

LEDA

Ausarbeitung zum Seminarvortrag vom 10.11.2010

von

Uwe Breckner

Einführung

Was ist LEDA?

LEDA (Library of Efficient Data Types and Algorithms) ist eine Bibliothek für C++, die dem Benutzer Zugriff auf verschiedene kombinatorische (z.B. Shortest Path, Planarität oder Unterteilung in Partitionen) und geometrische Algorithmen (z.B. Konvexe Hüllen oder Triangulationen) bietet.

Das Leitmotiv der Entwickler war: „LEDA soll die größten Errungenschaften der Algorithmenwelt enthalten und auch Laien zugänglich machen.“

LEDA wurde 1988 von Kurt Mehlhorn und Stefan Näher als Projekt gestartet. Sie begannen die Entwicklung an der Universität des Saarlandes in Saarbrücken und zogen dann 1990/91 an das Max-Planck-Institut, welches sich in unmittelbarer Nachbarschaft zu der Universität des Saarlandes befindet. Heute ist Stefan Näher der Leiter des Projekts und Maindesigner, bzw. -Programmierer.

Motivation

Es gab viele Beweggründe eine Bibliothek wie LEDA zu erstellen. Einige besonders wichtige möchte ich hier erwähnen.

Einerseits stellte das Team immer wieder fest, dass Algorithmen, die während einer Masterarbeit implementiert wurden, allzu oft nach dem Abgang der Studenten verloren gingen, da es keinen Ort gab an dem solche Werke gespeichert werden konnten.

Weiterhin erfuhr die Projektgruppe von ehemaligen Studenten, dass viele Erfolge der Algorithmenforschung oft gar nicht in der Wirtschaft eingesetzt werden konnten. Dies lag daran, dass Entwicklung passender Datenstrukturen und die fehlerfreie Implementation der Algorithmen einen viel zu hohen intellektuellen und programmiertechnischen Aufwand forderten, um noch kosteneffizient zu sein. Sie kamen zu dem Schluss, dass Implementation ein Feld in der Algorithmenforschung sein müsste, um dem Forschungsgebiet eine größere Plattform zu verschaffen.

Schon innerhalb ihrer relativ kleinen und eng verbundenen Forschungsgruppe entdeckte das LEDA-Team Redundanzen. So wurde zum Beispiel eine Datenstruktur für einen gleichgewichteten Baum mehrmals implementiert, was Mehrarbeit darstellt, die durch eine zentrale Basis auf der gearbeitet wird, verhindert werden kann.

Ein weiterer Beweggrund LEDA zu entwickeln und die Datenstrukturen und Algorithmen zu spezifizieren, waren die einschlägigen Bücher zum Thema Algorithmen, die die Autoren in ihrem Unterricht verwendeten. Oftmals waren die dort gegebenen Spezifikationen nicht genau bzw. abstrakt genug, um daraus eine einfache Implementation abzuleiten. Dies wurde in einem Versuch belegt, in dem zwei Gruppen von Studenten einerseits Warteschlangen mit Prioritäten und andererseits einen Shortest – Path – Algorithmus implementieren sollten. Als man diese beiden Teile zusammen benutzen wollte, funktionierte dies nicht, da die Spezifikationen, die für die Implementation genutzt wurden, nicht exakt genug waren.

Die 4 Designziele von LEDA

Bei der Entwicklung von LEDA gab es vier wesentliche Designziele: Einfachheit, Erweiterbarkeit, Validität und Effizienz. Ich erkläre hier wie sie erreicht wurden und was es mitunter für Schwierigkeiten auf diesem Weg gab.

Einfachheit:

Es wurde das Ziel verfolgt, die Komplexität der Algorithmen nicht an den Benutzer weiterzugeben, da um diese zu verstehen durchaus gehobene mathematische Kenntnisse vonnöten sein können. Die Implementation wurde gekapselt und so können auch Laien mit Grundkenntnissen des Problemgebiets die Algorithmen verwenden.

Dieser Gedanke wurde auch bei der Entwicklung der Datenstruktur berücksichtigt. So wurden Containertypen implementiert, die die Zusammensetzung der Container kapseln, aber auf die durch die sie verwendenden Datentypen leicht zugegriffen werden kann. Der Datentyp *graph* ist ein weiteres Beispiel der zu erreichenden Einfachheit der Bedienung. Durch ihn lassen sich Graphen leicht und intuitiv verwenden, was die Entwicklung von Graphen-, und Netzwerkalgorithmen erleichtert. Weiterhin wurden Regeln formuliert, um Konzepte wie Zuweisung, Kopieren und Vergleichsfunktionen einheitlich zu gestalten.

Erweiterbarkeit:

Die Algorithmenforschung ist eine lebendige und dynamische Wissenschaft. Es werden immer wieder neue Algorithmen entwickelt, die so noch nicht in LEDA vorkommen, und Alte verbessert. Um das Fortbestehen von LEDA zu garantieren und es zu einer Plattform für Algorithmen in Software zu machen, muss LEDA erweiterbar sein.

Ein Beispiel der Erweiterbarkeit der Bibliothek sind die Komplexen Datentypen und Algorithmen, welche auf den Basisdatentypen beruhen, bzw. auf anderen bereits implementierten Algorithmen.

Validität:

Validität ist ein wichtiges Ziel, da die richtigen Ergebnisse der Zweck dieser Bibliothek sind. Aber dieses Ziel ist nicht leicht zu erreichen, weil erstens die Genauigkeit der Zahltypen in der Informatik nicht immer gegeben ist und zweitens die Eingaben nicht immer von allgemeiner Form sind (besonders für Geometriealgorithmen wichtig).

Um diese Problemstellungen in LEDA zu lösen, wurden eigene Zahlendatentypen implementiert, welche ihren mathematischen Vorbildern sehr viel näher kommen oder sogar dem Konzept genau entsprechen. Ausserdem wurden auch mehrere Geometrie kernels entworfen.

Neben den herkömmlichen Arten, die Validität eines Algorithmus zu gewährleisten (Extensive Dokumentation, Ausführliches Testen) hat LEDA auch das Konzept der sprechenden Ergebnisse übernommen. Hierbei liefern die Algorithmen nicht nur Ergebnisse, sondern rechtfertigen diese auch durch Algorithmen-spezifische, zusätzliche Ausgaben.

Effizienz:

Die Effizienz der Bibliothek wird in erster Linie durch die Verwendung effizienter Algorithmen erreicht. LEDA enthält zu einer Problemstellung durchaus mehr als einen Algorithmus der die Lösung liefert. Hierbei haben alle Algorithmen die gleiche Worst-Case-Zeit, aber die durchschnittliche Zeit zur Problemlösung kann sich mitunter unterscheiden und ist von den Inputs abhängig. Es ist also dem Programmierer überlassen, welche Implementation er letztendlich für die vorgesehene Art von Eingaben als am sinnvollsten erachtet.

LEDA hat einen eigenen Speichermanager, der bei der Definition neuer Klassen auf Anweisung die Erzeugung ihrer *new()* und *delete()* Operatoren übernimmt und diese effizient implementiert.

Auch wenn LEDA auf Effizienz ausgelegt ist, kann man nicht erwarten, dass es die gleiche Laufzeiteffizienz besitzt, wie Tools, die auf ein Problemgebiet spezialisiert sind. LEDA deckt viele Gebiete ab und liefert überall zufriedenstellende Ergebnisse.

Anwendung und Vertrieb

LEDA findet in vielen Gebieten Anwendung, so z.B. in der Codeoptimierung, der Verkehrsplanung, Microprozessordesign oder der Computational Biology.

Die Bibliothek kommt weltweit zur Anwendung und wird an mehr als 1500 akademischen und forschenden Instituten genutzt, aber auch Wirtschaftsunternehmen, wie z.B. Deutsche Telekom AG, Sony Corp. oder die Siemens AG (welche der erste kommerzielle Nutzer LEDAs war) verwenden LEDA für ihre Zwecke.

Die LEDA – Bibliothek wird heute von der Algorithmic Solutions Software GmbH unter drei verschiedenen Lizenzen vertrieben, die sich sowohl im Preis, als auch im Inhalt unterscheiden.

Free Edition

Inhalt:

Diese Lizenz enthält die Datentypen, die Werkzeuge für die GUI, und die Grundbausteine des Systems (z.B. Speichermanager, Fehlerbehandlung).

Preis:

Kostenlos

Anmerkung: Der Originaltext bezüglich des Verbreitens von eigener Software, die mittels der LEDA Free Edition entstand, lautet wie folgt:

„The Licensee may use the Software for software development. The Licensee may distribute programs developed using the Software and may distribute those portions of the Software that are linked into such programs.“ Lizenzvereinbarung, Free Edition

Hieraus ist zu schliessen, dass allein entwickelte und mit Hilfe von LEDA implementierte Algorithmen durchaus vertrieben werden dürfen.

Research Edition

Inhalt:

Diese Lizenz enthält zusätzlich zu den Features aus der Free Edition auch sämtliche Algorithmen und Module, die in LEDA vorhanden sind.

Sie ist für den Gebrauch an Forschungs- oder Lehrinstituten vorgesehen.

Preis:

pro Forschungsgruppe : 1.200 €

pro Standort (Objectcode) : 4.000 €

pro Standort (Sourcecode) : 6.000 €

Professional Edition

Inhalt:

Diese Lizenz enthält zusätzlich zu den Features aus der Free Edition auch sämtliche Algorithmen und Module, die in LEDA vorhanden sind.

Sie ist für kommerzielle Institutionen vorgesehen bzw. zur kommerziellen Nutzung.

Preis:

pro Einzelanwender : 2.500 €

pro Standort (Objectcode) : 7.000 €

pro Standort (Sourcecode) : 16.000 €

Zusätzlich zu den einzelnen Lizenzen kann noch ein Abonnement erworben werden, welches für ein Jahr gilt und sämtliche in dieser Zeit erscheinenden Updates von LEDA beinhaltet.

LEDA – Ein Einblick

Die Bibliothek LEDA bietet viele der gängigen Datentypen der Informatik, wie z.B. Strings, Zahlendatentypen, Listen, Warteschlangen, Vektoren und auch Wörterbücher. Sie enthält noch den Datentyp *graph*, der es erlaubt Graphenalgorithmien leicht und elegant zu implementieren, genauso wie Tools für geometrische Zwecke und Tools für die grafische Darstellung von Lösungen.

In diesem Teil möchte ich einen kleinen Einblick in ein paar ausgewählte Themengebiete geben.

Zahlendatentypen

Zusätzlich zu den built-in Datentypen von C++ umfasst LEDA auch eigene Datentypen, die eine höhere Genauigkeit bei Berechnungen bieten.

Es gibt den Datentypen *integer*, welcher eine genaue Abbildung der Ganzen Zahlen aus der Mathematik ist. Er wird in LEDA als ein Vektor von *longs* realisiert, wobei Länge des Vektors und Vorzeichen getrennt gespeichert werden. Dies hat zur Folge, dass kein Überlauf erzeugt werden kann und die Zahlen tendenziell unendlich groß sein können, wobei sie immer noch durch den Speicherplatz begrenzt sind.

Weiterhin gibt es den Datentypen *rational*, welcher den Quotienten aus zwei Zahlen darstellt. Auch er ist eine genaue Abbildung seines mathematischen Pendanten. Für diesen Typen gibt es mehrere vordefinierte Operationen, um z.B. Zähler und Nenner zu bestimmen oder diese zu normalisieren.

Der Datentyp *bigfloat* aus LEDA erweitert die Fließkommazahlen aus C++. Sie bestehen auch aus einer Mantisse und werden mit einem Exponenten multipliziert, diese beiden Zahlen sind aber *integer* im Sinne von LEDA. Hierbei kann die Länge der Mantisse (→ Genauigkeit) vom Benutzer festgelegt werden. Zusätzlich gibt es spezielle Werte für ± 0 , $\pm\infty$ und NaN (Not a Number), da z.B. für manche Netzwerkalgorithmen Unendlichkeiten sinnvoll zu verwenden sind und sie durch die Werte hier genauer definiert sind als die herkömmlichen *MAXINT* und *MININT*.

LEDAs beste Näherung an die reellen Zahlen ist der Datentyp *real*. Er unterstützt das k-fache Wurzelziehen aus einer Zahl, wobei k eine beliebige ganze Zahl sein kann. Besonderer Wert wird hierbei auf die Aussage gelegt, dass *reals* die genaue Bestimmung eines Vorzeichens im mathematischen Sinne zulassen und nicht nur das Vorzeichen einer Näherung des exakten Wertes. Dies erlaubt es, Konditionen in Programmen genau zu formulieren, ohne einen gewissen Fehler erwarten zu müssen.

Die Vorzeichen werden Anhand des Konzepts der *separation bound* bestimmt. Hierbei ist die *separation bound* in LEDA voreingestellt und wird wie folgt verwendet:

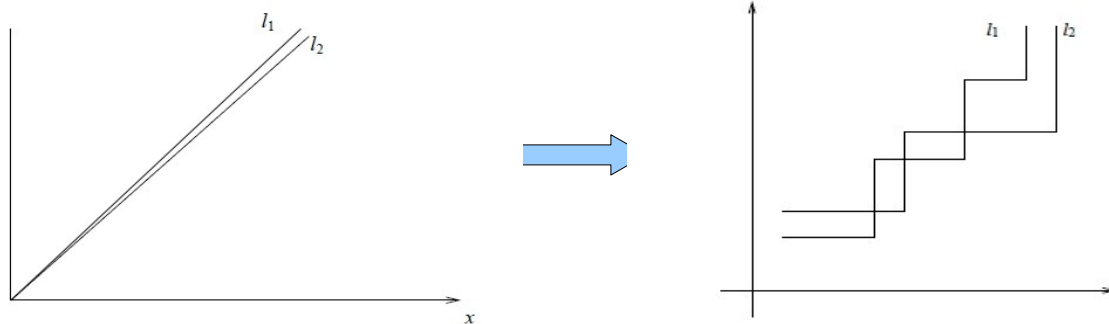
$$val(E) \neq 0 \text{ impliziert } |val(E)| \geq sep(E)$$

wobei *E* ein Ausdruck, *val(E)* sein Wert und *sep(E)* die *separation bound* ist.

Nun wird eine Annäherung *A* von *val(E)* berechnet mit $|A - val(E)| < sep(E)/2$. Die benötigte Mantissenlänge wird in einer Fehlerberechnung ermittelt, welche automatisiert im Datentypen abläuft und für den Benutzer unsichtbar ist.

Wenn $|A| \geq sep(E)/2$ dann ist das Vorzeichen des Ausdrucks *E* das Vorzeichen der Annäherung *A* und wenn nicht, dann ist es eine 0.

Hierdurch ist es z.B. möglich, für zwei Geraden, deren Steigung sich nur geringfügig unterscheidet, an jeder beliebigen Stelle genau zu bestimmen, welche Gerade steiler ist. Dies ist ansonsten in der Informatik meist nicht möglich, da die Datentypen es nur erlauben, die Geraden als Treppenfunktionen darzustellen.



Itemtypen

In LEDA werden für viele Datenstrukturen Itemtypen verwendet, welche auf den Inhalt des entsprechenden Elternobjekts verweisen. Im Allgemeinen werden sie durch Zeiger auf Containerobjekte realisiert, sind jedoch nicht in jedem Fall nur ein Zeiger.

Es gibt zwei verschiedene Arten von Itemtypen in LEDA: die Abhängigen, welche Items für Wörterbücher oder Warteschlangen beinhalten, und die Unabhängigen, zu welchen z.B. Punkte und Linien gehören.

Abhängige Typen können ohne die Collection, zu der sie gehören, nicht existieren bzw. würden keinen Sinn machen. Dies spiegelt sich auch in der Art ihres Zugriffs wider; auf sie wird über das Elternobjekt zugegriffen (*Dictionary.inf(item)*) während unabhängige Typen selbst den Zugriff auf die beinhalteten Werte liefern (*point.xcoord()*).

Wie erwähnt, sind die Itemtypen durch Pointer realisiert, doch auch hier unterscheiden sich die abhängigen von den unabhängigen Typen. Abhängige Typen werden direkt als C++ - Pointer implementiert, wohingegen die unabhängigen Typen als Klassen realisiert sind, deren einziges Feld ein Pointer auf den hiermit verbundenen Container ist. Diese Unterscheidung wurde wegen der Speicherverwaltung getroffen. Während abhängige Itemtypen gelöscht und freigegeben werden, wenn das dazugehörige übergeordnete Objekt gelöscht wird, können unabhängige Itemtypen durchaus von mehreren Objekten referenziert werden und dürfen somit nicht sofort gelöscht werden. Um festzustellen, wann keine Referenz mehr auf das Objekt zeigt, ist ein Referenzzähler in den Containerobjekten integriert. Wenn dieser Referenzzähler Null erreicht, kann das Containerobjekt freigegeben werden. Um diese Vorgänge für den Benutzer transparent zu machen, wird der Pointer auf den Container in einer Klasse gekapselt und die Pointeroperationen *Zuweisung* und *Zugriff* werden neu definiert. Somit ist der unabhängige Itemtyp genau genommen durch einen *Handletypen* realisiert.

Dadurch, dass unabhängige Itemtypen von mehreren Objekten referenziert werden können, darf sich ihr Inhalt im Nachhinein nicht mehr ändern, um so Inkonsistenzen zu vermeiden. Es werden von den Datentypen Funktionen zur Änderung angeboten, diese sind aber so implementiert, dass sie eine neues Objekt mit den veränderten Werten erzeugen.

In LEDA wurde der Einsatz von Itemtypen aus zwei Gründen gewählt. Einerseits erhöhen Itemtypen die Effizienz.

Dazu ein Beispiel:

Wenn man in einem Wörterbuch einen Datensatz anhand eines Schlüssels heraussuchen möchte, durchsucht man zuerst das Wörterbuch und liefert dann die Informationskomponente zurück. Diese möchte man nun verändern und den geänderten Datensatz wieder zurückschreiben. Wenn man ihn nun unter dem gleichen Schlüssel speichern möchte, wird ein erneutes Suchen in der Datenstruktur nötig. Hätte man sich aber beim ersten Mal schon die Position des Datums gemerkt, könnte man sich das zweite Suchen ersparen. Genau diese Funktionalität ist mit den Itemtypen realisiert.

Zusätzlich bieten die Itemtypen in LEDA Sicherheit beim Zugriff auf die Objekte, da nur lesend auf sie zugegriffen wird und so keine Verwirrung mit den Zeigern entstehen kann.

Initialisierung und Kopieren

In LEDA werden alle Datentypen initialisiert. Das heisst aber nicht, dass alle von Anfang an definiert belegt sind. Die eingebauten Typen von C++, wie z.B. *char*, *int* oder *float* werden mit einem undefinierten Wert belegt. Deshalb sollte der Benutzer hier eine sinngemäße Initialisierung selbst vornehmen. Die LEDA – Typen haben alle Defaultinitialisierungen, die im Allgemeinen den einfachsten Wert (leerer String, leere Liste, etc.) widerspiegeln.

Ausser der Initialisierung mit einem Defaultwert ist für jeden LEDA - Typen auch eine Initialisierung per Kopie möglich. Hierbei wird eine Kopie des im Konstruktor angegebenen Arguments erstellt und das neue Objekt damit initialisiert.

Zusätzlich zu diesen beiden Formen der Initialisierung haben bestimmte Datentypen noch zusätzliche Konstruktoren. So kann ein Array z.B. mit einem bestimmten Indexbereich erstellt oder ein Stack mit einer maximalen Kapazität angegeben werden.

LEDA bietet für einen Großteil seiner Datentypen Typparameter an und diese können sich auf die Initialisierung des Objekts auswirken. Ein Beispiel hierfür stellt das Array dar: Wenn es mit Typparameter und festen Indizes erzeugt wird, wird es nicht als leeres Array initialisiert, sondern als Array mit einer definierten Anzahl von Objekten in den festgelegten Indexgrenzen und die beinhalteten Objekte werden per Defaultinitialisierung des Typparameters erzeugt.

Hierbei ist noch zu erwähnen, dass bei parametrisierten Datentypen Werte, die in ein Maschinenwort passen, direkt in der Datenstruktur gespeichert werden und alles was mehr Platz braucht per Zeiger referenziert wird.

Da es in LEDA das Initialisieren per Kopie gibt, möchte ich an dieser Stelle nochmal den Vorgang des Kopierens beleuchten.

LEDA definiert ein paar strikte Regeln, die das Verständnis der Klassen und ihre Handhabung erleichtern sollen. So ist z.B. der Zuweisungsoperator für jede Klasse der Bibliothek auf eine gleichförmige Art und Weise definiert:

Eine Zuweisung $x = A$ weist der Variablen x eine Kopie des Wertes des Ausdrucks A zu.

In diesem Zusammenhang sollte man verstehen, wie die einzelnen Arten von Werten kopiert werden. Hierbei wird unterschieden zwischen primitiven Typen, Nicht-Item-basierten Typen und Item-basierten Typen.

Primitive Typen sind die bereits vorhandenen Typen von C++, Zeigertypen und Itemtypen. Wie von den eingebauten Typen bekannt, ist auch bei den anderen Typen eine Kopie des Wertes der Wert selbst.

Nicht-Item-basierte Typen sind z.B. Arrays oder Mengen. Wenn ein solcher Typ kopiert wird, wird ein neues Objekt mit der gleichen Anordnung (Indizes im Fall eines Arrays) erstellt, das neue

Objekte enthält. Diese Objekte enthalten Kopien der Werte, des zu kopierenden Objekts. Es wird also komponentenweise kopiert.

Für Item-basierte Typen, wie Wörterbücher oder Warteschlangen, werden für jedes Item, das im Original vorliegt ein neues Item in der Kopie erstellt. Diese Items werden mit den Kopien der Werte des Originalitems befüllt. Hierbei wird darauf geachtet, dass die neue kombinatorische Struktur, nach der die Items angeordnet sind, zu der Alten isomorph ist.

In der Informatik wird oft über tiefes oder flaches Kopieren gesprochen. In LEDA wird nicht eine dieser beiden Arten des Kopierens genutzt sondern beide in Kombination. Dies folgt aus dem semantischen Verständnis der Schöpfer LEDAs für das Kopieren.

Hier ein Beispiel:

```
class list
{ list * next
  E * info
}
```

Bei solch einer Struktur für eine einfach verbundenen Liste würde bei einer Kopie im LEDA-Kontext jeder Referenz *next* nachgegangen werden. Für jedes enthaltene Objekt würde in der Kopie ein neues Objekt erstellt werden und die Struktur der alten Liste würde auch auf der Neuen erhoben werden, was alles dem Konzept des tiefen Kopierens entspricht. Nun wird die Informationskomponente kopiert. Da diese aber nur durch einen Zeiger dargestellt wird und Kopien von Zeigern in LEDA direkt der Zeiger selbst sind, würde der Wert des Zeiger einfach in der neuen Liste gespeichert werden, was wiederum dem flachen Kopieren entspricht.

Speicherverwaltung

Objekte werden in LEDA üblicherweise genau dann erzeugt und initialisiert, wenn der Programmfluss sie erreicht und wieder zerstört, wenn das Programm aus dem Block springt, in dem sie definiert sind. Ausnahmen hiervon bilden die statischen und anonymen Objekte. Anonyme Objekte sind Objekte, die direkt per `new()` - Befehl erzeugt werden. Diese und auch nur diese können von dem `delete()` - Operator wieder zerstört werden. Hierbei wird der allokierte Speicher wieder freigegeben.

Da in LEDA die Datenstrukturen oft aus kleineren Einheiten bestehen, z.B. den Itemtypen, ist in der Bibliothek extra für diese kleinen Bausteine, nämlich Itemtypen, *nodes* und *edges*, ein spezieller Speichermanager implementiert worden.

Dieser kann auch vom Benutzer für selbst entworfene Klassen genutzt werden und muss dafür nur durch den Macro – Aufruf `LEDA_MEMORY(class)` eingebunden werden. Das Macro legt dann für den Benutzer `new()` und `delete()` Funktionen an, die der Richtlinie des LEDA – Speichermanagers folgen.

Ein bedeutender Vorteil des LEDA – Speichermanagers ist, dass der Speicher selten, dafür aber in großen Blöcken vom System angefordert wird. Hierdurch werden die Aufrufe der Allokationsfunktion `malloc()` reduziert und die Laufzeiteffizienz des Programms gefördert. Zugunsten der Effizienz wird hierdurch natürlich die Speicherplatznutzung erhöht. Dieser Speicherplatz kann jedoch im Programm selbst verwaltet werden und somit auch intern wieder verwendet werden.

Hier ein Beispiel der Laufzeiten in Sekunden für das Erstellen und anschließende Löschen von n Objekten mit und ohne den LEDA – Speichermanager.

<i>n</i>	LEDA memory	C++ memory
1000000	0.94	2.77

Wenn in einem LEDA – Programm ein Objekt erzeugt und Speicher angefordert wird, wird ein großer Block, meistens in der Größe von 4 bis 8 KByte, allokiert. Dieser wird dann in mehrere kleinere Teile von der Größe des angeforderten Objekts aufgeteilt. Diese einzelnen Teile des Blocks werden in einer einfach verbundenen Liste verwaltet. Für die LEDA – Datentypen ist für die Einzelteile eine Obergrenze von 256 Byte angelegt. Ab dieser Grenze werden Speichernachfragen nicht mehr über den Speichermanager verwaltet, sondern direkt an *malloc()* weitergegeben und direkt vom Betriebssystem angefordert. Diese Funktionalität ist deshalb so realisiert, weil sich das Aufteilen eines Großen Blocks nur dann lohnt, wenn die Relation von Blockgröße zu Blockteilen relativ groß ist. Die Obergrenze bezieht sich nur auf den Standard – Speichermanager, der in den Macroaufrufen von LEDA verwendet wird. Es können aber auch eigene Speichermanager erstellt werden, denen im Konstruktor eine selbst definierte obere Grenze mitgeteilt wird.

Im allgemeinen Fall sieht der Ablauf der Speicherallokation in LEDA so aus:

Ein Objekt wird im Programm erzeugt und braucht Speicherplatz. Nun wird in einer Tabelle, die die jeweils ersten Elemente der einzelnen Listen für die zugehörige Größe des Speicherteils verwaltet, nachgesehen (z.B. 128 Byte → *free_list[127]*). Ist diese Liste vorhanden, wird der erste Teil zurückgeliefert und kann verwendet werden. Ist die Liste noch leer, wird wie oben beschrieben ein größerer Block allokiert und in kleinere Einheiten unterteilt und dann wird das erste Element zurückgeliefert.

Wenn ein Speicherteil im Programm nicht mehr genutzt und per *delete()* freigegeben wird, wird der nun frei gewordene Speicherabschnitt an den Anfang der dazugehörige Liste angefügt und kann bei Bedarf wiederverwendet werden.

Im Buch ist von garbage collection die Rede. Hierbei denkt man an den Garbagecollector in Java, welcher aber mehr Funktionalität besitzt. Der Speichermanager regelt, wie für C++ - Programme üblich, den vom Programm verwendeten Speicher, aber nicht die Rückgabe an das Betriebssystem. Um aber Speicherknappheiten zu vermeiden bietet LEDA die Funktion *leda::std_memory_mgr.clear()* welche bei Aufruf den ungenutzten Speicher aus dem programminternen Speichermanagement wieder an das System abtritt.

Anwendungsbeispiel: Planarität

Das Problem der Planarität besteht darin festzustellen, ob ein Graph kreuzungsfrei in die Ebene gezeichnet (eingebettet) werden kann. In LEDA sind zwei verschiedene Arten des Planaritätstests implementiert, wobei der Algorithmus von Lempel / Even / Cederbaum und Booth / Lueker der Schnellere und häufiger Eingesetzte ist.

Anwendung findet der Test auf Planarität z.B. bei der Zerlegung eines Graphen in offene Ohren. Diese Ohren lassen sich aus einem planaren Graphen leicht bestimmen und dienen dazu festzustellen, ob ein Graph dreifach zusammenhängend ist. Diese Eigenschaft des „dreifach-zusammenhängens“ kann als Marke der Robustheit von Strukturen, die als Graphen dargestellt werden können, gesehen werden. So können z.B. Netzwerke bewertet werden.

Zusätzlich können die Knoten in einem planaren Graphen nach dem Vier-Farben-Satz immer mit 4 Farben graphentheoretisch korrekt gefärbt werden, was in der Kartographie zur Anwendung kommt.

Literaturverzeichnis:

- Kurt Mehlhorn, Stefan Näher, *LEDA Buch*, Cambridge University Press, 1999
- Markus Jochim, *Parallele Zerlegung von Graphen in Dreifach-Zusammenhangskomponenten*, Diplomarbeit am Fachbereich Informatik der Universität Dortmund, 1996
- Ingo Wegener, *Highlights aus der Informatik*, Springer, 1996
- Prof. Dr. Sebastian Iwanowski, Prof. Dr. Rainer Lang, *Neues Vorlesungsskript für die Vorlesung Diskrete Mathematik*, gehalten an der FH Wedel, 2010
- Website der Algorithmic Solutions GmbH, <http://www.algorithmic-solutions.com/>