Benutzerhandbuch
EMS-Graphenlabor V3.0

vorläufige Version

Peter Dahm
Jürgen Ebert
Christoph Litauer

13. Oktober 1994

Inhaltsverzeichnis

1 Einführung

1.1 Wozu EMS-Graphenlabor ?

Das EMS-Graphenlabor wurde mit dem Ziel entwickelt, einem Anwendungsprogrammierer (im folgenden als Benutzer bezeichnet) einen abstrakten Datentyp *Graph* mit einer möglichst leistungsfähigen Schnittstelle zur Verfügung zu stellen. Insbesondere werden folgende Anforderungen erfüllt:

- Knoten und Kanten sind eigenständige, identifizierbare Objekte.
- Mehrfachkanten, d.h. mehrere Kanten zwischen den gleichen Knoten, sind möglich.
- Die Kanten eines Graphen sind gerichtet.
- Der einem gerichteten Graphen zugrunde liegende ungerichtete Graph steht dem Benutzer unmittelbar zur Verfügung.
- Der Graph kann dynamisch geändert werden, d.h. während der Laufzeit der Anwendung können Knoten und Kanten eingefügt und gelöscht werden.
- Die Kanten, die mit einem Knoten in Berührung stehen, sind in einer vom Benutzer veränderbaren Reihenfolge angeordnet.
- Alle Knoten eines Graphen, alle Kanten eines Graphen und alle mit einem Knoten inzidenten Kanten können leicht traversiert werden.
- Zur Unterscheidung unterschiedlicher Knoten bzw. Kantenklassen können die Knoten bzw. Kanten typisiert werden.
- Die in einem Graphen verwendeten Typen können in einer Subtyp-Relation stehen.
- Die Knoten und Kanten können mit Werten versehen (*attributiert*) werden. Das dabei verwendete Attributierungsschema ist vom Typ des jeweiligen Knotens oder der Kante abhängig.
- Knoten und Kanten können kurzfristig mit Kontrollinformationen für Standardalgorithmen markiert werden. Diese temporäre Attributierung ist von den typabhängigen Attributierungsschemata unabhängig.

Die vom EMS-Graphenlabor verwalteten Graphen sind also i.allg. dynamische, gerichtete, angeordnete, typisierte und attributierte Graphen, die in der Literatur als TGraphen [EbeFra 94] bezeichnet werden.

1.2 Kleine Graphenterminologie

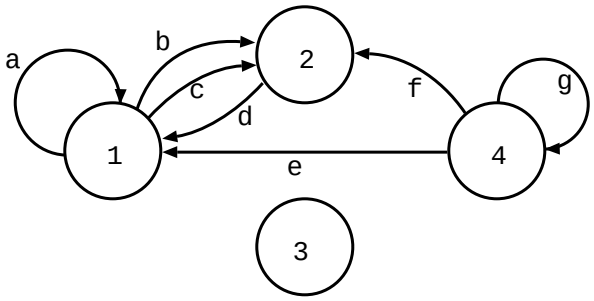
Graphen sind ein leistungsfähiges Modellierungsmittel, da sie zugleich anschaulich sind, mit den formalen Methoden der Graphentheorie behandelt werden können und effizient implementierbar sind. Im folgenden werden viele Begriffe aus der Graphentheorie verwendet:

1.2.1 Gerichtete Graphen

Ein *gerichteter Graph* (kurz auch nur *Graph*) umfaßt eine endliche Menge von *Knoten* V und eine endliche Menge von *gerichteten Kanten* (kurz auch nur *Kanten*) E. Dabei verbindet jede *Kante* $e \in E$ genau einen *Anfangsknoten* (oder *Startknoten*) $v \in V$ mit genau einem *Endknoten* (oder *Zielknoten*) $u \in V$. Man sagt dann auch, Kante e führt von Knoten v nach Knoten w. Falls $v = w$ gilt, heißt e auch eine *Schlinge*.

Die Beziehungen zwischen Kanten und ihren Anfangsknoten und Endknoten werden durch die Funktionen $\alpha : E \to V$ und $\omega : E \to V$ ausgedrückt. $\alpha(e)$ ist der Anfangsknoten der Kante e, $\omega(e)$ der Endknoten. Zwei verschiedene Kanten dürfen durchaus den gleichen Anfangs- und den gleichen Endknoten haben und werden dann als *Mehrfachkanten* bezeichnet.

Eine Kante e heißt inzident zu einem Knoten v, wenn er Anfangs- oder Endknoten dieser Kante ist. Eine Kante, deren Anfangsknoten bzw. Endknoten ein Knoten v ist, heißt auch *out-Kante* bzw. *in-Kante* bezüglich Knoten v. Ein Knoten, der weder Anfangsknoten noch Endknoten irgendeiner Kante ist, heißt *isoliert*. Ein Graph kann graphisch dargestellt werden:



Die Knotenmenge V ist hier $\{1, 2, 3, 4\}$, die Kantenmenge E ist $\{a, b, c, d, e, f, g\}$. Kanten b und c sind Mehrfachkanten, Kanten a und g sind Schlingen. Knoten 3 ist isoliert. Die Funktionen α und ω sind:

	a	b	c	d	e	f	g
α	1	1	1	2	4	4	4
ω	1	2	2	1	1	2	4

Das Graphenlabor repräsentiert die Menge aller Knoten und die Menge aller Kanten als (injektive) Folgen. *Vseq* bezeichnet die Folge aller Knoten, *Eseq* die Folge aller Kanten. Diese Folgen können mittels der Makros **G_forAllVertices** und **G_forAllEdges** durchlaufen werden.[1] Die Reihenfolge, in der das Labor die Werte zurückliefert, ist durch den Benutzer beeinflußbar. Zu weiteren Details siehe Abschnitt 3.8, S. 55.

Knoten und Kanten werden eindeutig natürliche Zahlen als Identifikationsnummern zugeordnet. Diese Zahlen werden bei der Ausgabe von Graphen durch Laborfunktionen zur Identifikation von Knoten oder Kanten verwendet. Sie werden evtl. nach Löschen eines Knotens oder einer Kante für andere, neu erzeugte Knoten oder Kanten wiederverwendet.

[1] Realisiert werden diese Makros durch first-, next- Funktionspaare, die auch einzeln dem Benutzer zur Verfügung stehen.

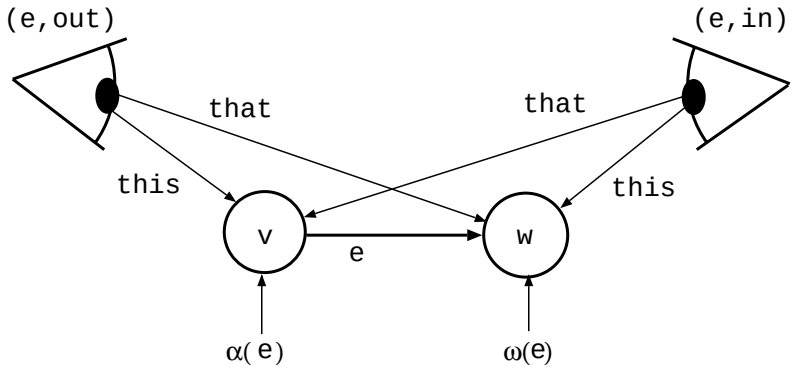
1.2.2 Ungerichtete Graphen

Ein *ungerichteter Graph* umfaßt eine endliche Menge von Knoten V und eine endliche Menge von *ungerichteten Kanten* E. Jede ungerichtete Kante $e \in E$ verbindet entweder zwei verschiedene Knoten $u, v \in V$ miteinander oder im Falle einer *ungerichteten Schlinge* einen Knoten $v \in V$ mit sich selbst.

Die Beziehung zwischen einer ungerichteten Kante und den durch sie verbundenen Knoten werden durch die Funktionen $this, that : E \to V$ ausgedrückt. $this(e)$ ist einer der beiden durch Kante e verbundenen Knoten, $that(e)$ der andere Knoten.[2] Falls e eine Schlinge ist, gilt $this(e) = that(e)$. Auch in ungerichteten Graphen sind Mehrfachkanten erlaubt.

1.2.3 Orientierung

Um gerichtete und ungerichtete Graphen gleich behandeln zu können, wird im EMS-Graphenlabor das Konzept der *Orientierung* eingeführt. Eine *orientierte Kante* $\vec{e}$ ist ein Paar $(e, d) \in E \times Dir$ mit $Dir = \{in, out\}$. Dies entspricht der Tatsache, daß man eine Kante von einem der durch sie verbundenen Knoten aus sehen kann.



Für jede gerichtete oder ungerichtete Kante e ist $\vec{e} = (e, out)$ die *normal* orientierte oder auch *normalisierte* Kante. Die normale Orientierung einer gerichtete Kante ist diejenige Orientierung, bei der man die Kante von ihrem Startknoten aus sieht, bei einer ungerichteten Kante e diejenige Orientierung, bei der man die Kante vom Knoten $this(e)$ aus sieht.

Im EMS-Graphenlabor werden Kanten mit ihren normal orientierten Kanten identifiziert. Die Funktionen α, ω, *this* und *that* werden auf die Menge der orientierten Kanten $E \times Dir$ wie folgt erweitert:

$$\begin{aligned} \alpha(e, out) &:= \alpha(e) \\ \alpha(e, in) &:= \alpha(e) \\ \omega(e, out) &:= \omega(e) \\ \omega(e, in) &:= \omega(e) \\ this(e, out) &:= \alpha(e) \end{aligned}$$

[2] Die Zuordnung ist hier willkürlich. Werden $this(e)$ und $that(e)$ vertauscht, ändert sich der Graph nicht. Dennoch ist die Zuordnung im weiteren als fest anzusehen.

$$
\begin{aligned}
\mathit{this}(e, \mathit{in}) &:= \omega(e) \\
\mathit{that}(e, \mathit{out}) &:= \omega(e) \\
\mathit{that}(e, \mathit{in}) &:= \alpha(e)
\end{aligned}
$$

Man beachte, daß die Werte der Funktionen α und ω von der Orientierung der betrachteten Kante unabhängig sind und nur für gerichtete Graphen sinnvoll sind, und daß die Werte der Funktionen *this* und *that* von der Orientierung abhängen. So ist $\mathit{this}(\vec{e})$ immer derjenige Knoten, von dem aus man die Kante e gerade betrachtet, und $\mathit{that}(\vec{e})$ der Knoten am anderen Ende der Kante.

Insgesamt wird dadurch erreicht, daß zu jedem gerichteten Graph der zugrundeliegende ungerichtete Graph unmittelbar zur Verfügung steht, indem man auf den gerichteten Graph ausschließlich mit den Funktionen *this*, *that* zugreift und mit orientierten Kanten arbeitet. Soll mit dem EMS-Graphenlabor ein ungerichteter Graph verwaltet werden, kann der Benutzer also einen gerichteten Graphen anlegen, dessen zugrundeliegender ungerichteter Graph der gewünschte Graph ist.

Bei der Ausgabe von Graphen wird die Orientierung von Kanten durch das Vorzeichen zur Identifikationsnummer ausgedrückt. Positives Vorzeichen bedeutet die Orientierung `out`, negatives Vorzeichen die Orientierung `in`.

1.2.4 Weitere Begriffe

Zu jedem Knoten v sind alle in- und out-Kanten als orientierte Kanten in einer Sequenz $\Lambda(v)$ angeordnet, die im folgenden als *Folge der zu Knoten v inzidenten Kanten* bezeichnet wird. In dieser Folge erscheinen in-Kanten mit der Orientierung *in*, out-Kanten mit der Orientierung *out*, also normalisiert. Schlingen treten doppelt, nämlich in beiden Orientierungen auf. Für jede orientierte Kante $\vec{e}$ aus $\Lambda(v)$ gilt $\mathit{this}(\vec{e}) = v$.

Formal ergibt sich für die Funktion Λ die Charakteristik $\Lambda : V \rightarrow \mathrm{seq}(E \times \mathit{Dir})$. Bei gerichteten Graphen enthält $\Lambda(v)$ die Teilfolge aller in-Kanten $\Lambda^-(v)$ und die Teilfolge aller out-Kanten $\Lambda^+(v)$.[3] Auch diese Folgen kann der Benutzer durch die vom Labor definierten Makros **G_forAllIncidentEdges**, **G_forAllInEdges** und **G_forAllOutEdges** durchlaufen. Ferner ist die Reihenfolge der Elemente der Sequenz $\Lambda(v)$ durch Laborfunktionen beeinflußbar.

Unter dem *Grad* $\delta(v)$ eines Knotens v versteht man die Anzahl der Kanten in der Kantenfolge zu v: $\delta(v) =\mid \Lambda(v) \mid$.[4] Für gerichtete Graphen definiert man analog noch den *Innengrad* $\delta^-(v)$ und *Außengrad* $\delta^+(v)$ eines Knoten v mittels $\delta^-(v) =\mid \Lambda^-(v) \mid$ und $\delta^+(v) =\mid \Lambda^+(v) \mid$.

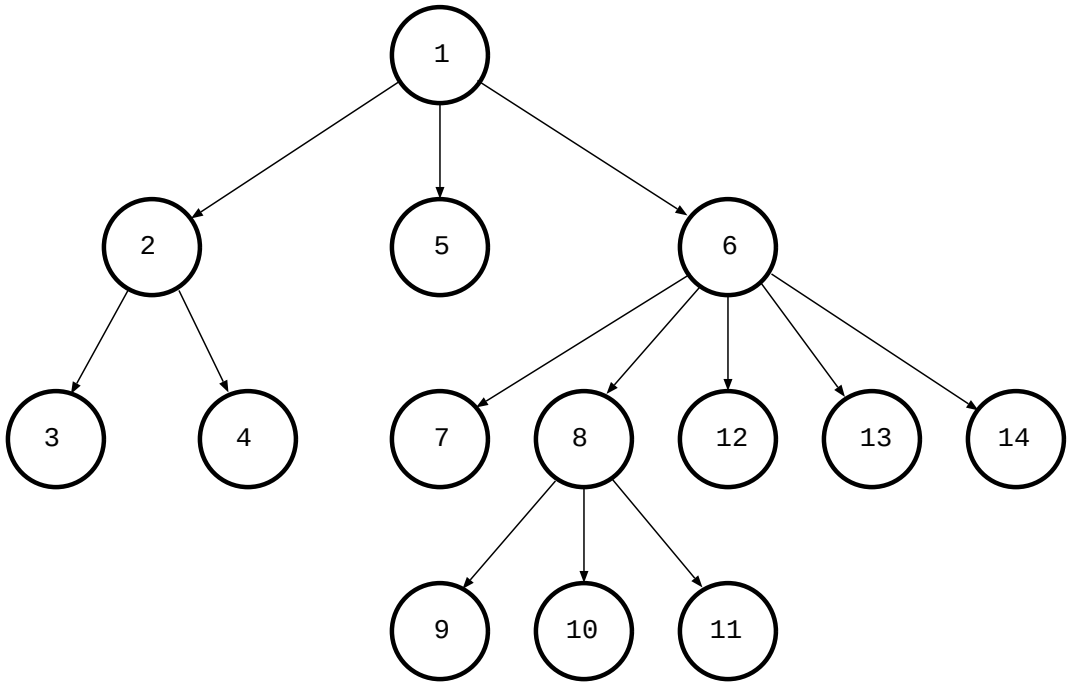
[3] Da es sich um Teilfolgen handelt, sind $\Lambda^-(v)$ und $\Lambda^+(v)$ durch $\Lambda(v)$ eindeutig beschrieben.

[4] Bei dieser Definition werden Schlingen daher doppelt gezählt.

1.3 Beispielanwendung

1.3.1 Aufgabenstellung

In diesem Abschnitt soll ein Programm[5] folgenden Baum aufbauen und in pre-Order traversieren:[6]



Als Ausgabe sollen alle Knoten in pre-order und gemäß ihrer Tiefe im Baum durch Punkte eingerückt erscheinen.

1.3.2 Quelltext

Man legt mit einem beliebigen Editor eine Datei `treeTrav.c` an und gibt den notwendigen Quelltext ein. Zunächst sind die Deklarationen des Graphenlabors dem Programm mitzuteilen:

```
#include <graph.h>
```

Ab jetzt stehen alle Typen, globale Variable und Funktionsprototypen des Graphenlabors zur Verfügung. Das Programm besteht aus dem Hauptprogramm `main`, einer Funktion `buildTree` zum Aufbau des Baums und einer rekursiven Funktion `traverse` zur Traversierung. Zunächst das Hauptprogramm:

[5] Der Quelltext dieses Beispielprogramms liegt in Verzeichnis `$EMSDEMO`(zu Verzeichnisangaben siehe Abschnitt 2.1, S. 38) vor.

[6] Traversierung von Graphen erfordert in der Regel eine Markierung der bereits besuchten Knoten, um nicht in Kreise zu gelangen. Eine Markierung von Knoten sollte aber durch die den Mechanismus der *temporären Attributierung* (siehe Abschnitt 1.6, S. 30) erfolgen und würde das Beispiel unnötig verkomplizieren.

```
void main ( void )
{
   G_graph g;
   G_vertex  root;

   root = buildTree (g);

   traverse (g, root, 0);
}
```

Beim Betreten der Funktion `main` wird zunächst der Konstruktor der Klasse `G_graph` für die Variable `g` aufgerufen. Dieser erzeugt einen leeren Graphen. Die Funktion `buildTree` erzeugt anschließend aus diesem leeren Graphen den obigen Baum und liefert den Wurzelknoten `root` zurück. Die Funktion `traverse` schließlich traversiert den Baum von der Wurzel `root` aus mit der aktuellen Rekursionstiefe 0.

Als nächstes ist die Funktion `buildTree` anzugeben. Der Baum soll interaktiv eingegeben werden.

```
G_vertex buildTree(G_graph &g)
{
    unsigned vCount, i, rootNo, alphaNo, omegaNo;

    cout << "Wieviele Knoten hat der Graph: ";
    cin >> vCount;

    for (i=1; i<=vCount; ++i)
        g.createVertex(i);

    cout << vCount << " Knoten wurden erzeugt." << endl
         << endl
         << "Welche Kanten hat der Graph?" << endl
         << "Eingabe von 0 als Startknoten beendet die Eingabe."
         << endl
         << endl;

    while (1) {
        cout << "Startknoten: ";
        cin >> alphaNo;
        if (alphaNo==0)
            break;
        if (!g.isVertex(g.getV(alphaNo)))
        {
            cout << alphaNo << " ist eine ungueltige Knotennummer."
                 << endl;
            continue;
        }
        cout << "Endknoten: ";
```

```
            cin >> omegaNo;
            if (!g.isVertex(g.getV(omegaNo)))
            {
                cout << omegaNo << " ist eine ungueltige Knotennummer."
                     << endl;
                continue;
            }
            g.createEdge(g.getV(alphaNo), g.getV(omegaNo));
            cout << "Kante von " << alphaNo << " nach "
                 << omegaNo << " wurde erzeugt." << endl
                 << endl;
        }

        cout << endl
             << "Welcher Knoten ist der Wurzelknoten: ";
        cin >> rootNo;
        cout << endl;

        return g.getV(rootNo);
    }
```

Zunächst wird die Anzahl der Knoten `vCount` eingegeben und entsprechend viele Knoten mit den Nummern 1 bis `vCount` mit der Methode **G_graph::createVertex** erzeugt. Zur Eingabe der Kanten wird die Nummer des Anfangsknoten `alphaNo` und die Nummer des Endknoten `omegaNo` eingegeben. Die Zulässigkeit der eingegebenen Nummern wird mit der Methode **G_graph::isVertex** überprüft. Die Methode **G_graph::getV** liefert dabei den zu einer Nummer gehörenden Knoten. Bei zulässigen Nummern wird die gewünschte Kante mit **G_graph::createEdge** erzeugt. Die Eingabe der Kanten wird durch Eingabe von 0 als Startknotennummer beendet. Zuletzt wird noch die Nummer des Wurzelknotens `rootNo` eingegeben und der Wurzelknoten an die aufrufende Funktion zurückgegeben.[7]

Schließlich noch die Funktion `traverse`:

```
void traverse(G_graph &g, G_vertex root, unsigned depth)
{
  G_edge e;
  int    i;

    if (depth>=g.vertexCount())
    {
        // circle detected
        cerr << "Der Graph enthaelt einen Kreis." << endl;
        abort();
    }
```

[7] Es wird nicht getestet, ob es sich bei dem eingegeben Graphen um einen Baum handelt.

```
  for (i=0; i<depth; ++i)
    cout << "..";

  cout << "Knoten(" << g.getVNo(root) << ")" << endl;

  G_forAllOutEdges(g, root, e)
  {
    traverse(g, g.omega(e), depth+1);
  }
}
```

Die Funktion selbst gibt nach dem Test auf einen Kreis[8] entsprechend der Rekursionstiefe viele Punktpaare und dann die Identifikationsnummer des gerade betrachteten Knoten `v` mittels der Methode **G_graph::getVNo** aus. Danach werden alle Kanten in $\Lambda^+(v)$ bearbeitet. Der Zugriff auf $\Lambda^+(v)$ erfolgt durch das Makro **G_forAllOutEdges**. Da die Endknoten `g.omega(e)` dieser Kanten `e` aus $\Lambda^+(v)$ die Kindknoten des gerade betrachteten Knoten `v` sind, wird dann `traverse` mit einer um eins erhöhten Rekursionstiefe für die Kindknoten erneut aufgerufen.

1.3.3 Übersetzen und Binden

Sobald der Quelltext unter dem Namen `treeTrav.c` abgespeichert ist, kann er mit

```
$CC -g -I$EMSINC treeTrav.c $EMSLIB/libgraphUdebug.a -o treeTrav
```

mit dem in der Environmentvariable `CC` angegebenen *C++*-Compiler übersetzt und mit der nötigen EMS-Library gebunden werden. Dazu sind die Environmentvariable des Graphenlabor zu setzen (siehe Abschnitt 2.1, S. 38).[9] Einfacher kann man es sich mit dem *make*-Kommando und einem entsprechenden *Makefile* machen. Ein möglicher Makefile steht im Verzeichnis `$EMSDEMO` zur Verfügung.

1.3.4 Ausführen

Sobald der Quelltext fehlerfrei übersetzt und gebunden worden ist, kann das Programm `treeTrav` ausgeführt werden. Für den Beispielbaum erhält man folgendes Ablaufprotokoll:

```
Wieviele Knoten hat der Graph:  14
14 Knoten wurden erzeugt.
Welche Kanten hat der Graph?
Eingabe von 0 als Startknoten beendet die Eingabe.
Startknoten:  1
Endknoten:  2
Kante von 1 nach 2 wurde erzeugt.
```

[8] Ein Kreis wird dadurch erkannt, daß die Rekursionstiefe die Knotenanzahl des Graphen überschreitet.
[9] Zur Wahl des Compilers siehe Anhang B.2, S. 81.

```
Startknoten:  1
Endknoten:  5
Kante von 1 nach 5 wurde erzeugt.

Startknoten:  1
Endknoten:  6
Kante von 1 nach 6 wurde erzeugt.

Startknoten:  2
Endknoten:  3
Kante von 2 nach 3 wurde erzeugt.

Startknoten:  2
Endknoten:  4
Kante von 2 nach 4 wurde erzeugt.

Startknoten:  6
Endknoten:  7
Kante von 6 nach 7 wurde erzeugt.

Startknoten:  6
Endknoten:  8
Kante von 6 nach 8 wurde erzeugt.

Startknoten:  6
Endknoten:  12
Kante von 6 nach 12 wurde erzeugt.

Startknoten:  6
Endknoten:  13
Kante von 6 nach 13 wurde erzeugt.

Startknoten:  6
Endknoten:  14
Kante von 6 nach 14 wurde erzeugt.

Startknoten:  8
Endknoten:  9
Kante von 8 nach 9 wurde erzeugt.

Startknoten:  8
Endknoten:  10
Kante von 8 nach 10 wurde erzeugt.

Startknoten:  8
Endknoten:  11
Kante von 8 nach 11 wurde erzeugt.

Startknoten:  0

Welcher Knoten ist der Wurzelknoten:  1

Knoten(1)
..Knoten(2)
....Knoten(3)
....Knoten(4)
```

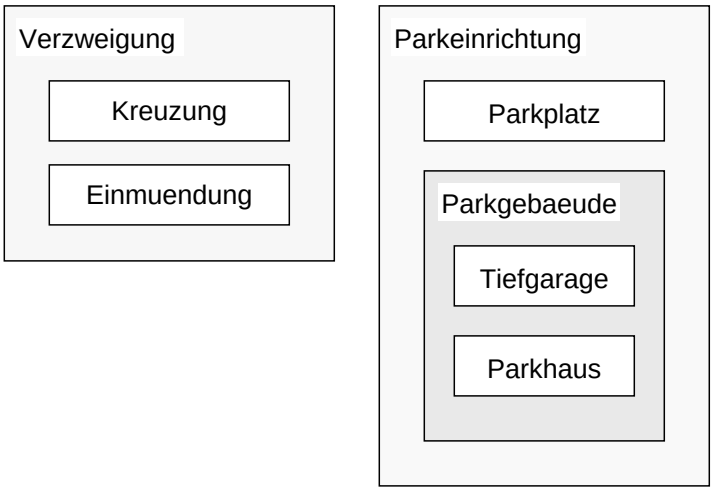


Abbildung 1: ER-Diagramm des Typisierungsbeispiels

```
..Knoten(5)
..Knoten(6)
....Knoten(7)
....Knoten(8)
......Knoten(9)
......Knoten(10)
......Knoten(11)
....Knoten(12)
....Knoten(13)
....Knoten(14)
```

1.4 Typisierung

Bei vielen Anwendungen von Graphen haben nicht alle Knoten bzw. Kanten ähnliche Bedeutungen. Vielmehr kann man Teilmengen von Knoten bzw. Kanten ähnlicher Bedeutung identifizieren.

Wenn man z.B. einen Stadtplan auf einen Graphen abbilden will, liegt es nahe, Straßenabschnitte als Kanten und Kreuzungen, Einmündungen, Parkplätze, Parkhäuser und Tiefgaragen als Knoten zu modellieren. Bei den Knoten fällt auf, daß man sie grob in zwei Mengen zerlegen kann, nämlich alle Verzweigungspunkte (Kreuzungen und Einmündungen) und alle Parkeinrichtungen, die ihrerseits wieder feiner zerlegt werden können (siehe Abb.1).

Die Einteilung von Knoten oder Kanten zu unterschiedlichen Mengen kann durch *Typisierung* realisiert werden. Die unterschiedlichen Mengen werden dabei durch *Typen* repräsentiert. Die Zugehörigkeit eines Knotens bzw. einer Kante zu einer bestimmten Menge wird dann dadurch ausgedrückt, daß dem Knoten bzw. der Kante der diese Menge repräsentierende Typ zugewiesen wird. Falls eine Menge, die durch einen Typ A repräsentiert wird, Teilmenge der durch Typ B repräsentierten Menge ist, ist A ein *Untertyp* von B (A *is-a* B) und B ein *Obertyp* von A. Z.B. ist `Parkhaus` ein Untertyp

von `Parkgebaeude`. In der Abbildung wird die Untertyprelation dadurch ausgedrückt, daß der Untertyp innerhalb des Obertyps steht. Die Untertyprelation ist reflexiv und transitiv, muß aber nicht notwendigerweise antisymmetrisch sein.

1.4.1 Realisierung

Im Graphenlabor wird Typisierung realisiert, indem Graphen ein geeignetes *Typsystem* (**G_typeSystem**-Instanz) zugeordnet wird. Dieses Typsystem beschreibt alle verwendeten Typen (**G_type**-Instanzen) und die Subtyp-Relation zwischen ihnen. Die Beschreibung eines Typs enthält eine innerhalb des Typsystems eindeutige Typbezeichnung (als String) und evtl. ein Schema für die Attributierung der Knoten oder Kanten des Typs (siehe dazu Abschnitt 1.5, S. 19). Jedes Typsystem enthält einen Nulltyp mit der Bezeichnung `"NULL"`, der automatisch Obertyp aller anderen Typen ist. Dem Nulltyp ist kein Attributierungsschema zugeordnet.

Jeder Knoten und jede Kante eines Graphen hat *genau einen* Typ des dem Graphen zugeordneten Typsystems.[10] Jeder Knoten und jede Kante hat nach der Erzeugung mit **G_graph::createVertex** oder **G_graph::createEdge** den Typ `"NULL"`, doch kann der Typ mit den Methoden **G_graph::setVType** für Knoten und **G_graph::setEType** für Kanten geändert werden.

1.4.2 Beispiel

In diesem Beispiel soll zunächst das oben beschriebene Typsystem definiert und anschließend eine Graphinstanz dazu erzeugt werden.

Definition des Typsystems

Um das oben beschriebene Typsystem im Graphenlabor anzulegen, kann man z.B. wie folgt vorgehen:

```
#include <graph.h>

G_typeSystem planTypSystem;

G_type verzweigung(planTypSystem, "Verzweigung");
G_type kreuzung(planTypSystem, "Kreuzung");
G_type einmuendung(planTypSystem, "Einmuendung");
G_type parkeinrichtung(planTypSystem, "Parkeinrichtung");
G_type parkplatz(planTypSystem, "Parkplatz");
G_type parkgebaeude(planTypSystem, "Parkgebaeude");
G_type parkhaus(planTypSystem, "Parkhaus");
G_type tiefgarage(planTypSystem, "Tiefgarage");
```

[10] Sollte z.B. ein Knoten eigentlich mehrere Typen haben, weil er unterschiedlichen Mengen angehört, ist es beim Graphenlabor notwendig, einen gemeinsamen Untertyp zu definieren und dem Knoten diesen gemeinsamen Untertyp zuzuordnen.

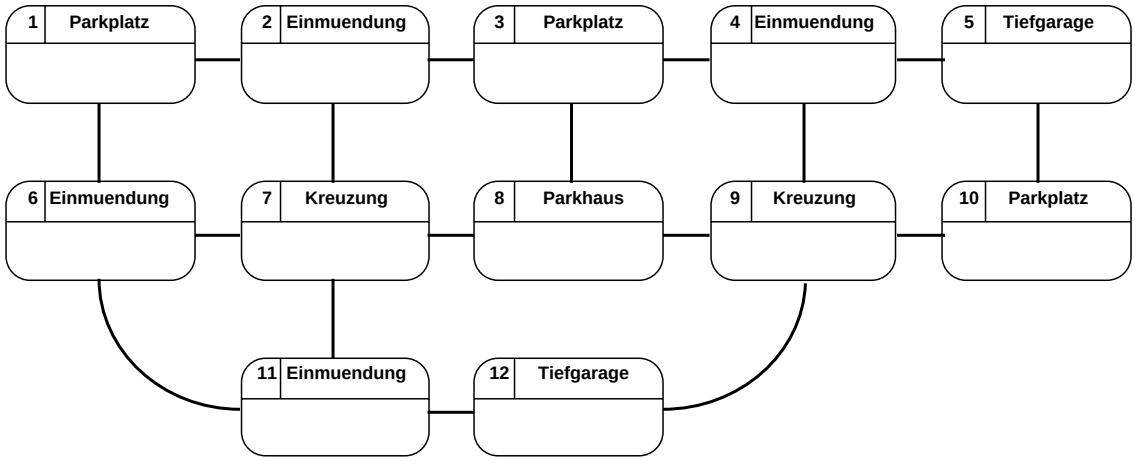


Abbildung 2: Graph des Typisierungsbeispiels

```
void initPlanTypSystem()
{
    planTypSystem.setIsA(kreuzung, verzweigung);
    planTypSystem.setIsA(einmuendung, verzweigung);

    planTypSystem.setIsA(parkplatz, parkeinrichtung);
    planTypSystem.setIsA(parkgebaeude, parkeinrichtung);
    planTypSystem.setIsA(parkhaus, parkgebaeude);
    planTypSystem.setIsA(tiefgarage, parkgebaeude);
}
```

Zunächst wird ein (bis auf den Nulltyp) leeres Typsystem `planTypSystem` durch den Konstruktor **G_typeSystem::G_typeSystem** erzeugt. Dann werden die benötigten Typen `verzeigung`, `einmuendung` usw. als Instanzen der Klasse **G_type** erzeugt und durch den Konstruktor **G_type::G_type** im Typsystem `planTypSystem` eingetragen. Anschließend wird die Subtyp-Relation auf den Typen durch die Methode **G_typeSystem::setIsA** beschrieben. Der transitive Abschluß auf der Relation erfolgt automatisch, d.h. nach Aufruf von `initPlanTypSystem` ist z.B. `tiefgarage` ein Untertyp von `parkeinrichtung`.

Erzeugen einer Graphinstanz

Im folgenden soll nun der Plan aus Abb.2 als Graph repräsentiert werden. Nach der Beschreibung des Typsystems kann ein Graph zu diesem Typsystem erzeugt werden. Dazu wird der Konstruktor **G_graph::G_graph** mit einem Zeiger auf das zu verwendende Typsystem als erstem Parameter aufgerufen[11].

```
G_graph plan(&planTypSystem);
```

[11] Erfolgt der Konstruktoraufruf ohne Parameter, ist der erzeugte Graph nicht typisiert.

Jetzt können Knoten erzeugt und ihnen mit der Methode **G_graph::setVType** die gewünschten Typen zugewiesen werden. Anschließend kann man sie durch Kanten miteinander verbinden.

```
void initPlan()
{
    plan.setVType(plan.createVertex(1), parkplatz);
    plan.setVType(plan.createVertex(2), einmuendung);
    plan.setVType(plan.createVertex(3), parkplatz);
    plan.setVType(plan.createVertex(4), einmuendung);
    plan.setVType(plan.createVertex(5), tiefgarage);
    plan.setVType(plan.createVertex(6), einmuendung);
    plan.setVType(plan.createVertex(7), kreuzung);
    plan.setVType(plan.createVertex(8), parkhaus);
    plan.setVType(plan.createVertex(9), kreuzung);
    plan.setVType(plan.createVertex(10), parkplatz);
    plan.setVType(plan.createVertex(11), einmuendung);
    plan.setVType(plan.createVertex(12), tiefgarage);

    plan.createEdge(plan.getV(1), plan.getV(2));
    plan.createEdge(plan.getV(2), plan.getV(3));
    plan.createEdge(plan.getV(3), plan.getV(4));
    plan.createEdge(plan.getV(4), plan.getV(5));
    plan.createEdge(plan.getV(1), plan.getV(6));
    plan.createEdge(plan.getV(2), plan.getV(7));
    plan.createEdge(plan.getV(4), plan.getV(9));
    plan.createEdge(plan.getV(5), plan.getV(10));
    plan.createEdge(plan.getV(6), plan.getV(7));
    plan.createEdge(plan.getV(7), plan.getV(8));
    plan.createEdge(plan.getV(8), plan.getV(9));
    plan.createEdge(plan.getV(9), plan.getV(10));
    plan.createEdge(plan.getV(6), plan.getV(11));
    plan.createEdge(plan.getV(7), plan.getV(11));
    plan.createEdge(plan.getV(9), plan.getV(12));
    plan.createEdge(plan.getV(11), plan.getV(12));
}
```

Man kann den erzeugten Graphen jetzt z.B. unter Beachtung von Typrestriktionen traversieren.

```
    G_vertex v;

    cout << "Alle Verzweigungsknoten sind:" << endl;
    G_forAllVerticesWithType(plan, verzweigung, v)
    {
        plan.printVertex(cout, v);
    }
```

```
    cout << "Alle Parkeinrichtungen sind:" << endl;
    G_forAllVerticesWithType(plan, parkeinrichtung, v)
    {
        plan.printVertex(cout, v);
    }
```

Dabei wird folgende Ausgabe erzeugt:

```
Alle Verzweigungsknoten sind:
vertex(2):
 -> ., 3(2), 7(6),
 <- 1(-1), ., .,
Type #3:Einmuendung
no attribute

vertex(4):
 -> ., 5(4), 9(7),
 <- 3(-3), ., .,
Type #3:Einmuendung
no attribute

vertex(6):
 -> ., 7(9), 11(13),
 <- 1(-5), ., .,
Type #3:Einmuendung
no attribute

vertex(7):
 -> ., ., 8(10), 11(14),
 <- 2(-6), 6(-9), ., .,
Type #2:Kreuzung
no attribute

vertex(9):
 -> ., ., 10(12), 12(15),
 <- 4(-7), 8(-11), ., .,
Type #2:Kreuzung
no attribute

vertex(11):
 -> ., ., 12(16),
 <- 6(-13), 7(-14), .,
Type #3:Einmuendung
no attribute

Alle Parkeinrichtungen sind:
```

```
vertex(1):
 -> 2(1), 6(5),
 <- ., .,
Type #5:Parkplatz
no attribute

vertex(3):
 -> ., 4(3),
 <- 2(-2), .,
Type #5:Parkplatz
no attribute

vertex(5):
 -> ., 10(8),
 <- 4(-4), .,
Type #9:Tiefgarage
no attribute

vertex(8):
 -> ., 9(11),
 <- 7(-10), .,
Type #8:Parkhaus
no attribute

vertex(10):
 -> ., .,
 <- 5(-8), 9(-12),
Type #5:Parkplatz
no attribute

vertex(12):
 -> ., .,
 <- 9(-15), 11(-16),
Type #9:Tiefgarage
no attribute
```

Auslagern in Dateien

Soll das erzeugte Typsystem und der erzeugte Graph in einem anderen Programm wiederverwendet werden, kann das Typsystem und der Graph in Dateien abgelegt werden. Die Typsystemdatei enthält alle Typen sowie die Untertypbeziehung, die Graphendatei neben der Graphenstruktur die Zuordnung der Typen zu Knoten und Kanten.

```
planTypSystem.store("typedemo.t");
plan.store("typedemo.g");
```

Wir wollen den so gespeicherten Graphen im Programm `typedemo2.c` wieder einlesen. Dazu muß zunächst das Typsystem gelesen werden. Dazu steht in der Klasse **G_typeSystem** ein entsprechender Konstruktor zur Verfügung:

```
G_typeSystem planTypSystem("typedemo.t");
```

Nach dem Lesen des Typsystems kann man sich auf bestimmte Typen nur mittels ihrer Typbezeichnung beziehen, da bislang keine *C++*-Bezeichner für die Typinstanzen vorhanden sind. Der Bezug auf einen bestimmten Typ erfolgt mit der Methode **G_typeSystem::getType**.

```
const G_type &verzweigung=planTypSystem.getType("Verzweigung");
const G_type &parkeinrichtung=planTypSystem.getType("Parkeinrichtung");
```

Nachdem das Typsystem geladen ist, kann auch der Graph mit der statischen Methode **G_graph::restore** gelesen werden. Diese Methode liefert im Erfolgsfall einen Zeiger auf den gelesenen Graphen, andernfalls einen `NULL`-Zeiger.

```
G_graph *pPlan;

pPlan=G_graph::restore("typedemo.g", &planTypSystem);
```

Um die gleiche Ausgabe wie das letzte Programm zu erzeugen, ist dann nur noch der folgende Code nötig:

```
cout << "Alle Verzweigungsknoten sind:" << endl;
G_forAllVerticesWithType(*pPlan, verzweigung, v)
{
    pPlan->printVertex(cout, v);
}

cout << "Alle Parkeinrichtungen sind:" << endl;
G_forAllVerticesWithType(*pPlan, parkeinrichtung, v)
{
    pPlan->printVertex(cout, v);
}
```

1.5 Attributierung

Für viele Anwendungen von Graphen ist es nötig, neben der reinen Graphenstruktur noch zusätzliche Daten an Knoten oder Kanten zu verwalten. Sollen Knoten oder Kanten mit Daten versehen werden, spricht man von *Attributierung* des Graphen. Man kann zwei Arten der Attributierung unterscheiden:

- Häufig reicht zur Abbildung eines Realitätsauschnitts auf einen Graphen die Unterscheidung von Knoten- oder Kantenarten mittels der Typisierung nicht aus. Vielmehr ist für einige (oder alle) Knoten und Kanten zusätzliche Information zu speichern. Die Art der Information hängt dabei in der Regel vom Typ des Knotens oder

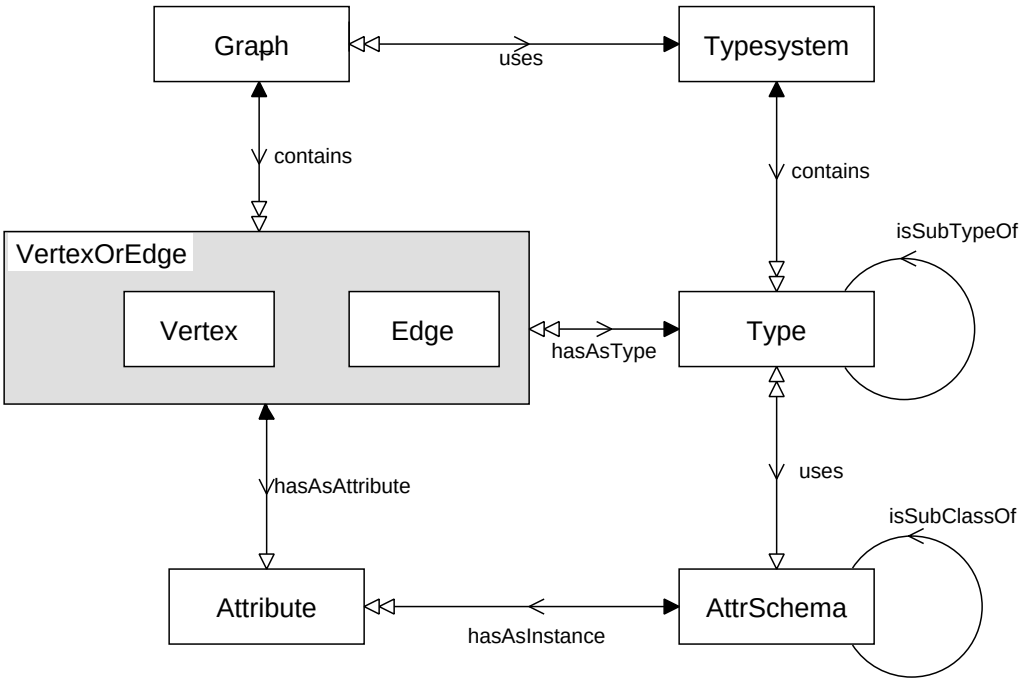


Abbildung 3: ER-Diagramm der Attributierung

der Kante ab. Im Beispiel aus dem vorangehenden Abschnitt könnte es notwendig sein, für die Parkeinrichtungen die Namen der Einrichtung, für Parkgebäude darüberhinaus die Öffnungszeiten zu speichern.

Im folgenden meint *Attributierung* immer Attributierung in diesem Sinne. Sie ist Thema dieses Abschnitts.

- Für viele Verfahren der Graphentheorie ist es notwendig, kurzfristig Steuerinformation an Knoten oder Kanten zu schreiben, die nach Ablauf des Verfahrens wieder gelöscht werden kann. Z.B. ist es für ein Suchverfahren wichtig, welche Knoten bereits behandelt wurden und welche nicht. Attributierung dieser Art wird im folgenden als *temporäre Attributierung* bezeichnet und im Abschnitt 1.6, S. 30 behandelt.

1.5.1 Realisierung

Bei der Definition eines Knoten- bzw. Kantentyps im Typsystem kann ein Attributschema vereinbart werden. Jeder Knoten und jede Kante dieses Typs *muß* dann ein Attribut tragen, das dem zugeordnetem Attributschema genügt. Beim Auslagern und Einlesen eines Graphen in bzw. aus einer Datei werden die Attribute berücksichtigt. Abb.3 zeigt den logischen Zusammenhang zwischen den relevanten Graphenlaborklassen.

Handelt es sich bei dem Typ um einen Untertyp verschiedener Obertypen mit unterschiedlichen Attributschemata, muß ein gemeinsames Attributschema erzeugt und dem

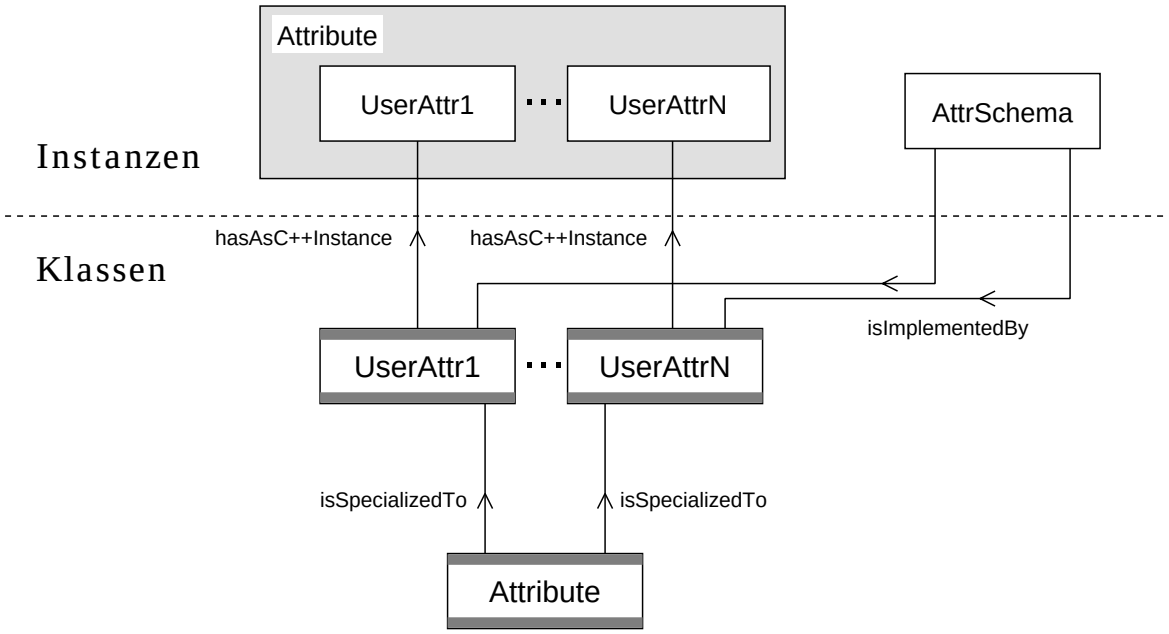


Abbildung 4: Implementierung von Attributen
In der Darstellung wird auf die mögliche Hierarchie (Unterklassen-Beziehung) innerhalb der Attributklassen UserAttr1 bis UserAttrN verzichtet.

Untertyp zugeordnet werden.[12]

Jedes Attributschema wird durch eine eigene *C++*-Klasse (*Attributklasse*) implementiert. Zusätzlich muß das Attributschema dem Labor durch die Erzeugung eines Objekts der Klasse **G_attrSchema** bekanntgegeben werden. Dieses Objekt beinhaltet eine global eindeutige Attributschemabezeichnung (als String) sowie (aus technischen Gründen) einen Verweis auf eine geeignete Methode zum Laden von Attributen des Schemas und realisiert so die Anbindung der vom Benutzer implementierten Attributklassen an das Graphenlabor. Der Zusammenhang zwischen den einzelnen Attributklassen, Attributen und Schemainstanzen ist in Abb.4 dargestellt.

Die Attributklassen selbst sind von der Klasse **G_attribute** abgeleitet. Sie enthalten u.a.:

- die gewünschten Datenelemente zur Aufnahme der für die Modellierung erforderlichen Information
- eine virtuelle Methode **print** zur Ausgabe von Attributen in einem möglichst übersichtlichen Format
- eine virtuelle Methode **store** zum Ausschreiben von Attributen in Dateien in einem möglichst leicht maschinell zu lesenden Format
- ggf. einen virtuellen Destruktor zum Löschen von mittels Zeigern realisierten Attributwerten

[12] Der von *C++* verwendete Mechanismus der mehrfachen Vererbung führt dabei leider zu Problemen, siehe Anhang A, S. 79.

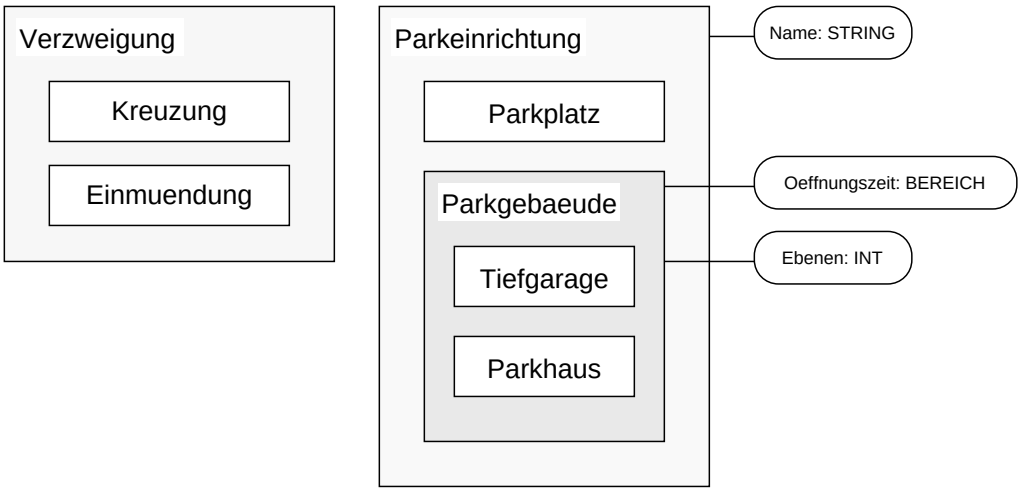


Abbildung 5: ER-Diagramm des Attributierungsbeispiels

- eine statische Methode **restore** zum Einlesen von Attributwerten aus Dateien und zur Erzeugung einer Instanz dieser Klasse mit den gelesenen Werten (auf diese Methode wird in der **G_attrSchema**-instanz des Schemas verwiesen)

Aus technischen Gründen enthalten die Attributklassen noch ein paar zusätzliche Methoden. Natürlich kann der Benutzer zusätzliche Methoden zur Attributbehandlung hinzufügen.

Vereinfachte Erzeugung von Attributschemata

Um die Definition von Attributschemata zu vereinfachen, steht im Verzeichnis `$EMSBIN` das Hilfsprogramm `makeattr` zur Verfügung. Es wird mit einem Parameter *ident* aufgerufen und erzeugt dann im aktuellen Verzeichnis die Dateien *ident*`Attr.h` und *ident*`Attr.c`, die dann modifiziert werden müssen.

1.5.2 Beispiel

Wir nehmen das Beispiel aus dem letzten Abschnitt wieder auf. Bei den Parkeinrichtungen soll der Name der Einrichtung gespeichert werden, bei den Parkgebäuden die Anzahl der Parkebenen und die Öffnungszeiten. Die gewünschte Attributierung ist in Abb.5 dargestellt. Wir benötigen also ein Attributschema für die Parkgelegenheiten und ein Schema für die Parkgebäude.

Wir definieren zunächst die Attributschemata und ordnen sie später den entsprechenden Typen zu. Anschließend erzeugen wir einen Graphen zu diesem Typsystem.

Definition der Attributschemata

Mit dem Aufruf „`makeattr parkeinrichtung`“ erzeugt man die Dateien `parkeinrichtungAttr.h` und `parkeinrichtungAttr.c`. Die erzeugten Dateien haben dann folgenden Inhalt:

parkeinrichtungAttr.h

```
/* parkeinrichtungAttr.h, generated by makeattr */

#ifndef _PARKEINRICHTUNG_DEFINED
#define _PARKEINRICHTUNG_DEFINED

#ifndef _GRAPH_DEFINED
#include <graph.h>        /* include the declarations of graph lab */
#endif

class parkeinrichtungAttribute: public G_attribute
    // change the base class, if the actual class
    // specializes another attribute class

{
public:
    // insert your required data members here

virtual ostream &print(ostream &);
    // pretty output, called by G_graph::print etc.

virtual void store(ostream &);
    // easy to read output, called by G_graph::store

static G_attribute *restore(istream &iS, unsigned length);
    // restore attribute from the next 'length' characters in
    // stream 'iS' written by 'store',
    // called by G_graph::restore

virtual ~parkeinrichtungAttribute();
    // virtual destructor, optional
    // you should provide a destructor if you use a pointer structure
    // called by G_graph::deleteVertex etc.

    // feel free to declare additional methods to
    // handle your attributes


    // do not modify the following declarations !!!
public:
virtual G_attrNo getAttrNo();
```

```
static const G_attrSchema &getSchema();

protected:
virtual void *getPAttr() { return this; };

private:
static const G_attrSchema *pAttrSchema;
};

#endif
/********* parkeinrichtungAttr.h **********************************/
```

parkeinrichtungAttr.c

```
/* parkeinrichtungAttr.c, generated by makeattr */

#include "parkeinrichtungAttr.h"

static const char schemaID[]="PARKEINRICHTUNG";

ostream &parkeinrichtungAttribute::print(ostream &oS)
{
    // write code to print an attribute in a pretty format,
    // called by G_graph::print etc.
    return oS;
}

void parkeinrichtungAttribute::store(ostream &oS)
{
    // write code to store an attribute in a practical format
    // so that it can easily be restored,
    // called by G_graph::store
}

G_attribute *parkeinrichtungAttribute::restore(istream &iS,
                                               unsigned length)
{
    // write code to restore an attribute from the next 'length'
    // characters in stream 'iS' written by 'store',
    // called by G_graph::restore

    parkeinrichtungAttribute *pAttr=new parkeinrichtungAttribute;
    // fill the data members with the information found in 'iS'
    return pAttr;
}

parkeinrichtungAttribute::~parkeinrichtungAttribute()
```

```
{
    // write code to destroy an attribute, if you use
    // a pointer structure,
    // called by G_graph::deleteVertex etc.
}

    // insert the implementation of your additional methods here


    // do not modify the following definitions !!!
G_attrNo parkeinrichtungAttribute::getAttrNo()
{
    return getSchema().getAttrNo();
}

const G_attrSchema *parkeinrichtungAttribute::pAttrSchema= &getSchema();


const G_attrSchema &parkeinrichtungAttribute::getSchema()
{
    if (!pAttrSchema)
        pAttrSchema=new G_attrSchema(schemaID, restore);
    return *pAttrSchema;
}

    // insert the implementation of your additional methods here

/********* parkeinrichtungAttr.c ***********************************/
```

Modifikation der Dateien

Diese Dateien müssen natürlich noch ergänzt werden. Wir beginnen mit `gebuehrAttr.h`:

- Hinter `// insert your required data members here` fügt man das benötigte Datenelement ein:

  ```
  char *name;
  ```

- Ferner benötigen wir noch einen Konstruktor, der `parkeinrichtung` sofort initialisiert, und einen Konstruktor ohne Parameter für das Erzeugen eines Attributs beim Einlesen eines Graphen, ohne daß die Attributwerte bereits bekannt sind. Wir deklarieren sie hinter `// feel free to declare additional methods to handle your attributes`:

  ```
  public:
      parkeinrichtungAttribute(const char *name)
  ```

```
        { this->name=strdup(name); };

protected:
    parkeinrichtungAttribute() {};
```

Beim Namen der Parkeinrichtung wird nicht der Zeiger auf den String kopiert, sondern der String selbst kopiert. Der dazu von der Speicherverwaltung angeforderte Platz muß natürlich bei der Freigabe eines Attributes ebenfalls (durch den Destruktor) freigegeben werden.

Die Änderung in `parkeinrichtungAttr.h` ist damit abgeschlossen. Die Datei `parkeinrichtungAttr.c` muß wie folgt geändert werden:

- In der Methode `print` ersetzt man „`return oS;`“ durch

  ```
  return oS << name;
  ```

- In der Methode `store` fügen wir folgende Zeile ein:

  ```
  oS << strlen(name) << ' ' << name;
  ```

  Um vor dem Lesen des Strings ausreichend viel Speicherplatz anfordern zu können, wird zunächst die Länge des Strings und dann der String selbst gespeichert.

- In der Methode `restore` fügen wir folgende Zeile ein:

  ```
  unsigned nameLength;

  iS >> nameLength;
  iS.ignore(1);
  pAttr->name=new char[nameLength+1];
  iS.get(pAttr->name, nameLength+1, '\0');
  ```

- Im Destruktor muß der von `name` belegt Speicherplatz freigegeben werden.

  ```
  delete name;
  ```

Schema für Parkgebäude

Als nächstes erzeugt man das Attributschema für die Parkgebäude mittels „`makeattr parkgebaeude`“. In `gebaeudeAttr.h` ändert man an folgenden Stellen:

- `parkgebaeudeAttribute` soll von `parkeinrichtungAttribute` abgeleitet werden. Dazu müssen die Deklarationen der Basisklassen bekannt sein. Zu Beginn ist also einzufügen:

  ```
  #include "parkeinrichtungAttr.h"
  ```

- Der Datentyp `bereich` muß definiert werden:

```
struct bereich {
    int von;
    int bis;
};
```

- Die Basisklasse muß in der Klassendefinition angegeben werden:

```
class gebaeudeAttribute: public parkeinrichtungAttribute
```

- Hinter `// insert your required data members here` fügt man ein:

```
    int ebenen;
    bereich oeffnungszeit;
```

- Da keine zusätzlichen Speicherbereiche auf dem Heap angelegt werden, kann der Destruktor auskommentiert werden:

```
// virtual ~parkgebaeudeAttribute();
```

- Hinter `// feel free to declare additional methods to handle your attributes` fügen wir noch zwei Konstruktoren ein:

```
public:
    parkgebaeudeAttribute(char *name, int ebenen, int von, int bis)
     : parkeinrichtungAttribute(name)
    { this->ebenen=ebenen; oeffnungszeit.von=von;
      oeffnungszeit.bis=bis; };

protected:
    parkgebaeudeAttribute() {};
```

Folgende Änderungen in `parkgebaeudeAttr.c` sind nötig:

- In Methode `print` ersetzt man `return oS;` durch

```
    parkeinrichtungAttribute::print(oS);
    oS << ", " << ebenen << " Ebenen, geoeffnet von "
       << oeffnungszeit.von << " bis " << oeffnungszeit.bis;
```

  Es wird zunächst die `print`-Methode der Basisklasse aufgerufen, und dann werden die zusätzlichen Attributwerte ausgegeben.

- In Methode `store` fügt man ein:

```
parkeinrichtungAttribute::store(oS);
oS << ' ' << ebenen << ' ' << oeffnungszeit.von
   << ' ' << oeffnungszeit.bis;
```

- In Methode `restore` fügt man ein:

```
unsigned nameLength;

iS >> nameLength;
iS.ignore(1);
pAttr->name=new char[nameLength+1];
iS.get(pAttr->name, nameLength+1, '\0');
iS >> pAttr->ebenen >> pAttr->oeffnungszeit.von
   >> pAttr->oeffnungszeit.bis;
```

- Der Destruktor muß auch hier auskommentiert oder gelöscht werden.

Zuordnung der Schemata zu Typen

In der Datei `attrdemo.c` realisieren wir nun die Zuordnung der definierten Schemata zu den Typen des Typsystems. Dazu werden die **G_type**-Instanzen direkt mit dem Verweis auf das zu verwendende Attributschema konstruiert. Dieser Verweis erfolgt über das der Attributklasse zugeordnete Objekt der Klasse **G_attrSchema**.[13] Dieses Objekt kann durch die statische Methode `getSchema` der Attributklasse erreicht werden. Bei Typen ohne Attribute wird das Attributschema einfach weggelassen.

```
G_typeSystem planTypSystem;

G_type verzweigung(planTypSystem, "Verzweigung");
G_type einmuendung(planTypSystem, "Einmuendung");
G_type kreuzung(planTypSystem, "Kreuzung");

G_type parkeinrichtung(planTypSystem, "Parkeinrichtung",
                       parkeinrichtungAttribute::getSchema());
G_type parkplatz(planTypSystem, "Parkplatz",
                       parkeinrichtungAttribute::getSchema());
G_type parkgebaeude(planTypSystem, "Parkgebaeude",
                     parkgebaeudeAttribute::getSchema());
G_type parkhaus(planTypSystem, "Parkhaus",
                 parkgebaeudeAttribute::getSchema());
G_type tiefgarage(planTypSystem, "Tiefgarage",
                   parkgebaeudeAttribute::getSchema());
```

Die Definition der Subtyp-Relation erfolgt wie im Beispiel des letzten Abschnitts.

[13] Dadurch kann das Graphenlabor zum einen überprüfen, ob das einem Knoten oder einer Kante zugeordnete Attribut dem Schema des Typs des Knotens bzw. der Kante entspricht, zum anderen ist es beim Einlesen eines Graphen aus einer Datei möglich, die zu einem Typ passende Lesemethode für Attribute zu finden.

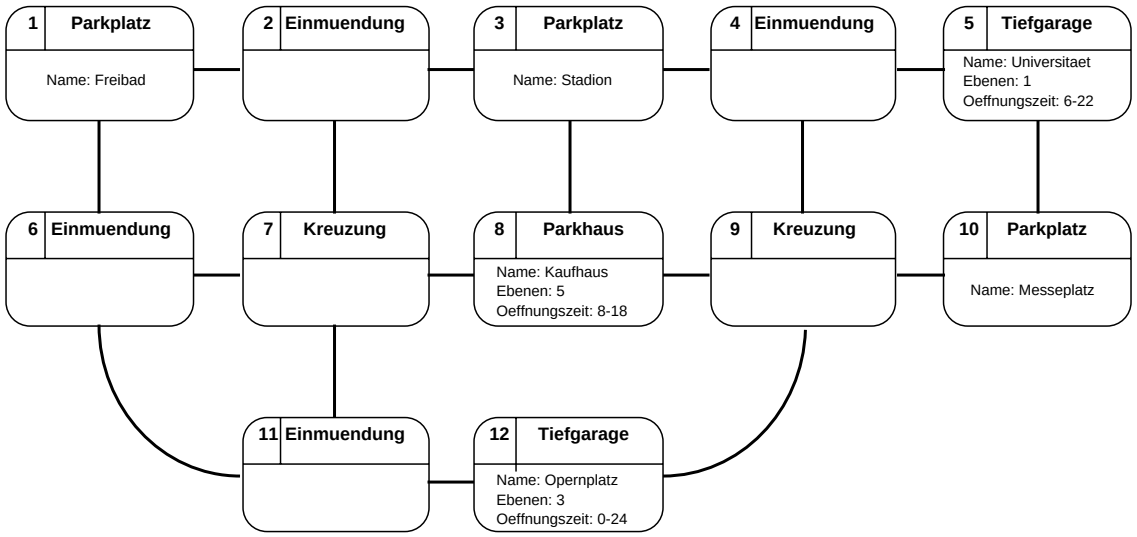


Abbildung 6: Graph des Attributierungsbeispiels

Anlegen von Attributen an Knoten oder Kanten

Es soll jetzt der Plan aus Abb.6 als Graph repräsentiert werden. Nach der Vereinbarung eines Attributschemas bei der Typdefinition kann einem Knoten bzw. einer Kante dieser Typ nur dann zugeordnet werden, wenn ihm bzw. ihr gleichzeitig ein Attribut des Schemas zugeordnet wird. Dazu wird die Methode **G_graph::setVType** mit einem Zeiger auf ein geeignetes Attribut aufgerufen. Ist dem Typ kein Attributschema zugeordnet, entfällt der Attributzeiger einfach.

```
G_graph plan(&planTypSystem);

void initPlan()
{
    plan.setVType(plan.createVertex(1), parkplatz,
                  new parkeinrichtungAttribute("Freibad"));
    plan.setVType(plan.createVertex(2), einmuendung);
    plan.setVType(plan.createVertex(3), parkplatz,
                  new parkeinrichtungAttribute("Stadion"));
    plan.setVType(plan.createVertex(4), einmuendung);
    plan.setVType(plan.createVertex(5), tiefgarage,
                  new parkgebaeudeAttribute("Universitaet",
                                            1, 6, 22));
    plan.setVType(plan.createVertex(6), einmuendung);
    plan.setVType(plan.createVertex(7), kreuzung);
    plan.setVType(plan.createVertex(8), parkhaus,
                  new parkgebaeudeAttribute("Kaufhaus", 5, 8, 18));
    plan.setVType(plan.createVertex(9), kreuzung);
    plan.setVType(plan.createVertex(10), parkplatz,
```

```
                new parkeinrichtungAttribute("Messeplatz"));
    plan.setVType(plan.createVertex(11), einmuendung);
    plan.setVType(plan.createVertex(12), tiefgarage,
                new parkgebaeudeAttribute("Opernplatz",
                                          3, 0, 24));
```

Wird nun z.B. der Knoten mit der Nummer 8 mit der Methode **G_graph::print** ausgegeben, erhält man folgende Ausgabe:

```
vertex(8):
 -> ., 9(11),
 <- 7(-10), .,
Type #7:Parkhaus
Attr #2:PARKGEBAEUDE
Kaufhaus, 5 Ebenen, geoeffnet von 8 bis 18
```

Um einzelne Attributwerte zu ändern, kann man mit der Methode **G_graph::getPVAttr** auf Knotenattribute bzw. **G_graph::getPEAttr** auf Kantenattribute zugreifen. Der Rückgabewert dieser Methoden hat den Typ `G_attribute *` und kann durch eine Typumwandlung in einen geeigneten Zeiger umgeformt werden.

Falls zum Beispiel die Kaufhausverwaltung beschließt, das Parkhaus um eine Ebene aufzustocken, kann man dem mit folgenden Anweisungen Rechnung tragen:

```
parkgebaeudeAttribute *pAttr=(parkgebaeudeAttribute *)
                              plan.getPVAttr(plan.getV(8));
++pAttr->ebenen;
```

Der Graph und das Typsystem können wie im letzten Abschnitt in Dateien abgespeichert werden.

```
planTypSystem.store("attrdemo.t");
plan.store("attrdemo.g");
```

Die Typsystemdatei enthält dabei die Zuordnung von Attributschemata zu Typen sowie die Subtyp-Relation, die Graphendatei neben der Graphenstruktur (Knoten, Kanten, Inzidenzen etc.) die Instanzen der Attributklassen.

1.6 Temporäre Attributierung

Viele Graphenalgorithmen erfordern die kurzfristige Speicherung von Kontrolldaten an Knoten oder Kanten von Graphen. Man könnte diese Daten in die Attributstruktur des Typsystems aufnehmen, aber im Grunde sind sie unabhängig von den der Modellierung dienenden Attributdaten.

1.6.1 Realisierung

Im Graphenlabor wird der Ansatz verfolgt, temporäre und typabhängige Attributierung zu trennen. Jeder Knoten und jede Kante kann Daten in Form von *temporären Attributen* tragen. Der Typ des Knotens bzw. Kante spielt dabei keine Rolle. Ein temporäres Attribut ist dabei ein Objekt einer von **G_tempAttribute** abgeleiteten Klasse (*temporärer Attributierungsklasse*). Eine solche Klasse kann folgende Methoden definieren:

- Eine virtuelle `print`-Methode zur Ausgabe des Attributs bei **G_graph::printVertex** etc.
- Einen virtuellen Destruktor, falls das Attribut Zeiger auf dynamisch angeforderte Speicherbereiche enthält, die freigegeben werden sollten.

Temporäre Attribute werden in *Schichten* angelegt. Es gibt Schichten zur Attributierung von Knoten und Schichten zur Attributierung von Kanten. Es sind immer nur die Attribute der zuletzt erzeugten Schicht für Knoten und die der zuletzt erzeugten Schicht für Kanten sichtbar.[14] Durch die Verwendung von Schichten ist es möglich, daß ein Verfahren, das temporäre Attribute verwendet, sich eines anderen Verfahrens bedient, das ebenfalls temporäre Attribute verwendet, ohne daß sich die Attributierungen gegenseitig stören.

Eine Schicht darf auch Elemente unterschiedlicher Attributierungsklassen enthalten. Dabei muß das Verfahren natürlich wissen, welcher Klasse ein Element angehört.

1.6.2 Beispiel

Es soll eine (ungerichtete) Tiefensuche in einem Graphen durchgeführt werden. Während des Ablaufs einer Tiefensuche ist es wichtig zu wissen, ob und über welche Kante ein bestimmter Knoten schon einmal besucht wurde. Diese Daten wollen wir als temporäre Attribute „an die Knoten schreiben“.

Dazu müssen wir erst einmal eine geeignete Attributierungsklasse definieren.

```
class dfsMarking : public G_tempAttribute
{
public:
    int     number;
    G_edge  parent;

            dfsMarking(unsigned n, G_edge p)
            { number=n; parent=p; };

    virtual ostream & print(ostream &oS)
    { return oS << "Number = " << number
                << ", parent = " << parent; };
};
```

[14] Die Schichten bilden also zwei Stapel.

Der Eintrag `number` gibt an, als wievielter Knoten der Knoten besucht wurde, und der Eintrag `parent`, über welche Kante (orientiert) der Knoten erreicht wurde. Das `parent`-Feld enthält **G_EdgeNull**, falls der Knoten ein Startknoten der Suche[15] ist. Wurde der fragliche Knoten noch nicht besucht, existiert kein temporäres Attribut an ihm.

In der Funktion `depthFirstSearch` wird zunächst eine neue Schicht für Knoten mit der Methode **G_graph::createVTemp** erzeugt. Dann wird in einer Schleife nach einem unmarkierten Knoten gesucht. Dieser ist dann der Startknoten einer Zusammenhangskomponente, die mit der rekursiven Funktion `dfs` abgesucht wird. Alle Knoten dieser Komponente sind danach markiert, so daß der nächste unmarkierte Knoten in einer noch nicht besuchten Komponente liegen muß. Wird kein unmarkierter Knoten mehr gefunden, ist die Suche abgeschlossen. Jetzt geben wir den temporär attributierten Graphen aus und löschen dann die Attributierungschicht.

```
void depthFirstSearch(G_graph &g)
{
    G_vertex v;
    int num=0;

    g.createVTemp();

    G_forAllVertices(g, v)
    {
         if (g.getPVTemp(v)==NULL)
         {
             cout << "Beginn einer Komponente" << endl;
             dfs(g, num, v, G_EdgeNull, 0);
             cout << "Ende der Komponente." << endl;
         }
    }

    cout << g;

    g.deleteVTemp();
}
```

In der rekursiven Funktion `dfs` wir der Graph `g` ab dem Knoten `v` abgesucht. Dazu wird der Knoten `v` zunächst als der als `num`-ter besuchte Knoten markiert, der über die Kante `parent` erreicht wurde. Danach wird der Knoten ausgegeben. Die Ausgabe erfolgt eingerückt entsprechend der in `level` verfolgten Rekursionstiefe mit der nicht abgedruckten Funktion `indent`. Anschließend werden alle mit dem Knoten `v` inzidenten Kanten `e` untersucht. Wurde der andere Knoten `w` dieser Kante noch nicht besucht, handelt es sich um eine Gerüstkante und die Suche wird ab `w` fortgesetzt. Andernfalls kann es sich um eine Sehne handeln. Die Sehnen sollen nur als Rückwärtskanten innerhalb des Gerüstes ausgerichtet ausgegeben werden. Dazu werden die Besuchsnummern der beteiligten Knoten verglichen und ggf. die Sehne ausgegeben.

[15] Es handelt sich dann um den Wurzelknoten des in der Zusammenhangskomponente aufgespannten Baums.

```
void dfs(G_graph &g, int &num, G_vertex v, G_edge parent, int level)
{
    G_edge e;
    G_vertex w;
    dfsMarking *pM1, *pM2;

    ++num;
    g.setPVTemp(v, pM1=new dfsMarking(num, parent));
    indent('.', level*2);
    cout << "Knoten(" << g.getVNo(v) << ')' << endl;

    G_forAllIncidentEdges(g, v, e)
    {
        w=g.thatV(e);
        pM2=(dfsMarking *)g.getPVTemp(w);
        if (pM2==NULL)
        {
            indent(' ', level*2);
            cout << "--- Geruestkante(" << g.getENo(e) << ')'
                 << endl;
            dfs(g, num, w, e, level+1);
        }
        else if (   !g.areEqualEdges(e, parent)
                 && pM2->number<pM1->number)
        {
            indent(' ', level*2);
            cout << "--- Sehne(" << g.getENo(e) << ") nach Knoten("
                 << g.getVNo(w) << ')' << endl;
        }
    }
}
```

Zum Test des Verfahrens lesen wir noch einmal den Graphen aus dem Beispiel in Abschnitt 1.5, S. 19 ein und rufen das Verfahren auf.

```
int main()
{
    G_typeSystem planTS("attrdemo.t");
    G_graph *pG;

    pG=G_graph::restore("attrdemo.g", &planTS);

    depthFirstSearch(*pG);

    return 0;
}
```

Man erhält mit diesem Programm dann folgende Ausgabe:

```
Beginn einer Komponente
Knoten(1)
--- Geruestkante(1)
..Knoten(2)
  --- Geruestkante(2)
....Knoten(3)
    --- Geruestkante(3)
......Knoten(4)
      --- Geruestkante(4)
........Knoten(5)
        --- Geruestkante(8)
..........Knoten(10)
          --- Geruestkante(-12)
............Knoten(9)
            --- Sehne(-7) nach Knoten(4)
            --- Geruestkante(-11)
..............Knoten(8)
              --- Geruestkante(-10)
................Knoten(7)
                --- Sehne(-6) nach Knoten(2)
                --- Geruestkante(-9)
..................Knoten(6)
                  --- Sehne(-5) nach Knoten(1)
                  --- Geruestkante(13)
....................Knoten(11)
                    --- Sehne(-14) nach Knoten(7)
                    --- Geruestkante(16)
......................Knoten(12)
                      --- Sehne(-15) nach Knoten(9)
Ende der Komponente.
Graph: 12 of 1000 vertices, 16 of 1000 edges.
vertices:
vertex(1):
 -> 2(1), 6(5),
 <- ., .,
Type #5:Parkplatz
Attr #1:PARKEINRICHTUNG
Freibad
temp attribute(1):
Number = 1, parent = 0

vertex(2):
 -> ., 3(2), 7(6),
 <- 1(-1), ., .,
Type #2:Einmuendung
```

```
no attribute
temp attribute(1):
Number = 2, parent = 1

vertex(3):
 -> ., 4(3),
 <- 2(-2), .,
Type #5:Parkplatz
Attr #1:PARKEINRICHTUNG
Stadion
temp attribute(1):
Number = 3, parent = 2

vertex(4):
 -> ., 5(4), 9(7),
 <- 3(-3), ., .,
Type #2:Einmuendung
no attribute
temp attribute(1):
Number = 4, parent = 3

vertex(5):
 -> ., 10(8),
 <- 4(-4), .,
Type #8:Tiefgarage
Attr #2:PARKGEBAEUDE
Universitaet, 1 Ebenen, geoeffnet von 6 bis 22
temp attribute(1):
Number = 5, parent = 4

vertex(6):
 -> ., 7(9), 11(13),
 <- 1(-5), ., .,
Type #2:Einmuendung
no attribute
temp attribute(1):
Number = 10, parent = -9

vertex(7):
 -> ., ., 8(10), 11(14),
 <- 2(-6), 6(-9), ., .,
Type #3:Kreuzung
no attribute
temp attribute(1):
Number = 9, parent = -10
```

```
vertex(8):
 -> ., 9(11),
 <- 7(-10), .,
Type #7:Parkhaus
Attr #2:PARKGEBAEUDE
Kaufhaus, 5 Ebenen, geoeffnet von 8 bis 18
temp attribute(1):
Number = 8, parent = -11

vertex(9):
 -> ., ., 10(12), 12(15),
 <- 4(-7), 8(-11), ., .,
Type #3:Kreuzung
no attribute
temp attribute(1):
Number = 7, parent = -12

vertex(10):
 -> ., .,
 <- 5(-8), 9(-12),
Type #5:Parkplatz
Attr #1:PARKEINRICHTUNG
Messeplatz
temp attribute(1):
Number = 6, parent = 8

vertex(11):
 -> ., ., 12(16),
 <- 6(-13), 7(-14), .,
Type #2:Einmuendung
no attribute
temp attribute(1):
Number = 11, parent = 13

vertex(12):
 -> ., .,
 <- 9(-15), 11(-16),
Type #8:Tiefgarage
Attr #2:PARKGEBAEUDE
Opernplatz, 3 Ebenen, geoeffnet von 0 bis 24
temp attribute(1):
Number = 12, parent = 16

edges:
edge(1): 1 -> 2
Type #0:NULL
```

```
no attribute

...
```

2 Nutzung des EMS-Graphenlabors

2.1 Verzeichnisstruktur des Graphenlabors

Zur Benutzung des Graphenlabor sollten zur Abkürzung von Verzeichnispfaden zu EMS-Dateien folgende Environmentvariable gesetzt werden:[16]

`EMS` :
: Wurzel des Directorybaums aller EMS-Dateien.

`EMSSRC` :
: Alle Quelltexte zum Labor.

`EMSINC` :
: Alle include-Dateien zur Übersetzung von EMS-Anwendungen.

`EMSLIB` :
: Alle Bibliotheken zum Binden von EMS-Anwendungen.

`EMSMSG` :
: Alle Meldungsdateien für die Laufzeitmeldungen des EMS-Graphenlabor.

`EMSDEMO` :
: Alle in diesem Handbuch aufgeführten Beispiele.

`EMSMAN` :
: Dieses Handbuch.

2.2 Übersetzen und Binden von Anwendungssoftware

2.2.1 Übersetzen

Zur Übersetzung von Quelltextdateien zu EMS-Anwendungen müssen diese nur die Datei `graph.h` im Verzeichnis `$EMSINC` mit dem `#include`-Befehl einfügen. Am einfachsten geschieht dies durch den Befehl `#include <lab.h>` im Quelltext und die Angabe des include-Verzeichnisses bei Aufruf des Compilers mit der Option `-I$(EMSINC)` in einem Makefile. Durch weitere Makro-Definitionen kann das Verhalten des Präprozessors noch gesteuert werden:

`-DNO_TRC` :
: Alle das Tracing (siehe Abschnitt 2.5, S. 41) betreffenden Anweisungen werden ignoriert.

`-DNO_CHK` :
: Alle das Checking (siehe Abschnitt 2.6, S. 43) betreffenden Anweisungen werden ignoriert.

[16] Die Werte für die Universität Koblenz stehen in Anhang B.1, S. 81.

2.2.2 Binden

Zum Binden einer EMS-Anwendung zu einem lauffähigen Programm muß eine der Bibliotheken `libgraph`*xxx*`.a` aus dem Verzeichnis `$EMSLIB` verwendet werden. Die Varianten sind:

`libgraphdebug.a` :
: Das ist die Bibliothek für noch in der Entwicklung befindliche Programme. Sie erlaubt uneingeschränkt Checking und Tracing. Sie ist mit der Compiler-Option `-g` erstellt worden und enthält somit auch die Debugging-Information für die Graphenlaborfunktionen.

`libgraph.a` :
: Das ist die Bibliothek für ausgetestete Anwendungen. Sie vollzieht kein Checking der den Laborfunktionen übergebenen Parameter und erlaubt weder Tracing noch Debugging der Laborfunktionen.

2.3 Nutzung von Environment-Variablen

Das Verhalten des Graphenlabors wird durch einige Environment-Variable beeinflußt. Z.B. existiert eine Variable **G_NMAX**, die angibt, wieviele Knoten ein Graph zunächst enthalten kann. Falls diese Variable nicht definiert sind, wird mit Defaultwerten gearbeitet. Die Defaultwerte sind als Konstante gleichen Namens in `graphconfig.h` definiert.

Welche Variablen durch diesen Mechanismus behandelt werden und welche Bedeutung sie haben, kann man Anhang D, S. 90 entnehmen.

2.4 Meldungen

Die Funktionen des EMS-Graphenlabor erzeugen Warn- bzw. Fehlermeldungen auf dem Standardfehlergerät `stderr`, sofern sie eine unerwartete oder eine Fehlersituation erkennen. Dies ist z.B. der Fall, wenn eine Datei nicht geöffnet werden oder der Inhalt einer Datei nicht interpretiert werden kann. Sofern *Checking* (siehe Abschnitt 2.6, S. 43) eingeschaltet ist, erzeugen Graphenlaborfunktionen auch bei offensichtlich unsinnigen Parametern Fehlermeldungen.

Die Meldungen werden in drei Klassen eingeteilt:

Warnungen werden bei unerwarteten Situationen ausgegeben, die jedoch nicht notwendigerweise einen Fehler darstellen.

Fehlermeldungen werden in Fehlersituationen ausgegeben, in denen das System evtl. noch sinnvoll die Abarbeitung fortsetzen kann. Diese Fehlersituationen werden in der statischen Variable **G_msg::errorCount** gezählt und das Programm solange fortgesetzt, bis eine maximale Fehlerzahl (in der statischen Variable **G_msg::maxErrorCount**, initialisiert mit der Environment-Variable **G_MaxError**) erreicht ist. Dann wird das Programm (mit `exit(1)`) abgebrochen.

fatale Fehlermeldungen treten auf, wenn die Fehlersituation so schwerwiegend ist, daß die Fortsetzung des Programms wahrscheinlich sinnlos ist. In diesem Fall wird das Programm unverzüglich (mit `exit(2)`) abgebrochen.

Jede Fehlermeldung besteht aus folgenden Teilen:

- die Art (Warnung, Fehler oder fataler Fehler) des Fehlers
- die Funktion, die die Fehlersituation erkannt hat
- die Beschreibung der Fehlersituation (dazu können evtl. aktuelle Parameter der die Fehlersituation feststellenden Funktion zählen)
- das Verhalten des Programms in dieser Fehlersituation
- eine Empfehlung, wie diese Fehlersituation vermieden werden kann

Eine Meldung wird durch eine *Modulbezeichnung*[17] und eine Fehlernummer identifiziert. Dadurch wird es ermöglicht, dieselben Nummern in unterschiedlichen Moduln für unterschiedliche Fehler zu verwenden.

Versucht man zum Beispiel, einen Knoten mit einer bereits vergebenen Identifikationsnummer zu erzeugen, gibt das Labor folgende Fehlermeldung aus:

```
****** ERROR grf002 in G_graph::createVertex(1)
****** Condition found  : vertex 1 already exists
****** Action taken     : no vertex created, G_VertexNull returned
****** Reaction proposed: use unique numbers
```

Die erste Zeile gibt die Fehlerart (hier: `ERROR`), die den Fehler feststellende Funktion (`G_graph::createVertex`) sowie Modulbezeichnung (`grf`) und Fehlernummer (`002`) an. Die zweite Zeile beschreibt das Verhalten des Systems, nämlich den Abbruch der Funktion mit dem Rückgabewert **G_VertexNull**. Die dritte Zeile gibt eine Empfehlung, wie dieser Fehler zu vermeiden ist: man sollte eindeutige Identifikationsnummern verwenden.

Die Texte aller Fehlermeldungen sind nicht im Quelltext des Graphenlabors fest vorgegeben. Die für alle Meldungen gleichen Texte (also hier „`****** ERROR`“, „`****** Condition found : `“, „`****** Action taken : `“ und „`****** Reaction proposed: `“) sowie das Format (gemäß `printf`), in dem Fehlermeldungsdatei, Fehlernummer und Funktion ausgegeben werden („ `%s%3d in %s`“), werden der Datei `msgems` in einem der in der Environmentvariable **G_MsgLibs** angegeben Verzeichnisse entnommen. Mit der Methode **G_msg::addDirectory** können weitere Verzeichnisse angegeben werden. Die für den Fehler spezifischen Texte (diese enthalten je nach Fehler ebenfalls Formatangaben gemäß `printf` für die Ausgabe von Parametern etc.) befinden sich in einer der Dateien `msg`*module* (im Beispiel also `msggrf`). Jedes Modul hat also eine eigene Fehlermeldungsdatei. Dort werden die Texte anhand der Fehlernummer (im Beispiel 2) gesucht. Zum Format dieser Dateien siehe Anhang C.3, S. 87.

[17] Unter einem Modul ist hier (nur) eine logisch zusammenhängende Menge von Funktionen und Methoden zu verstehen.

Sollen die Meldungen in einer anderen Sprache als Englisch erfolgen, sind die Texte in diesen Dateien entsprechend zu ändern oder ein Verzeichnis mit entsprechenden geänderten Dateien in der Environment-Variable **G_MsgLibs** anzugeben.

Das Graphenlabor stellt dem Benutzer Funktionen zur Verfügung, mit denen er in seinem Programm Meldungen in analoger Weise erzeugen kann. Zu Details siehe Abschnitt 8.2, S. 75.

2.5 Tracing

Das Graphenlabor enthält einige Funktionen und Makros, die eine Laufzeitverfolgung (im weiteren nur *Tracing* genannt) einer EMS-Anwendung während ihrer Entwicklung ermöglichen. Einige Funktionen geben dann zur Laufzeit in eine Datei aus, in welcher Reihenfolge sie mit welchen Parametern in welcher Schachtelungstiefe aufgerufen werden und welchen Wert sie zurückgeben. Dabei ist zu beachten, daß jeweils für alle Funktionen, die in derselben Quelltextdatei *xxx*`.c` definiert sind, nur zusammen Tracing ein- bzw. ausgeschaltet werden kann.

2.5.1 Wie kann Tracing eingeschaltet werden ?

Zunachst muß die Anwendung, für die Tracing genutzt werden soll, mit der Bibliothek `libgraphUdebug.a` gebunden werden (siehe Abschnitt 2.2.2, S. 39). Dann kann mit der Methode **G_trace::set** für einzelne Quelltextdateien Tracing ein bzw. ausgeschaltet werden. Die Methode erwartet einen Dateinamen (Der Name darf auch die üblichen Jokerzeichen „`*`“ für beliebige Zeichenfolgen und „`?`“ für ein beliebiges Zeichen enthalten) und einen logischen `int`-Wert. Ist der `int`-Wert gleich `true`, wird Tracing für den oder die angegebenen Quelltextdateien ein-, andernfalls ausgeschaltet. Mit einer gleichnamigen Methode mit nur einem logischen `int`-Parameter kann Tracing auch ganz aus- bzw. eingeschaltet werden.

Die Tracing-Ausgaben können mit der Methode **G_trace::setFile** auch in eine Datei umgeleitet werden.

2.5.2 Wie können Funktionen Tracing-fähig programmiert werden ?

Das Graphenlabor stellt (über `graph.h`) die Makros **G_TrcDeclaration**, **G_trcEnter**, **G_trcLeave** und **G_trc** zur Verfügung. Das Makro **G_TrcDeclaration** definiert lokal in der `.c`-Datei ein Steuerobjekt, in dem zur Laufzeit festgehalten wird, ob Tracing für diese Quelltext-Datei ein- oder ausgeschaltet ist.

Die übrigen Makros werden mit einem Stück C-Code verwendet: `G_trcEnter(`*codeFragment*`)`, `G_trcLeave(`*codeFragment*`)` bzw. `G_trcEnter(`*codeFragment*`)`. Die Makros testen dann zur Laufzeit, ob momentan Tracing für die aktuelle Funktion (bzw. die Quelltextdatei, in der diese definiert wird) eingeschaltet ist. Ist dies der Fall, werden gemäß der Schachtelungstiefe von Funktionsaufrufen viele Einrückungszeichen in die Tracing-Datei ausgegeben und dann werden die in *codeFragment* angegeben C-Anweisungen ausgeführt (Dies werden in der Regel Ausgabeanweisungen in den Tracing-Stream sein, der über die

Zeigervariable **G_trace::out** vom Typ `ostream *` erreicht wird.) Zusätzlich verwalten `G_trcEnter` und `G_trcLeave` noch einen internen Zähler für die Schachtelungstiefe.

Dazu ein Beispiel:

Gegeben sei folgende Funktion `exampleFunc`:

```
G_TrcDeclaration

...

int exampleFunc ( int x, char *txt)
{
    int result;
    FILE *fp;

    G_trcEnter( *G_Trace.out << "exampleFunc("
                                    << x << ", " << txt
                                    << ')'; )


    ...
    name="xyz.dat";
    fp=fopen(name, flags);
    if (fp==NULL)
    {
        G_trc( *G_Trace.out << "cannot open file \""
                                   << name << '"'; )
    }
    ...
    result = ... ;
    G_trcLeave( *G_Trace.out << result; )
}
```

Wenn Tracing für diese Funktion eingeschaltet ist und sie in der Schachtelungstiefe 3 mit den Parametern `x=4, txt="abcdef"` aufgerufen wird, die Datei `xyz.dat` nicht öffnen kann und den Wert `result=12` berechnet, werden in die Tracing-Datei folgende Ausgaben geschrieben:

```
| | | exampleFunc(4, abcdef))
| | | | cannot open file "xyz.dat"
| | | ->12<-
```

Um bei voll ausgetesteten und für fehlerfrei befundenen Anwendungen volle Ausführungsgeschwindigkeit zu erreichen, ist es nicht notwendig, alle Tracing-Makros aus den Quelltexten zu entfernen. Es genügt, alle Quelltexte mit der zusätzlichen Definition `NO_TRC` zu übersetzen (siehe Abschnitt 2.2.1, S. 38). Dann löscht der Präprozessor alle Tracing-Makros, so daß zur Laufzeit in Bezug auf Tracing nichts mehr geschieht.

2.6 Checking

Um den Benutzer des Graphenlabor die Entwicklung seiner Anwendung zu erleichtern, prüft nahezu jede Laborfunktion ihre Aufrufparameter hinsichtlich ihrer Plausibilität und erzeugt ggf. Fehlermeldungen. Damit Checking durchgeführt werden kann, muß die Anwendung mit der Bibliotheksversion `liblabdebug.a` gebunden werden. Checking ist dann eingeschaltet, kann aber zur Laufzeit über die globale, logische Variable **G_Chk** vom Typ `int` mit dem Wert `false` aus- und mit dem Wert `true` eingeschaltet werden.

In eigenen Modulen kann derselbe Checking-Mechanismus verwendet werden. Dazu müssen die Prüfanweisungen mit dem Makro **G_chk** eingeklammert werden:

```
/* "normale" Anweisungen
...
*/

G_chk(
    /* checking Anweisungen */
        ...
)

/* weitere normale Anweisungen */
...
*/
```

Das Makro **G_chk** ist in `graph.h` definiert und sorgt dafür, daß die geklammerten Anweisungen nur dann ausgeführt werden, wenn die Variable **G_Chk** mit einem Wert $\neq 0$ belegt ist. Die geklammerten Anweisungen können mittels bedingter Compilierung gelöscht werden, falls der Quelltext mit der Compileroption `-DNO_CHK` übersetzt wird, so daß dann in Bezug auf Checking zur Laufzeit nichts mehr geschieht.

3 Die Klasse G_graph

Die Klasse **G_graph** realisiert die eigentliche Datenstruktur Graph. Instanzen dieser Klasse sind Graphen. Intern wird alle Knoten- und Kanteninformation in Tabellen gespeichert. Die Längen dieser Tabellen geben an, wieviele Knoten bzw. Kanten der Graph gerade enthalten kann. Reichen die Tabellen durch Erzeugen eines Knotens bzw. einer Kante nicht mehr aus, werden sie verdoppelt.

Zur Beschreibung dieser Klasse werden zunächst die benötigten Typen vorgestellt. Es folgen, nach Gruppen[18] geordnet, Beschreibungen der zur Verfügung gestellten Methoden.

3.1 Typen und Nullwerte

Die Klasse **G_graph** arbeitet mit folgenden Datentypen:

G_vertex :
: Variable dieses Typs repräsentieren Knoten. Der Nullwert ist **G_VertexNull** und wird von einigen Methoden als Rückgabewert im Fehlerfall verwendet. Die Identifikationsnummer eines Knotens erhält man durch Anwendung der Methode **G_graph::getVNo** auf diesen Knoten. Umgekehrt erhält man den Knoten mit einer bestimmten Identifikationsnummer mit der Methode **G_graph::getV**.

G_edge :
: Variable dieses Typs repräsentieren orientierte Kanten. Der Nullwert ist **G_EdgeNull** und wird von einigen Methoden als Rückgabewert im Fehlerfall verwendet. Die Identifikationsnummer einer Kante mit Orientierungsvorzeichen erhält man durch Anwendung der Methode **G_graph::getENo** auf diese Kante. Umgekehrt erhält man die Kante mit einer bestimmten Identifikationsnummer mit der Methode **G_graph::getE**.

Ferner nimmt **G_graph** Bezug auf folgende Klassen, die in eigenen Abschnitten beschrieben werden:

G_typeSystem, G_type :
: Graphen können Typsysteme zugeordnet werden, in denen alle verwendeten Typen beschrieben sind.

G_attrSchema, G_attribute :
: Knoten oder Kanten eines attributierten Graphen können ein Attribut tragen, das dem Schema des Typs genügen muß.

G_tempAttribute :
: Knoten oder Kanten eines temporär attributierten Graphen können ein temporäres Attribut tragen.

Logische Werte werden, wie in der Sprache C üblich, als `int`-Werte repräsentiert. Ein Wert $=0$ repräsentiert logisch `false`, ein Wert $\neq 0$ repräsentiert logisch `true`.

[18] Die Einteilung nach Gruppen entspricht der Aufteilung der Methoden auf die sie definierenden Quelltexte. Da der Name des zugeordneten Quelltextes für das Ein-/Ausschalten des Tracing wichtig ist, findet sich dieser in der Überschrift der entsprechenden Gruppe wieder.

3.2 Graphen anlegen und löschen: grfall

Graphen werden mit dem Konstruktor **G_graph** angelegt und dem Destruktor **~G_graph** gelöscht.

G_graph::G_graph

Prototyp

```
G_graph(G_typeSystem *pTypeSys=NULL,
        unsigned vMax=G_getEnvVar("G_NMAX", G_NMAX),
        unsigned eMax=G_getEnvVar("G_MMAX", G_MMAX));
```

Beschreibung

`G_graph(pTypeSys, vMax, eMax)` initialisiert eine Graphinstanz. Der Graph ist leer, d.h. er enthält keine Knoten und keine Kanten. Als Typsystem wird das mit `pTypeSys` angegebene verwendet. Wird hier ein NULL-Zeiger übergeben, ist der Graph untypisiert. Die Parameter `vMax` und `eMax` geben die gewünschten (Start-)Größen der internen Tabellen an. Falls sie weggelassen werden, werden die Werte den Environmentvariablen **G_NMAX** und **G_MMAX** entnommen.

Fehlerbehandlung

Falls der Speicher nicht mehr zur Aufnahme der Tabellen ausreicht, werden Warnmeldungen ausgegeben und die Werte `vMax` und `eMax` reduziert.

G_graph::~G_graph

Prototyp

```
~G_graph();
```

Beschreibung

`~G_graph()` löscht den Graphen. Bei attributierten Graphen werden die Destruktoren aller Attribute und die Destruktoren aller temporären Attribute aller Schichten aufgerufen. Anschließend wird der für interne Tabellen verwendete Speicherplatz wieder freigegeben.

3.3 Graphenstruktur manipulieren: grfman

Mit folgenden Funktionen können Knoten oder Kanten hinzugefügt und wieder gelöscht werden. Außerdem können Start- und Endknoten von Kanten geändert werden. Zu Identifikationsnummern können die zugehörigen Knoten oder Kanten gefunden werden und umgekehrt.

G_graph::createVertex

Prototyp

```
G_vertex createVertex();
```

Beschreibung

`createVertex()` erzeugt einen neuen Knoten im Graphen und gibt diesen zurück. Der Knoten ist isoliert, das heißt mit keiner Kante inzident, hat den Typ **G_TypeNull**, trägt kein Attribut und keine temporäreren Attribute. In die Folge aller Knoten des Graphen wird der neue Knoten als letzter eingetragen. Das Labor gibt dem Knoten willkürlich eine freie Identifikationsnummer. Reichen die internen Tabellen nicht für einen weiteren Knoten aus, werden sie automatisch vergrößert.

Fehlerbehandlung

Wenn die Vergrößerung der internen Tabellen wegen Speichermangels nicht möglich ist, wird ein Fehler gemeldet, kein Knoten erzeugt und **G_VertexNull** zurückgegeben.

G_graph::createVertex

Prototyp

```
G_vertex createVertex(unsigned vNo);
```

Beschreibung

`createVertex(vNo)` erzeugt einen Knoten mit der vorgegebenen Identifikationsnummer `vNo` und verhält sich sonst wie die parameterfreie Variante.

Fehlerbehandlung

Falls die Identifikationsnummer bereits vergeben ist oder nicht im gültigen Bereich (d.h. zwischen eins und der momentan maximalen Knotenzahl) liegt, wird ein Fehler gemeldet, kein Knoten erzeugt und **G_VertexNull** zurückgegeben.

G_graph::createEdge

Prototyp

```
G_edge createEdge(G_vertex alpha, G_vertex omega);
```

Beschreibung

`createEdge(alpha, omega)` erzeugt eine gerichtete Kante von Knoten `alpha` nach Knoten `omega` und gibt diese zurück. Die Kante hat den Typ **G_TypeNull**, trägt kein Attribut und keine temporären Attribute. In die Folge aller Kanten des Graphen wird die neue Kante als letzte eingetragen. Auch in die Folgen Λ(`alpha`) und Λ(`omega`) wird die neue Kante als letzte eingetragen.[19] Das Labor gibt der Kante willkürlich eine freie Identifikationsnummer. Reichen die internen Tabellen nicht für eine weitere Kante aus, werden sie automatisch vergrößert.

Fehlerbehandlung

Wenn die Vergrößerung der internen Tabellen wegen Speichermangels nicht möglich ist, wird ein Fehler gemeldet, keine Kante erzeugt und **G_EdgeNull** zurückgegeben.

[19] Handelt es sich um eine Schlinge, wird die Kante erst als out-Kante und dann als in-Kante an die Folge angehängt.

G_graph::createEdge

Prototyp

```
G_edge createEdge(unsigned eNo, G_vertex alpha, G_vertex omega);
```

Beschreibung

`createEdge(eNo, alpha, omega)` erzeugt einen Kante mit der vorgegebenen Identifikationsnummer `eNo` und verhält sich sonst wie die Variante mit zwei Parametern.

Fehlerbehandlung

Falls die Identifikationsnummer bereits vergeben ist oder nicht im gültigen Bereich (d.h. zwischen eins und der momentan maximalen Kantenzahl) liegt, wird ein Fehler gemeldet, keine Kante erzeugt und **G_EdgeNull** zurückgegeben.

G_graph::deleteVertex

Prototyp

```
void deleteVertex(G_vertex v);
```

Beschreibung

`deleteVertex(v)` löscht den Knoten `v`. Alle mit `v` inzidenten Kanten werden durch Aufrufe von **deleteEdge** ebenfalls gelöscht. Falls vorhanden werden die Destruktoren des Attributs und der temporären Attribute aller Schichten aufgerufen.

Fehlerbehandlung

Es wird eine Fehlermeldung erzeugt, falls `v` kein gültiger Knoten ist.

G_graph::deleteEdge

Prototyp

```
void deleteEdge(G_edge e);
```

Beschreibung

`deleteEdge(e)` löscht den Knoten `e`. Falls vorhanden werden die Destruktoren des Attributs und der temporären Attribute aller Schichten aufgerufen.

Fehlerbehandlung

Es wird eine Fehlermeldung erzeugt, falls `e` keine gültige Kante des Graphen ist.

G_graph::changeAlpha

Prototyp

```
void changeAlpha(G_edge e, G_vertex v);
```

Beschreibung

`changeAlpha(e, v)` legt für die (bereits vorhandene) Kante `e` den neuen Startknoten `v` fest. Dazu wird `e` aus $\Lambda(\alpha(\texttt{e}))$ entfernt und als out-Kante an $\Lambda(\texttt{v})$ angehängt. Stimmt `v` mit dem alten Startknoten von `e` überein, wird die Kante in $\Lambda(\texttt{v})$ an das Ende verschoben.

Fehlerbehandlung

Existiert die Kante `e` oder der Knoten `v` nicht, wird ein Fehler gemeldet.

G_graph::changeOmega

Prototyp

```
void changeOmega(G_edge e, G_vertex v);
```

Beschreibung

`changeOmega(e, v)` legt für die Kante `e` den neuen Endknoten `v` fest. Die Methode arbeitet völlig analog zu **G_changeAlpha**.

G_graph::changeThis

Prototyp

```
void changeThis(G_edge e, G_vertex v);
```

Beschreibung

`changeThis(e, v)` legt für die orientierte Kante `e` den neuen *this*-knoten `v` fest.[20] Die Methode arbeitet völlig analog zu **G_changeAlpha**.

G_graph::changeThat

Prototyp

```
void changeThat(G_edge e, G_vertex v);
```

Beschreibung

`changeThat(e, v)` legt für die orientierte Kante `e` den neuen *that*-knoten `v` fest. Die Methode arbeitet völlig analog zu **G_changeAlpha**.

G_graph::getV

Prototyp

```
G_vertex getV(unsigned vNo);
```

Beschreibung

`getV(vNo)` liefert den Knoten mit der Identifikationsnummer `vNo`.

[20] Ist also `e` normal orientiert, wird $\alpha(\texttt{e})$, ansonsten $\omega(\texttt{e})$ geändert.

G_graph::getE

Prototyp

```
G_edge getE(int eNo);
```

Beschreibung

`getE(eNo)` liefert die Kante mit der Identifikationsnummer `eNo`. Ist `eNo` negativ, wird die Kante als in-Kante geliefert, andernfalls als out-Kante.

G_graph::getVNo

Prototyp

```
unsigned getVNo(G_vertex v);
```

Beschreibung

`getVNo(v)` liefert die Identifikationsnummer des Knoten `v`.

G_graph::getENo

Prototyp

```
int getENo(G_edge e);
```

Beschreibung

`getENo(e)` liefert die Identifikationsnummer der Kante `e`. Falls `e` als in-Kante orientiert ist, liefert die Methode den Wert mit einem negativen Vorzeichen.

3.4 Graphen typisieren und attributieren: grfatr

Die Knoten und Kanten eines Graphen können typisiert und attributiert werden.

G_graph::setVType

Prototyp

```
int setVType(G_vertex v, const G_type &type,
             G_attribute *pAttr=NULL);
```

Beschreibung

`setVType(v, type, pAttr)` setzt am Knoten `v` den Typ `type` und das Attribut `pAttr`. Das Attribut muß zu dem Attributschema passen, das dem Typ im Typsystem zugeordnet ist. Gehört zu dem Typ kein Attributschema, darf der Methode kein Attribut übergeben werden. Ein evtl. vorhandenes altes Attribut am Knoten `v` wird durch seinen Destruktor gelöscht. Im Erfolgsfall wird `true` zurückgegeben.

Bemerkung

Das dem Labor übergebene Attribut muß auf dem dynamischen Speicher alloziert worden sein, da das Labor beim Löschen des Knotens den Destruktor aufrufen wird.

Fehlerbehandlung

Falls der Knoten `v` nicht gültig ist, der Graph untypisiert ist, das dem Graphen zugeordnete Typsystem den Typ `type` nicht enthält oder das Attribut nicht zum Typ paßt, wird ein Fehler gemeldet und `false` zurückgegeben. Der alte Typ und das alte Attribut des Knotens bleiben erhalten.

G_graph::setEType

Prototyp

```
int setEType(G_edge e, const G_type &type,
             G_attribute *pAttr=NULL);
```

Beschreibung

`setEType(e, type, pAttr)` arbeitet völlig analog zu **setVType** für Kanten.

G_graph::getVType

Prototyp

```
const G_type & getVType(G_vertex v);
```

Beschreibung

`getVType(v)` liefert den Typ des Knoten `v`.

Fehlerbehandlung

Falls `v` kein gültiger Knoten ist, wird ein Fehler gemeldet und **G_TypeNull** geliefert.

G_graph::getEType

Prototyp

```
const G_type & getEType(G_edge e);
```

Beschreibung

`getEType(e)` liefert den Typ der Kante `e`.

Fehlerbehandlung

Falls `e` kein gültige Kante ist, wird ein Fehler gemeldet und **G_TypeNull** geliefert.

G_graph::isAV

Prototyp

```
int isAV(G_vertex v, const G_type &type);
```

Beschreibung

`isAV(v, type)` liefert `true`, falls der Typ des Knoten `v` identisch mit dem Typ `type` oder ein Untertyp davon ist.

Fehlerbehandlung

Falls der Knoten `v` nicht gültig ist, der Graph untypisiert ist oder das dem Graphen zugeordnete Typsystem den Typ `type` nicht enthält, wird ein Fehler gemeldet und `false` zurückgegeben.

G_graph::isAE

Prototyp

```
int isAE(G_edge e, const G_type &type);
```

Beschreibung

`isAE(e, type)` arbeitet völlig analog zu **isAV** für Kanten.

G_graph::getPVAttr

Prototyp

```
G_attribute * getPVAttr(G_vertex v);
```

liefert einen Zeiger auf das Attribut des Knoten `v`. Trägt dieser Knoten kein Attribut, liefert die Methode `NULL`.

Fehlerbehandlung

Falls `v` kein gültiger Knoten ist, wird ein Fehler gemeldet und `NULL` geliefert.

G_graph::getPEAttr

Prototyp

```
G_attribute * getPEAttr(G_edge e);
```

liefert einen Zeiger auf das Attribut der Kante `e`. Trägt diese Kante kein Attribut, liefert die Methode `NULL`.

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und `NULL` geliefert.

3.5 Graphen temporär attributieren: grftmp

Graphen können mit folgenden Funktionen temporär attributiert werden.

G_graph::createVTemp

Prototyp

```
int createVTemp();
```

Beschreibung

`createVTemp()` erzeugt eine neue, leere Schicht temporärer Attribute für Knoten. Evtl. vorhandene Schichten werden verdeckt. Alle Knoten des Graphen tragen anschließend kein temporäres Attribut in der aktuellen Schicht. Die Methode liefert die Anzahl der Attributierungsschichten für Knoten.

Fehlerbehandlung

Falls keine Attributierungsschicht angelegt werden kann, erzeugt die Methode eine Fehlermeldung und liefert den Wert 0.

G_graph::setPVTemp

Prototyp

```
int setPVTemp(G_vertex v, G_tempAttribute *pTemp);
```

Beschreibung

`setPVTemp(v, pTemp)` setzt für Knoten `v` das temporäre Attribut `pTemp` in der aktuellen Schicht und liefert `true`. War der Knoten bereits in der aktuellen Schicht markiert, wird das alte Attribut per Destruktor gelöscht. Wird der Methode anstelle eines Attributs der `NULL`-Zeiger übergeben, wird nur das alte Attribut gelöscht. Der Knoten ist anschließend in der aktuellen Schicht nicht attributiert.

Bemerkung

Das dem Labor übergebene Attribut muß auf dem dynamischen Speicher alloziert worden sein, da das Labor beim Löschen des Knotens den Destruktor aufrufen wird.

Fehlerbehandlung

Falls keine temporäre Attributierungsschicht mit **createVTemp** erzeugt worden ist, wird ein Fehler gemeldet, das übergebene Attribute gelöscht und `false` zurückgegeben.

G_graph::deleteVTemp

Prototyp

```
void deleteVTemp();
```

Beschreibung

`deleteVTemp()` löscht die aktuelle Schicht temporärer Attribute und alle in ihr befindlichen Attribute. Die darunter liegende Schicht wird wieder sichtbar.

Fehlerbehandlung

Falls keine Attributierungsschicht mit **createVTemp** erzeugt worden ist, wird ein Fehler gemeldet.

G_graph::getPVTempLevel

Prototyp

```
int getPVTempLevel();
```

Beschreibung

`getPVTempLevel()` liefert die Anzahl der Attributierungsschichten für Knoten. Falls keine Schicht vorhanden ist, liefert die Methode den Wert 0.

Die Methoden **createETemp**, **setPETemp**, **deleteETemp** und **getETempLevel** arbeiten völlig analog für temporäre Kantenattributierung.

3.6 Graphen ausgeben: grfprt

Die folgenden Funktionen dienen der übersichtlichen Ausgabe von Knoten, Kanten oder ganzen Graphen.

G_graph::printVertex

Prototyp

```
ostream & printVertex(ostream &oS, G_vertex v);
```

Beschreibung

`printVertex(oS, v)` schreibt alle verfügbare Information zu Knoten `v` in den Ausgabestrom `oS` und liefert diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifiaktionsnummern der über out-Kanten erreichbaren Knoten sowie die der zugehörigen out-Kanten.
- die Identifiaktionsnummern der über in-Kanten erreichbaren Knoten sowie die der zugehörigen out-Kanten.
- Bei typisierten Graphen der Knotentyp (als Typnummer und Identifikationsstring), das zugehörige Attributschema (als Schemanummer und Identifikationsstring) sowie (falls vorhanden) das Attribut (durch Aufruf der `print`-Methode desselben)
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

Fehlerbehandlung

Falls der Knoten `v` nicht gültig ist, wird ein Fehler gemeldet und keine Ausgabe auf `oS` vorgenommen.

G_graph::printEdge

Prototyp

```
ostream & printEdge(ostream &oS, G_edge v);
```

Beschreibung

`printEdge(oS, e)` schreibt alle verfügbare Information zu Kante `e` in den Ausgabestrom `oS` und liefert diesen zurück. Im einzelnen werden ausgegeben:

- die Identifikationsnummer
- die Identifikationsnummen von Start- und Endknoten
- Bei typisierten Graphen der Kantentyp (als Typnummer und Identifikationsstring), das zugehörige Attributschema (als Schemanummer und Identifikationsstring) sowie (falls vorhanden) das Attribut (durch Aufruf der `print`-Methode desselben)
- bei temporär attributierten Graphen die temporären Attribute aller Schichten

Fehlerbehandlung

Falls die Kante `v` nicht gültig ist, wird ein Fehler gemeldet und keine Ausgabe auf `oS` vorgenommen.

G_graph::print

Prototyp

```
ostream & print(ostream &oS);
```

Beschreibung

`print(oS)` schreibt alle verfügbare Information zum Graphen in den Ausgabestrom `oS` und liefert diesen zurück. Im einzelnen werden ausgegeben:

- aktuelle Knoten und Kantenanzahl
- Größe der internen Tabellen (d.h. die maximale Anzahl von Knoten oder Kanten, die der Graph ohne Vergrößerung der Tabellen aufnehmen kann)
- Information für alle Knoten (in der Reihenfolge *Vseq* durch Aufruf von **printVertex**)
- Information für alle Kanten (in der Reihenfolge *Eseq* durch Aufruf von **printEdge**)

Bemerkung

Der Operator `<<` ist so überladen, daß mit ihm **G_graph::print** aufgerufen werden kann. Also `oS<<g` führt zum Aufruf `g.print(oS)`.

3.7 Graphen in Dateien ablegen und laden: grfsto

Mit den folgenden Methoden können Graphen in Dateien gespeichert und wieder geladen werden.

G_graph::store

Prototyp

```
int store(char *fileName);
```

Beschreibung

`store(fileName)` schreibt alle Information zum Graphen mit Ausnahme der temporären Attributierung in die Datei mit dem Namen `fileName`. Falls bereits eine Datei dieses Namens existiert, wird die alte Datei überschrieben. Es werden gespeichert:

- die Folge aller Knoten *Vseq* samt inzidenter Kanten $\Lambda(v), v \in V$
- die Folge aller Kanten
- die im Graphen verwendeten Typen
- die Typisierung und Attributierung aller Knoten und aller Kanten

Zur Speicherung der Attributierung werden die `store`-Methoden der verwendeten Attributklassen verwendet. Im Erfolgsfall liefert die Methode `true`.

Fehlerbehandlung

Falls die Datei nicht für Schreibzugriffe geöffnet werden kann, wird ein Fehler gemeldet und `false` zurückgegeben.

G_graph::restore

Prototyp

```
static G_graph * restore(char *fileName,
                         G_typeSystem *pTypeSystem = NULL);
```

Beschreibung

`restore(fileName)` liest einen mit **G_store** geschriebenen Graphen zum durch `pTypeSystem` bezeichneten Typsystem wieder ein und liefert einen Zeiger auf den gelesenen Graphen zurück. Das Typsystem muß alle im zu lesenden Graphen verwendeten Typen beinhalten. Falls der zu lesende Graph untypisiert ist, kann der Parameter `pTypeSystem` entfallen. Im Erfolgsfall liefert die Methode `true`.

Fehlerbehandlung

Falls die Datei nicht für Lesezugriffe geöffnet werden kann oder nicht dem erwarteten Format entspricht, wird ein Fehler gemeldet und `false` zurückgegeben.

Bemerkung

Die Methode ist `static` und sollte daher nicht mit einer **G_graph**-Instanz aufgerufen (etwa `g.restore(fileName, pTypeSystem)`), sondern mit dem Scope-Operator (also `G_graph::restore(fileName, pTypeSystem)`).

3.8 Graphen traversieren: grftra

Mit den folgenden Makros ist es leicht möglich, Graphen zu traversieren. Es kann auch typ-abhängig traversiert werden, siehe dazu den nächsten Abschnitt.

G_forAllVertices

Verwendung

```
G_forAllVertices(G_graph g, G_vertex v)
            { statements }
```

Beschreibung

`G_forAllVertices(g, v)` { *statements* } führt für jeden Knoten des Graphen `g` die Anweisungen *statements* aus. Der aktuelle Knoten steht dabei jeweils in der Variable `v`. Die Reihenfolge entspricht der Folge aller Knoten des Graphen $Vseq$.

Beispiel

Durch folgenden Code wird `process` für alle Knoten des Graphen `g` aufgerufen:

```
G_forAllVertices(g, x)
{
        process(g, x);
}
```

Bemerkung

Im Gegensatz zu Methoden muß bei Makros der Graph natürlich explizit angegeben werden.

Innerhalb der Schleife darf die Folge aller Knoten *Vseq* nicht durch Erzeugen, Löschen oder Umordnen von Knoten verändert werden. Die Schleife wird mit den Methoden **G_firstVertex** und **G_nextVertex** realisiert.

G_forAllEdges

Verwendung

`G_forAllEdges(G_graph g, G_edge e)` { *statements* }

Beschreibung

`G_forAllEdges(g, e)` { *statements* } führt für jede (normalisierte) Kante des Graphen `g` die Anweisungen *statements* aus. Die aktuelle Kante steht dabei jeweils in der Variable `e`. Die Reihenfolge entspricht der Folge aller Kanten des Graphen *Eseq*.

Bemerkung

Innerhalb der Schleife darf die Folge aller Kanten *Eseq* nicht durch Erzeugen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden **G_firstEdge** und **G_nextEdge** realisiert.

G_forAllIncidentEdges

Verwendung

`G_forAllIncidentEdges(G_graph g, G_vertex v, G_edge e)` { *statements* }

Beschreibung

`G_forAllIncidentEdges(g, v, e)` { *statements* } führt für jede mit Knoten `v` inzidente Kante des Graphen `g` die Anweisungen *statements* aus. Die aktuelle Kante steht dabei jeweils in der Variable `e`. Die Kanten sind so orientiert, daß `this(e)=v` gilt. Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht $\Lambda(v)$. Schlingen werden zweimal, nämlich für beide Orientierungen bearbeitet.

Bemerkung

In der Schleife darf die Folge $\Lambda(v)$ der zu `v` inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden **G_first** und **G_next** realisiert.

G_forAllInEdges

Verwendung

```
G_forAllInEdges(G_graph g, G_vertex v, G_edge e)
        { statements }
```

Beschreibung

`G_forAllInEdges(g, v, e)` { *statements* } führt für jede in-Kante zu Knoten `v` des Graphen `g` die Anweisungen *statements* aus. Die aktuelle Kante steht dabei jeweils in der Variable `e`. Die Kanten sind so orientiert, daß `this(e)=v` gilt. Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht $\Lambda^-(v)$.

Bemerkung

In der Schleife darf die Folge $\Lambda(v)$ der zu `v` inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden **G_firstIn** und **G_nextIn** realisiert.

G_forAllOutEdges

Verwendung

```
G_forAllOutEdges(G_graph g, G_vertex v, G_edge e)
        { statements }
```

Beschreibung

`G_forAllOutEdges(g, v, e)` { *statements* } führt für jede out-Kante zu Knoten `v` des Graphen `g` die Anweisungen *statements* aus. Die aktuelle Kante steht dabei jeweils in der Variable `e`. Die Kanten sind so orientiert, daß `this(e)=v` gilt. Die Reihenfolge, in der die Kanten bearbeitet werden, entspricht $\Lambda^+(v)$.

Bemerkung

In der Schleife darf die Folge $\Lambda(v)$ der zu `v` inzidenten Kanten nicht durch Hinzufügen, Löschen (auch indirekt durch Löschen eines Knotens) oder Umordnen von Kanten verändert werden. Die Schleife wird mit den Methoden **G_firstOut** und **G_nextOut** realisiert.

Zur Realisierung obiger Makros sind folgende Methoden definiert worden, die dem Benutzer auch direkt zur Verfügung stehen:

Folge	Startmethode	Iterationsmethode
Vseq	`G_vertex firstVertex()`	`G_vertex nextVertex(G_vertex curV)`
Eseq	`G_edge firstEdge()`	`G_edge nextEdge(G_edge curE)`
$\Lambda(v)$	`G_edge first(G_vertex v)`	`G_edge next(G_edge curE)`
$\Lambda^+(v)$	`G_edge firstIn(G_vertex v)`	`G_edge nextIn(G_edge curE)`
$\Lambda^-(v)$	`G_edge firstOut(G_vertex v)`	`G_edge nextOut(G_edge curE)`

Dabei liefert die Startmethode immer das erste Element der Folge (oder **G_VertexNull** bzw. **G_EdgeNull**, falls die Folge leer ist), die Iterationsmethode das auf das aktuelle Element (`curV` bzw. `curE`) folgende Element (oder **G_VertexNull** bzw. **G_EdgeNull**, falls das Ende der Folge erreicht ist). Bei ungültigen Parametern wird immer ein Fehler gemeldet und **G_VertexNull** bzw. **G_EdgeNull** zurückgegeben.

3.9 Graphen typabhängig traversieren: grftra2

Bei typisierten Graphen will man häufig in Abhängigkeit der Typen traversieren. Dabei interessiert man sich nicht für eine ganze Folge von Knoten oder Kanten, sondern nur für die Teilfolgen mit allen Elementen eines bestimmten Typs (und der Untertypen desselben).

Für diese Zwecke stellt das Labor Varianten der Makros und Funktionen des letzten Abschnitts bereit:

G_forAllVerticesWithType

Verwendung

```
G_forAllVerticesWithType(G_graph g, const G_type &t,
                         G_vertex v)
        { statements }
```

Beschreibung

`G_forAllVerticesWithType(g, t, v)` { *statements* } führt für jeden Knoten des Graphen `g`, dessen Typ `t` oder ein Untertyp von `t` ist, die Anweisungen *statements* aus. Der aktuelle Knoten steht dabei jeweils in der Variable `v`. Die Reihenfolge entspricht der Folge aller Knoten des Graphen *Vseq*.

Bemerkung

Innerhalb der Schleife darf *Vseq* nicht durch Erzeugen, Löschen oder Umordnen von Knoten verändert werden. Die Schleife wird mit den Methoden **G_firstVertex** und **G_nextVertex** (mit dem Typ als zusätzlichem Parameter) realisiert.

Völlig analog sind die Makros **G_forAllEdgesWithType**, **G_forAllIncidentEdgesWithType**, **G_forAllInEdgesWithType** und **G_forAllOutEdgesWithType** definiert.

Zur Realisierung obiger Makros sind folgende Methoden definiert worden, die ebenfalls dem Benutzer direkt zur Verfügung stehen:

Folge	Startmethode	Iterationsmethode
Vseq	`G_vertex firstVertex (const G_type &t)`	`G_vertex nextVertex (const G_type &t, G_vertex curV)`
Eseq	`G_edge firstEdge (const G_type &t)`	`G_edge nextEdge (const G_type &t, G_edge curE)`
$\Lambda(v)$	`G_edge first (const G_type &t, G_vertex v)`	`G_edge next (const G_type &t, G_edge curE)`
$\Lambda^+(v)$	`G_edge firstIn (const G_type &t, G_vertex v)`	`G_edge nextIn (const G_type &t, G_edge curE)`
$\Lambda^-(v)$	`G_edge firstOut (const G_type &t, G_vertex v)`	`G_edge nextOut (const G_type &t, G_edge curE)`

3.10 Knoten- oder Kantenanordnung erfragen und manipulieren: grford

Mit folgenden Funktionen kann die Reihenfolge der Folgen *Vseq*, *Eseq* oder $\Lambda(v), v \in V$ erfragt und geändert werden.

G_graph::isVertexBefore

Prototyp

```
int isVertexBefore(G_vertex v1, G_vertex v2);
```

Beschreibung

`isVertexBefore(v1, v2)` liefert `true`, falls Knoten `v1` in *Vseq* irgendwo vor Knoten `v2` steht.

Fehlerbehandlung

Falls `v1` oder `v2` keine gültigen Knoten sind, wird ein Fehler gemeldet.

G_graph::putVertexBefore

Prototyp

```
void putVertexBefore(G_vertex v1, G_vertex v2);
```

Beschreibung

`putVertexBefore(v1, v2)` verändert die Position des Knotens `v1` in *Vseq* so, daß nach dem Aufruf der Knoten `v1` unmittelbar vor Knoten `v2` steht.

Fehlerbehandlung

Falls `v1` oder `v2` keine gültigen Knoten oder identisch sind, wird ein Fehler gemeldet.

G_graph::putVertexAfter

Prototyp

```
void putVertexAfter(G_vertex v1, G_vertex v2);
```

Beschreibung

`putVertexAfter(v1, v2)` verändert die Position des Knotens `v1` in *Vseq* so, daß nach dem Aufruf der Knoten `v1` unmittelbar nach Knoten `v2` steht.

Fehlerbehandlung

Falls `v1` oder `v2` keine gültigen Knoten oder identisch sind, wird ein Fehler gemeldet.

G_graph::isEdgeBefore

Prototyp

```
int isEdgeBefore(G_edge e1, G_edge e2);
```

Beschreibung

`isEdgeBefore(e1, e2)` liefert `true`, falls Kante `e1` in *Eseq* irgendwo vor Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten sind, wird ein Fehler gemeldet.

G_graph::putEdgeBefore

Prototyp

```
void putEdgeBefore(G_edge e1, G_edge e2);
```

Beschreibung

`putEdgeBefore(e1, e2)` verändert die Position der Kante `e1` in *Eseq* so, daß nach dem Aufruf die Kante `e1` unmittelbar vor Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten oder identisch sind, wird ein Fehler gemeldet.

G_graph::putEdgeAfter

Prototyp

```
void putEdgeAfter(G_edge e1, G_edge e2);
```

Beschreibung

`putEdgeAfter(e1, e2)` verändert die Position der Kante `e1` in *Eseq* so, daß nach dem Aufruf die Kante `e1` unmittelbar hinter Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten sind identisch sind, wird ein Fehler gemeldet.

G_graph::isBefore

Prototyp

```
int isBefore(G_edge e1, G_edge e2);
```

Beschreibung

`isBefore(e1, e2)` liefert `true`, falls Kante `e1` in $\Lambda(\mathit{this}(e1))$ irgendwo vor Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten sind oder $\mathit{this}(e1) \neq \mathit{this}(e2)$ gilt, wird ein Fehler gemeldet.

G_graph::putBefore

Prototyp

```
void putBefore(G_edge e1, G_edge e2);
```

Beschreibung

`putBefore(e1, e2)` verändert die Position der Kante `e1` in $\Lambda(this(e1))$ so, daß nach dem Aufruf die Kante `e1` unmittelbar vor Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten sind oder $this(e1) \neq this(e2)$ gilt, wird ein Fehler gemeldet.

G_graph::putAfter

Prototyp

```
void putAfter(G_edge e1, G_edge e2);
```

Beschreibung

`putAfter(e1, e2)` verändert die Position der Kante `e1` in $\Lambda(this(e1))$ so, daß nach dem Aufruf die Kante `e1` unmittelbar hinter Kante `e2` steht.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültigen Kanten sind oder $this(e1) \neq this(e2)$ gilt, wird ein Fehler gemeldet.

3.11 Knoten, Kanten testen: grftst

Die folgenden Funktionen dienen der sauberen Datenabstraktion und ermöglichen das Vergleichen von vertex- bzw. edge-Werten.

G_graph::areEqualVertices

Prototyp

```
int areEqualVertices(G_vertex v1, G_vertex v2);
```

Beschreibung

`areEqualVertices(v1, v2)` liefert `true`, falls `v1` derselbe Knoten wie `v2` ist. Sonst liefert die Funktion `false`.

Fehlerbehandlung

Falls `v1` oder `v2` kein gültiger Knoten ist (**G_VertexNull** ist allerdings erlaubt.), wird ein Fehler gemeldet und `false` zurückgegeben.

G_graph::areEqualEdges

Prototyp

```
int areEqualEdges(G_edge e1, G_edge e2);
```

Beschreibung

`areEqualEdges(e1, e2)` liefert `true`, falls `e1` dieselbe Kante (evtl. in unterschiedlicher Orientierung) wie `e2` ist. Sonst liefert die Funktion `false`.

Fehlerbehandlung

Falls `e1` oder `e2` keine gültige Kanote ist (**G_EdgeNull** ist allerdings erlaubt.), wird ein Fehler gemeldet und `false` zurückgegeben.

G_graph::isVertexNull

Prototyp

```
int isVertexNull(G_vertex v);
```

Beschreibung

`isVertexNull(v)` liefert `true`, falls `v` **G_VertexNull** ist. Sonst liefert die Funktion `false`.

G_graph::isEdgeNull

Prototyp

```
int isEdgeNull(G_edge e);
```

Beschreibung

`isEdgeNull(e)` liefert `true`, falls `e` **G_EdgeNull** ist. Sonst liefert die Funktion `false`.

G_graph::isNormal

Prototyp

```
int isNormal(G_edge e);
```

Beschreibung

`isNormal(e)` liefert `true`, falls `e` im normalisiert ist. Sonst liefert die Funktion `false`.

G_graph::isVertex

Prototyp

```
int isVertex(G_vertex v);
```

Beschreibung

`isVertex(v)` liefert `true`, falls `v` ein gültiger Knoten ist.

G_graph::isEdge

Prototyp

```
int isEdge(G_edge e);
```

Beschreibung

`isEdge(e)` liefert `true`, falls `e` eine gültige Kante ist.

3.12 Kanteninformation erfragen: grfaux

Die folgenden Funktionen geben Strukturinformationen zu Kanten.

G_graph::alpha

Prototyp

```
G_vertex alpha(G_edge e);
```

Beschreibung

`alpha(e)` liefert den Anfangsknoten der Kante `e`.

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_VertexNull** zurückgegeben.

G_graph::omega

Prototyp

```
G_vertex omega(G_edge e);
```

Beschreibung

`omega(e)` liefert den Endknoten der Kante `e`.

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_VertexNull** zurückgegeben.

G_graph::thisV

Prototyp

```
G_vertex thisV(G_edge e);
```

Beschreibung

`thisV(e)` liefert den Knoten, von dem aus man die orientierte Kante `e` betrachtet (sh. Abschnitt 1.2.3, S. 6).

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_VertexNull** zurückgegeben.

G_graph::thatV

Prototyp

```
G_vertex thatV(G_edge e);
```

Beschreibung

`thatV(e)` liefert den Knoten am anderen Ende der orientierten Kante `e` (sh. Abschnitt 1.2.3, S. 6).

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_VertexNull** zurückgegeben.

G_graph::normal

Prototyp

```
G_edge normal(G_edge e);
```

Beschreibung

`normal(e)` liefert die Kante `e` des Graphen `g` normalisiert (sh. Abschnitt 1.2.3, S. 6).

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_EdgeNull** zurückgegeben.

G_graph::reverse

Prototyp

```
G_edge reverse(G_edge e);
```

Beschreibung

`reverse(e)` dreht die Orientierung der Kante `e` um (sh. Abschnitt 1.2.3, S. 6).

Fehlerbehandlung

Falls `e` keine gültige Kante ist, wird ein Fehler gemeldet und **G_EdgeNull** zurückgegeben.

3.13 Grad von Knoten erfragen: grfdeg

Mit folgenden Funktionen kann der Grad von Knoten abgefragt werden.

G_graph::degree

Prototyp

```
int degree(G_vertex v);
```

Beschreibung

`degree(v)` liefert die Anzahl der mit Knoten `v` inzidenten Kanten. Schlingen werden dabei doppelt gezählt.

Fehlerbehandlung

Falls `v` kein gültiger Knoten ist, wird ein Fehler gemeldet und `-1` zurückgegeben.

G_graph::inDegree

Prototyp

```
int inDegree(G_vertex v);
```

Beschreibung

`inDegree(v)` liefert die Anzahl der in-Kanten zu Knoten `v`.

Fehlerbehandlung
Falls `v` kein gültiger Knoten ist, wird ein Fehler gemeldet und `-1` zurückgegeben.

G_graph::outDegree

Prototyp
```
int outDegree(G_vertex v);
```

Beschreibung
`outDegree(v)` liefert die Anzahl der out-Kanten zu Knoten `v`.

Fehlerbehandlung
Falls `v` kein gültiger Knoten ist, wird ein Fehler gemeldet und `-1` zurückgegeben.

3.14 Globale Graphdaten erfragen: grfmsc

Zur Abfrage globaler Information zu Graphen dienen folgende Methoden.

G_graph::vertexCount

Prototyp
```
int vertexCount();
```

Beschreibung
`vertexCount()` liefert die Anzahl der Knoten des Graphen.

G_graph::edgeCount

Prototyp
```
int edgeCount();
```

Beschreibung
`edgeCount()` liefert die Anzahl der Kanten des Graphen.

G_graph::edgeFromTo

Prototyp
```
G_edge edgeFromTo(G_vertex v, G_vertex w);
```

Beschreibung
`edgeFromTo(alpha, omega)` liefert die erste, normalisierte Kante in der Folge der zu `alpha` inzidenten Kanten, die zum Knoten `omega` führt. Existiert keine derartige Kante, wird **G_EdgeNull** zuückgegeben.

Fehlerbehandlung
Falls `alpha` oder `omega` keine gültigen Knoten sind, wird ein Fehler gemeldet und **G_EdgeNull** zurückgegeben.

Bemerkung

Diese Funktion kann auch boolesch verwendet werden, d.h., der Rückgabewert ist `true`, falls eine Kante existiert, und `false` sonst.

G_graph::edgeBetween

Prototyp

```
G_edge edgeBetween(G_vertex v, G_vertex w);
```

Beschreibung

`edgeBetween(v, w)` liefert die erste Kante `e` zwischen Knoten `v` und `w` in der Folge der zu `v` inzidenten Kanten so orientiert, daß `this(e)=v` gilt. Existiert keine derartige Kante, wird **G_EdgeNull** zuückgegeben.

Fehlerbehandlung

Falls `v` oder `w` keine gültigen Knoten sind, wird ein Fehler gemeldet und **G_EdgeNull** zurückgegeben.

Bemerkung

Diese Funktion kann auch boolesch verwendet werden, d.h., der Rückgabewert ist `true`, falls eine Kante existiert, und `false` sonst.

4 Die Klasse G_typeSystem

Die Klasse **G_typeSystem** verwaltet die von Graphen verwendeten Typen. Jedem (typisierten) Graphen ist genau eine Instanz dieser Klasse zugeordnet. Das Typsystem beinhaltet eine Menge von Typen (repräsentiert durch **G_type**-Instanzen) und die Subtyp-Relation unter diesen Typen. Typsysteme werden unabhängig von Graphen in Dateien gespeichert. Jedes Typsystem enthält einen Nulltyp (**G_TypeNull**) mit der Bezeichnung `"NULL"`, dem kein Attributschema zugeordnet ist. Dieser Typ ist (automatisch) Obertyp aller anderen Typen.

G_typeSystem::G_typeSystem

Prototyp

```
G_typeSystem(unsigned typeMax=G_getEnvVar("G_MaxType",
             G_MaxType));
```

Beschreibung

`G_typeSystem(typeMax)` initialisiert ein Typsystem. Das Typsystem enthält nur den Typ **G_TypeNull** und kann maximal `typeMax` Typen (einschließlich **G_TypeNull**) aufnehmen.

Fehlerbehandlung

Falls der Speicherplatz nicht für das Typsystem der geforderten Größe ausreicht, wird eine Warnung ausgegeben und ein kleineres Typsystem initialisiert.

Die Aufnahme eines neuen Typs in ein Typsystem erfolgt automatisch durch den Konstruktor des Typs **G_type::G_type** (siehe S.69).

G_typeSystem::store

Prototyp

```
int store(const char *fileName);
```

Beschreibung

`store(fileName)` speichert das Typsystem in eine Datei mit dem Namen `fileName` und liefert im Erfolgsfall `true` zurück.

Fehlerbehandlung

Falls die Datei nicht für Schreibzugriffe geöffnet werden kann, wird ein Fehler gemeldet und `false` zurückgegeben.

G_typeSystem::G_typeSystem

Prototyp

```
G_typeSystem(const char *fileName);
```

Beschreibung

`G_typeSystem(fileName)` liest ein Typsystem aus der Datei mit dem Namen `fileName` ein. Die in dem Typsystem verwendeten Attributschemata sollten

bekannt sein. Dazu sollten zum gerade ablaufenden Programm die entsprechenden Attributklassen definierenden Dateien mit dem Linker hinzugefügt worden sein. Falls Attributschemata nicht bekannt sind, werden Warnmeldungen ausgegeben und die zugehörigen Typen dennoch erzeugt. Beim Einlesen von Graphen werden jedoch die Attribute der Knoten oder Kanten dieser Typen ignoriert.

Fehlerbehandlung

Falls beim Lesen der Datei ein Fehler auftritt, wird dieser gemeldet und das Typsystem leer initialisiert.

G_typeSystem::setIsA

Prototyp

```
void setIsA(const G_type &subType, const G_type &superType);
```

Beschreibung

`setIsA(subType, superType)` setzt die Subtyp-Relation zwischen Typ `subType` und `superType`.

Bemerkung

Die Methode bildet selbständig den transitiven Abschluß über der Relation. Nach `setIsA(typA, typB)` und `setIsA(typB, typC)` stehen auch `typA` und `typC` in der Subtyp-Relation.

G_typeSystem::isA

Prototyp

```
int isA(const G_type &subType, const G_type &superType);
```

Beschreibung

`isA(subType, superType)` testet, ob `subType` und `superType` in der Subtyp-Relation stehen. Falls `subType` ein Untertyp von `superType` ist, liefert die Methode `true`, andernfalls `false`.

G_typeSystem::getType

Prototyp

```
const G_type & getType(const char *id);
```

Beschreibung

`getType(id)` liefert das Typobjekt mit der Bezeichnung `id`. Existiert kein Typ mit dieser Bezeichnung in dem Typsystem, liefert die Methode **G_TypeNull**.

G_typeSystem::contains

Prototyp

```
int contains(const char *id);
```

Beschreibung

`contains(id)` liefert `true`, falls das Typsystem einen Typ mit der Bezeichnung `id` enthält, andernfalls `false`.

5 Die Klasse G_type

Instanzen der Klasse **G_type** repräsentieren Typen. Jeder Typ ist einem Typsystem zugeordnet und umfaßt eine innerhalb des Typsystems eindeutige Bezeichnung, die den Typ beim Schreiben und Lesen aus Dateien identifiziert und ein Attributschema (ggf. das Nullschema), das die Attributierung von Knoten oder Kanten dieses Typs beschreibt.

Ein Typobjekt darf nur dann gelöscht werden, wenn das zugehörige Typsystem auch gelöscht wird, da dieses ansonsten einen ungültigen Zeiger auf einen freigegebenen Speicherbereich enthält. Dies bedeutet, das Typobjekte in der Regel entweder statisch (also nicht innerhalb von Funktionen) oder dynamisch mittels `new`-Operator definiert werden sollten, jedoch nie als als lokale Objekte von Funktionen.

Mit Ausnahme des Nulltyps kann kein Typobjekt in mehreren Typsystemen zugleich existieren. Es können aber „gleiche“ Typen (also Typen mit derselben Bezeichnung und demselben Attributschema) in mehreren Typsystemen erzeugt werden.

G_type::G_type

Prototyp

```
G_type(G_typeSystem &tSys,
       const char *idString,
       const G_attrSchema &aSchema=NullSchema);
```

Beschreibung

`G_type(tSys, idString, aSchema)` erzeugt einen neuen Typ im Typsystem `tSys` mit der Bezeichnung `idString`, der das Attributschema `aSchema` verwendet. Dabei muß der Konstruktor des Typsystems bereits abgearbeitet worden sein.

Fehlerbehandlung

Falls bereits ein Typ mit der gewünschten Bezeichnung im Typsystem existiert, wird ein Fehler gemeldet.

6 Die Klasse G_attrSchema

Instanzen der Klasse **G_attrSchema** binden Attributierungsklassen an das Graphenlabor an. Jede **G_attrSchema**-Instanz umfaßt eine eindeutige Bezeichnung, die das Schema beim Schreiben in und Lesen aus Dateien identifiziert, und einen Zeiger auf die Lesemethode der das Schema realisierenden Attributklasse, mit der Attribute des Schemas gelesen werden können. Zusätzlich hat jedes Schema noch eine eindeutige Schemanummer des Typs **G_attrNo**. Der spezielle Wert **G_AttrNoUnknown** kennzeichnet dabei ein nicht verwendbares Attributschema nach dem Auftreten von Fehlern.[21]

Alle Schemaobjekte werden in einer statischen Tabelle verwaltet, so daß ein Attributschema auch nach seiner Bezeichnung gesucht werden kann.

Das Zusammenspiel von **G_attrSchema**-Instanzen und **G_attribute**-Klassen ist nicht trivial, so daß zur Definition von eigenen Attributierungsklassen das Hilfsprogramm `makeattr` verwendet werden sollte. Das von diesem Hilfsprogramm erzeugte Programmgerüst enthält nämlich schon die korrekte Definition und Initialisierung der **G_attrSchema**-Instanz, die zur Anbindung einer Attributklasse an das Graphenlabor notwendig ist.

G_attrSchema::G_attrSchema

Prototyp

```
G_attrSchema(const char *idString,
             G_attribute *(*restore(istream &iS,
             unsigned length)));
```

Beschreibung

`G_attrSchema(idString, restore)` erzeugt ein neues Attributierungsschema mit der Bezeichnung `idString`. Zum Lesen von Attributen des Schemas wird die Funktion verwendet, auf die `restore` zeigt. Dieser Funktion wird dann der Eingabestrom `iS` übergeben, in dem der String Länge `length` steht, der das zu lesende Attribut beschreibt.

Fehlerbehandlung

Existiert bereits ein Attributschema mit dem gewünschten Namen oder kann die Tabelle kein weiteres Schema aufnehmen, wird ein Fehler gemeldet und das (neue) Schema nicht in die statische Tabelle eingetragen und erhält die Nummer **G_AttrNoUnknown**.

G_attrSchema::getAttrId

Prototyp

```
const char * getAttrId();
```

Beschreibung

`getAttrId()` liefert die Bezeichnung des Attributschemas zurück.

[21] Beim Einlesen eines Typsystems aus einer Datei kann es z.B. vorkommen, daß die Attributklasse zu einer Schemabezeichnung nicht vorhanden ist.

G_attrSchema::getAttrNo

Prototyp

```
G_attrNo getAttrNo();
```

Beschreibung

`getAttrNo()` liefert die Nummer des Attributs zurück.

Bemerkung

Die Attributnummer **G_AttrNoUnknown** deutet auf einen Fehler hin.

G_attrSchema::getAttrNo

Prototyp

```
static G_attrNo getAttrNo(const char *idString);
```

Beschreibung

`getAttrNo(idString)` liefert die Nummer des Schemas mit der Bezeichnung `idString`. Falls kein Schema mit dieser Bezeichnung existiert, liefert die Methode **G_AttrNoUnknown**.

Bemerkung

Die Methode ist `static` und sollte daher nicht mit einer **G_attrSchema**-Instanz aufgerufen (etwa `SchemaXY.getAttrNo("idXYZ")`), sondern mit dem Scope-Operator (also `G_attrSchema::getAttrNo("idXYZ")`).

G_attrSchema::getSchema

Prototyp

```
static const G_attrSchema & getSchema(G_attrNo aNo);
```

Beschreibung

`getSchema(aNo)` liefert eine Referenz auf das Schemaobjekt mit der Nummer `aNo`.

Fehlerbehandlung

Falls `aNo` keine gültige Attributnummer ist, wird ein Fehler gemeldet und Referenz auf das Nullschema geliefert.

Bemerkung

Auch diese Methode ist `static`.

7 Abstrakte Klassen

Das Graphenlabor definiert als Basisklassen für Attributierungsklassen und Klassen zur temporären Attributierung die abstrakten Klassen **G_basicAttribute**, **G_attribute** und **G_tempAttribute**. **G_basicAttribute** umfaßt die für alle Attributierungsklassen (auch temporär) verwendeten virtuellen Methoden, **G_attribute** definiert zusätzliche Methoden für Attributierungsklassenzur Modellierung, **G_tempAttribute** realisiert zusätzlich die Verwaltung unterschiedlicher Schichten temporärer Attributierung.

7.1 G_basicAttribute

Sowohl temporäre als auch zur Modellierung dienende Attribute sollen mittels `print`-Methoden ausgeben und eventuell gelöscht werden können. Die dafür notwendigen virtuellen Methoden sind in **G_basicAttribute** deklariert und sollten in allen abgeleiteten Klassen entsprechend redefiniert werden.

G_basicAttribute::print

Prototyp

```
virtual ostream & print(ostream &oS);
```

Beschreibung

`print(oS)` schreibt den Inhat des Attributs in den Ausgabestrom `oS` und liefert diesen zurück. In welcher Form die Ausgabe genau erfolgen soll, ist natürlich für jede Klasse unterschiedlich, so daß diese Methode in jeder abgeleiteten Klassen redefiniert werden sollte. Sie wird vom Graphenlabor bei den Methoden **G_graph::printVertex** und **G_graph::printEdge** für das angehängte Attribut und alle temporären Attribute aufgerufen.

G_basicAttribute::~G_basicAttribute

Prototyp

```
virtual ~G_basicAttribute();
```

Beschreibung

Falls beim Konstruktor eines Objektes mittels `free`- oder `new`-Aufrufen von der Speicherverwaltung zusätzlicher Speicher angefordert wurde, sollte dieser durch den Destruktor wieder freigegeben werden. In diesen Fällen ist also in den abgeleiteten Klassen der Destruktor zu redefinieren.

7.2 G_attribute

Für Attributierungsklassen müssen noch über die in **G_basicAttribute** deklarierten Methoden hinaus Methoden zum Schreiben und Lesen von Attributen in Dateien und zur Anbindung an die **G_attrSchema**-Klasse bereitgestellt werden.[22]

[22] Wie bereits in Abschnitt 6, S. 70 erwähnt, erleichtert das Hilfsprogramm `makeattr` die Anbindung an die **G_attrSchema**-Klasse erheblich.

G_attribute::store

Prototyp

```
virtual void store(ostream &oS);
```

Beschreibung

`store(oS)` schreibt ein Attribut in den Ausgabestrom `oS` in einem für das Wiedereinlesen der Daten praktischen Format. Im Unterschied zur `print`-Methode kommt es hier jedoch nicht auf die Übersichtlichkeit der Ausgabe an.

G_attribute::restore

Prototyp

```
static G_attribute * restore(istream &iS, unsigned length);
```

Beschreibung

`restore(iS, length)` soll die nächsten `length` Zeichen aus dem Eingabestrom `iS` lesen, daraus ein Attribut erzeugen und einen Zeiger auf dieses zurückgeben.

Bemerkung

Diese Methode muß in jeder Attributierungsklasse definiert werden und ist beim Erzeugen der der Klasse zugeordneten **G_attrSchema**-Instanz anzugeben. Das Format der zu lesenden Daten ist durch die `store`-Methode der Klasse vorgegeben.

G_attribute::getAttrNo

Prototyp

```
virtual G_attrNo getAttrNo();
```

Beschreibung

`getAttrNo()` liefert die Nummer des Attributschemas, zu dem das Attribut gehört.

Bemerkung

Diese Methode wird bei Verwendung von `makeattr` automatisch deklariert, definiert und muß vom Benutzer nicht modifiziert werden.

7.3 G_tempAttribute

Die Klasse **G_tempAttribute** ist Basisklasse aller Klassen zur temporären Attributierung und umfaßt `private`-Methoden zur Verwaltung von Attibutierungsschichten. Die für den Benutzer sichtbare Schnittstelle ist mit der von **G_basicAttribute** identisch.

8 Weitere Klassen

Das Graphenlabor stellt noch einige weitere Klassen zur Verfügung. Die Klasse **G_trace** realisiert das *Tracing* (siehe auch 2.5, S. 41), die Klasse **G_msg** die Ausgabe von Meldungen.

8.1 G_trace

Die Klasse **G_trace** verwaltet Steuerinformation zur Laufzeitverfolgung. Für jeden Quelltext (also *name*`.c`-Datei), in dem zu verfolgende Funktionen oder Methoden stehen, sollte mit dem Makro **G_TrcDeclaration** ein Steuerobjekt (unmittelbar nach den `include`-Anweisungen) definiert werden. Dieses Objekt wird durch seinen Konstruktor in eine globale Tabelle mit dem Namen der Quelltextdatei eingetragen, ist ansonsten aber nur in dem Quelltext unter dem Bezeichner `G_TrcObj` sichtbar (also als `static` deklariert). Die durch die Makros **G_trc**, **G_trcEnter** und **G_trcLeave** geklammerten Anweisungen werden nur dann ausgeführt, wenn für den Quelltext Tracing eingeschaltet ist. Tracing kann für Quelltexte mit der Methode **G_trace::set** ein- bzw. ausgeschaltet werden. Tracing-Ausgaben erfolgen immer in den Ausgabestrom, auf den die Variable **G_trace::out** (Typ: `ostream *`) zeigt. Sie zeigt zunächst auf `cout`, die Ausgabe kann aber mit der Methode **G_trace::setFile** in eine Datei umgeleitet werden.

G_trace::G_trace

Prototyp

```
G_trace(char *name);
```

Beschreibung

`G_trace(name)` erzeugt ein Kontrollobjekt und trägt es unter dem Namen `name` in die globale Tabelle der Kontrollobjekte ein.

Bemerkung

Der Benutzer sollte diesen Konstruktor nicht direkt aufrufen, sondern das Makro **G_TrcDeclaration** verwenden.

G_trace::set

Prototyp

```
static int set(char *pattern, int flag);
```

Beschreibung

`set(pattern, flag)` schaltet für alle Quelltextdateien, deren Namen dem Suchmuster `pattern` genügen, Tracing gemäß `flag` aus (`flag=false`) oder ein (`flag=true`). Das Muster darf die Wildcards „`*`“ (null oder mehrere beliebige Zeichen) und „`?`“ (genau ein beliebiges Zeichen) enthalten. Die Methode liefert die Anzahl der durch das Muster getroffenen Dateien zurück.

Bemerkung

Einige Compiler liefern beim Makro `__NAME__` den Dateinamen mit kompletter Pfadangabe, andere nur den Dateinamen zurück.

G_trace::set

Prototyp

```
static void set(int flag);
```

Beschreibung

`set(flag)` schaltet Tracing global gemäß `flag` aus oder ein, ohne die Auswahl an Quelltextdateien zu verändern.

G_trace::setFile

Prototyp

```
static void setFile(const char *fileName);
```

Beschreibung

`setFile(fileName)` öffnet eine Datei unter dem Namen `fileName` für Schreibzugriffe und lenkt alle folgenden Tracingausgaben in diese um. Falls zuvor bereits Tracing umgeleitet war, wird die alte Datei geschlossen.

8.2 G_msg

Die Klasse **G_msg** dient der Steuerung der Ausgabe von Warn- und Fehlermeldungen. Es gibt keine Objekte der Klasse **G_msg**, sie dient nur dazu, einen eigenen Gültigkeitsbereich zu bilden und vermeidet dadurch globale Bezeichner. Alle definierten Methoden sind statisch und sollten mit dem Scope-Operator angesprochen werden. Fehlermeldungen werden in der Variable **G_msg::errorCount** gezählt. Die Variable **G_msg::maxErrorCount** gibt an, nach wievielen Fehlern eine Anwendung abgebrochen werden soll. Dabei bedeutet der Wert 0, daß die Anwendung nie abgebrochen werden soll. Die Variable wird mit der Environmentvariable **G_MaxError** initialisiert. Die auszugebenden Meldungen werden aus einer Datei gelesen, die sich einem der von Environmentvariable **G_MsgLibs** angegebenen Verzeichnisse befindet. Mit der Methode **G_addDirectory** können weiter Verzeichnisse angegeben werden.

G_msg::error

Prototyp

```
static void error(int no, char *modName, char *funcName, ...);
```

Beschreibung

`error(no, modName, funcName, ...)` gibt die Meldung mit der Nummer `no` zum Modul `modName` evtl. mit weiteren Parametern in die Standardfehlerausgabe (`stderr`) als Fehlermeldung aus. Dabei wird `funcName` als diejenige Funktion angegeben, die die Fehlersituation erkannt hat. `funcName` darf Umwandlugsspezifizierer enthalten, für die weitere Parameter übergeben werden müssen. Das Labor lädt dazu den Text dieser Meldung aus einer Datei mit dem Namen `msg<modName>` aus einem der in Variable **G_MsgLibs** angegebenen Verzeichnisse. Enthält dieser Text Umwandlungsspezifizierer gemäß der

der Standardfunktion `printf` (z.B. `%d`, `%s` etc.), müssen der Methode entsprechend weitere Parameter übergeben werden. Der Fehler wird gezählt und bei Erreichen der maximalen Fehlerzahl wird die Anwendung mit `exit(1)` abgebrochen.

Beispiel

Die Methode **G_graph::restore** soll Graphendateien lesen. Dazu wird ihr der Name einer Graphendatei übergeben. Falls diese Datei nicht existiert, soll eine Fehlermeldung erzeugt werden und der aufrufenden Funktion ein `NULL`-Zeiger geliefert werden. Zur Erzeugung der Meldung ruft die Methode **G_error** wie folgt auf:

```
G_msg::error(30, "grf", "G_graph::restore(%s)",
                 fileName, fileName);
```

Dabei enthält `fileName` den Namen der unauffindbaren Datei. Der zugehörige Meldungstext befindet sich in der Datei `msggrf`:

```
MSG030
C {unable to open file ‘%s’ for reading}
A {no graph created, NULL pointer returned}
R {check your filename}
```

Soll z.B. eine Datei mit dem Namen ¨`Gibts_nicht`¨ gelesen werden, die nicht existiert, wird folgende Nachricht ausgegeben:

```
****** ERROR grf030 in G_graph::restore(Gibts_nicht)
****** Condition found  : unable to open file ‘Gibts_nicht’ for
                         reading
****** Action taken     : no graph created, NULL pointer returned
****** Reaction proposed: check your filename
```

Zum Format der Meldungstextdateien siehe Anhang C.3.1, S. 87.

Fehlerbehandlung

Falls die Fehlermeldungsdatei nicht geöffnet werden kann, wird anstelle des Fehlermeldungstextes die Nachricht „`Can't open message file` *fileName*“ ausgegeben. Falls die Meldungsdatei zwar geöffnet werden kann, sie aber keine Meldung mit der geforderten Nummer enthält, wird die Nachricht „`Can't find message number` *number*“ ausgegeben.

G_msg::warning

Prototyp

```
static void warning(int no, char *modName, char *funcName, ...);
```

Beschreibung

G_error verhält sich wie **G_error**, nur wird die Meldung als Warnmeldung gekennzeichnet und nicht als Fehler gezählt.

G_msg::fatalError

Prototyp

```
static void fatalError(int no, char *modName, char *funcName, ...);
```

Beschreibung

G_fatalError verhält sich wie **G_error**, nur wird die Meldung als fataler Fehler gekennzeichnet und die Anwendung mit `exit(2)` abgebrochen.

G_msg::setHeaders

Prototyp

```
static void setHeaders(char *fileName);
```

Beschreibung

`setHeaders(fileName)` liest die Einleitungstexte der Meldungen aus der Datei `fileName` ein. Zum Format dieser Datei siehe Anhang C.3, S. 87.

G_msg::addDirectory

Prototyp

```
static void addDirectory(const char *directoryName);
```

Beschreibung

`addDirectory(directoryName)` fügt das Verzeichnis `directoryName` zu den Verzeichnissen hinzu, in denen nach Meldungsdateien gesucht werden soll.

9 Funktionen zum Lesen von Environment-Variable

Mit folgenden Funktionen können die Werte von Environment-Variable gelesen werden:

G_getEnvVar

Prototyp

```
int G_getEnvVar(const char *varId, int defaultValue);
```

Beschreibung

`G_getEnvVar(varId, defaultValue)` liefert den Wert der Environment-Variable `varId` als `int`-Wert. Existiert diese Variable nicht oder enthält sie keinen `int`-Wert, liefert die Funktion `defaultValue`.

G_getEnvVar

Prototyp

```
void G_getEnvVar(const char *varId, char *dest,
                 const char *defaultValue, int maxLength);
```

Beschreibung

`G_getEnvVar(varId, dest, defaultValue, maxLength)` liest den Wert der Environment-Variable `varId` als String nach `dest`. Es werden maximal `maxLength`-1 Zeichen gelesen und das Resultat mit dem Stringende-Zeichen abgeschlossen. Existiert die Variable nicht, wird `defaultValue` verwendet.

A Mehrfache Vererbung bei Attributierungsklassen

Soll von unterschiedlichen Typen (z.B. `TypeA` und `TypeB`) mit unterschiedlichen Attributierungsklassen (`ClassA` und `ClassB`) ein gemeinsamer Untertyp gebildet werden, muß auch eine gemeinsame Attributierungsklasse (`ClassAB`) für diesen Typ definiert werden. Damit die Instanzen dieser Klasse die Elemente von **G_attribute** nur einmal enthalten, muß **G_attribute** virtuelle Basisklasse von `ClassA` und `ClassB` sein.

```
class ClassA : public virtual G_attribute
{
public:
  int getA();
  ...
};

class ClassB : public virtual G_attribute
{
public:
  int getB();
  ...
};

class ClassAB : public virtual ClassA, public virtual ClassB
{
  ...
};
```

Leider führt dieses Vorgehen zu einigen Problemen. In *C++* ist es derzeit[23] nicht möglich, einen Zeiger auf die virtuelle Basisklasse zu einem Zeiger auf die davon abgeleitete Klasse umzuwandlen. Im Beispiel ist es also nicht möglich, einen Zeiger des Typs `G_attribute *` (z.B. einen Rückgabewert von **G_graph::getPVAttr**) in einen Zeiger des Typs `ClassA *` (mittels *type cast*) umzuwandeln. Im Graphenlabor wird dieses Problem zu lösen versucht, indem eine zusätzliche virtuelle Methode **G_attribute::getPAttr** definiert wird, die einen Zeiger des Typs `void *` auf das „ganze“ Attribut liefert. Diese Methode muß in jeder abgeleiteten Klasse redefiniert werden. Der Rückgabewert dieser Methode kann dann (mittels *type cast*) in einen Zeiger des benötigten Typs umgewandelt werden. Dadurch wird z.B. folgender Programmtext möglich:

```
G_attribute *pAttr1;
void        *pAttr2;
ClassA      *pAttr3;
int         valueA;

pAttr1=g.getPVAttr(v);
```

[23] Im nächsten *C++* Standard soll aber wohl dynamische Typumwandlung im Zusammenhang mit der Laufzeittypinformation enthalten sein.

```
pAttr2=pAttr1->getPAttr();
pAttr3=(ClassA *)pAttr2;
valueA=pAttr3->getA();
```

Ist das Attribut des Knoten `v` jedoch nicht exakt vom Typ `ClassA`, sondern vom Typ `ClassAB`, führt obiger Programmtext zu unvorhersehbaren Resultaten, obwohl er problemlos übersetzt werden kann.

B Zusatzinformation für die Universität Koblenz

B.1 Environmentvariable für Universität Koblenz

Variable	Wert
EMS	/home/ems/GraLab3
EMSSRC	$EMS/src
EMSINC	$EMS/src/include
EMSLIB	$EMS/lib
EMSBIN	$EMS/bin
EMSMSG	$EMS/msg
EMSDEMO	$EMS/doc/demo
EMSMAN	$EMS/doc/manual
G_NMAX	1000
G_MMAX	1000
G_MsgPath	;$EMSMSG
G_MaxAttr	200
G_MaxType	200
G_MaxError	10

Durch Einfügen der Zeile „`source /home/ems/GraLab3/bin/setems`“ in die eigene `.cshrc`-Datei werden die Variable automatisch gesetzt.

B.2 Welche Compiler für welche Rechnerarchitekturen ?

An der Universität Koblenz steht das Graphenlabor derzeit für folgende Architektur zur Verfügung:

sun4 :
: Die Graphenlaborbibliotheken befinden sich in Verzeichnis `$EMS/lib/sun4`. Sie sind mit dem Compiler `g++` übersetzt worden, so daß nach Möglichkeit dieser Compiler zur Übersetung von EMS-Anwendungen verwendet werden sollte.

C Dateiformate

Im folgenden werden die vom Graphenlabor verwendeten Dateiformate beschrieben.

C.1 Format der Graphen-Dateien

Graphendateien werden von **G_graph::store** geschrieben und **G_graph::restore** gelesen.

C.1.1 Grammatik der Graphen-Dateien

```
<graph>         ::= <header> <structure> <usedTypes> <typesAttributes>

<header>        ::= "graph(" <vMax> " "  <eMax>
                    " " <vCount> " " <eCount> ")\n"
                    "\n"
<vMax>          ::= <CONST>
<eMax>          ::= <CONST>
<vCount>        ::= <CONST>
<eCount>        ::= <CONST>

<structure>     ::= <vStructure> <eList>
<vStructure>    ::= "structure:\n"
                    <vertex>*
                    "\n"
<vertex>        ::= <vertexNo>  ( " " <incidentEdge> )* " 0 \n"
<incidentEdge>  ::= [ "-" ] <edgeNo>
<eList>         ::= "global edge list:\n"
                    ( <edgeNo> "\n" )*
                    "\n"
<vertexNo>      ::= <CONST>
<edgeNo>        ::= <CONST>

<usedTypes>     ::= "used types:" <typeCount> "\n"
                    "max type number:" <typeMaxNo> "\n"
                    ( <typeId> " " <typeNo> "\n" )*
                    "\n"
<typeCount>     ::= <CONST>
<typeMaxNo>     ::= <CONST>
<typeId>        ::= <noWhiteSpaceString>
<typeNo>        ::= <CONST>

<typesAttributes>
                ::= <vT&A> <eT&A>
<vT&A>          ::= "vertex attributes:\n"
                    ( <typeNo> [ " " <attribute> ] "\n" ) *
```

```
                    "\n"
<vT&A>          ::= "edge attributes:\n"
                    ( <typeNo> [ " " <attribute> ] "\n" ) *
                    "\n"
<attribute>     ::= <length> <spaces> ":" <string>
<length>        ::= <CONST>

<CONST>         ::= <DIGIT>+
<DIGIT>         ::= "0" | "1" | ... | "9"
<STRING>        ::= ( any character ) *
<noWhiteSpaceString>
                ::= ( any character but " ", "\t", "\n" ) +
<spaces>        ::= " " *
```

Zur Bedeutung:

- `vMax` und `eMax` geben die Größen der internen Tabellen an.
- `vCount` gibt die Anzahl der Knoten an, `eCount` die Anzahl der Kanten.
- `vertexNo` und `edgeNo` sind die Nummern von Knoten bzw. Kanten.
- `typeCount` gibt die Anzahl der verwendeten Typen an.
- `typeMaxNo` gibt die höchste Typnummer an.
- `typeId` identifiziert einen Typ.[24].
- `typeNo` ist eine Typnummer zum Zeitpunkt des Schreibens der Datei. Beim Lesevorgang wird sie in die aktuelle Typnummer des Typs mit der gleichen Bezeichnung übersetzt.
- `attribute` ist eine Beschreibung eines Attributs, das mit der `store`-Methode der entsprechenden Attributklasse geschrieben wird.
- `length` gibt die Länge des folgenden Attributstrings `attribute` an.

Natürlich müssen die einzelnen Werte zueinander passen.

C.1.2 Beispiel

Das Attributierungsbeispiel in Abschnitt 1.5.2, S. 22 erzeugt folgende Datei `attrdemo.g`:

```
graph(1000 1000 12 16)

structure:
1 1 5 0
2 -1 2 6 0
```

[24] Vgl. dazu die Beschreibung der Klasse **G_type** in Abschnitt 5, S. 69

```
3 -2 3 0
4 -3 4 7 0
5 -4 8 0
6 -5 9 13 0
7 -6 -9 10 14 0
8 -10 11 0
9 -7 -11 12 15 0
10 -8 -12 0
11 -13 -14 16 0
12 -15 -16 0

global edge list:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

used types:7
max type number:15
Einmuendung 2
Kreuzung 3
gebuehrenpflichtiger_Parkplatz 8
gebuehrenfreier_Parkplatz 9
gebuehrenpflichtiges_Parkhaus 12
gebuehrenpflichtige_Tiefgarage 14
gebuehrenfreie_Tiefgarage 15

vertex attributes:
9
2
9
2
15 20    :6 22 12 Universitaet
2
```

```
3
12 19    :8 18 8 Kaufhaus 150
3
8 2     :50
2
14 22    :0 24 10 Opernplatz 100

edge attributes:
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

C.2 Format der Typsystem-Dateien

Typsystemdateien werden mit der Methode **G_typeSystem::store** geschrieben und dem Konstruktor **G_typeSystem::G_typeSystem** gelesen.

C.2.1 Grammatik

```
<typeSystem>    ::= <header> <types> <is-a-relation>

<header>        ::= "type system\n"
                    "max no. of types: " <typeMaxNo> "\n"
                    "no. of types: " <typeNo> "\n"
                    "\n"
<typeMaxNo>     ::= <CONST>
<typeNo>        ::= <CONST>

<types>         ::= { <typeId> " " <attrSchemaId> "\n" ) *
                    "\n"
<typeId>        ::= <noWhiteSpaceString>
<attrSchemaId>  ::= <noWhiteSpaceString>
```

```
<is-a-relation> ::= "is-a relation:\n"
                    ( <HEXCONST> " " ) * "\n"

<CONST>         ::= <DIGIT>+
<DIGIT>         ::= "0" | "1" | ... | "9"
<noWhiteSpaceString>
                ::= ( any character but " ", "\t", "\n" ) +

<HEXCONST>      ::= <HEXDIGIT>+
<HEXDIGIT>      ::= <DIGIT> | "a" | "b" | ... | "f"
```

Zur Bedeutung:

- `typeMaxNo` gibt die Größe der Typsystemtabellen an.
- `typeNo` gibt die Anzahl der Typen (ohne **G_typeNull** an.
- `typeId` identifiziert den Typ.
- `attrSchemaId` identifiziert das dem Typ zugeordnete Attributschema.
- Die Subtyp-Relation[25] wird mittels Bitstrings hexadezimal gespeichert.

C.2.2 Beispiel

Das Attributierungsbeispiel in Abschnitt 1.5.2, S. 22 erzeugt folgende Datei `attrdemo.t`:

```
type system
max no. of types: 200
no. of types: 15

Verzweigung NULL
Einmuendung NULL
Kreuzung NULL
Parkeinrichtung NULL
gebuehrenpflichtige_Einrichtung GEBUEHR
gebuehrenfreie_Einrichtung NULL
Parkplatz NULL
gebuehrenpflichtiger_Parkplatz GEBUEHR
gebuehrenfreier_Parkplatz NULL
Parkgebaeude GEBAEUDE
Parkhaus GEBAEUDE
gebuehrenpflichtiges_Parkhaus GEBAEUDEMITGEBUEHR
Tiefgarage GEBAEUDE
gebuehrenpflichtige_Tiefgarage GEBAEUDEMITGEBUEHR
gebuehrenfreie_Tiefgarage GEBAEUDE
```

[25] genaugenommen ihr transitiver Abschluß

```
is-a relation:
00000001
00000003
00000007
0000000b
00000011
00000031
00000051
00000091
000001b1
000002d1
00000411
00000c11
00001c31
00002411
00006431
0000a451
```

C.3 Format der Meldungsdateien

Meldungstexte zerfallen in zwei Teile, nämlich die einleitenden Texte, die bei allen Meldungen gleich sind (Meldungsköpfe), und die für jede Meldung individuellen Texte.

C.3.1 Meldungsköpfe

In einer Datei des folgenden Formats sind die Texte angegeben, mit denen Fehlermeldungen eingeleitet werden und die einzelnen Teile einer Meldung voneinander abgegrenzt werden. Sie wird mit der Methode **G_msg::setHeaders** gelesen.

```
<msgHeads> ::= <MsgStr> "\n"
               <ErrStr> "\n"
               <FErStr> "\n"
               <InfStr> "\n"
               <ConFndStr> "\n"
               <ActTakStr> "\n"
               <ReaProStr> "\n"

<MsgStr>    ::= <STRING>
<ErrStr>    ::= <STRING>
<FErStr>    ::= <STRING>
<InfStr>    ::= <STRING>
<ConFndStr> ::= <STRING>
<ActTakStr> ::= <STRING>
<ReaProStr> ::= <STRING>
<PCHAR>     ::= any printable character but "\n"
```

```
<STRING>    ::= <PCHAR>*
```

Alle Strings dürfen maximal 100 Zeichen lang sein. Sie haben folgende Bedeutung:

`<MsgStr>` Anfang einer Warnmeldung

`<ErrStr>` Anfang einer Fehlermeldung

`<FErStr>` Anfang der Meldung eines fatalen Fehlers

`<InfStr>` Formatstring gemäß `printf`, der Art der Ausgabe von Modulname, Meldungsnummer und Funktionsname. Dieser String muß also „`%s`“„`%d`“ und „`%s`“ in dieser Reihenfolge enthalten.

`<ConFnd>` Anfang der Beschreibung der Fehlerbedingung

`<ActTak>` Anfang der Beschreibung des Verhaltens des Graphenlabors (bzw. Anwendungsprogramms bei eigenen Fehlermeldungen)

`<ReaPro>` Anfang des Vorschlags zur Fehlerbeseitigung

Beispiel

Das Graphenlabor verwendet standardmäßig die Texte aus der Datei `msgems`:

```
****** MESSAGE
****** ERROR
****** FATAL ERROR
 %s%03d in %s
****** Condition found  :
****** Action taken     :
****** Reaction proposed:
```

C.3.2 Meldungstexte

Für jedes Modul existiert eine Meldungsdatei mit dem Namen `msg<moduleName>`. Diese Dateien werden bei jeder Meldung durch die Methoden **G_msg::error**, **G_msg::warning** und **G_msg::fatalError** gelesen.

```
<messages> ::= <message>*
<message>  ::= <msgHead> [<conFnd>] [<actTak>] [<reaPro>] "\n"
<msgHead>  ::= "MSG" <DIGIT> <DIGIT> <DIGIT> "\n"
<conFnd>   ::= "C {" <STRING> "}\n"
<actTak>   ::= "A {" <STRING> "}\n"
<reaPro>   ::= "R {" <STRING> "}\n"
<CHAR>     ::= any character but "}"
<STRING>   ::= <CHAR>*
```

Zur Ausgabe von zusätzlichen Parametern müssen die Strings in `<conFnd>`, `<actTak>` und `<reaPro>` zusammen die erforderlichen Umwandlungsspezifizierer („`%d`“ etc.) enthalten, deren Anzahl und Reihenfolge vom Programm für jede Meldung fest vorgegeben ist. Als Beispiel für einen Eintrag in diesen Dateien siehe Beschreibung der Funktion **G_msg::error** (Abschnitt 8.2, S. 75).

D Liste der Environment-Variablen

Folgende Environment-Variable beeinflußen das Verhalten des Graphenlabor. Falls sie nicht gesetzt sind, werden Defaultwerte aus `graphconfig.h` verwendet.

G_NMAX :
Der `int`-Wert gibt die Anzahl der Knoten an, die ein Graph unmittelbar nach der Erzeugung aufnehmen kann. Der Wert kann durch die explizite Angabe dieser Größe im Konstruktoraufruf **G_graph::G_graph** ignoriert werden. Werden mehr Knoten erzeugt, wird für den jeweiligen Graphen die Anzahl verdoppelt.

G_MMAX :
Der `int`-Wert gibt die Anzahl der Kanten an, die ein Graph unmittelbar nach der Erzeugung aufnehmen kann. Der Wert kann durch die explizite Angabe dieser Größe im Konstruktoraufruf **G_graph::G_graph** ignoriert werden. Werden mehr Kanten erzeugt, wird für den jeweiligen Graphen die Anzahl verdoppelt.

G_MaxAttr :
Der `int`-Wert gibt die Anzahl der maximal verfügbaren Attributschemata an.

G_MaxType :
Der `int`-Wert gibt die Anzahl der maximal in einem Typsystem verfügbaren Typen an. Der Wert kann durch die explizite Angabe dieser Größe im Konstruktoraufruf **G_typeSystem::G_typeSystem** ignoriert werden.

G_MaxError :
Der `int`-Wert gibt an, nach wievielen Fehlern das Graphenlabor das Programm abbricht. Er wird in der statischen Variable **G_msg::maxErrorCount** bei der Initialisierung der Message-Komponente gespeichert und kann dort überschrieben werden.

G_MsgPath :
Der String gibt an, in welchen Verzeichnissen nach Meldungsdateien gesucht wird. Mit der Methode **G_msg::addDirectory** können zusätzliche Verzeichnisse angegeben werden.

Literatur

[EbeFra 94] J. Ebert, A. Franzke. *A Declarative Approach to Graph Based Modeling.* In: E. Mayr, G. Schmidt, G. Tinhofer (Eds.): Graph Theoretic Concepts in Computer Science. Berlin: Springer, LNCS 1994.

[Strou 92] B. Stroustrup. *Die C++-Programmiersprache,* 2. überarb. Aufl. Bonn: Addison-Wesley 1992.

Index