Praktikum Diskrete Optimierung

1 LEDA

LEDA (Library of Efficient Data types and Algorithms) is a library of C++-classes which was developed at the MPI in Saarbrücken, and which provides a variety of higher data structures and tools for visualization and animation. Since February 2001 Algorithmic Solutions Software GmbH is the sole distributor of LEDA. By now, a royalty has also to be paid for the research edition. More to LEDA can be found in the internet:

`http://www.algorithmic-solutions.com/`

In this practical course we will use LEDA 5.2 and the compiler g++ (version 4.3.x).

2 Installations

LEDA can be found at `/usr/local/LEDA` on the LINUX computers of the chair. The main directory that contains the files which have to be included is `/usr/local/LEDA/incl/LEDA`. To work with LEDA, the following environment variables must possibly be set:

bash / ksh:

```
export LEDAROOT=/usr/local/LEDA
export LD_LIBRARY_PATH=$LEDAROOT:$LD_LIBRARY_PATH
```

tcsh / csh:

```
setenv LEDAROOT /usr/local/LEDA
setenv LD_LIBRARY_PATH ${LEDAROOT}:${LD_LIBRARY_PATH}
```

3 Usage

To compile and link a C++-program using LEDA the compiler must be given the directory with the LEDA-includes and the LEDA-libraries. You can find an appropriate `makefile` and an example program `dfs.cpp` (along with the associated `control.h`) on the website of the course.

After copying the three files to a separate directory you should be able to compile `dfs.cpp` by `make dfs`. A program `foo.cpp` written by yourself can analogously be compiled by calling `make foo`. As the course progresses test graphs will be provided as inputs for your LEDA-programs on the website of the course. Those should be copied into the directory where your programs are started (in order to load them more simply using the function "Load Graph").

If you want to use a LEDA-class `foo` which is located in the subdirectory `bar` (i.e., the path is `/usr/local/LEDA/incl/LEDA/bar/foo.h`) for your program, then you just have to incorporate the corresponding header by `#include <LEDA/bar/foo.h>`. Some of the classes we will use are:

- `string` (`<LEDA/core/string.h>`): Similar to `char *` of C++ but has more features
- `random_source` (`<LEDA/core/random_source.h>`): Generation of random numbers
- `stack` (`<LEDA/core/stack.h>`)
- `queue` (`<LEDA/core/queue.h>`)
- `list` (`<LEDA/core/list.h>`)
- `set` (`<LEDA/core/set.h>`): Set of elements
- `partition` (`<LEDA/core/partition.h>`): Partition of a set
- `map` (`<LEDA/core/map.h>`): Mapping from one type to another
- `p_queue` (`<LEDA/core/p_queue.h>`): Priority queue
- `graph` (`<LEDA/graph/graph.h>`): LEDA-Graph
- `node_array` (`<LEDA/graph/node_array.h>`): Assignment of values to nodes
- `edge_array` (`<LEDA/graph/edge_array.h>`): Assignment of values to edges
- `node_map` (`<LEDA/graph/node_map.h>`): Dynamic variation of `node_array`
- `edge_map` (`<LEDA/graph/edge_map.h>`): Dynamic variation of `edge_array`
- `node_set` (`<LEDA/graph/node_set.h>`): Set of nodes
- `edge_set` (`<LEDA/graph/edge_set.h>`): Set of edges
- `node_partition` (`<LEDA/graph/node_partition.h>`): Partition of the node set of a graph
- `node_pq` (`<LEDA/graph/node_pq.h>`): Priority queue of nodes of a graph
- `color` (`<LEDA/graphics/color.h>`): Definitions of colors
- `window` (`<LEDA/graphics/window.h>`): Screen window
- `GraphWin` (`<LEDA/graphics/graphwin.h>`): Display of graphs on the screen, and user interface

More in-depth description of these classes can be found, e.g., in the online-manual-viewer. Furthermore, `LEDA/system/basic.h` provides several useful functions which can be looked up in the section "misc" of the manual.

We illustrate a queue of LEDA as simple example. The expression `#include <LEDA/core/queue.h>` provides the template-type `queue<T>` where `T` is an arbitrary type and specifies the type of the elements of the queue. For instance,

`queue<node> Q` declares a queue of nodes. We can append nodes to the queue by `Q.append(v)`, and pop a node from the queue by `v=Q.pop()`. The expression `Q.empty()` tests a queue for emptyness.

The most complex class used in the practical couse probably is `GraphWin` which is used to display the graph on the screen. We will use this class to let the user input or load a graph which is then used to visualize how the algorithm works as well as the result of the algorithm. To this end, we can use a variety of functions for the modification of the visualization, and of the labels of nodes and edges. Note that GraphWin-graphs are always directed. Undirected graphs are realized by visualizing the edges as undirected edges. We can iterate over all incident edges of a node by `forall_inout_edges(e,v)` (see the example program `dfs.cpp`).

The include-file `control.h` used by `dfs.cpp` realizes a small control window which must be made visible by `create_control()` at the beginning of the program. At the end it should be destroyed by `destroy_control()`. The control window realizes some kind of "remote control" which we can use to control the animation process (Stop, Continue, etc.) if the program uses the function `control_wait()` for delays.

4 Assignments

For solving the assignment sheets you can use (most of) the computers of the chair located in room 03.09.034. If you want to use your own computer make sure to use version 5.2 or LEDA. In any case, we must be able to compile your programs using the `Makefile` of the website and the computers of the chair. Please use the respective names proposed in the assignment sheets for your programs (for example `bfs` for the first program of the first sheet).

You will work in teams of two persons. It is recommended to work and implement the programs together in close collaboration, and not to partition the different assignments amongst you.

We emphasize that you must provide your own solution and not use programs of other groups as "template". If you have problems with the implementation don't use some solution of another group. (Of course, you are allowed and encouraged to discuss ways of implementing certain aspects of your programs with other groups but you are not allowed to share source code.) Instead, you can talk to your respective advisor in the consultation-hour if you have matters with understanding or the implementation.

5 Submission of solutions

You must submit your solution until the respective due date (normally a week after the release of the assignment sheet). To this end, send an email with the subject

`Optprak SS2013 Gruppe` x `Blatt` y

and your programs as the attachment to algoprak@in.tum.de where x is your group number and y is the number of the assignment sheet. The programs should have the respective names proposed in the assignment sheet.

We review the solutions and test them on correctness using the test inputs provided on the website as well as additional test data. Furthermore, we will test the solutions on efficient implementation. We will judge your submissions either "OK" or "not OK". A submission will be judged "OK" if it meets the following criteria:

- Correctness of the calculated results (using the given test inputs and our additional inputs)
- Efficiency of the implementation (avoidance of inefficient constructs such that the Worst-Case running time is met when removing the animation procedures)
- Quality of the animation (the algorithm should be vividly visualized)
- Readability of the source code (sufficient useful comments which help the reader understanding the source code)

Submissions that cannot be judged "OK" will be sent back to the authors with comments on the bugs or deficits. In this case, the authors are allowed to rework their solution and to submit a revised version within at most one additional week. This opportunity does not apply to the case of detected plagiarism. The deadlines are firm and cannot be extended.

6 Certificates

Each team member obtains a certifcate if

- *all* assignments were treated,
- *all except for two* of the submissions of the team were jugded "OK", and
- the oral examination at the end of the semester was passed.

Within the oral exam at the end of semester, we expect every student to be able to answer questions to *all* assignments of the course. This also includes the code of her/his group.

7 F.A.Q.

1. **Q:** Should I use `p_queue` or `node_pq` for nodes?
   A: `node_pq`

2. **Q:** I get a segmentation fault error, what's wrong?
   A: Often, the reason for this error is that the data structure representing the graph is not synchronous with the visual representation. Usually, this can be repaired by adding the command `gw.update_graph();` (where `gw` is the GraphWin object) at the respective position in the code (after modifying the graph or before calling commands that use the graphical representation like, e.g., the edit mode).

3. **Q:** Why shouldn't I use copy constructors?
   A: They do not work the same way as in Java, so use references or pointers instead.

4. **Q:** Why shouldn't I use 2D arrays?
   A: Just don't!

8 Example program: dfs.cpp

```
// Animation einer Tiefensuche in ungerichteten Graphen
// (unbesuchter Teil wird gelb (Default) angezeigt, erledigter Teil blau,
// noch in Bearbeitung befindlicher Teil rot, Rückwärtskanten grün)

#include <iostream>
#include <climits>
#include <LEDA/graphics/graphwin.h>
#include <LEDA/graphics/color.h>
#include <LEDA/system/basic.h>
#include "control.h" // Fernbedienung

using leda::graph;
using leda::node;
using leda::edge;
using leda::node_array;
using leda::user_label;
using leda::GraphWin;
using leda::red;
using leda::blue;
using leda::green;
using leda::yellow;
using leda::string;

// 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: nächste 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 färben
    gw.redraw();                                        // Darstellung aktualisieren
    control_wait(0.5);                                  // 0.5 sec warten

    // GraphWin-Graphen sind immer gerichtet, auch wenn auf dem Bildschirm
    // keine Pfeile sichtbar sind

    edge e;
```

```
    forall_inout_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 färben
                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 färben
                control_wait(0.5);             // 0.5 sec warten
            } else {    // Knoten w war schon besucht
                if (dfsnum[w] < dfsnum[v]) { // Rückwärtskante
                    gw.set_color(e, green);    // grün färben
                    control_wait(0.5);         // 0.5 sec warten
                }
            }
        }
    }
    gw.set_color(v, blue); // Knoten blau färben
    gw.redraw(); // Darstellung aktualisieren (zur Sicherheit)
}

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

    gw.display(); // Fenster auf den Bildschirm bringen
    create_control(); // "Fernbedienung" anzeigen
    gw.set_directed(false); // ungerichtete Darstellung (keine Pfeile an Kanten)
    if (argc > 1)    // falls Name als Parameter, Graph laden
        gw.read(argv[1]);

    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) {  // Ende, wenn der Graph leer ist.
        gw.close(); destroy_control();
        exit(1);
    }

    // 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
    // sowie initialisieren den Graphen gelb.
    node v;
    forall_nodes(v, g) {
        gw.set_label_type(v, user_label);    // User-Label anzeigen (statt Index-Label)
        gw.set_user_label(v, string("%d", dfsnum[v])); // User-Label auf dfsnum[v] setzen
        gw.set_color(v, yellow);
    }
    edge e;
    forall_edges(e, g)
        gw.set_color(e, yellow);

    // in dieser Variable merken wir uns die nächste 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
        // auswählen (wenn er danebenklickt, wird NULL zurückgeliefert),
```

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

    // Aufräumen und Ende
    gw.close();
    destroy_control();
    exit(0);
}
```