Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

Folien: Graphen

Autor: Stefan Edelkamp

Institut für Informatik

Georges-Köhler-Allee

Albert-Ludwigs-Universität Freiburg

1 Graphen

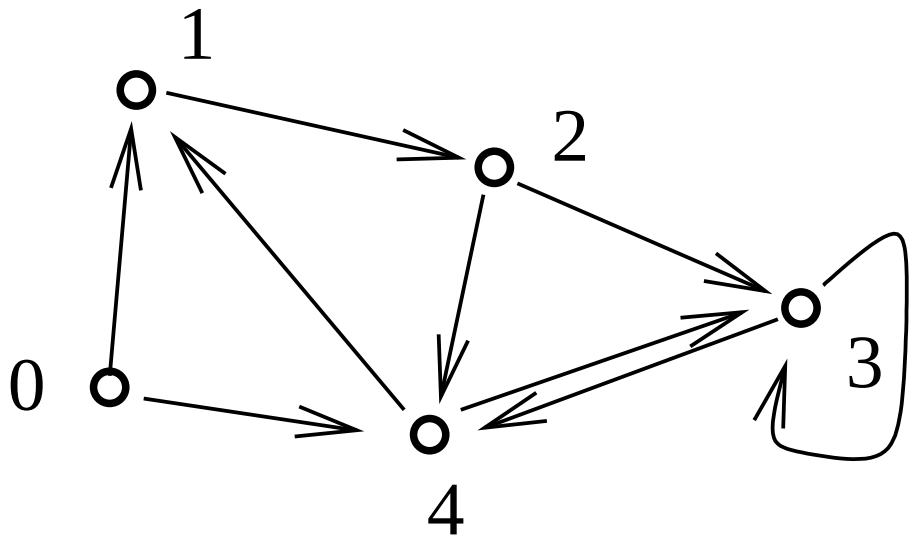
(Gerichtete) Graphen: $G = (V, E)$ bestehen aus Knoten V und Kanten E

Beispiel: und eignen sich zur Repräsentation z.B. von Bus- und Straßenbahnverbindungen der Stadt Freiburg.

Repräsentation: Man unterscheidet zwei grundlegend verschiedene Implementationen von Graphen.

Die Adjazenzliste: An ein Array von Knoten wird für jeden Eintrag eine Liste der Nachbarn angehängt.

Die Adjazenzmatrix: Der Eintrag an der Stelle (i, j) der Matrix aus Wahrheitswerten gibt an, ob zwischen dem Knoten i und dem Knoten j eine Kante existiert oder nicht.



Implementation

Knoten:

```
class Knoten {
  Knoten next;
  int n;
  Knoten(int n) {
    next = null;
    this.n = n;
  }
  Knoten(int n, Knoten k) {
    next = k;
    this.n = n;
  }
}
```

Graph:

```
class Graph {
  public static void main(String args[]) {
    Adliste Al = new Adliste(10,0.3);
    Admatrix Am = new Admatrix(10,0.3);
    System.out.println(Am);
    System.out.println(Al);
  }
}
```

Implementation Adjazenzmatrix

Konstruktion: Kantendichte gemäß vorgegebener Dichte ausgewürfelt

```
class Adliste {
  Knoten ad[];

  Adliste(int n, double Dichte) {
    int i; int j;
    ad = new Knoten[n];
    for(i=0;i<ad.length;i++) {
      for (j=0;j<ad.length;j++) {
        if (Math.random() < Dichte) {
          ad[i] = new Knoten(j,ad[i]);
        }
      }
    }
  }
```

Ausgabe:

```
  public String toString() {
    StringBuffer st = new StringBuffer("");
    int i; Knoten j;
    for(i=0;i<ad.length;i++) {
      for (j=ad[i];j!=null;j=j.next) {
        st.append(j.n+" ");
      }
      st.append("\n");
    }
    return st.toString();
  }
}
```

Implementation Adjazenzliste

Konstruktion: Kantendichte gemäß vorgegebener Dichte ausgewürfelt

```
class Admatrix {
  boolean ad[][];

  Admatrix(int n, double Dichte) {
    int i; int j;
    ad = new boolean[n][n];
    for(i=0;i<n;i++) {
      ad[i] = new boolean[n];
      for(j=0;j<n;j++) {
        if (Math.random() < Dichte)
          ad[i][j] = true;
      }
    }
  }
```

Ausgabe:

```
  public String toString() {
    StringBuffer st = new StringBuffer("");
    int i; int j;
    for (i=0;i<ad.length;i++) {
      for(j=0;j<ad[i].length;j++) {
        st.append(ad[i][j] + " ");
      }
      st.append("\n");
    }
    return st.toString();
  }
}
```

Tiefensuche

Hauptschleife:

```
public void DFS() {
  boolean visited[] = new boolean[ad.length];
  for (int i=0;i<ad.length;i++) {
    if (!visited[i])
      DFS(i,visited);
  }
}
```

Rekursion:

```
void DFS(int i, boolean visited[]) {
  System.out.println("Visiting node " +i);
  visited[i] = true;
  for (Knoten j=ad[i];j!=null;j=j.next) {
    if (!visited[j.n])
      DFS(j.n,visited);
  }
}
```

2 Topologische Sortierung

Definition: Die Relation $\leq$ ist eine vollständige bzw. partielle Ordnung auf M, wenn (1)-(4) bzw. (2)-(4) gelten.

(1) for all $x, y \in M : x \leq y$ oder $y \leq x$
(2) for all $x \in M : x \leq x$
(3) for all $x, y \in M : x \leq y$ und $y \leq x \Rightarrow x = y$
(4) for all $x, y \in M : x \leq y$ und $y \leq z \Rightarrow x \leq z$

Beispiel: Potenzmenge 2^M einer Menge M bezüglich Teilmengenrelation partiell aber nicht vollständig geordnet.

Topologische Sortierung: Gegeben M und $\leq$ partiell. Gesucht Einbettung in vollständige Ordnung, d.h. die Element von M in eine Reihenfolge $m_1, \ldots, m_n$ bringen, wobei $m_i \leq m_j$ für $i < j$ gilt.

Lemma: Jede partiell geordnete Menge läßt sich topologisch sortieren.

Beweisidee: Es genügt, die Existenz eines minimalen Elements z in M zu zeigen. z muß nicht eindeutig sein. Wähle $m_1 = z$ und sortiere $M - \{z\}$ topologisch.

Verwendete Datenstrukturen

A: Array der Länge n, $A[i]$ natürliche Zahl
V: Array der Länge n, $V[i]$ Zeiger auf $L[i]$
$L[i]$: Liste von Zahlenpaaren
Q: Queue (Schlange), die ebenfalls Zahlen enthält

Algorithmus:

Eingabe: p Paare (i, j), mit $x_i \leq x_j$ und
$M = \{x_1, \ldots, x_n\}$

(1) Setze alle $A[i]$ auf 0, alle $L[i]$ und Q seien leer
(2) Wird (j, k) gelesen, wird $A[k]$ um 1 erhöht und in $L[j]$ eingetragen.
(3) Durchlaufe A: Schreibe alle k mit $A[k] = 0$ in Q
(4) While $Q \neq \{\}$
    j = Q.dequeue()
    Gebe x_j aus
    Durchlaufe $L[j]$: Wird k gefunden, wird $A[k]$ um 1 verringert. Falls $A(k) = 0$ ist, Q.enqueue(k)

Satz: Der Algorithmus löst das Problem Topologische Sortierung in $O(n + p)$ Zeit.

Implementation

```
public void topsort() {
  int [] indeg = new int[ad.length];
  for(int i=0;i<ad.length;i++) {
    for (Knoten j=ad[i];j!=null;j=j.next) {
      indeg[j.n]++;
    }
  }
  Queue Q =  new Queue(ad.length);
  for(int i=0;i<ad.length;i++) {
    if (indeg[i] == 0)
      Q.enqueue(i);
  }
  int lfdNr = 0;
  while (!Q.empty()) {
    int i = Q.dequeue();
    System.out.println(i); lfdNr++;
    for (Knoten j=ad[i];j!=null;j=j.next) {
      indeg[j.n]--;
      if (indeg[j.n] == 0)
        Q.enqueue(j.n);
    }
  }
  if (lfdNr != ad.length)
    System.out.println("Graph hat einen Zyklus");
}
```