**LK-Q1-V**

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(c) Bestimmung von Wegen in Graphen im Anwendungskontext (Tiefensuche, Breitensuche)



**Wegsuche im Graphen**

**Aufgabe**

*Entwickeln Sie einen Algorithmus, der einen beliebigen Weg von einem Startknoten S zu einem Zielknoten Z in einem gegebenen Graphen bestimmt. Jeder im Weg enthaltene Knoten darf nur einmal besucht werden, d.h. der Weg darf keine Kreise (Zyklen) haben.*

Lösung:

Es gibt mehrere Algorithmen, um dieses Problem zu lösen:

1. **Tiefensuche - Backtracking**

Ein Backtracking-Algorithmus geht nach dem Versuch-und-Irrtum-Prinzip (*trial and error*) vor. Es wird versucht, eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt beziehungsweise werden die letzten Schritte zurückgenommen und es werden stattdessen alternative Wege probiert. Auf diese Weise ist sichergestellt, dass alle in Frage kommenden Lösungswege ausprobiert werden können. Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert. Backtracking wird am einfachsten rekursiv implementiert und ist ein prototypischer Anwendungsfall von Rekursion. Selbstverständlich kann ein Backtracking-Algorithmus mit Hilfe eines Stacks auch iterativ implementiert werden (Übung!).

Ein Beispiel für Backtracking ist die Tiefensuche im Graphen. Bei diesem Verfahren wird versucht den Zielknoten auf möglichst direktem Wege zu finden und sich den zurückgelegten Weg dabei zu merken. Man geht vom Startknoten zum ersten Nachbarknoten, von diesem zu dessen erstem Nachbarknoten, der noch nicht besucht wurde, usw. bis man den Zielknoten erreicht hat. Wenn man in einer Sackgasse landet, d.h. bei einem Knoten der nur Nachbarn hat, die schon besucht wurden, geht man einen oder bei Bedarf auch mehrere Schritte zurück (Backtracking) und geht dann zum nächsten noch nicht besuchten Nachbarknoten usw. Dieses Verfahren wird fortgeführt bis man den Zielknoten erreicht hat, oder kein Schritt mehr möglich ist.

**Beispiel:**

Graph: s.o. Startknoten I, Zielknoten L

Die Nachbarknoten sollen jeweils in alphabetischer Reihenfolge besucht werden:

|  |  |  |
| --- | --- | --- |
| Schritt | Knotenfolge | Kommentar |
| 1 | I |  |
| 2 | IG |  |
| 3 | IGA |  |
| 4 | IGAB |  |
| 5 | IGABC |  |
| 6 | IGABCE |  |
| 7 | IGABCEF |  |
| 8 | IGABCEFK | Sackgasse zurück |
| 9 | IGABCEF | Sackgasse zurück |
| 10 | IGABCE |  |
| 11 | IGABCEL |  |

**Umgangssprachlich formulierter formalisierter Algorithmus**

**SucheWeg** von:start nach:ziel

Markiere start als besucht

start in die Wegliste

Wenn ziel noch nicht erreicht

Solange Weg nicht gefunden und start hat Nachbarknoten

Nächsten knoten aus Liste der Nachbarknoten holen

wenn knoten nicht markiert

**SucheWeg** von:knoten nach:ziel //rekursiver Aufruf knoten wird start

Wenn Weg noch nicht gefunden

gehe letzten Schritt zurück //Backtracking

sonst

ziel in Wegliste einfügen

Mit dem rekursiven Aufruf wird der Nachbarknoten zum neuen Start-Knoten, mit dem die Wegsuche fortgesetzt wird.

Dieser Algorithmus in Java implementiert ergibt folgende Methode:

public boolean sucheWeg(Vertex knoten, Vertex ziel, List<Vertex> weg){

boolean wegGefunden = false;

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (!wegGefunden && nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

wegGefunden = sucheWeg(nachbar, ziel, weg);

if (!wegGefunden) {

//Zug rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

weg.remove();

}

}

nachbarn.next();

}

} else {

wegGefunden = true;

}

return wegGefunden;

}

Die Methode liefert als Rückgabewert true, wenn ein Weg gefunden wurde, und false, wenn kein Weg gefunden wurde. Die Knotenfolge des Weges wird in der Liste verwaltet, die als Parameter übergeben wird. Der Aufruf der Methode muss mit einer leeren Liste als aktuellen Parameter erfolgen. Da bei Objekten als Parameter in Java grundsätzlich nur eine Referenz auf das Objekt übergeben wird, enthält die Liste nach Beendigung der Methode die Knotenfolge des gefundenen Weges.

**Aufgabe**

*Analysieren Sie die Java-Methode mit dem Startknoten F und dem Zielknoten L und geben Sie den gefundenen Weg als Knotenfolge an.*

Durch wenige Modifikationen der Java-Implementation lassen sich **alle** Wege vom Start- zum Zielknoten finden.

public void sucheAlleWege(Vertex knoten, Vertex ziel, List<Vertex> weg,

List<List<Vertex>> alleWege){

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

sucheAlleWege(nachbar, ziel, weg, alleWege);

//Zug rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

weg.remove();

}

nachbarn.next();

}

} else {

List<Vertex> einWeg = new List<Vertex>();

copy(weg, einWeg); //Hilfsmethode

alleWege.append(einWeg);

}

}

**Aufgabe**

*Vergleichen Sie die Java-Methode sucheWeg mit der Methode sucheAlleWege und beschreiben Sie die Unterschiede.*

*Begründen Sie, warum sich mit sucheAlleWege alle Wege im Graphen vom Start- zum Zielknoten erzeugen lassen.*

*Erläutern Sie, wie diese Wege in der Methode verwaltet werden.*

**2. Breitensuche**

Der Algorithmus zur Breitensuche vermeidet Sackgassen. Vom Startknoten ausgehend wird geprüft, ob einer seiner Nachbarknoten der Zielknoten ist. Wenn das nicht der Fall ist, werden nacheinander die Nachbarknoten der Nachbarknoten, die bisher nicht besucht wurden, betrachtet usw. bis der Zielknoten gefunden wurde. In einer Liste werden alle Knoten entsprechend der Besuchsreihenfolge gespeichert, bis der Zielknoten gefunden ist.

**Beispiel:**

Wenn von I aus der Zielknoten K gesucht wird, werden, wie man der folgenden Abbildung entnehmen kann, die Knoten in folgender Reihenfolge betrachtet: I GHK AF E B C L



10

9

8

7

6

5

4

3

2

1

Um den Algorithmus zu implementieren, werden die Nachbarknoten eines Knotens, die nicht markiert sind, an eine Schlange (Queue) angefügt, die von vorne nach hinten abgearbeitet wird. Der Algorithmus ist beendet, wenn der Zielknoten aus der Schlange geholt wurde oder wenn die Schlange leer ist. Im zweiten Fall wurde der Zielknoten nicht gefunden.

Die folgende Java-Methode implementiert die Breitensuche in einem Graphen:

public boolean breitenSuche(Vertex knoten, Vertex ziel,

List<Vertex> besucht){

Queue<Vertex> schlange = new Queue<Vertex>();

schlange.enqueue(knoten);

knoten.setMark(true);

boolean gefunden = false;

while (!schlange.isEmpty() && !gefunden){

knoten = schlange.front();

besucht.append(knoten);

schlange.dequeue();

if (knoten != ziel) {

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()){

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()) {

schlange.enqueue(nachbar);

nachbar.setMark(true);

}

nachbarn.next();

}

} else {

gefunden = true;

}

}

return gefunden;

}

**Aufgabe**

*Analysieren Sie die Java-Methode zur Breitensuch mit dem Startknoten E und dem Zielknoten H und geben Sie den Inhalt der Schlange nach jedem Durchlauf der äußeren while-Schleife an.*

Die Liste der besuchten Knoten enthält im Gegensatz zur Tiefensuche keinen Weg vom Start- zum Zielknoten. Bei der Breitensuche von I nach L enthält die Liste die Knotenfolge IGHKAFEBCL. Da es z.B. keine Kante zwischen H und K gibt, kann es sich nicht um einen Weg von I nach L handeln. Dennoch lässt sich aus dieser Liste ein Weg konstruieren, weil alle Knoten eines Weges von I nach L in der Liste enthalten sein müssen, sonst wäre L nicht erreicht worden. Die Strategie besteht darin, vom Zielknoten L aus rückwärts den Weg zu suchen. Wenn man die Liste von hinten nach vorne durchläuft, ist E der erste Knoten, der eine Kante mit H hat, also Vorgängerknoten von L ist. Der erste Vorgänger von E ist F, Vorgänger von F ist K, erster Vorgänger von K ist G und von G I, sodass sich der Weg IGKFEL konstruieren lässt.

In der folgenden Java-Methode ist der beschriebene Algorithmus mithilfe eines Stacks implementiert. Als Parameter wird die Liste der besuchten Knoten an die Methode übergeben.

public List<Vertex> konstruiereWeg(List<Vertex> knotenListe){

Stack<Vertex> stack = new Stack<Vertex>();

knotenListe.toFirst();

while (knotenListe.hasAccess()){

stack.push(knotenListe.getContent());

knotenListe.next();

}

List<Vertex> weg = new List<Vertex>();

Vertex knoten1 = stack.top();

weg.insert(knoten1);

weg.toFirst();

stack.pop();

while (!stack.isEmpty()){

Vertex knoten2 = stack.top();

stack.pop();

Edge edge = bspGraph.getEdge(knoten1, knoten2);

if (edge != null){

weg.insert(knoten2);

weg.toFirst();

knoten1 = knoten2;

}

}

return weg;

}

**Aufgabe**

*Analysieren und erläutern Sie die Java-Methode konstruiereWeg mit der durch Breitensuche von I nach L generierten Liste der besuchten Knoten.*

*Geben Sie den Weg an, den die Methode zurückliefert.*

Alle Algorithmen sind in dem BlueJ-Projekt **WegSuche** implementiert und können getestet werden.

 