1 Definitionen

1.1 Graphen

1.1.1 Gerichteter Graph

Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit einer endlichen Menge $V \neq \emptyset$ und einer endlichen Menge $E \subseteq \{(u, v) | u, v \in V, u \neq v\}$. Die Elemente von V heißen Knoten (auf englisch: vertex) von G . Die Elemente von E heißen Kanten (auf englisch: edge) von G. Eine Kante $e \in E$ ist also von der Form $e = (u, v)$ mit $u, v \in V, u \neq v$, wobei u als Anfangsknoten und v als Endknoten von e bezeichnet wird.

1.2 Mengen

1.2.1 Untere Schranke

Sei M eine Teilmenge von $\mathbb{R}$. Dann nennt man eine Zahl u, die kleiner gleich jedem Element von M ist, eine untere Schranke. Es ist also $x \geq u$ für alle $x \in M$.

1.2.2 Infimum

Sei M eine Teilmenge von $\mathbb{R}$. Das Infimum i einer Menge M ist die größte untere Schranke von M. Das Infimum wird charakterisiert über die beiden Eigenschaften:

- Für jedes $y \in M$ ist $y \geq i$.
- Jede Zahl x größer als i ist keine untere Schranke von M: Für alle $x > i$ gibt es mindestens eine Zahl $y \in M$ mit $x > y$.

2 Datentyp für Graphen und Netzwerke

(→ LEDA)

2.1 Definition eines Datentyps

Beispiel 2.1
Ein *stack* ist eine Folge von Elementen vom Typ T.
$Stack < T >$ parametrisierter Typ

- Konstruktion
  $stack < int > S(100)$
  Java: $S = stack < int > (100)$
- Operationen (auf einem Stack S)
  - void S.push(T x)
    Fügt x an der Top-Seite hinzu.
  - T S.pop()
    Entfernt und liefert das Top-Element.
    Precondition: S ist nicht leer.
  - S.empty()
- Iteration → Makros
  Liste list<int> L
  $int\ x;\ forall(x, L)\ \{sum+ = x_i\}$
  Makro
  $\#\ define\ forall(x, L)\ for(list\_elem\ p = L.first();\ p \neq null;\ p = L.succ(p))$

Bemerkung:
Der allgemeine Graph-Datentyp in LEDA: „*graph*“
Der Datentyp repräsentiert gerichtete Graphen.

Ein Graph G besteht aus zwei Typen von Objekten:

Knoten vom Typ „*node*“

Kanten vom Typ „*edge*“

Mit jedem Knoten v sind zwei Listen vom Typ $list < edge >$ assoziiert:
out_edges und in_edges.

Eine Kante verläuft zwischen genau 2 Knoten.

2.1.1 Operationen auf G

- Update

  $\underline{node}\ G.new\_node();$
  Erzeugt einen neuen Knoten in G und gibt ihn (seine Referenz) zurück.
  $\underline{edge}\ G.new\_edge(node\ v, node\ w)$
  $\underline{void}\ G.del\_edge(edge\ e)$

- Access

  $\underline{list < edge >}\ G.out\_edges(node\ v);$
  $\underline{int}\ G.outdeg(node\ v);$
  $\underline{node}\ G.source(edge\ e);$
  $\underline{node}\ G.target(edge\ e);$

- Iteration

$$
\begin{aligned}
&forall\_nodes(v, G)\{\dots\}\\
&forall\_edges(e, G)\\
&forall\_out\_edges(e, V)\\
&forall\_in\_edges(e, V)\\
&forall\_edges(e, G) \rightarrow node\ v;\\
&\qquad\qquad forall\_nodes(v, G)\{\\
&\qquad\qquad\quad edge\ e;\\
&\qquad\qquad\quad forall\_out\_edges(e, v)\{\dots\}\\
&\qquad\qquad \}
\end{aligned}
$$

2.1.2 Daten für Knoten und Kanten

Es gibt zwei Arten von Daten für Knoten und Kanten.

i) Parametrisierte Graphen: graph<node_type, edge_type>
Beispiel: graph<stadt, autobahn> Verkehrsnetz;
graph<transistor, wire> Schaltkreis;

ii) Temporäre Daten
Beispiel: besucht[v] = true;
dist[u] = 17;

LEDA stellt dafür zwei Datentypen bereit.

- node_array<T> A(G,x) (Initialisierung optional)
Es ist ein Feld über die Knoten des Graphen G.
Der Zugriff erfolgt über A[v].

- edge_array<T> B(G,y)
Es ist ein Feld über die Kanten des Graphen G.
Der Zugriff erfolgt über B[e].

Verwendung:
Dieser Datentyp findet Verwendung für temporäre Daten im Algorithmus und für Eingabedaten und Resultate.

Anwendung im topologischen Sortieren:
Es existiert eine injektive Abbildung $topnum : v \to \{1 \dots n\}$
mit $\forall\ (v, w) \in E : topnum[v] < topnum[w]$

Algorithmus Topsort

```
1  bool Topsort(const graph& G, node_array<int>& topnum){
2       int count = 0;
3       list<node> ZERO; // leere Liste von Knoten
4       node_array<int> indeg(G); // aktueller Indegree
5       node v;
6       forall_nodes(v,G){
7            INDEG[v] = indeg(v);
8            if(INDEG[v] == 0)
9                 ZERO.append(v);
10      }
11      while(!ZERO.empty()){
12           u = ZERO.pop();
13           topnum[u] = count+1
14           edge e;
15           forall_out_edges(e,u){
16                v = G.target(e);
17                if(--INDEG[v] == 0)
18                     ZERO.append(v);
19           }
20      }
21      return count == G:number_of_nodes();
22 }
```

Listing 2.1: Topsort

Auf diesem Algorithmus aufbauend lässt sich das Problem lösen, ob ein Graph azyklisch ist.

Gegeben:
Graph G = (V,E)

Idee:
(→ siehe topologische Sortierung)
Entferne jeweils einen Knoten v mit $indeg(v) = \emptyset$ bis der Graph G leer ist.
Falls wir keinen solchen Knoten finden, ist G zyklisch.
Falls G am Ende leer ist, ist G azyklisch.

Algorithmus Acyclic

```
1  bool Acyclic(graph G){
2       list<node> ZERO;
3       node v;
4       forall_nodes(v,G){
5            if(G.indeg(v) == 0)
6                 ZERO.append(v);
7       }
8       while(!ZERO.empty()){
9            node u = ZERO.pop();
10           edge e;
11           forall_out_edges(e,u){
12                node w = G.target(e);
13                G.del_edge(e);
14                if(G.indeg(w) == 0)
15                     ZERO.append(w);
16           }
17      }
18      return G.empty();
19 }
```

Listing 2.2: Acyclic

Erklärung:
Zuerst werden alle Knoten, die keine eingehenden Kanten besitzen in die Menge $ZERO$ aufgenommen. Danach wird solange noch Knoten in der Menge $ZERO$ enthalten sind, jeweils einer entfernt und für diesen Knoten v überprüft, ob einer der Knoten w, die direkt über eine von v ausgehende Kante mit v verbunden sind, keine eingehenden Kanten mehr hat, wenn diese eine Kante gelöscht würde. Ist dies der Fall, wird w in die Menge $ZERO$ aufgenommen.

Algorithmus Tiefensuche (DFS) (rekursiv)

```
1  void DFS(const graph& G, node_array<int>& dfsnum,
       node_array<int>& compnum){
2        int count1 = 0;
3        int count2 = 0;
4        node_array<bool> visited(G,false);
5        node v;
6        forall_nodes(v,G){
7             if(!visited[v])
8                  dfs(G, v, count1, count2, dfsnum,
                       compnum);
9        }
10 }
```

Listing 2.3: Hauptprogramm

```
1  void dfs(const graph& G, node v; int& count1. int&
       count2, node_array<int>& dfsnum, node_array<int>&
       compnum){
2        dfsnum[v] = ++count1;
3        visited[v] = true;
4        edge e;
5        forall_out_edges(e,v){
6             node w = G.target(e);
7             if(!visited[w])
8                  dfs(G,w,count1, count2, dfsnum, compnum)
9        }
10       compnum[v] = ++count2;
11 }
```

Listing 2.4: Rekursive Funktion

Erklärung:
Zu Beginn des Algorithmus sind alle Knoten als „noch nicht besucht“ markiert. Das Hauptprogramm ruft für den ersten Knoten v, der noch nicht besucht wurde die rekursive Funktion auf. Diese speichert in der Variable „dfsnum“ zu welchem Zeitpunkt der Knoten v das erste mal besucht wurde und markiert den Knoten v als „besucht“. Dann wird für einen Knoten w, der über eine ausgehende Kante mit v verbunden ist und noch nicht besucht wurde, die rekursive Funktion aufgerufen. Dabei wird aber, anderes als bei der Breitensuche, der nächste von v direkt erreichbare

Knoten erst aufgerufen, wenn alle von w ausgehenden Kanten abgearbeitet wurden. Für seine ausgehenden Kanten gilt aber wieder das gleiche, wie für die ausgehenden Kanten von v.
Sind alle ausgehenden Kanten von v abgearbeitet, wird in der Variablen „compnum“ genau dieser Zeitpunkt gespeichert.

Eine eventuell Bessere Variante wäre die folgende:

```
class DFS{
      aonst graph G,
      int count1;
      int count2;
      node_array<int>&dfsnum;
      ...

      void dfs(node v){...}
      void run(){ /* DFS */ }
      ...
}
```

Eine Anwendung von DFS ist die Berechnung von starken Zusammenhangskomponenten.