03.02.2002

Theoretische Informatik 1, WS 2001/2002

Übungsblatt 12

Keine Garantie für die Korrektheit der hier vorgestellten Lösungen.

Aufgabe 12.2

(a)

Beschreibung:

Durchlaufe den Graphen nach Tiefensuche. Am aktuellen Knoten wird jedem seiner Nachbarknoten, denen noch kein T(v) zugewiesen wurde (Knoten.Tv = null), sein Index als T(v) zugewiesen.
Der T(v) Wert von s wird zu Beginn auf ∞ gesetzt, was hier bedeutet, dass s.T(v) kein Wert gesetzt wurde.

Algorithmus:

```
Startknoten s
KnotenListe beinhaltet alle Knoten in aufsteigender Reihenfolge
Knoten.Tv beinhaltet den Wert von T(v)
Knoten.index beinhaltet Position eines Knotens in der Tiefensuche
Knoten.Ende vgl. Skript S. 131

globale Variable index = 1;

init() {
        MaxKnotenIndex = Anzahl der Knoten in KnotenListe;
        for (alle Knoten k ∈ KnotenListe)
                k.Tv = null;
                s.Tv = ∞;
                k.besucht = 0;
                k.index = 0

        for (alle Knoten k ∈ KnotenListe)
                if(k.besucht = 0) tiefenSuche1(k);

        /* In Aufgabenstellung steht nach Algorithmus S. 131 und der beinhaltet auch die Variable
           Ende. */
        for (alle Knoten k, aufsteigen nach ihrem Index sortiert)
                k.Ende = MaxKnotenIndex;
                MaxKnotenIndex--;

}

tiefenSuche1(k) {
        k.besucht = 1;
        k.index = index;
        for (alle Nachbarknoten k.nachbar von k)
                if (k.nachbar.Tv = null)
                        k.nachbar.Tv = k.index;
        index++;
        for (alle Nachbarknoten k.next von k in aufsteigender Reihenfolge)
                if(k.next.besucht = 0) tiefenSuche1(k.next);

}
```

(b)

Beschreibung:

Jeder Knoten beinhaltet eine Liste *Knoten.Nachbar*, die alle an den jeweiligen Knoten anschließende Kanten enthält (Adjazenzliste).
Außerdem gibt es für jeden Knoten ein Variable *Index*, die seine Position in der Tiefensuche enthält (vgl. Variable *Anfang* im Algorithmus tsuche() Skript S. 131).
Aus dem durch Tiefensuche() erhaltene Tiefensuche-Baum erzeuge für jeden Knoten eine Liste (*Knoten.Kinder*) mit seinen Nachfolgern im Tiefensuche-Baum.
Wenn ein Knoten im Tiefensuche-Baum mehr als ein Kind hat, sind die Kanten zu diesen Kindern potentielle Brücken. Für jedes dieser Kinder betrachte seine Nachbarknoten im Graphen. Wenn es eine Kante von einem Kind zu einem Knoten mit Index $\leqslant$ Vater.Index gibt, ist die Kante (Vater, diesesKind) keine Brücke. Wenn es keine solchen Knoten gibt, prüfe rekursiv für die Kinder von Kind. Gelangt man bis zu einem Blatt, gibt es in diesem Zweig keine Brücken.
Der Tiefensuche-Baum dient nur dazu potentielle Brücken zu finden.

Algorithmus:

```
Startknoten s (Knoten mit kleinstem Wert)
Liste Knoten.Nachbar
Liste Knoten.Kinder

init() {
        // im Graphen
        for (alle Knoten i) {
                erzeuge eine Liste i.Nachbar. Darin ist jeder Knoten j gespeichert, zu dem es eine
                Kante (i,j) gibt. Gibt es zu einem Knoten mehrere Kanten (i,j), schreibe den Knoten j
                genauso oft in die Liste, wie es Kanten (i,j) gibt.
        }
        Tiefensuche(s);
        /* Nach dem Durchführen der Tiefensuche ist jedem Knoten des Graphen ein Index
        zugeordnet (vgl. (a), Variable Knoten.index ).*/

        /* Tiefensuche() liefert auch den Tiefensuche-Baum.*/
        Erzeuge aus Tiefensuche-Baum für jeden Knoten i eine Liste i.Kinder, die alle seine Kinder
        im Tiefensuche-Baum enthält.
}

run() {
        for (alle Knoten i mit Anzahl Elemente in i.Kinder > 1)
                // die Kanten (i, i.Kind) sind potentielle Brücken
                pruefePotentielleBruecken(i);
}

// prüft ob potentielle Brücken wirkliche Brücken sind
pruefePotentielleBruecken(j) {
        for (alle Knoten k aus j.Kinder, beginnend bei k mit kleinstem k.index)
                streiche j in k.Nachbar;
                if (es gibt eine Kante (k,l) im Graphen, wobei l.Index ⩽ j.Index)
                        Kante (j, j.Kind) ist keine Brücke
                else
                        pruefe(k);
}
```

Laufzeit:

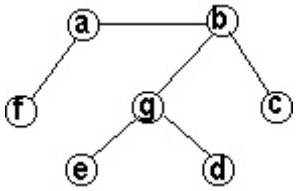
Erzeugen der Liste i.Nachbar (Adjazenzliste für ungerichtete Graphen) O(|V| + 2|E|).
Tiefensuche O(|V| + |E|).
Erzeugen der Liste i.Kinder (Adjazenzliste für gerichtete Graphen) O(|V| + |E|).
pruefePotentielleBruecken() O(|V| + |E|).
Laufzeit insgesamt O(|V| + |E|).

Aufgabe 12.3

Prim's Algorithmus (einzelnen Baum wachsen lassen)

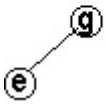
(a,b), (b,g), (g,e), (g,d), (a,f), (b,c)

Es entsteht folgender minimaler Spannbaum:

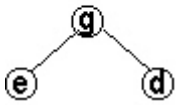


Kruskal's Algorithmus (Bäume verschmelzen)
sortierte Kanten:
(a,b) = 1 - (g,e) = 2 - (g,d) = 3 - (e,d) = 4 - (g,b) = 5 - (g,a) = 6 - (f,a) = 7 - (f,e) = 8 - (b,c) = 9 - (c,d) = 10

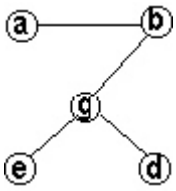
Nach den ersten Schritte ergeben sich folgende Bäume:



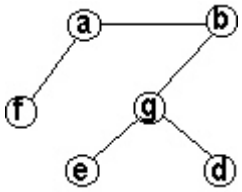
Kante (g,d) einbauen:



Kante (e,d) schließt einen Kreis, verwerfen.
Kante (g,b) einbauen:



Kante (g,a) schließt einen Kreis, verwerfen.
Kante (f,a) einbauen.



Kante (f,e) schließt einen Kreis, verwerfen.
Kante (b,c) einbauen: