[A08] Tiefensuche: Zusammenhängender Graph

Aufgabe

Die Tiefensuche (depth-first search) ist ein für Bäume und Graphen gebräuchlicher Algorithmus. Während bei der Breitensuche ausgehend von einem Startknoten alle über die Kanten erreichbaren Knoten in eine Queue eingefügt und der Reihe nach abgearbeitet werden, kann die Tiefensuche entweder rekursiv oder mit Hilfe eines Stacks (anstatt der Queue) erfolgen.

Die Tiefensuche hat für Graphen gleich wie die Breitensuche eine Laufzeit von O(V+E), da für einen vollständigen Durchlauf im schlechtesten Fall alle Knoten und Kanten je einmal besucht bzw. überprüft werden müssen. Schreiben Sie einen Algorithmus, der mit Hilfe der Tiefensuche feststellt, ob ein Graph zusammenhängend ist bzw. aus wie vielen zusammenhängenden Komponenten ein Graph besteht.

Dort finden Sie auch JUnit-Tests, mit denen Sie Ihre Lösung überprüfen können. Die Methode `getNumberOfComponents(Graph g)` soll die Anzahl der zusammenhängenden Komponenten eines Graphs retournieren. Ist der Graph vollständig zusammenhängend, liefert die Funktion den Wert **1**.

Möglicher Algorithmus

1. Starten Sie die Tiefensuche an einem beliebigen Knoten
2. Markieren Sie im Zuge der Tiefensuche alle besuchten Knoten
3. Überprüfen Sie mit einer gesonderten Schleife, ob alle Knoten besucht worden sind

- Wenn ja: fertig.
- Wenn nein: Graph ist nicht zusammenhängend, neue Zusammenhangskomponente gefunden. Neue Tiefensuche bei einem unbesuchten Knoten starten. Weiter mit Schritt 2.

4. Retournieren der Anzahl der gefundenen Komponenten

Welche Laufzeit hat dieser Algorithmus?

Lösung

```java
public int getNumberOfComponents(Graph g) {
  int componentCount = 0;
  int nV = g.numVertices();
  boolean[] wasVisited = new boolean[nV];
  for (int i = 0; i < nV; i++) {
    if (!wasVisited[i]) {
      wasVisited[i] = true;
      Stack<Integer> stack = new Stack<>();
      stack.push(i);
      while(true) {
        try {
          int currentVertex = stack.pop();
          List<WeightedEdge> edgesToNeighbors = g.getEdges(currentVertex);
          for (WeightedEdge e : edgesToNeighbors) {
            int j = e.to_vertex;
            if (!wasVisited[j]) {
              stack.push(j);
              wasVisited[j] = true;
            }
          }
        } catch (StackEmptyException e) {
          componentCount++;
          break;
        }
      }
    }
  }
  return componentCount;
}
```

Laufzeit

Die erste `for`-Schleife führt durch alle Knoten, und hat eine Laufzeit von *O(V)*.

Ausgehend von jedem (noch) unbesuchten Knoten, wird eine Tiefensuche ausgeführt. Sei *V(u)* die Anzahl aller Knoten in der Komponente, in der unser unbesuchter Knoten *u* liegt und *E(u)* die Anzahl aller Kanten zwischen den Knoten in dieser Komponente, dann ist die Laufzeit für die Tiefensuche innerhalb dieser Komponente gleich *O(V(u) + E(u))*. Warum?

Jeder Knoten wird einmal besucht (markiert) und einmal in den Stapel eingefügt und wieder entfernt. Sowohl das Markieren als auch die beiden Operationen am Stapel haben eine konstante Laufzeit *O(1)*. Das bedeutet, dass dies in *O(V(u))* erledigt wird.

Dabei wird jede Kante zwischen den Knoten maximal einmal für den Weg zum Nachbar verwendet, was im schlimmsten Fall für jede Kante in der Komponente stattfindet. Dies resultiert in einer zusätzlichen Laufzeit von *O(E(u))*, also insgesamt *O(V(u) + E(u))*.

Wenn alle Laufzeiten für alle Komponenten zusammengezählt werden, ergibt sich eine Laufzeit von *O(V + E)*.

Somit hat diese Implementierung des Algorithmus eine Laufzeit von *O(V + E)*.