Hinweise zur Benutzung von LEDA

1 Allgemeines

LEDA (Library of Efficient Data types and Algorithms) ist eine Bibliothek von C++-Klassen, die am MPI in Saarbrücken entwickelt wurde und die eine Vielzahl von höheren Datenstrukturen sowie Hilfsmittel zur Visualisierung und Animation zur Verfügung stellt. Für Zwecke der Forschung und der Ausbildung darf LEDA frei benutzt werden; für kommerzielle Anwendungen muß eine Lizenz käuflich erworben werden. Näheres zu LEDA ist im WWW unter der folgenden URL zu finden:

`http://www.mpi-sb.mpg.de/LEDA`

Im Praktikum werden wir LEDA Version 3.5.2 benutzen. Es sind aber bereits die neueren Versionen 3.6 und 3.6.1 verfügbar, die beim Arbeiten zu Hause ebenfalls benutzt werden können.

2 Installationen

LEDA ist in der HP-Halle unter `/usr/proj/leda` und auf den hpmayr-Rechnern unter `/usr/local/lea/LEDA-3.5.2` installiert. Im Unterverzeichnis `Manual` auf den hpmayr-Rechnern liegt auch ein komplettes LEDA-Manual als PostScript-Datei (mit gzip komprimiert). Im Unterverzeichnis `graphen` werden im Verlauf des Praktikums Testgraphen bereitgestellt, die als Eingabe für LEDA-Programme dienen können. Sie sollten in das Verzeichnis kopiert werden, in dem auch die LEDA-Programme gestartet werden (zwecks einfacheren Einladens mittels der Funktion "Load Graph"). Der Online-Manual-Viewer von LEDA kann in der HP-Halle mit `/usr/proj/leda/bin/xlman` und auf den hpmayr-Rechnern mit `/usr/local/lea/bin/xlman` gestartet werden.

LEDA kann ohne Probleme auch auf einem PC unter Linux oder unter Windows installiert werden. Zur Installation von LEDA 3.5.2 unter Linux kann das tgz-Archiv verwendet werden, das ebenfalls in den oben genannten Verzeichnissen bereitliegt. (Das Archiv entpacken, dann `./lconfig g++ shared` eingeben, dann `make lib`.) Wer LEDA 3.6.1 verwenden will, kann sich die vorkompilierte Bibliothek über obige URL bei der LEDA Software GmbH besorgen (kostenlose *research license*).

3 Verwendung

Um ein C++-Programm, das LEDA verwendet, zu übersetzen und zu linken, müssen dem Compiler das Verzeichnis mit den LEDA-Includes und die LEDA-Bibliotheken bekannt gemacht werden. Auf den hpmayr-Rechnern funktioniert dies (als C++-Compiler muß auf den hpmayr-Rechnern `g++27` verwendet werden, in der HP-Halle dagegen `g++`) etwa durch Angabe von `-I/usr/local/lea/LEDA-3.5.2/incl` für die Includes und `-L/usr/local/lea/LEDA-3.5.2 -lW -lP -lG -lL -lX11 -lm` für die Bibliotheken. Ein entsprechendes `Makefile` sowie ein Beispielprogramm `dfs.C` (mit zugehörigem

`control.h`) liegen in den LEDA-Verzeichnissen in der HP-Halle und auf den hpmayr-Rechnern als Vorlage bereit. Nach Kopieren der drei Dateien in ein eigenes Praktikumsverzeichnis sollte sich `dfs.C` mittles `make dfs` übersetzen lassen. Ein selbst erstelltes Programm `foo.C` kann dann mittels `make foo` übersetzt werden.

Soll eine Klasse `xyz` aus LEDA im eigenen C++-Programm verwendet werden, so muß nur mittels `#include <LEDA/xyz.h>` der entsprechende Header eingebunden werden. Einige der Klassen, die wir im Praktikum verwenden werden, sind:

> `string` (ähnlich `char *` in C++, aber mit zusätzlichen Features),
> `random_source` (Erzeugung von Zufallszahlen), `stack`, `queue`, `list`,
> `set` (Menge von Elementen), `partition` (Partition einer Menge),
> `map` (Abbildung von einem Typ auf einen anderen),
> `p_queue` (Prioritäts-Warteschlange), `graph` (LEDA-Graph),
> `node/edge_array` (Zuordnung von Werten zu Knoten bzw. Kanten),
> `node/edge_map` (dynamische Variante von `node/edge_array`),
> `node/edge_set` (Menge von Knoten bzw. Kanten),
> `node_partition` (Partition der Knotenmenge eines Graphen),
> `node_pq` (Prioritäts-Warteschlange von Knoten eines Graphen),
> `color` (Definition von Farben), `window` (Bildschirm-Fenster) und
> `GraphWin` (Darstellung von Graphen auf dem Bildschirm und Interaktion mit dem Benutzer).

Näheres zur Funktionalität und Benutzung dieser Klassen sollte z.B. im Online-Manual-Viewer nachgelesen werden. Außerdem stehen in `LEDA/basic.h` verschiedene nützliche Funktionen zur Verfügung, die im Manual-Abschnitt "misc" erklärt sind.

Als einfaches Beispiel sei kurz die Benutzung einer Queue (Warteschlange) aus LEDA skizziert. Nach `#include <LEDA/queue.h>` steht der Template-Typ `queue<T>` zur Verfügung, wobei `T` einen beliebigen Typ für die Elemente der Queue spezifizieren kann. Etwa kann mit `queue<node> Q` eine Queue von Knoten (`node` ist der Typ für Knoten eines LEDA-Graphen) deklariert werden. An diese Queue kann mit `Q.append(v)` ein Knoten angehängt und mit `v=Q.pop()` ein Knoten herausgeholt werden. Mit `Q.empty()` kann Leerheit der Queue getestet werden.

Die komplexeste im Praktikum verwendete Klasse ist wohl `GraphWin`, die Klasse zur Darstellung von Graphen auf dem Bildschirm. Wir werden diese Klasse verwenden, um den Benutzer einen Graphen eingeben oder laden zu lassen und um anschließend die Arbeitsweise und das Ergebnis unserer Algorithmen zu visualisieren. Dazu können wir vielerlei Funktionen zur Veränderung der Darstellung und der Labels von Knoten und Kanten auf dem Bildschirm einsetzen.

Das von `dfs.C` verwendete Include-File `control.h` realisiert ein kleines Kontrollfenster, das am Programmanfang mit `create_control()` sichtbar gemacht werden muß und am Ende mit `destroy_control()` wieder geschlossen werden sollte. Das Kontrollfenster realisiert eine Art "Fernbedienung", mit der der Animationsablauf kontrolliert werden kann (Stop, Continue, etc.), wenn im Programm die Funktion `leda_wait()` für Verzögerungen verwendet wird.

Warnung: LEDA ist zwar recht zuverlässig, aber nicht fehlerfrei! Man sollte beim Debuggen also auch berücksichtigen, daß ein Fehler in LEDA die Ursache sein kann.

Beispielprogramm: dfs.C

```
//
// Animation einer Tiefensuche in ungerichteten Graphen
// (unbesuchter Teil wird gelb (Default) angezeigt, erledigter Teil blau,
// noch in Bearbeitung befindlicher Teil rot, Rueckwaertskanten gruen)
//
#include <iostream.h>
#include <LEDA/graphwin.h>
#include <LEDA/color.h>
#include <LEDA/basic.h>

#include "control.h" // Fernbedienung

// rekursive Funktion zur Realisierung der Tiefensuche (dfs);
// Parameter:
//    parent: Knoten, von dem aus der aktuelle Knoten besucht wird
//            (parent == v, falls der aktuelle Knoten der Startknoten ist)
//    v: aktueller Knoten
//    g: zu durchsuchender Graph (als Referenz)
//    gw: Darstellungsfenster des Graphen (als Referenz)
//    dfsnum: Feld zur Zuordnung von DFS-Nummern zu Knoten (als Referenz)
//    akt: naechste freie Nummer (als Referenz)
void dfs(node parent, node v, graph &g, GraphWin &gw,
         node_array<int> &dfsnum, int &akt)
{
    dfsnum[v] = akt++;                          // DFS-Nummer zuweisen
    gw.set_user_label(v,string("%d",dfsnum[v]));// DFS-Nummer anzeigen
    gw.set_color(v,red);                        // Knoten rot faerben
    leda_wait(0.5);                             // 0.5 sec warten
    edge e;
    forall_adj_edges(e,v) // alle Nachbarkanten von v
        {
        node w = g.opposite(v,e); // Knoten am anderen Ende von e
        if (w != parent) // die Kante zum parent ignorieren wir
            {
            if (dfsnum[w]<0) // falls Knoten w noch nicht besucht
                {
                gw.set_color(e,red);      // Kante zu w rot faerben
                gw.set_width(e,2);        // Kante fett anzeigen
                dfs(v,w,g,gw,dfsnum,akt); // rekursiver Aufruf fuer w
                gw.set_color(e,blue);     // Kante blau faerben
                leda_wait(0.5);           // 0.5 sec warten
                }
            else // Knoten w war schon besucht
                {
                if (dfsnum[w]<dfsnum[v]) // Rueckwaertskante
                        {
                        gw.set_color(e,green); // gruen faerben
                        leda_wait(0.5);        // 0.5 sec warten
                        }
                }
            }
        }
    gw.set_color(v,blue); // Knoten blau faerben
}

// Haupt-Programm
main(int argc, char *argv[])
{
// Fenster der Groesse 800 x 600 zur Graphendarstellung erzeugen
GraphWin gw(800.0,600.0);
```

```
gw.display(); // Fenster auf den Bildschirm bringen

create_control(); // "Fernbedienung" anzeigen

gw.set_directed(false); // ungerichtete Darstellung (keine Pfeile an Kanten)

// Hintergrundfarbe auf grau setzen (Kantenlabels werden manchmal
// weiss angezeigt und sind bei weissem Hintergrund unsichtbar)
gw.set_bg_color(grey1);

if (argc>1) gw.read(argv[1]); // falls Name als Parameter, Graph laden

gw.edit(); // in Editier-Modus gehen, bis der Benutzer "done" klickt

// jetzt holen wir uns den Graphen, den der Benutzer eingegeben oder
// geladen hat
graph & g = gw.get_graph();

if (g.number_of_nodes()==0) // wenn der Graph leer ist, tun wir gar nichts
        {
        gw.close(); destroy_control();
        exit(1);
        }

// GraphWin-Graphen sind immer gerichtet, auch wenn auf dem Bildschirm
// keine Pfeile sichtbar sind; wir wollen aber einen ungerichteten Graphen
g.make_undirected();

// Jetzt deklarieren wir ein Feld, das jedem Knoten eine Nummer zuordnet,
// und initialisieren es mit "-1"
node_array<int> dfsnum(g,-1);

// Nun zeigen wir fuer alle Knoten den dfsnum-Wert als User-Label an
node v;
forall_nodes(v,g)
        {
        gw.set_user_label(v,string("%d",dfsnum[v]));
        gw.set_label_type(v,user_label);
        }

// in dieser Variable merken wir uns die naechste zu vergebende Nummer
// (und gleichzeitig die Zahl der schon besuchten Knoten)
int akt = 0;

do {
   // jetzt lassen wir den Benutzer mit der Maus einen unbesuchten Knoten
   // auswaehlen (wenn er danebenklickt, wird NULL zurueckgeliefert),
   v = NULL;
   while ( (v = gw.read_node()) == NULL || dfsnum[v]>=0) ;

   // nun rufen wir die rekursive DFS-Funktion auf
   dfs(v, v, g, gw, dfsnum, akt);
} while (akt < g.number_of_nodes()); // bis alle Knoten besucht wurden

gw.acknowledge("Ready!"); // Dialogbox anzeigen und bestaetigen lassen

gw.edit(); // nochmal in den Edit-Modus, zum Anschauen :)

// Jetzt raeumen wir das Fenster auf und machen Schluss
gw.close(); destroy_control();
exit(0);
}
```