Praktikum: Algorithmenentwurf

1 LEDA

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. Seit Februar 2001 ist Algorithmic Solutions Software GmbH alleiniger Distributor für LEDA geworden. Für eine Version von LEDA muss mittlerweile auch für Lehre und Forschung eine Nutzungsgebühr entrichtet werden. Näheres zu LEDA ist im WWW unter der folgenden URL zu finden:

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

Im Praktikum werden wir LEDA 5.2 mit dem Compiler g++ (Version 4.3.x) benutzen.

2 Installationen

LEDA ist unter `/usr/local/LEDA` auf den den LINUX Rechnern des Lehrstuhls installiert. Das Hauptverzeichnis der einzubindenden Dateien ist `/usr/local/LEDA/incl/LEDA`. Um mit LEDA zu arbeiten, sind gegebenenfalls noch folgende Umgebungsvariablen zu setzen:

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 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. Ein entsprechendes `Makefile` sowie ein Beispielprogramm `dfs.cpp` (mit zugehörigem `control.h`) erhalten Sie auf der Praktikumswebpage.

Nach Kopieren der drei Dateien in ein eigenes Verzeichnis sollte sich `dfs.cpp` mittels `make dfs` übersetzen lassen. Ein selbst erstelltes Programm `foo.cpp` kann dann amalog mit dem Aufruf `make foo` übersetzt werden. Auf der Praktikumswebpage 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”).

Soll eine Klasse `foo` aus LEDA, das im Unterverzeichnis `bar` liegt (d.h. der Pfad ist `/usr/local/LEDA/incl/LEDA/bar/foo.h`) im eigenen C++-Programm verwendet werden, so muss nur mittels `#include <LEDA/bar/foo.h>` der entsprechende Header eingebunden werden. Einige der Klassen, die wir im Praktikum verwenden werden, sind:

- `string` (`<LEDA/core/string.h>`): ähnlich `char *` in C++, aber mit zusätzlichen Features
- `random_source` (`<LEDA/core/random_source.h>`): Erzeugung von Zufallszahlen
- `stack` (`<LEDA/core/stack.h>`)
- `queue` (`<LEDA/core/queue.h>`)
- `list` (`<LEDA/core/list.h>`)
- `set` (`<LEDA/core/set.h>`): Menge von Elementen
- `partition` (`<LEDA/core/partition.h>`): Partition einer Menge
- `map` (`<LEDA/core/map.h>`): Abbildung von einem Typ auf einen anderen
- `p_queue` (`<LEDA/core/p_queue.h>`): Prioritäts-Warteschlange
- `graph` (`<LEDA/graph/graph.h>`): LEDA-Graph
- `node_array` (`<LEDA/graph/node_array.h>`): Zuordnung von Werten zu Knoten
- `edge_array` (`<LEDA/graph/edge_array.h>`): Zuordnung von Werten zu Kanten
- `node_map` (`<LEDA/graph/node_map.h>`): Dynamische Variante von `node_array`
- `edge_map` (`<LEDA/graph/edge_map.h>`): Dynamische Variante von `edge_array`
- `node_set` (`<LEDA/graph/node_set.h>`): Menge von Knoten
- `edge_set` (`<LEDA/graph/edge_set.h>`): Menge von Kanten
- `node_partition` (`<LEDA/graph/node_partition.h>`): Partition der Knotenmenge eines Graphen
- `node_pq` (`<LEDA/graph/node_pq.h>`): Prioritäts-Warteschlange von Knoten eines Graphen
- `color` (`<LEDA/graphics/color.h>`): Definitionen von Farben
- `window` (`<LEDA/graphics/window.h>`): Bildschirm-Fenster
- `GraphWin` (`<LEDA/graphics/graphwin.h>`): 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/system/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/core/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. Zu beachten ist, dass GraphWin-Graphen intern immer gerichtet sind und man ungerichtete Graphen dadurch realisiert, dass man die Kanten am Bildschirm ungerichtet darstellen lässt und die Nachbarkanten eines Knotens mittels `forall_inout_edges(e,v)` durchläuft (siehe `dfs.cpp`-Beispielprogramm).

Das von `dfs.cpp` 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 `control_wait()` für Verzögerungen verwendet wird.

4 Bearbeitung der Aufgaben

Für die Bearbeitung der Praktikumsaufgaben können die (meisten) Praktikumsrechner am Lehrstuhl (Raum 03.09.034) verwendet werden. Bei der Bearbeitung auf anderen Rechnern ist darauf zu achten, dass LEDA in der Version 5.2 verwendet wird. In jedem Fall müssen sich die abgegebenen Programme auf den Rechnern am Lehrstuhl mit dem im Netz zur Verfügung gestellten `Makefile` kompilieren lassen. Als Name für Ihr Programm verwenden Sie bitte den auf dem Aufgabenblatt vorgeschlagenen Namen (bspw. `bfs` für das erste Programm des ersten Aufgabenblattes).

Die Aufgaben werden in Zweierteams bearbeitet. Es wird dringend empfohlen, die Aufgaben nicht aufzuteilen, sondern in Zusammenarbeit zu lösen und zu implementieren.

Wir legen großen Wert darauf, dass die Lösungen selbst erarbeitet und nicht Programme anderer Gruppen als "Vorlage" verwendet werden. Falls Probleme bei der Implementierung einer Lösung auftreten, sollte keinesfalls auf die evtl. schon fertigen Lösungen einer anderen Gruppe zurückgegriffen werden. (Diskussionen über Implementierungsmöglichkeiten sind natürlich erlaubt und erwünscht, die direkte Weitergabe von Quelltexten ist jedoch nicht gestattet.) Stattdessen können Verständnis- oder Implemen-

tierungsprobleme mit dem jeweiligen Betreuer in der Sprechstunde besprochen werden.

5 Abgabe

Bis zum jeweiligen Abgabetermin (in der Regel eine Woche nach Ausgabe des Blattes) muss jedes Team seine Lösung abgeben. Dazu wird eine E-Mail mit dem Betreff

`Algoprak WS2013 Gruppe` x `Blatt` y

an algoprak@in.tum.de mit den Programmen im Anhang geschickt, wobei x die Gruppennummer und y die Number des Aufgabenblattes ist. Die Programme sollten den auf dem Übungsblatt angegebenen Namen tragen.

Die Lösungen werden von uns durchgesehen und anhand von auf der Webseite zur Verfügung gestellten Testeingaben und weiteren Beispieldaten auf Korrektheit getestet. Ferner werden die Abgaben auf effiziente Implementierung geprüft. Jede Abgabe wird entweder mit "OK" oder "nicht OK" bewertet. Eine Lösung wird dann mit "OK" bewertet, wenn folgende Kriterien erfüllt sind:

- Korrektheit der berechneten Ergebnisse (bei den Beispielgraphen und beliebigen weiteren Eingaben)
- Effizienz der Implementierung (Vermeidung ineffizienter Konstrukte, so dass bei Entfernung der Animationsroutinen die ggf. geforderte Worst-Case-Laufzeit eingehalten wird)
- Qualität der Animation (die Arbeitsweise des Algorithmus soll anschaulich visualisiert werden)
- Lesbarkeit des Quelltextes (ausreichend Kommentare, die das Verständnis erleichtern)

Aufgaben die nicht mit "OK" bewertet werden können, werden unter Hinweis auf die entsprechenden Fehler zurückgeschickt. In einem solchen Fall (außer bei Duplikaten!) ist es möglich, das Programm zu überarbeiten und *eine* korrigierte Version nachzureichen. Hierfür steht jeweils höchstens *eine* weitere Woche zur Verfügung. Diese Fristen sind fest und müssen eingehalten werden.

6 Scheinerwerb

Ein Teammitglied erhält einen Schein, wenn

- *alle* Aufgaben bearbeitet werden,
- *alle bis auf höchstens zwei* der abgegebenen Lösungen des Teams mit "OK" bewertet werden und

- die mündliche Prüfung am Semesterende bestanden wird.

In der mündlichen Prüfung am Ende des Semesters wird erwartet, dass jeder Student Fragen zu *allen* Praktikumsaufgaben beantworten kann, dies beinhaltet auch die Implementierungen seiner Gruppe.

7 F.A.Q.

1. **Q:** Soll ich `p_queue` oder `node_pq` für Knoten verwenden?
   A: `node_pq`

2. **Q:** Ich erhalte einen segmentation fault error, wo liegt der Fehler?
   A: Eine häufige Quelle für diesen Fehler ist, dass die Datenstruktur, die den Graphen speichert nicht mehr in Synchronisation mit der visuellen Darstellung des Graphen ist. Der Fehler kann oftmals beseitigt werden, indem an der passenden Stelle des Codes (nach Hinzufügen von Kanten, vor Setzen von Kantenlabels, oder vor Start des Edit-Modus) der Befehl `gw.update_graph();` aufgerufen wird, wobei `gw` das betreffende GraphWin-Objekt bezeichnet.

3. **Q:** Wieso soll ich keine copy constructors verwenden?
   A: Diese funktionieren in C++ nicht so wie in Java. Benutze stattdessen Referenzen oder Pointer.

4. **Q:** Wieso soll ich keine 2D-Arrays verwenden?
   A: Tu's einfach nicht! Aus irgendwelchen Gründen funktionieren diese in LEDA nicht wie gewollt.

8 Beispielprogramm: dfs.cpp