



Nagl, Ranger, Wörzberger

Übungen zur Vorlesung „Grundgebiete der Informatik 2: Algorithmen und Programmiertechniken“

— Lösung zu Blatt 11 —

25. Aufgabe

Freiwillige Zusatzaufgabe: hier wurde der Floyd-Algorithmus verwendet, der die Länge *aller* Wege berechnet.

```
#include <iostream>
#include "AdjacencyGraph.h"

using namespace std;

AdjacencyGraph::AdjacencyGraph()
{
    /// Konstruktor nutzt makeNull zum initialisieren
    makeNull();
};

void AdjacencyGraph::makeNull( )
{
    for (NodeIdT from=0; from<MAX_NODES; from++) {
        for (NodeIdT to=0; to<MAX_NODES; to++) {
            deleteEdge(from, to);
        };
    };
};

void AdjacencyGraph::deleteEdge( NodeIdT from, NodeIdT to )
{
    if (from < MAX_NODES && to < MAX_NODES) {
        /// sicherheitshalber werden from und to ueberprueft
        adjacencyMatrix[from][to] = INFINITE; /// Kante loeschen
    };
};

void AdjacencyGraph::insertEdge( NodeIdT from, NodeIdT to )
{
    if (from < MAX_NODES && to < MAX_NODES) {
        // Kante einfuegen mit Defaultlaenge 1. Ohne Zusatzaufgabe waere das true,
        // so ein Wert != INFINITE.
        adjacencyMatrix[from][to] = 1;
    };
};
```

```

void AdjacencyGraph::insertEdge( NodeIdT from, NodeIdT to, DistanceT length )
{
    if (from < MAX_NODES && to < MAX_NODES) {
        adjacencyMatrix[from][to] = length; /// Kante beliebiger positiver Laenge einfuegen
    };
};

void AdjacencyGraph::maxOut( NodeIdT &node, int &degree )
{
    int currentDegree;
    degree = 0; node = 0;
    for (NodeIdT from=0; from<MAX_NODES; from++) {
        currentDegree = 0;          /// Auslaufgrad von Knoten #from

        for (NodeIdT to=0; to<MAX_NODES; to++) {
            if (INFINITE != adjacencyMatrix[from][to]) {
                currentDegree++;
            };
        };
        /// neuer maximaler Auslaufgrad?
        if (currentDegree > degree) {
            degree = currentDegree;
            node = from;
        };
    };
};

void AdjacencyGraph::maxIn( NodeIdT &node, int &degree )
{
    int currentDegree;
    degree = 0; node=0;
    for (NodeIdT to=0; to<MAX_NODES; to++) {
        currentDegree = 0;          /// Einlaufgrad von Knoten #from

        for (NodeIdT from=0; from<MAX_NODES; from++) {
            if (INFINITE != adjacencyMatrix[from][to]) {
                currentDegree++;
            };
        };
        /// neuer maximaler Einlaufgrad?
        if (currentDegree > degree) {
            degree = currentDegree;
            node = to;
        };
    };
};

DistanceT AdjacencyGraph::distance( NodeIdT from, NodeIdT to )
{
    DistanceT distances[MAX_NODES][MAX_NODES];

    /* Der Floyd-Algorithmus berechnet alle kuerzesten Wege in einem Graphen.
       Eine Beschreibung ist z.B. im Sedgewick zu finden. */
    DistanceT alternativeDistance;

    // Weglaengen mit Kantenlaengen initialisieren.
    for (NodeIdT initFrom=0; initFrom<MAX_NODES; initFrom++) {
        for (NodeIdT initTo=0; initTo<MAX_NODES; initTo++) {
            distances[initFrom][initTo] = adjacencyMatrix[initFrom][initTo];
        };
    };
};

```

```

};

// durch sukzessive Hinzunahme von Kanten Wege verkuerzen
for (NodeIdT alternative=0; alternative<MAX_NODES; alternative++) {
    for (NodeIdT fromCursor=0; fromCursor<MAX_NODES; fromCursor++) {
        if (distances[fromCursor][alternative] != INFINITE) {
            for (NodeIdT toCursor=0; toCursor<MAX_NODES; toCursor++) {
                /* Versuche den Weg von "fromCursor" nach "toCursor" zu verkuerzen, indem
                   ein Umweg ueber "alternative" gemacht wird */
                if (INFINITE != distances[alternative][toCursor]) {
                    alternativeDistance =
                        distances[fromCursor][alternative] + distances[alternative][toCursor];
                    if (alternativeDistance < distances[fromCursor][toCursor]) {
                        distances[fromCursor][toCursor] = alternativeDistance;
                    }
                }
            }
        }
    }
}
// Laenge zurueckgeben
return distances[from][to];
};

```

26. Aufgabe

- (a) Hier ist nur die Schnittstelle angegeben, die Implementierung gibt es unter <http://www-i3.informatik.rwth-aachen.de/ggdi>.

Datentypmodul Graph

```

#ifndef __INC_InOutGraph_h__
#define __INC_InOutGraph_h__
typedef unsigned int NodeIdT;

class InOutGraph {
public:
    static const NodeIdT MAX_NODES = 10;    // maximale Knotenanzahl
    InOutGraph();

    NodeIdT insertNode(); // Knoten einfuegen, Ergebnis ist neue Knotennr
    void deleteNode( NodeIdT n ); // Knoten #n loeschen
    void insertEdge( NodeIdT from, NodeIdT to );
    void deleteEdge( NodeIdT from, NodeIdT to );
    void print();
private:
    // Realisierung mit sequentiellen Adjazenzlisten
    //I besser: Adjazenzvektoren
    bool node[MAX_NODES]; // true falls Knoten existiert
    NodeIdT out[MAX_NODES][MAX_NODES]; // auslaufende Kanten
    unsigned int firstFreeOut[MAX_NODES]; // Anzahl auslaufender Kanten
                                         // von Knoten i

    NodeIdT in[MAX_NODES][MAX_NODES]; // einlaufende Kanten
    unsigned int firstFreeIn[MAX_NODES]; // Anzahl einlaufender Kanten
                                         // von Knoten i
};
#endif

```

- (b) • `insertNode()`: Keine Veränderung.

- `deleteNode()`: Am Löschen auslaufender Kanten ändert sich nichts, das Löschen einlaufender Kanten wird aber einfacher, da nicht mehr die Kantenlisten *aller* anderen Knoten durchsucht werden müssen.
- `insertEdge()`: Die neue Kanten muß jetzt als einlaufende und auslaufende Kante eingetragen werden, der Aufwand bleibt aber größenordnungsmäßig gleich.
- `deleteEdge()`: Die Kanten muß jetzt als einlaufende und auslaufende Kante gelöscht werden, der Aufwand bleibt aber größenordnungsmäßig gleich.

(c) Nach dem Löschen des dritten Knotens:

Knoten: 1, 2, 4, 5, 6

auslaufende Kanten: (1, 2), (2, 1), (4, 4), (4, 6), (5, 6)

einlaufende Kanten: (1, 2), (2, 1), (4, 4), (6, 5), (6, 4)