Lehrstuhl Algorithmen
und Komplexität
Prof. Dr. Fabian Kuhn
Sebastian Daum, Oleksii Saukh

Algorithmen und Datenstrukturen
(Informatik II) SS 2014

http://ac.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Übungsblatt 7

Abgabe: Mittwoch, 2. Juli, 23:59 Uhr

Laden Sie Ihre Lösungen für theoretische *Übungen im Format "loesung-xx-y.pdf" hoch, wobei "xx" die Nummer des aktuellen Übungsblattes ist (bei Bedarf mit führender Null) und "y" die Aufgabennummer, d.h., erzeugen Sie* pro (theoretische) Aufgabe *eine* .pdf*-Datei. Scans, Bilder von Scans etc. bitte zuerst ins pdf-Format umwandeln. Bei Programmieraufgaben müssen Sie – wann immer es relevant/durchführbar ist – Unit Tests sowie einen Style-check durchführen. Laden Sie außerdem eine Datei "erfahrungen.txt" hoch, in welcher Sie Ihre Erfahrungen und Meinungen zu dem Übungsblatt teilen.*

Aufgabe 1 (5 Punkte)

Implementieren Sie die Repräsentation von Graphen via Adjazenzlisten. Programmieren Sie dazu eine Klasse *Graph*, welche ein Array von Adjazenzlisten enthält, und zu welchem man Knoten via *addNode* und Kanten via *addEdge* hinzufügen kann.

Aufgabe 2 (5 Punkte)

Auf dem *public* SVN Verzeichnis zu diesem Übungsblatt ist ein Verzeichnis *mazes*, in welchem Sie mehrere ASCII-Dateien finden: diverse Testdateien und die Datei *inputmaze.log*, die offizielle Eingabedatei. Jede dieser Dateien stellt ein quadratisches Labyrinth da. Die erste Zeile in jeder dieser Dateien sagt ihnen die Seitenlänge des Labyrinths, z.B. l, und in den nächsten l Zeilen befinden sich strings aus jeweils l Zeichen. Jedes Labyrinth besteht aus Wänden (mit '1' gekennzeichnet) und Gehwegen (mit '0' gekennzeichnet), hat einen Startpunkt ('s') irgendwo in der Mitte des Labyrinths und einen Ausgang ('e') am Rand des Labyrinths, wobei 's' und 'e' auch als Gehweg zählen.

In *mazes* wird ein Programm zur Verfügung gestellt - *MazeConverter.class*, mit dem so ein ASCII Labyrinth visualisiert werden kann. Um eine *.log* Datei zu visualisieren, führen Sie Folgendes im Terminal aus: *java MazeConvert <fileName>.log*. Der *MazeConverter* erstellt eine neue Datei *<fileName>.log.png* die das Bild des Labyrinths enthält. (Sie müssen zum Ausführen von *MazeConverter* Java Runtime Environment installiert haben.)

In dieser Übung sollen Sie Ihre Implementierung aus der vorhergehenden Aufgabe anwenden, um das Labyrinth *inputmaze.log* als Graph abzuspeichern. Nullen (Gehwege) sollen dabei einzelne Knoten sein, und nebeneinanderliegende Nullen sind als benachbart zu betrachten, d.h., sind durch eine Kante verbunden. Beachten Sie, dass ca. 50% aller ASCII Zeichen Nullen sind, d.h., der Graph hat etwa $l^2/2$ Knoten.

Erweitern Sie, falls nötig, Ihre Implementierung aus der ersten Aufgabe, um Informationen wie Koordinaten oder Knotentyp (Gehweg, Startpunkt, Ausgang) mitabspeichern zu können.

Überlegen Sie sich außerdem eine Lösung (und programmieren Sie diese), um bei Kenntnis der Koordinaten (i, j) eines Knotens *schnell* auf den entsprechenden Knoten in Ihrem Array (mit den angehängten Adjazenzlisten) zugreifen zu können.

Hinweis: Es gibt diverse Lösungen (uns sind 4 sehr verschiedene bekannt), um die zuletzt erwähnte Anforderung zu erfüllen. Zwei dieser Lösungen, eine davon sehr elegant, können erziehlt werden, indem man *jedes* ASCII Zeichen als Knoten betrachtet, auch Wände – die kann man ja problemlos als isolierte Knoten betrachten und abspeichern. Programmieren Sie nicht blind drauflos, es hilft hier wirklich, eine Weile nachzudenken.

Aufgabe 3 (5 Punkte)

Implementieren Sie sowohl *BFS* als auch *DFS* als Methode der Graphenklasse aus Aufgabe 1, wobei beiden Prozeduren ein *Source*-Knoten r als Eingabe übergeben werden soll. Pseudo-Code zu beiden Prozeduren gibt es bei Bedarf auf den Vorlesungsfolien.

Zum Testen können Sie ja mal Ihr in der vorhergenden Aufgabe eingelesenes Labyrinth traversieren....

Aufgabe 4 (5 Punkte)

Kombinieren Sie die 3 vorhergehenden Teilaufgaben, um mit DFS einen Ausweg aus dem Labyrinth zu finden, d.h., starten Sie eine Traversierung des Baumes beim Startknoten s, bis Sie auf e stoßen. Nutzen Sie den DFS-Baum, um den Pfad P von s zu e zu bekommen. Generieren Sie eine Datei *dfs_outputmaze.log*, welche identisch zur Datei *inputmaze.log* ist, außer dass alle Nullen, welche Knoten auf dem Pfad P entsprechen, durch ein y ersetzt werden sollen (verfahren Sie mit Startpunkt und Ausgang nach Belieben).

Finden Sie dann analog mit Hilfe von BFS den *kürzesten* Weg aus dem Labyrinth und erstellen Sie eine Datei *bfs_outputmaze.log*, in welcher alle betreffenden Nullen durch x ersetzt sind.

Sie können *MazeConverter* auch auf die beiden Ausgabedateien anwenden, um sich die Pfade farblich darstellen zu lassen.

Bemerkung: Ein Labyrinth mit Seitenlänge l hat $n \in \Theta(l^2)$ Knoten, eine Adjazenzmatrix würde demnach $\Theta(l^4) = \Theta(n^2)$ an Speicher belegen. Ein Knoten selbst hat aber maximal 4 Nachbarn, d.h., die meisten Einträge in der Adjazenzmatrix wären Nullen. In diesem Fall belegen Adjazenzlisten daher nur $O(n)$ Speicherplatz, um einen Faktor in $\Theta(n)$ weniger als die Adjazenzmatrix.