LEDA
Eine Plattform für kombinatorisches und geometrisches Rechnen.

Stefan Näher *

1 Einführung

Kombinatorisches und geometrisches Rechnen stellt eines der zentralen Gebiete der Informatik dar. Aus diesem Grund enthalten auch die meisten Universitätslehrpläne Kurse über Datenstrukturen und effiziente Algorithmen. Typische Objekte dieses Gebietes sind Graphen, Folgen, Wörterbücher, Bäume, kürzeste Wege, Flüsse, Abbildungen, Punkte, Segmente, Linien, konvexe Hüllen und Voronio-Diagramme; es bildet die Grundlage für solche Anwendungungen wie diskrete Optimierung, Planung, Verkehrskontrolle, CAD, Grafik und vieles mehr. Dennoch gibt es keine Standard-Bibliothek für Datenstrukturen und Algorithmen. Dies steht in klarem Gegensatz zu anderen Disziplinen wie beispielsweise Statistik (SPSS), numerische Analyse (LINPACK, EISPACK), symbolisches Rechnen (MAPLE, MATHEMATICA) oder lineares Programmieren (CPLEX).

Das Fehlen einer Bibliothek beeinträchtigt den Nutzen des Gebietes für die gesamte Informatik ganz erheblich. Das sich ständig wiederholende Neuimplementieren von grundlegenden Datenstrukturen und Algorithmen behindert den Fortschritt nicht nur innerhalb der Forschung, sondern in sogar noch stärkerem Maße auch außerhalb der Universitäten. Letzteres vor allem deshalb, weil außerhalb der Forschung Zweifel an der Wiederverwendbarkeit von Software dazu führen, daß der Aufwand zum Implementieren einer effizienten Lösung nicht investiert wird, so daß man schließlich zu weniger effizienten oder sogar trivialen Lösungen greift. Dies hat zur Folge, daß neue Erkenntnisse über Algorithmen und Datenstrukturen nur sehr langsam Eingang in die Praxis finden.

Doch worauf ist nun das Fehlen einer Bibliothek von effizienten Algorithmen und Datenstrukturen zurückzuführen? Einer der größten Unterschiede zwischen kombinatorischem oder geometrischem Rechnen und anderen Gebieten wie Statistik, numerischer Analyse und linearem Programmieren ist der Gebrauch komplexer Datentypen. Während die in

*email: naeher@informatik.uni-halle.de

Programmiersprachen eingebauten Typen wie ganze oder reell Zahlen, Vektoren und Matrizen normalerweise für andere Gebiete ausreichen, beruht kombinatorisches und geometrisches Rechnen vor allem auf Typen wie Keller, Schlangen, Listen, Wörterbüchern, Folgen, Graphen, Punkten, Linien und konvexe Hüllen. Es erfordert deshalb eine Programmierumgebung, die all diese Typen zur Verfügung stellt. Mit der Einführung der objektorientierten Programmierung wurde solch eine Erweiterung in effizienter und eleganter Weise möglich.

Im Jahre 1989 haben wir das LEDA Projekt (Library of Efficient Data Types and Algorithms) ins Leben gerufen, um eine Bibliothek der Datentypen und Algorithmen für kombinatorisches und geometrisches Rechnen aufzubauen. Die wesentlichen Eigenschaften von LEDA sind:

- LEDA stellt eine beträchtliche Anzahl von Datentypen und Algorithmen in einer Form zur Verfügung, die es auch Nicht-Experten erlaubt, sie zu benutzen. Diese Sammlung beinhaltet die meisten der Datentypen und Algorithmen, die in Textbüchern dieses Gebietes beschrieben sind: Keller, Schlangen, Listen, Mengen, Wörterbücher, gerichtete, ungerichtete und planare Graphen, Linien, Punkte und Ebenen; dazu kommen viele Algorithmen der Graphen- und Netzwerktheorie sowie der algorithmischen Geometrie.

- LEDA gibt eine präzise und lesbare Spezifikation für jeden der erwähnten Datentypen und Algorithmen. Die Spezifikationen sind kurz (in der Regel nicht länger als eine Seite), allgemein (sie erlauben verschiedene Implementierungen) und abstrakt (Implementierungsdetails sind versteckt).

- Viele Datentypen in LEDA sind parametrisiert; beispielsweise funktioniert der Wörterbuchtyp für beliebige Schlüssel- und Informationstypen. Ein spezielles Objekt D mit Schlüsseltyp **string** und Informationstyp **int** kann z.B. durch **dictionary<string, int>** D definiert werden.

- LEDA enthält die effizientesten Implementierungen, die für seine Datentypen bekannt sind. Für viele der Typen kann der Benutzer zwischen verschiedenen Implementierungen wählen, beispielsweise ab-Bäumen, $BB[a]$-Bäumen, dynamisches perfektes Hashing oder Skiplisten für Wörterbücher. So realisiert beispielsweise die Definition **_dictionary <string,int,skip_list>** D ein Wörterbuch durch Skiplisten.

- Für viele effizienten Datenstrukturen ist der Zugriff über eine Position wichtig. LEDA benutzt ein neues Item-Konzept um Positionen abstrakt behandeln zu können.

- LEDA enthält einen komfortablen Typ graph. Er bietet Standarditerationen wie 'für alle Knoten v eines Graphs G tue' (= '**forall_nodes**(v, G)') oder 'für alle Nachbarn w von v tue' (='**forall_adj_nodes**(w, v)'); Knoten und Kanten können beliebig hinzugefügt oder gelöscht werden. Auch werden Felder und Matrizen bereitgestellt, die mit den Knoten oder Kanten indiziert werden können. Mithilfe

des Datentyps graph können die Programme zum Lösen von Graphenproblemen fast genauso geschrieben werden, wie sie normalerweise in Textbüchern zu finden sind. Wir betonen, daß alle hier abgedruckten Programme übersetz- und ausführbar sind. Das Ziel ist die Gleichung 'Algorithmus + LEDA = Programm'.

- LEDA ist in C++ geschrieben und alle Datentypen und Algorithmen sind in der Bibliothek als vorübersetzte Objekt-Module gespeichert. Dies führt zusammen mit der Tatsache, daß aufgrund der großen Ausdruckskraft von LEDA Anwendungsprogramme kurz sind, zu kurzen Übersetzungszeiten.
- Viele geometrische Algorithmen benutzen beliebig genaue Arithmetik und sind deshalb frei von Rundungsfehlern. Darüber hinaus kommen sie mit alle degenerierten (= entarteten) Fällen klar.
- LEDA unterstützt eine Vielfalt von Anwendungsbereichen. So wurde es schon in so unterschiedlichen Gebieten wie Code-Optimierung, VLSI-Design, Roboter-Bewegungsplanung, Verkehrsplanung, Maschinenlernen und algorithmischer Biologie benutzt.

In den folgenden Abschnitten illustrieren wir zunächst LEDA anhand von 4 Beispielen und diskutieren dann theoretische Ergebnisse, die die Qualität unseres "Produkts" wesentlich erhöht haben. Die vier Beispiele demonstrieren verschiedene Teile der Bibliothek: grundlegende Datenstrukturen, Graphen, Geometrie und Grafik. Die Beispiele zeigen vor allem auch, wie leicht man mithilfe von LEDA von einem Algorithmus zu einem lauffähigen Programm gelangen kann. Sie verdeutlichen, daß LEDA nicht einfach eine Bibliothek von Datentypen und Algorithmen ist, sondern eher eine Plattform darstellt, auf der Anwendungen aufsetzen können.

1.1 Worte zählen

Wir möchten eine Folge von Worten (Zeichenketten, Strings) von der Standardeingabe lesen, die Anzahl der Vorkommen jedes einzelnen Strings zählen und schließlich eine Liste aller Strings zusammen mit ihrer Häufigkeit ausgeben. Die hierfür geeigneten LEDA Typen sind *string* und *dictionary array*. Der parametrisierte Typ dictionary array (**d_array<I,E>**) realisiert Felder mit Indextyp **I** und dem Elementtyp **E**. Hier verwenden wir ein Array mit Indextyp **string** und Elementtyp **int**. Das vollständige Programm ist in Abbildung 1 zu sehen. Es beginnt mit einer include Anweisung für dictionary arrays. In der ersten Zeile des Hauptprogramms definieren wir ein dictionary array N mit Indextyp **string** und Elementtyp **int** und initialisieren alle Einträge des Array auf 0. (Die Implementierung von **d_array** speichert alle Einträge, die von Null verschieden, in einem balancierten Suchbaum mit Schlüsseltyp string). In der zweiten Zeile definieren wir einen String s. Die dritte Zeile verrichtet die ganze Arbeit. Der Ausdruck $(cin \gg s)$ gibt true zurück, wenn der Eingabestrom nicht leer ist, andernfalls gibt er false zurück. Im ersteren Fall wird der erste String vom Eingabestrom entfernt und der Variablen s zugewiesen. Dann wird der Eintrag $N[s]$ des arrays N inkrementiert.

Die Iteration **forall_defined**(s, N) in der letzten Zeile weist nacheinander alle Strings, für die auf N während der Ausführung des Programms zugegriffen wurde, der Variablen s zu. All diese Strings werden zusammen mit der Häufigkeit ihres Auftretens auf die Standardausgabe ausgegeben.

```
#include<LEDA/d_array.h>
main()
{ d_array<string,int> N(0);
  string s;
  while (cin >> s) N[s]++;
  forall_defined (s,N)
    cout << s << " " << N[s] << endl;
 }
```

Abbildung 1: Ein Programm, das die Anzahl der Vorkommen jedes Strings in einer Folge von Strings zählt.

1.2 Kürzeste Wege in Graphen

Ein gerichteter Graph besteht aus einer Menge V von Knoten und einer Menge E von Kanten. Eine Kante $e = (v, w)$ ist eine gerichtete Verbindung von ihrem Startknoten v zu ihrem Endknoten w. Nehmen wir nun an, daß für jede Kante e eine Reisezeit $cost[e]$ in Minuten gegeben ist und daß es unsere Aufgabe ist, die minimale Reisezeit von einem bestimmten Knoten s zu jedem beliebigen anderen Knoten zu berechnen (Abbildung 2 zeigt ein Beispiel).

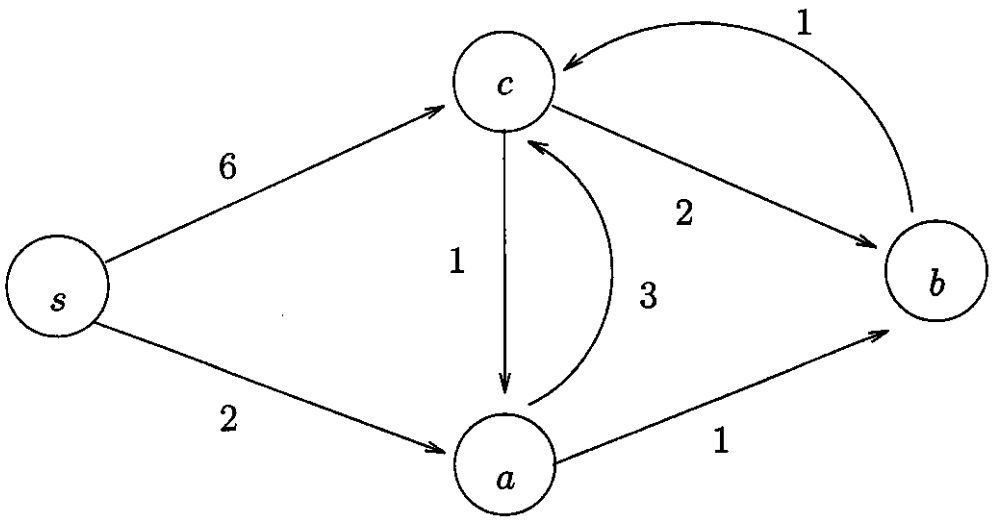

Abbildung 2: Ein Graph mit Kantenbeschriftungen. Diese geben die Reisezeit (in Minuten) über die jeweilige Kante an. Die kürzeste Reisezeit von Knoten s zu Knoten c beträgt 4 Minuten.

Dijkstra [Dij59] hat einen einfachen Algorithmus gefunden, der dieses Problem löst. Zu jedem Zeitpunkt während der Ausführung des Algorithmus gibt es eine Menge $S \in V$ von Knoten, für die die minimale Reisezeit bereits bekannt ist, während für alle anderen Knoten lediglich eine obere Schranke für die Reisezeit bekannt ist, d.h., eine Zeitspanne, in der man die Strecke auf jeden Fall bewältigen kann, die jedoch nicht notwendigerweise minimal ist.

Für jeden Knoten v bezeichnen wir mit $dist[v]$ die kürzeste bisher bekannte Reisezeit von s nach v. Anfangs ist $dist[s] = 0$, $dist[v] = \infty$ für jeden Knoten $v \neq s$, und $S = \emptyset$. In jedem Schritt wählen wir einen Knoten $u \in V - S$ mit dem kleinsten Wert $dist[u]$ (am Anfang ist $u = s$) und fügen ihn zu S. Für jede von u ausgehende Kante $e = (u, w)$ setzen wir $dist[v]$ auf den Wert $dist[u]+cost[e]$, wenn dieser Wert kleiner ist als der aktuelle Wert von $dist[v]$. In unserem Beispiel ist s der erste Knoten, der zur Menge S kommt. Wenn dies geschieht, wird $dist[a]$ auf 2 gesetzt und $dist[c]$ auf 6. Dann fügen wir a zu S, $dist[b]$ wird auf 3 gesetzt und $dist[c]$ wird auf 5 erniedrigt, ... Die Korrektheit dieses Algorithmus soll hier nicht bewiesen werden (Beweisidee: man zeigt, daß für alle Knoten v der Wert von $dist[v]$ immer die kürzeste Reisezeit von s nach v entlang eines Pfades angibt, dessen Knoten alle mit Ausnahme des Endknotens zu S gehören). Wie können wir nun den Knoten in $V - S$ mit dem kleinsten *dist*-Wert schnell finden? Eine geeignete Datenstruktur ist eine *Warteschlange*. Abbildung 3 zeigt die LEDA - Implementierung von Dijkstras Algorithmus. Neben Graphen benutzt er **node_array** (Knotenfeld), **edge_array** (Kantenfeld) und **node_pq** (Knoten-Warteschlange). Knoten- und Kantenfelder sind Felder, die mit Knoten bzw. Kanten indiziert sind. Wir benutzen ein **edge_array<int>** *cost*, um die Reisezeiten entlang der Kanten zu speichern, ein **node_array<int>** *dist* um die minimale Reisezeit zu jedem Knoten zu speichern und eine **node_pq<int>** $PQ(G)$. An dem Parameter G in der Definition von PQ erkennt LEDA, daß G der zugrundeliegende Graph ist. Die Knoten-Warteschlange PQ enthält stest alle Knoten von G, die in $V - S$ liegen, zusammen mit ihren aktuellen *dist*-Werten. Am Anfang werden alle Knoten in PQ eingefügt. Bei jeder Iteration wird der Knoten mit dem kleinsten zugehörigen Wert aus PQ gelöscht ($u = PQ.del\_min()$) und alle von u ausgehenden Kanten e werden überprüft (**forall_adj_edges**(e, u)). Falls notwendig wird dabei der *dist*-Wert des Zielknotens verringert ($PQ.decrease\_inf(v, c)$).

Wir möchten besonderes auf die große Ähnlichkeit zwischen dem LEDA Programm und der Beschreibung des Algorithmus hinweisen. Die gleiche Ähnlichkeit ist bei den meisten Graphenalgorithmen anzutreffen. In diesem Sinne erfüllt LEDA die Gleichung "**Algorithmus + LEDA = Programm**".

Erwähnenswert ist sicher auch, daß ein Programmierer von Dijkstras Algorithmus nichts über die innere Funktionsweise von Graphen und Knoten-Warteschlangen zu wissen braucht. Die Kenntnis der relevanten Seiten im Handbuch genügt vollkommen. Abbildung 4 zeigt die Handbuchseite für Knoten-Warteschlangen. Im letzten Abschnitt gibt diese Seite die Laufzeiten der verschiedenen Operationen für Warteschlangen an: konstante Zeit für *insert*, *empty* und *decrease_inf*, und logarithmische Zeit für *del_min*. In einem Graph mit n Knoten und m Kanten benötigt Dijkstras Algorithmus n *inserts*, *empty*-Tests und *del_mins* und höchstens m *decrease_infs*. Seine Laufzeit ist deshalb proportional zu $m + n \log n$.

```
#include <LEDA/graph.h>
#include <LEDA/node_pq.h>

void DIJKSTRA(const graph& G, node s, const edge_array<int>& cost,
node_array<int>& dist)
{ node_pq<int>  PQ(G);
  node v;
  forall_nodes(v,G)
  { dist[v] = (v == s) ? 0 :MAXINT;
    PQ.insert(v,dist[v]);
   }
  while (! PQ.empty())
  { node u = PQ.del_min();
    edge e;
    forall_adj_edges(e,u)
    { v = target(e);
      int c = dist[u] + cost[e];
      if (c < dist[v])
      { PQ.decrease_p(v,c);
        dist[v] = c;
       }
     }
   }
}
```

Abbildung 3: Dijkstras Algorithmus für das Problem der Berechnung aller kürzesten Wege von einem Knoten aus.

1.3 Konvexe Hüllen

Stellen Sie sich eine endliche Menge L von Punkten in der Ebene vor, die von einem Gummiband umschlossen ist. Das Gummiband zeigt den Verlauf der sogenannten konvexen Hülle von L. Abbildung 5 zeigt ein Beispiel. Die konvexe Hülle ist eine der grundlegenden Strukturen in der algorithmischen Geometrie. Sie wird bei der Bildverarbeitung oder Mustererkennung oft als eine Annäherung an die Form einer Punktmenge benutzt. Wir erklären im folgenden, wie wir die konvexe Hülle berechnen, wenn die Punktmenge gegeben ist: wir beschränken uns der Einfachheit halber auf die sogenannte obere Hülle. Wenn wir die konvexe Hülle nämlich durch eine Gerade durch den Punkt der am weitesten links und den Punkt der am weitesten rechts liegt, durchschneiden, zerteilen wir sie in die obere und die untere Hülle von L (siehe Abbildung 5). Dabei ist ein Punkt p links von einem Punkt q, wenn entweder die x-Koordinate von p kleiner ist als die von q ist oder wenn die x-Koordinaten der beiden Punkte gleich sind und die y-Koordinate von p ist kleiner ist als die von q.

Node priority queues (node_pq)

1. Definition

An instance Q of the parametrized data type **node_pq⟨I⟩** is a set of pairs (v, i), where v is a node of some graph G and i belongs to some linearly ordered type **I**; i is called the information associated with node v. For any node v of G there can be at most one pair $(v, \ )$ in Q.

2. Creation

node_pq⟨I⟩ $Q(G)$;

creates an empty instance Q of type **node_pq⟨I⟩** for the nodes of graph G.

3. Operations

void	*Q.insert*(**node** v, **I** i)	adds the node v with information i to Q. *Precondition:* There is no pair $(v, \ )$ in Q.
I	*Q.inf*(**node** v)	returns the information of node v.
bool	*Q.member*(**node** v)	returns true if (v, i) in Q for some i, false otherwise.
void	*Q.decrease_inf*(**node** v, **I** i)	makes i the new information of node v. (*Precondition:* $i \leq Q.inf(v)$).
node	*Q.find_min*()	returns a node with the minimal information(**nil** if Q is empty).
void	*Q.del*(**node** v)	removes the pair $(v, \ )$ from Q.
node	*Q.del_min*()	removes a node with minimal information from Q and returns it (**nil** if Q is empty).
int	*Q.size*()	returns the number of pairs in Q.
void	*Q.clear*()	makes Q the empty node priority queue.
bool	*Q.empty*()	returns true if Q is the empty node priority queue, false otherwise.

4. Implementation

Node priority queues are implemented by fibonacci heaps and node arrays. Operations *insert*, *del_node*, *del_min* take time $O(\log n)$, *find_min*, *decrease_inf*, *empty* take time $O(1)$ and *clear* takes time $O(m)$, where m is the size of Q. The space requirement is $O(n)$, where n is the number of nodes of G.

Abbildung 4: Die Handbuchseite für Knoten-Warteschlangen.

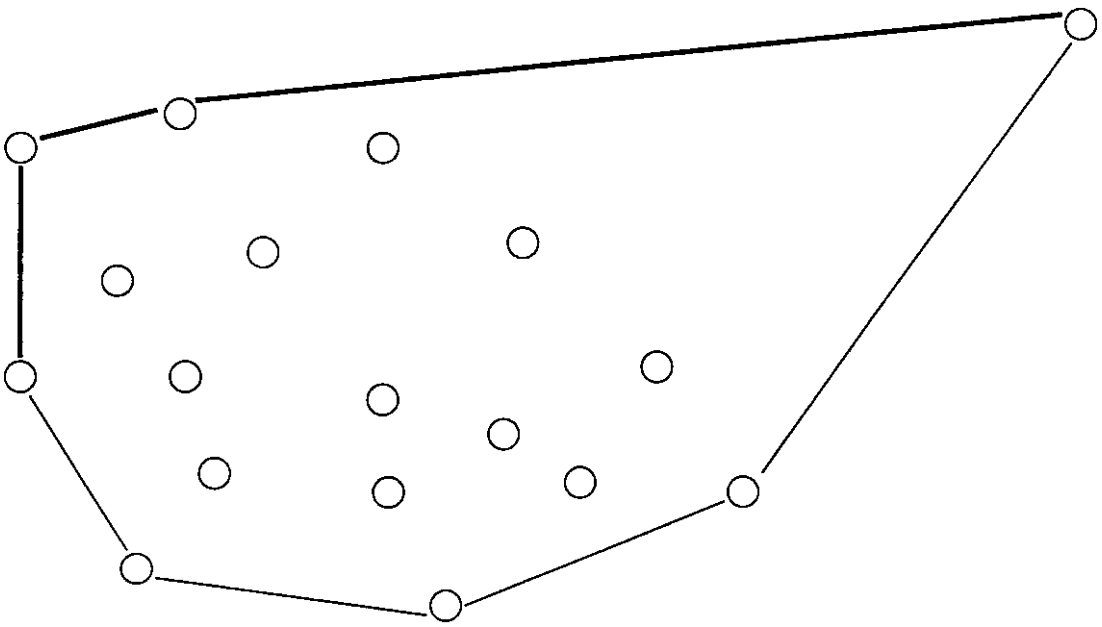

Abbildung 5: Die konvexe Hülle einer Punktemenge in der Ebene. Die obere Hülle ist dick gezeichnet.

Wir berechnen die obere Hülle einer Punktmenge L wie folgt: Zuerst sortieren wir die Menge L gemäß der oben definierten Links-Rechts Reihenfolge ihrer Punkte. Sei $p_1, p_2, \ldots, p_n$ die gemäß dieser Reihenfolge aufsteigend sortierte Folge. Wir konstruieren die obere Hülle der Punkte $p_1, \ldots, p_i$ inkrementell für alle i, $1 \leq i \leq n$. Die Initialisierung ist einfach. Die obere Hülle von p_1 ist p_1. Setzen wir nun voraus, daß die obere Hülle der Punkte $p_1, \ldots, p_i$ bereits berechnet ist und daß wir den Punkt p_{i+1} abarbeiten. Wenn $p_i = p_{i+1}$, dann ist nichts zu tun. Wenn aber $p_i \neq p_{i+1}$, dann löschen wir den jeweils letzten Punkt der aktuellen oberen Hülle, solange diese mindestens aus zwei Punkten besteht und mit dem neuen Punkt p_{i+1} keine Rechtskurve bildet (siehe Abb. 6). Dann addieren wir p_{i+1} zur oberen Hülle; damit ist der Iterationsschritt abgeschlossen.

Abbildung 7 zeigt die LEDA Implementierung von diesem Algorithmus. Wegen der schon erwähnten großen Ähnlichkeit zwischen der algorithmischen Beschreibung und dem LEDA Programm sind nur wenige zusätzliche Worte notwendig, um das Programm zu erklären: *L.sort*() sortiert die Liste L gemäß einer vordefinierten Ordnung auf ihren Elementen. Für Punkte ist dies die bereits erwähnte Links-Rechts Reihenfolge. *L.pop*() löscht das erste Element der Liste und gibt es zurück. *Uh.append*(p) hängt den Punkt p an die Liste Uh an, *L.empty*() gibt true zurück, wenn L leer ist, *Uh.length*() gibt die Länge der Liste Uh zurück, und *Uh.Pop*() löscht das letzte Element der Liste Uh. In LEDA wird eine Liste als eine Folge sogenannter Items (mit dem Typ **list_item**) betrachtet, wobei jedes dieser Items ein Element der Liste enthält. *Uh.last*() gibt das letzte Item der liste Uh zurück und für ein Item *it* kann mit $Uh[it]$ auf den Inhalt des Items zugegriffen werden. Das Vorgänger-Item von *it* in der Liste Uh erhält man mit *Uh.pred*(*it*).

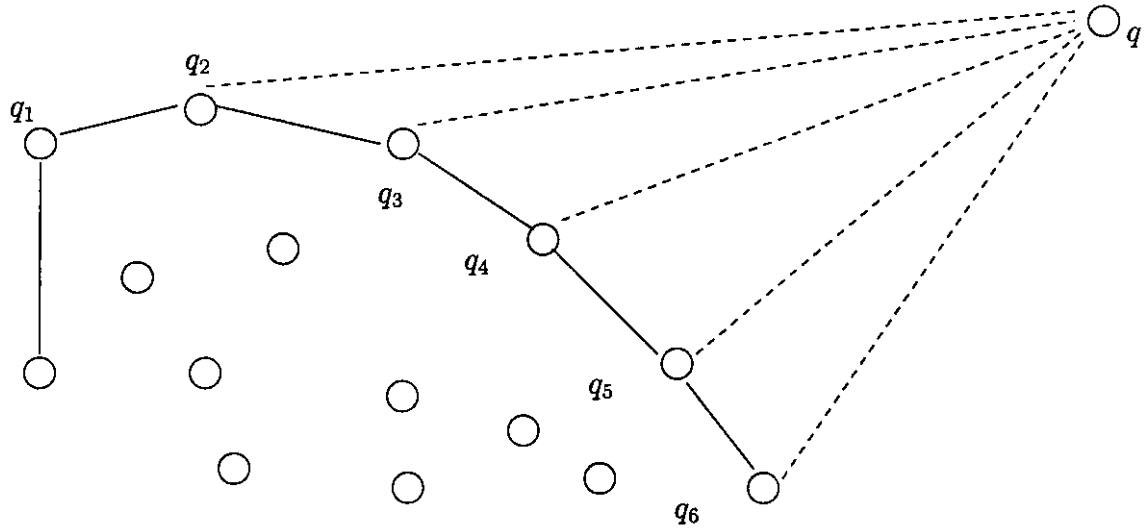

Abbildung 6: Die oberer Hülle nach dem Abarbeiten aller Punkte außer q. Wenn der Knoten q hinzugefügt wird, werden die letzten vier Punkte der aktuellen Hülle gelöscht, weil (q_i, q_{i+1}, q) für $2 \leq i \leq 5$ keine Rechtskurve bildet.

Es gibt noch eine weitere interessante Beobachtung bei dem Programm zur Berechnung der konvexen Hülle. Ein Punkt in LEDA kann beliebige rationale Koordinaten enthalten und alle Prädikate werden exakt (d.h. mit genauer rationaler Arithmetik) ausgerechnet. Beachten Sie auch, daß das Programm mehrfaches Auftreten des gleichen Punktes ebenso korrekt verarbeitet wie kollineare Punkte (solche Situationen werden degenerierte Fälle genannt). Es ist vorgesehen, daß alle geometrischen Algorithmen in LEDA mit allen degenerierten Fällen zurecht kommen und momentan trifft dies bereits für viele zu. Weitere Details hierzu können in [BMS94b], [BMS94a] und [MN94b] nachgelesen werden.

1.4 Grafik

Der LEDA Datentyp **window** ist eine Schnittstelle zum X11 Window System. Abbildung 8 illustriert den Gebrauch dieses Datentyps. Das Programm liest eine Folge von Punkten ein, gibt sie in einem Fenster aus und zeichnet ihre obere Hülle als Linienzug. Wieder reichen wenige Erklärungen aus: Die Definition **window** W definiert ein grafisches Fenster und öffnet es für die Mauseingabe. Durch Drücken der linken Maustaste gibt man einen Punkt ein ($W \gg p$); beim Betätigen der rechten Maustaste wird $W \gg p$ als false ausgewertet. Der Punkt wird an die Liste L gehängt und im Fenster W gezeigt. Danach wird die obere Hülle berechnet und als Linienzug gezeichnet. Abbildung 9 zeigt ein Beispiel.

```
#include<LEDA/list.h>
#include<LEDA/plane.h>

list<point> u_hull(list<point> L)
{ L.sort();  // into left-to-right order
  list<point> Uh;
  point p = L.pop();
  Uh.append(p);
  while (!L.empty())
  { point q = L.pop();  // deletes the first element from L
    if (p == q) continue;
    list_item it = Uh.last();
    while (Uh.length() >= 2 &&
           right_turn(Uh[Uh.pred(it)],Uh[it],q))
    { Uh.del_item(it);  // deletes the last element from Uh
      it = Uh.last();
     }
    Uh.append(q);
    p = q;
   }
  return Uh;
}
```

Abbildung 7: Ein Programm zum Berechnen der oberen Hülle einer Punktmenge.

2 Größe der Bibliothek

LEDA ist sehr umfangreich. Die Bibliothek enthält die meisten Datenstrukturen und Algorithmen, die in den einschlägigen Textbüchern beschrieben sind (siehe [AHU83], [CLR90], [Kin90], [Meh84], [NH93], [Woo93]). Sie ist in die folgenden sechs Gebiete unterteilt: (1) Basistypen, (2) Zahlen, Vektoren und Matrizen, (3) Wörterbücher und Warteschlangen, (4) Graphen, (5) Geometrie und (6) Grafik.

Die Basisdatentypen sind Strings, Listen, Schlangen, Keller, Felder, Partitionen und Bäume.

Die Zahlentypen umfassen sowohl die eingebauten Typen (int, float, double) als auch Typen mit beliebiger Genauigkeit (integer, real). Der Typ integer entspricht der Menge der ganzen Zahlen im mathematischen Sinne; real entspricht der Klasse der floating point Zahlen mit Mantisse und Exponent in beliebiger Genauigkeit. Es gibt auch Vektoren und Matrizen für all diese Zahlentypen.

In der dritten Gruppe gibt es Warteschlangen, Wörterbücher, Wörterbuch- und Hashingfelder, Sortierte Folgen und persistente Wörterbücher.

```
#include<LEDA/window.h>

main()
{ window W;
  list<point> L;
  point p;
  while (W >> p)
  { L.append(p);
    W.draw_point(p);
   }
  W.draw_polygon(u_hull(L));
  W.read_mouse
}
```

Abbildung 8: Ein Programm, das den Algorithmus für die obere Hülle und die Schnittstelle zu XWindows illustriert.

Der Graphenteil bietet verschiedene Arten von Graphen: gerichtete Graphen, ungerichtete Graphen und planare Graphen. Darüber hinaus gibt es noch weitere Datenstrukturen auf Graphen wie etwa Felder, die mit Knoten oder Kanten indiziert sind, Warteschlangen für Knoten, Knotenpartitionen und andere. Zusätzlich ist eine große Anzahl von Algorithmen auf Graphen und Netzwerken verfügbar: Kürzeste Wege, zweifache Zusammenhangskomponenten, starke Zusammenhangskomponenten, transitive Hülle, topologisches Sortieren, ungewichtetes und gewichtetes bipartites Matching, ungewichtetes allgemeines Matching, Netzwerkfluß, Netzwerkfluß mit minimalen Kosten, Planaritätstest, planare Einbettung, minimaler aufspannender Baum, usw.. Das LEDA-Handbuch [NU95] sagt Ihnen, was es sonst noch alles gibt.

3 Implementierung

Alle Datentypen und Algorithmen in LEDA sind vorkompiliert und in Bibliotheken abgespeichert. Ein Anwenderprogramm braucht nur die Header Files von den Datentypen, die in der Anwendung benutzt werden, einzubinden. Diese sind im allgemeinen kurz (sie bestehen nämlich nur aus den Deklarationen der member Funktionen des Typs) und enthalten wenig Programmtext. Da LEDA erlaubt, die Anwenderprogramme auf hohem Niveau zu formulieren, sind diese normalerweise kurz und elegant. Insgesamt resultiert daraus eine kurze Übersetzungszeit.

Alle Datentypen in LEDA sind durch die asymptotisch effizienteste Implementierung realisiert, die bekannt ist. Für viele Datentypen werden verschiedene Implementierungen zur Verfügung gestellt. Beispielsweise kann der Benutzer bei den Wörterbüchern unter ab-Bäumen, AVL-Bäumen, $BB[\alpha]$-Bäumen, Rot-Schwarz-Bäumen, Skiplisten und randomisierten Suchbäumen wählen. Der Mechanismus zum Auswählen einer anderen Im-

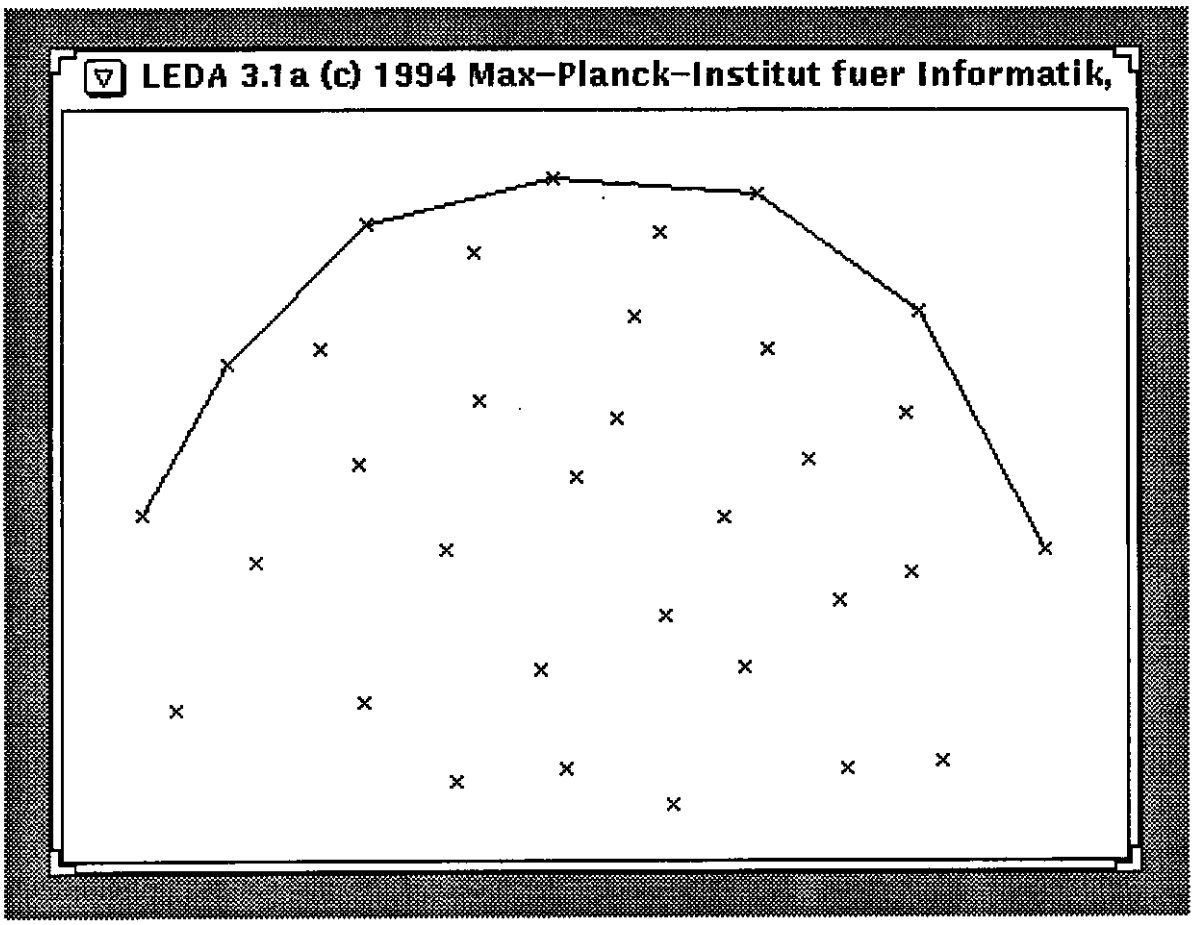

Abbildung 9: Eine Ausgabe des Programms zum Berechnen der oberen Hülle.

plementierung ist ziemlich bequem. Wenn man zum Beispiel die Definition des Wörterbuchfeldes N im Wörterzähl-Programm durch **_d_array<string, int,** *skip_list* **>** $N(0)$ ersetzt, werden automatisch Skiplisten benutzt.

Abstrakte Datentypen verstecken die Implementierungsdetails einer Datenstruktur. Allerdings kann die Abstraktion, wenn sie nicht sorgfältig durchgeführt wird, auch einen Effizienzverlust mit sich bringen, wie folgendes Beispiel erläutert. Ein Wörterbuch mit Schlüsseltyp **K** und Informationstyp **I** wird gewöhnlich als Abbildung von **K** nach **I** betrachtet. Nehmen wir nun an, jemand möchte zuerst auf die Information zugreifen, die zu einem Schlüssel k gehört, dann diese Information verändern, um schließlich die neue Information an den Schlüssel k zu binden. Ist ein Wörterbuch auf herkömmliche Weise definiert, werden dabei zwei Suchvorgänge notwendig: Beim ersten Suchen werden der Schlüssel k und die dazugehörige Information lokalisiert. Der zweite Suchvorgang lokalisiert k noch einmal und assoziiert eine neue Information mit k. Ein direkter Zugriff auf die Datenstruktur könnte jedoch die zweite Suche sparen. Man merkt sich einfach einen Zeiger auf die Position des Schlüssels k in der Datenstruktur und ersetzt den zweiten Suchvorgang durch direkten Zeigerzugriff. In LEDA wurde ein neues Item-Konzept eingeführt, um die durch die Abstraktion eingeführte Ineffizienz zu umgehen. Ein Wörterbuch wird als eine Sammlung von items (mit dem Typ **dic_item**) betrachtet, von denen

jedes einen Schlüssel und eine Information enthält. Die Schlüssel, die mit verschiedenen Informationen assoziiert sind, sind verschieden. Eine Suche in einem Wörterbuch nimmt einen Schlüssel und gibt ein Item zurück (und nicht die darin enthaltene Information). Auf die Information kann nun über dieses Item zugegriffen werden. Das Wesentliche dabei ist, daß man das Item speichern und später direkt über dieses Item zugreifen kann. Auf diese Weise stellen Items die Abstraktion einer Position in einer Datenstruktur dar. Sie ermöglichen einerseits Effizienz und erlauben andererseits vollständige Verkapselung der zugrundeliegenden Datenstruktur. Unserer Meinung nach ist das Item-Konzept ein Schlüsselfaktor für die Effizienz von LEDA. Weitere Informationen dazu finden sie im LEDA-Handbuch [NU95] und in [MN92]

Natürlich müssen wir einen Preis für die Universalität von LEDA zahlen. Dr. Ullrich Lauther von der Siemens AG in München [Lau92] hat intensive Vergleiche zwischen mit LEDA implementierten Graphen- und Netzwerkalgorithmen und von Hand in C geschriebenen durchgeführt. Er erklärt, die LEDA Versionen seien um einen Faktor zwischen 2 und 10 (normalerweise etwa 4) langsamer als seine eigenen Versionen und benötigten zwischen 2 und 3,5 mal soviel Speicherplatz. Wir glauben, daß man dies in Anbetracht der Bequemlichkeit, die LEDA bietet, durchaus akzeptieren kann. Als Reaktion auf Lauthers Bericht haben wir mittlerweile einige grundlegenden Datenstrukturen (in erster Linie graph) neu implementiert, wodurch der Overhead auf einen Faktor von etwa 2 reduziert wurde.

4 Programmüberprüfung

Wie stellen wir Korrektheit der Programme in der LEDA-Bibliothek sicher?

- Wir gehen von korrekten Algorithmen aus (die Algorithmenforschung legt, Gott sei Dank, großen Wert auf Korrektheit und Analyse).
- Wir dokumentieren unsere Programme ausführlich (zumindest in jüngerer Zeit), siehe z.B. [MN94a, MZ93].
- Wir testen unsere Programme intensiv und die große Nutzergemeinde von LEDA tut es auch.
- Wir entwickeln Prüfprogramme, die die Ausgaben unserer Programme überprüfen.

Den letzten Punkt vertiefen wir durch ein Beispiel und verweisen den Leser auf den Artikel [MNSSSSU96] für eine ausführlich Diskussion. In LEDA gibt es schon seit 1990 eine Funktion *bool* PLANAR(*graph G*), die testet, ob ein Graph G planar ist. 1992 bekamen wir einen planaren Graphen zugesandt, den unser Programm für nicht planar erklärt hatte. Bei der Reimplementierung des Algorithmus [MMN93] erkannten wir, daß wir nur dann Vertrauen in die Korrektheit unserer Implementierung haben könnten, wenn diese *ihre Antwort belegen würde.* Die neue Implementierung tut daher folgendes: Wenn der Eingabegraph planar ist, dann liefert das Programm eine planare Einbettung, und wenn der Eingabegraph nicht planar ist, dann liefert das Programm einen sogenannten Kuratowski-Untergraphen, der die Nichtplanarität bezeugt. Ein Nutzer der Neuimplementierung muß nun nichts mehr glauben, er erhält vielmehr für jeden Eingabegraphen einen Beweis für die Korrektheit der Ausgabe. Falls der Leser Zugang zur LEDA-Bibliothek hat, sollte er an dieser Stelle die Vorführung *plan_demo* ausprobieren. Er wird erfahren, daß der Planaritätstest und das Erstellen der Zeichnung sehr schnell ist (die entsprechenden Programme laufen in Linearzeit), daß aber das Finden der Kuratowskigraphen vergleichsweise lange dauert (die Laufzeit ist quadratisch). Wir haben auch für das letztere Problem jüngst einen Linearzeitalgorithmus gefunden, aber noch nicht implementiert [HMN96].

5 Schlußfolgerungen

Die Arbeit an LEDA wurde 1989 begonnen und eine erste Version wurde 1990 für die Öffentlichkeit zugänglich gemacht. Seither wird die Bibliothek ständig weiterentwickelt. Die aktuelle Version ist über ftp@mpi-sb.mpg.de abrufbar. LEDA wird inzwischen an sehr vielen Hochschulen und Forschungsinstituten eingesetzt, an der Universität Dortmund, etwa in den Fachbereichen Informatik, Physik, Mathematik, Chemietechnik, Maschinenbau und Elektrotechnik. Es ist uns also gelungen, die Ergebnisse der Algorithmenformschung auch ausßerhalb der Informatik zugänglich zu machen.

LEDA ist nicht die einzige Bibliothek für Datenstrukturen. Andere sind NIHL [GOP90], Booch components [Boo87] ([Loc94] gibt einen Überblick) und die Standard Template Library (STL). Die Haupteigenschaft, die LEDA von den anderen Bibliotheken unterscheidet ist ihr Umfang. Keine andere Bibliothek enthält soviele Datentypen und Algorithmen aus dem Gebiet des kombinatorischen und geometrischen Rechnens.

Das LEDA-Projekt hat uns sehr viel Befriedigung gebracht.

- Die Bibliothek ist weit verbreitet. Das zeigt, daß die Ergebnisse der Algorithmenforschung gebraucht werden (wenn man sie denn nur richtig aufbereitet).
- Die Bibliothek ist eine Quelle schwieriger und gut motivierter theoretischer Forschungsaufgaben. Die Lösung dieser Aufgaben erlaubte es, die Qualität des Produkts wesentlich zu verbessern. Dies zeigt zum einen, daß auch unsere theoretische Forschung von der Bibliothek profitiert und zum anderen, daß **nur** Algorithmenforscher die Ergebnisse des Gebiets auch implementieren können.

Literatur

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, 1983.

[BMS94a] Ch. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[BMS94b] Ch. Burnikel, K. Mehlhorn, and St. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *LNCS*, 1994. Proceedings of ESA'94, volume 855.

[Boo87] G. Booch. *Software Components with Ada.* Benjamin/Cummings Publishing Company, 1987.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill Book Company, 1990.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[GOP90] K.E. Gorlen, S.M. Orlow, and P.S. Plexico. *Data Abstraction and object-oriented programming in C++.* John Wiley and Sons Ltd., 1990.

[HMN96] C. Hundack, K. Mehlhorn, and S. Näher A Simple Linear Time Algorithm for Identifying Kuratowski Subgraphs of Non-Planar Graphs to appear, 1996

[Kin90] J.H. Kingston. *Algorithms and Data Structures.* Addison-Wesley Publishing Company, 1990.

[Lau92] U. Lauther. Untersuchung der library of efficient data types and algorithms (LEDA). Technical report, Siemens AG, ZFE, München, 1992.

[Loc94] N. Locke. C++ FTP Libraries. *C++ Report*, 6(2):61–65, 1994.

[Meh84] K. Mehlhorn. *Data Structures and Efficient Algorithms.* Springer Verlag, 1984.

[MMN93] K. Mehlhorn, P. Mutzel, and St. Näher. An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Technical Report MPI–I–93–151, Max–Planck–Institut für Informatik, Saarbrücken, 1993.

[MN92] K. Mehlhorn and St. Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1. Elsevier Science Publishers B.V., 1992.

[MN94a] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[MN94b] K. Mehlhorn and St. Näher. The implementation of geometric algorithms. 1994. IFIP94, to appear.

[MNSSSSU96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and Ch. Uhrig. Checking Geometric Programsor Verification of Geometric Structures 1996 Computational Geomtry, to appear

[MZ93] M. Müller and J. Ziegler. An implementation of a convex hull algorithm. Technical Report MPI-I-94–105, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[NU95] St. Näher and Ch. Uhrig. LEDA manual. Technical Report MPI-I-95-1-02, Max-Planck-Institut für Informatik, 1995.

[NH93] J. Nievergelt and K.H. Hinrichs. *Algorithms and Data Structures.* Prentice Hall Inc., 1993.

[O'R94] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[Sed91] R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1991.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*, volume 44. CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

[Wyk88] C.J. van Wyk. *Data Structures and C programs.* Addison-Wesley Publishing Company, 1988.

[Woo93] D. Wood. *Data Structures, Algorithms, and Performance.* Addison-Wesley Publishing Company, 1993.