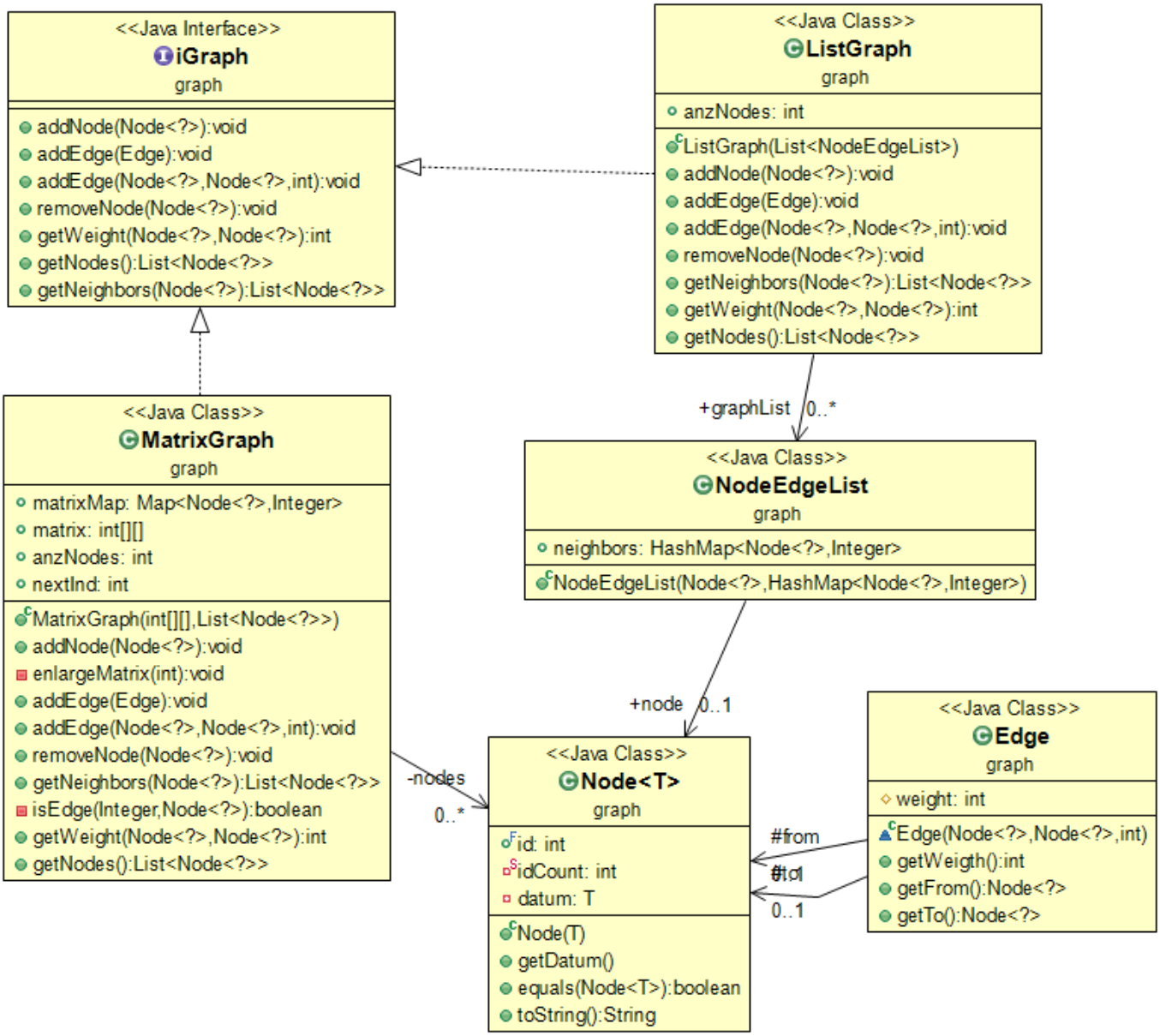

Abbildung 1: UML Diagramm, der Implementation der Graphendarstellungen

1.3 Algorithmus von Dykstra

Wir verzichten den Algorithmus von Dykstra zu erklären. Dies ist bereits im Skript behandelt worden.

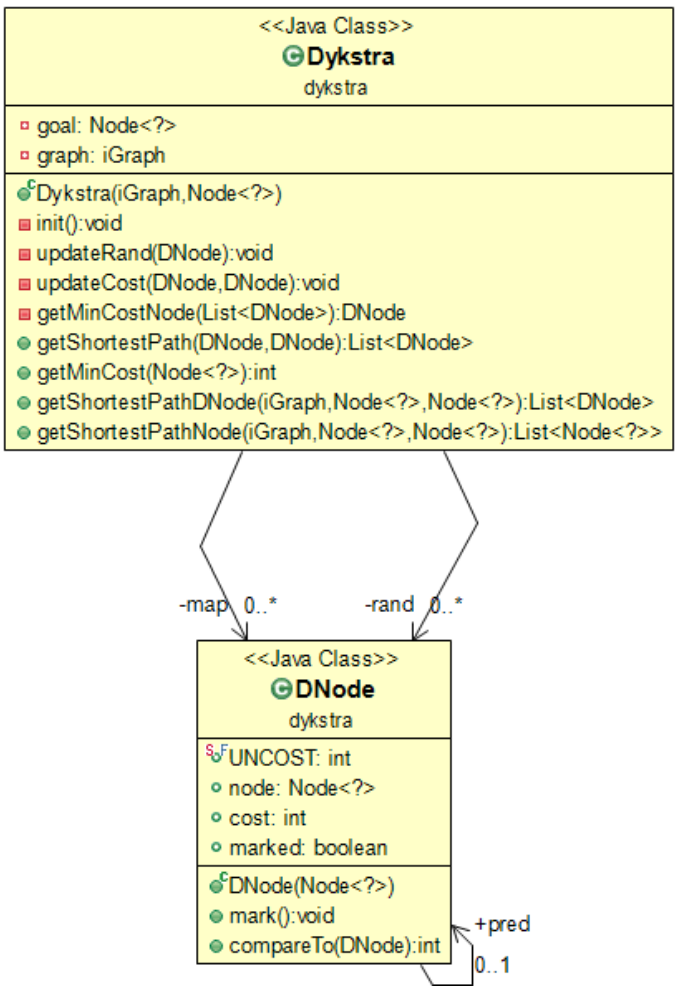

Abbildung 2: UML Diagramm der Dykstra Implementierung

1.4 Zählerintegration

Aufgrund von Zeitgründen wurde kein schlaues Pattern für die Zählerintegration verwendet. Er wurde deswegen einfach in die innersten Schleifen eingefügt. Es wurden nur die Graphenmethoden `getNeigbors()` und `getWeight()` getrackt, da diese im Dykstra verwendet werden.

2 Messung

Für die Messung wurde wegen Resourcenknappheit nur bis 10^3 gemessen, dafür aber die drei ersten Knoten genommen und aus deisen der Mittelwert genommen. Zum Graphen selber ist zu sagen, dass jeder Knoten zu 50% der andern Knoten eine verbindung ausweist.

Auf den ersten Blick sehen alle Abbildung 3, 4 und 5 gleich aus, dies liegt aber an der logarithmischen Darstellung. Die der Algorithmus mit dem Listengraphen weißt eine Komplexität von $O(n^3)$ und die mit

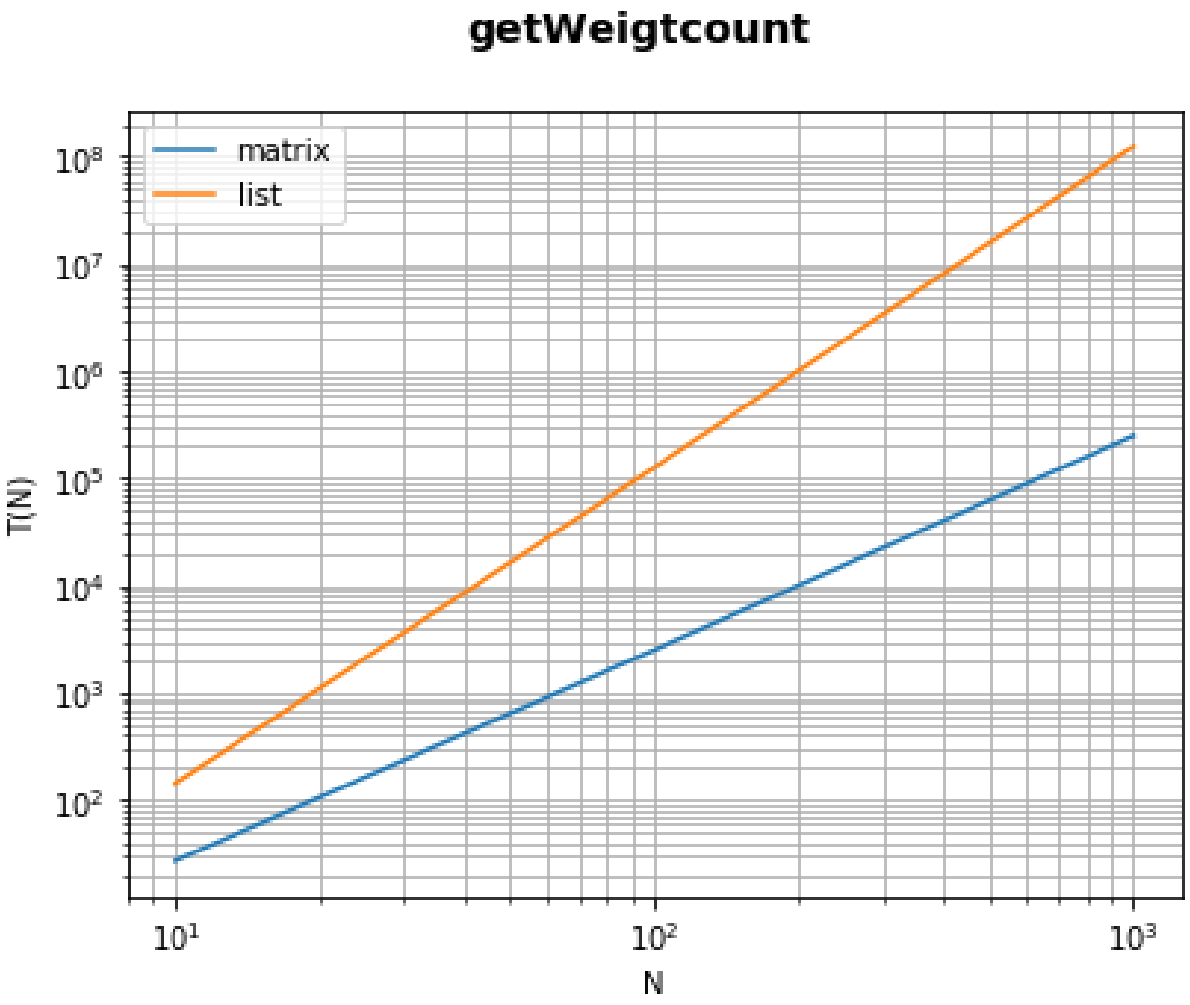

Abbildung 3: Durchläufe der inneren Schleife von getWeight() während Pathfindung mit Dykstra in Abhängigkeit von der Knotenanzahl N

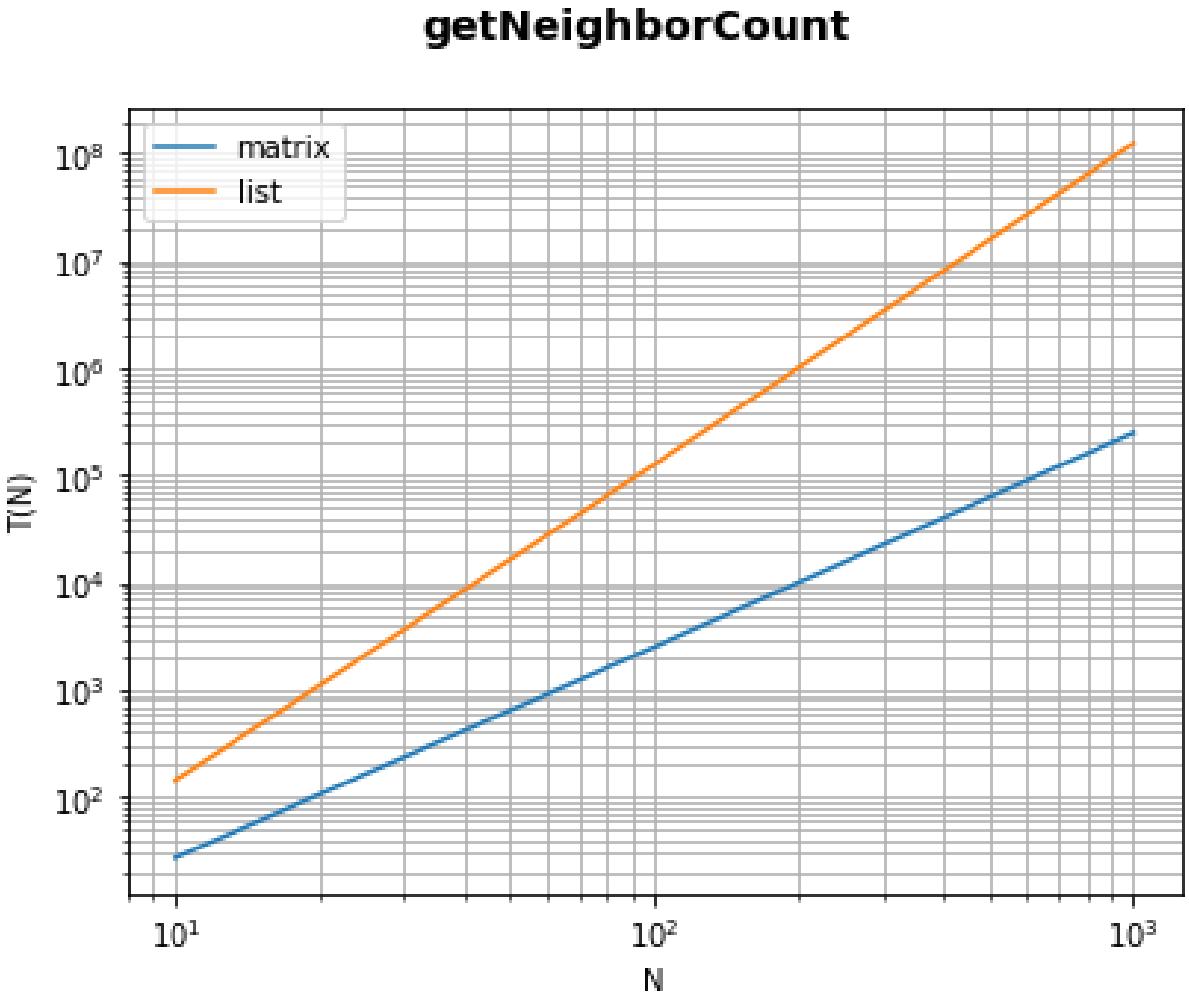

Abbildung 4: Durchläufe der inneren Schleife von getNeighbors() während Pathfindung mit Dykstra in Abhängigkeit von der Knotenanzahl N

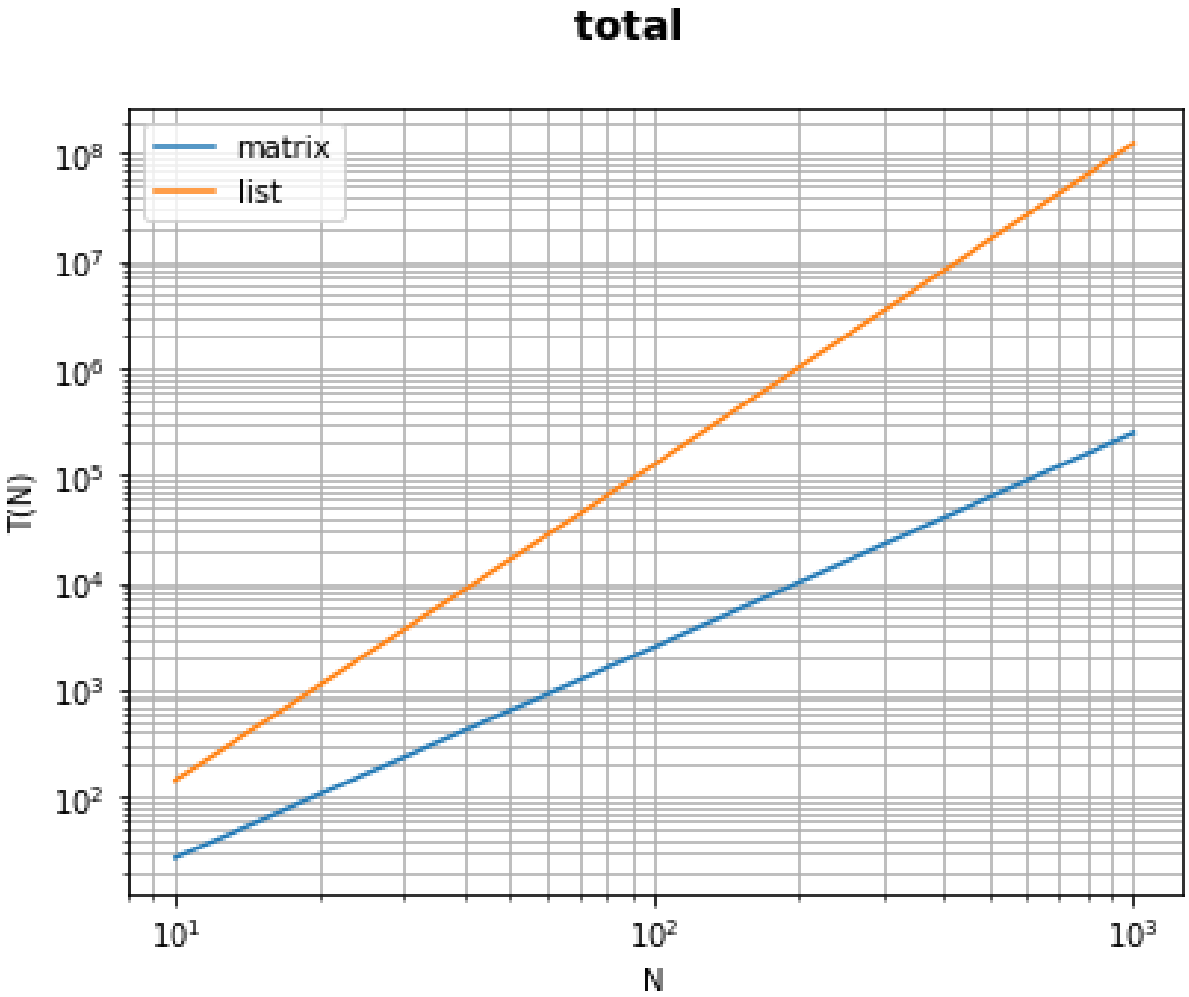

Abbildung 5: getNeighbors() + getWeight()

dem MatrixGraphen $O(n^2)$. Dies ist ein wenig widererwartend, da von der Liste eine schnellere Rückgabe der Nachbarn erwartet wird. Eventuell haben wir uns hier vertan.

3 Anhang

Es ist das Javadoc für die Documentation des Codes zu betrachten.