Research Seminar on Algorithm Engineering
Chapter 8 — Libraries

Roman Dementiev and Johannes Singler

Institute for Theoretical Computer Science
University of Karlsruhe

Dagstuhl, September 7th, 2006
1 Introduction
2 Library Overview
3 Libraries as Building Blocks
4 Basic Design Goals and Paradigms
5 Fundamental Operations
6 Advanced Number Types
7 Basic Data Structures and Algorithms
8 Graph Data Structures and Algorithms
9 Computational Geometry
10 Conclusion
algorithm engineering

realistic models

design

falsifiable hypotheses

induction

implementation

experiments

real Inputs

algorithm-libraries

analysis
deduction

performance guarantees
1 Introduction

2 Library Overview

3 Libraries as Building Blocks

4 Basic Design Goals and Paradigms

5 Fundamental Operations

6 Advanced Number Types

7 Basic Data Structures and Algorithms

8 Graph Data Structures and Algorithms

9 Computational Geometry

10 Conclusion
Motivation

Precarious Questions

- Have you never programmed the same (non-trivial) thing twice?
- **Have you ever refrained from fully implementing something because of some “intricate” part?**
- Have you ever given up to get “some code” running on your machine?
- Have you never regretted to have copied-and-pasted some code instead of writing a versatile version?

Solution

- Use software libraries!
- “You talk like a *Software Engineer*? We are in Algorithmics here!?”
- “Software libraries are also very important in Algorithm Engineering!”
- Software Libraries + Algorithm Engineering $\Rightarrow$ Algorithm Libraries
Precarious Questions

- Have you never programmed the same (non-trivial) thing twice?
- Have you ever refrained from fully implementing something because of some “intricate” part?
- Have you ever given up to get “some code” running on your machine?
- Have you never regretted to have copied-and-pasted some code instead of writing a versatile version?

Solution

- Use software libraries!
- “You talk like a Software Engineer? We are in Algorithmics here!?”
- “Software libraries are also very important in Algorithm Engineering!”
- Software Libraries + Algorithm Engineering $\Rightarrow$ Algorithm Libraries
Definition of a *Library*

- “A collection of program components that can be used in many programs” [Plauger2000]
  - Function Libraries, Class Libraries, Component Libraries
- To distinguish from Software Systems
  - e.g. computer algebra systems, numerical computation systems like MatLab
  - Library and system differ in level
  - Library and program are based on same foundations, have same possibilities
  - “Program” in a software system has limited access to the resources of the machine

- Common ground: foster reuse of already implemented functionality

*Algorithm* libraries are a “major goal” of Algorithm Engineering
Definition of a Library

- “A collection of program components that can be used in many programs” [Plauger2000]
  - Function Libraries, Class Libraries, Component Libraries
- To distinguish from Software Systems
  - e.g. computer algebra systems, numerical computation systems like MatLab
  - Library and system differ in level
  - Library and program are based on same foundations, have same possibilities
  - “Program” in a software system has limited access to the resources of the machine
- Common ground: foster reuse of already implemented functionality

Algorithm libraries are a “major goal” of Algorithm Engineering
The Role of Libraries in Algorithm Engineering

Algorithm Engineers do both

- use libraries
- develop libraries

Major Questions of this Topic

- Why is library XY useful for us in Algorithm Engineering? ⇒ Æ
- Has Algorithm Engineering played an important role in developing/designing library XY? Æ ⇒
Introduction

The Role of Libraries in Algorithm Engineering

Major Questions of this Topic

- Why is library XY useful for us in Algorithm Engineering? \(\Rightarrow \exists \)  
- Has Algorithm Engineering played an important role in developing/designing library XY? \(\exists \Rightarrow \)
Introduction

Library Overview

Libraries as Building Blocks

Basic Design Goals and Paradigms

Fundamental Operations

Advanced Number Types

Basic Data Structures and Algorithms

Graph Data Structures and Algorithms

Computational Geometry

Conclusion
Why C++?

We limit our deep investigations to C++ libraries

- **Accepted in AE community and industry**
- Object-oriented
- Generic programming
- Zero-overhead
- Fast native binary code, excellent compilers for most platforms
- ISO standardized
  - No interface self-inspection
  - Sometimes lengthy recompilations
### Most popular problems

<table>
<thead>
<tr>
<th>Rank</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>shortest-path</td>
</tr>
<tr>
<td>2</td>
<td>kd-trees</td>
</tr>
<tr>
<td>3</td>
<td>dictionaries</td>
</tr>
<tr>
<td>4</td>
<td>traveling-salesman</td>
</tr>
<tr>
<td>5</td>
<td>convex-hull</td>
</tr>
<tr>
<td>6</td>
<td>nearest-neighbor</td>
</tr>
<tr>
<td>7</td>
<td>minimum-spanning-tree</td>
</tr>
<tr>
<td>8</td>
<td>voronoi-diagrams</td>
</tr>
<tr>
<td>9</td>
<td>triangulations</td>
</tr>
<tr>
<td>10</td>
<td>sorting</td>
</tr>
<tr>
<td>11</td>
<td>graph-data-structures</td>
</tr>
<tr>
<td>12</td>
<td>string-matching</td>
</tr>
<tr>
<td>13</td>
<td>sux-trees</td>
</tr>
<tr>
<td>14</td>
<td>priority-queues</td>
</tr>
<tr>
<td>15</td>
<td>geometric-primitives</td>
</tr>
</tbody>
</table>
**C++ Standard Template Library (STL)**

- Basic data structures and algorithms
- Specification standardized with C++ in 1998, many implementations (belong to C++ compilers)

**Boost library**

- Extends STL, e.g. by graphs

**Library of Efficient Data Structures and Algorithms (LEDA)**

- Outrider of algorithmic libraries, development started in 1988
- From the Algorithm Engineering community

**CGAL + Exacus**

- (Curved) Computational Geometry
# Library Overview

## Combinatorial and Geometric Computing

### C++ Standard Template Library (STL)
- Basic data structures and algorithms
- Specification standardized with C++ in 1998, many implementations (belong to C++ compilers)

### Boost library
- Extends STL, e.g. by graphs

### Library of Efficient Data Structures and Algorithms (LEDA)
- Outrider of algorithmic libraries, development started in 1988
- From the Algorithm Engineering community

### CGAL + Exacus
- (Curved) Computational Geometry
### Combinatorial and Geometric Computing

<table>
<thead>
<tr>
<th>Library Overview</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C++ Standard Template Library (STL)</strong></td>
</tr>
<tr>
<td>- Basic data structures and algorithms</td>
</tr>
<tr>
<td>- Specification standardized with C++ in 1998, many implementations (belong to C++ compilers)</td>
</tr>
<tr>
<td><strong>Boost library</strong></td>
</tr>
<tr>
<td>- Extends STL, e.g. by graphs</td>
</tr>
<tr>
<td><strong>Library of Efficient Data Structures and Algorithms (LEDA)</strong></td>
</tr>
<tr>
<td>- Outrider of algorithmic libraries, development started in 1988</td>
</tr>
<tr>
<td>- From the Algorithm Engineering community</td>
</tr>
<tr>
<td><strong>CGAL + Exacus</strong></td>
</tr>
<tr>
<td>- (Curved) Computational Geometry</td>
</tr>
</tbody>
</table>
### Combinatorial and Geometric Computing

#### C++ Standard Template Library (STL)
- Basic data structures and algorithms
- Specification standardized with C++ in 1998, many implementations (belong to C++ compilers)

#### Boost library
- Extends STL, e.g. by graphs

#### Library of Efficient Data Structures and Algorithms (LEDA)
- Outrider of algorithmic libraries, development started in 1988
- From the Algorithm Engineering community

#### CGAL + Exacus
- (Curved) Computational Geometry
Graph Drawing

Goals

- Embed graphs nicely into two or three dimensions
- Avoid crossings, draw labels

Libraries

- AGD: Algorithms for Graph Drawing (academic, outdated)
- Visone: visualize social networks (academic, new)
- Commercial libraries: yFiles and TomSawyer
Graph Drawing

Goals

- Embed graphs nicely into two or three dimensions
- Avoid crossings, draw labels

Libraries

- AGD: Algorithms for Graph Drawing (academic, outdated)
- Visone: visualize social networks (academic, new)
- Commercial libraries: yFiles and TomSawyer
**Goals**

- Solve numerical problems
- Trade-off between speed and accuracy

**Libraries**

Provide matrix calculations, basic linear algebra, eigenvalue calculations, statistics, advanced number types

- Packages EISPACK, LINPACK, MINPACK: eigenvalues, singular value decompositions, linear equations, least-squares, function minimization
- Matrix Template Library: template metaprogramming for performance-critical code generations
- GNU Multiple Precision Arithmetic Library (GMP), CORE: number types ⇒ later
**Goals**

- Solve numerical problems
- Trade-off between speed and accuracy

**Libraries**

Provide matrix calculations, basic linear algebra, eigenvalue calculations, statistics, advanced number types

- Packages EISPACK, LINPACK, MINPACK: eigenvalues, singular value decompositions, linear equations, least-squares, function minimization
- Matrix Template Library: template metaprogramming for performance-critical code generations
- GNU Multiple Precision Arithmetic Library (GMP), CORE: number types ⇒ later
### External Memory

- Input is larger than main memory
- **Minimize the number of I/Os**
- Libraries: TPIE, LEDA-SM, STXXL

### Cache-Efficiency

- Cache = fast memory, main memory = slow memory
- No explicit control of I/O operations
- No library exists
External Memory

- Input is larger than main memory
- Minimize the number of I/Os
- Libraries: TPIE, LEDA-SM, STXXL

Cache-Efficiency

- Cache = fast memory, main memory = slow memory
- No explicit control of I/O operations
- No library exists
Parallel Computers

1. Communication and Synchronization: CGMlib, BSPlib, OpenMP, ClusterOpenMP, POSIX threads
2. Algorithmic libraries: STAPL, parallel BGL, CGMgraph, MCSTL
Library Overview

Broad Functionality Overview

- Basic Data Structures and Algorithms
- Advanced Number Types
- Graphs
- Basic Graph Drawing
- Computational Geometry
- STL
- Boost
- CGAL
- LEDA
- GMP, CORE
Evaluation of Library Usage

Ideal properties of an algorithm library

- Easy to use, platform independent
- Exactly specified and documented
- Well-tested, robust (against misuse)
- Broad “basic/versatile” functionality
- Efficient (asymptotic and/or in practice)

Advantages from using a library in implementing new algorithms

- Faster development
  - Complex subparts are provided by library
    ⇒ Algorithm cannot be harder than its description
  - Sometimes making the project worthwhile
- Improved correctness
- Actual/higher performance improvement
  - “Stand on the shoulder of giants!”
Evaluation of Library Usage

Ideal properties of an algorithm library

- Easy to use, platform independent
- Exactly specified and documented
- Well-tested, robust (against misuse)
- Broad “basic/versatile” functionality
- Efficient (asymptotic and/or in practice)

Advantages from using a library in implementing new algorithms

- Faster development
  - Complex subparts are provided by library
    ⇒ Algorithm cannot be harder than its description
  - Sometimes making the project worthwhile
- Improved correctness
- Actual/higher performance improvement
  - “Stand on the shoulder of giants!”
### Evaluation of Library Usage

#### Disadvantages from using a library

- Compromises in running time and memory usage for the sake of generality
- Lacking insight (in particular when no source code available)
- Monetary cost
- Learn how to use (although easy)
- “Not invented here”
- Incompatible / badly interoperable libraries: decision necessary

#### Comparisons against library implementations

- Libraries set a *lower bound* in terms of performance for a problem because of
  - General and easy to use, why use specialized version then?
  - Software Engineering benefits, e.g. code sharing
Disadvantages from using a library

- Compromises in running time and memory usage for the sake of generality
- Lacking insight (in particular when no source code available)
- Monetary cost
- Learn how to use (although easy)
- “Not invented here”
- Incompatible / badly interoperable libraries: decision necessary

Comparisons against library implementations

- Libraries set a \textit{lower bound} in terms of performance for a problem because of
  - General and easy to use, why use specialized version then?
  - Software Engineering benefits, e.g. code sharing
## Usual Algorithmic Lifecycle

1. New problem arises, abstract model is constructed
2. First, simple algorithm is given, usually not efficient/optimal, but easy to implement
3. This algorithm is used in practice
4. Algorithm Engineering cycle starts
5. More evolved algorithms are published, based on (many) advanced data structures as foundations
6. No/little impact on practice

## Algorithm Lifecycle with Library Integration

7. Efficient implementations are integrated into a library
8. Academic and industrial use start
9. Accelerated Algorithm Engineering cycle starts
Usual Algorithmic Lifecycle

1. New problem arises, abstract model is constructed
2. First, simple algorithm is given, usually not efficient/optimal, but easy to implement
3. This algorithm is used in practice
4. Algorithm Engineering cycle starts
5. More evolved algorithms are published, based on (many) advanced data structures as foundations
6. No/little impact on practice

Algorithm Lifecycle with Library Integration

7. Efficient implementations are integrated into a library
8. Academic and industrial use start
9. **Accelerated Algorithm Engineering cycle starts**
1 Introduction

2 Library Overview

3 Libraries as Building Blocks

4 Basic Design Goals and Paradigms

5 Fundamental Operations

6 Advanced Number Types

7 Basic Data Structures and Algorithms

8 Graph Data Structures and Algorithms

9 Computational Geometry

10 Conclusion
We consider the “major” libraries mentioned in this talk

- LEDA
- STL + Boost
- CGAL + EXACUS

Questions asked

- What major aims does the library strive for?
- Which of the properties of an ideal library are realized?
- In what order are those prioritized?
- Is the functionality restricted to a certain area?
We consider the “major” libraries mentioned in this talk

- LEDA
- STL + Boost
- CGAL + EXACUS

Questions asked

- What major aims does the library strive for?
- **Which of the properties of an ideal library are realized?**
- In what order are those prioritized?
- Is the functionality restricted to a certain area?
LEDＡ Goals

Four principal design goals, in this order

- Ease of Use
- Extensibility
- Correctness
- Efficiency

Comments on the Goals

- Correctness less important than something else?
- Efficiency last although part of the library’s name? “Worthless without other three goals fulfilled”
  - Asymptotic efficiency is guaranteed, at least with high probability

Other

- Automatic API doc generation from code comments, test cases
- LEDA extension packages
LEDAs Basic Design Goals and Paradigms

LEDAs Goals

Four principal design goals, in this order:

- Ease of Use
- Extensibility
- Correctness
- Efficiency

Comments on the Goals:

- Correctness less important than something else?
- **Efficiency last although part of the library’s name?**
  “Worthless without other three goals fulfilled”
  - Asymptotic efficiency is guaranteed, at least with high probability

Other:

- Automatic API doc generation from code comments, test cases
- LEDA extension packages
## LEDA Goals

### Four principal design goals, in this order
- Ease of Use
- Extensibility
- Correctness
- Efficiency

### Comments on the Goals
- Correctness less important than something else?
- **Efficiency last although part of the library’s name?**
  “Worthless without other three goals fulfilled”
  - Asymptotic efficiency is guaranteed, at least with high probability

### Other
- Automatic API doc generation from code comments, test cases
- LEDA extension packages
LEDА Functionality

“Horizontal” Dimension

- Used “side by side”
- Functionality is broad, covers many areas of computer science
- Combinatorial computing and geometric computing incl. graphs
- Information theory (compression, encryption)
- **Usually many solutions for a problem, to chose from**
- LEDА extension packages (high-d-geometry, external memory)

“Vertical” Dimension

- Used “on top” of each other
- Provide *everything* a program needs to run
- Basic functionality like console and file I/O
  - Again, LEDА’s age plays a role
- Portable graphic interface, but no sophisticated GUI
**LED A Functionality**

**“Horizontal” Dimension**

- Used “side by side”
- Functionality is broad, covers many areas of computer science
- Combinatorial computing and geometric computing incl. graphs
- Information theory (compression, encryption)
- **Usually many solutions for a problem, to chose from**
- LEDA extension packages (high-d-geometry, external memory)

**“Vertical” Dimension**

- Used “on top” of each other
- Provide *everything* a program needs to run
- Basic functionality like console and file I/O
  - Again, LEDA’s age plays a role
- Portable graphic interface, but no sophisticated GUI
Both STL and Boost

- Fully exploit the features of the programming language
  - Use generic programming paradigm extensively
  - Standardization of C++ accompanied by development of STL
  - Sometimes even try to extend

- Usually only one “solution per problem”

STL

- “Core of the most widely used facilities”: Basic data structures and algorithms, I/O
- Specification only, many implementations freely available
Both STL and Boost

- Fully exploit the features of the programming language
  - Use generic programming paradigm extensively
  - Standardization of C++ accompanied by development of STL
  - Sometimes even try to extend

- Usually only one “solution per problem”

STL

- “Core of the most widely used facilities”:
  Basic data structures and algorithms, I/O

- Specification only, many implementations freely available
Boost

- Extension to the STL, “playground” for future extensions to C++ (TR1)
- Everything a reasonably large number of programmers might like to use: e.g. graphs, parsers
- “Extensions” to the programming language, in particular by means of template metaprogramming
  - Expression templates, lambda expressions
- Specification and implementation
- Community-reviewed
Basic Design Goals and Paradigms

CGAL/Exacus paradigms

Goals

- **Make geometric algorithms available for industrial applications**
  - Be correct: produce *mathematically* correct results (even for degenerated inputs)
    - Exacus: robust *non-linear* geometry (e.g. for curves)
  - Be flexible:
    - work with any user code
    - *user* decides which underlying basic types or/and implementations to use
    - *user* can choose a compromise between efficiency and exactness
  - Be easy to use: shortcuts and defaults for novices
  - Be efficient: many fast implementations, freedom to choose the best

Quality of code

- Well documented (LEDAtyle)
- Community-reviewed
CGAL/Exacus paradigms

Goals

- **Make geometric algorithms available for *industrial* applications**
- Be correct: produce *mathematically* correct results (even for degenerated inputs)
  - Exacus: robust non-linear geometry (e.g. for curves)
- Be flexible:
  - work with any user code
  - *user* decides which underlying basic types or/and implementations to use
  - *user* can choose a compromise between efficiency and exactness
- Be easy to use: shortcuts and defaults for novices
- Be efficient: many fast implementations, freedom to choose the best

Quality of code

- Well documented (LEDAn-style)
- Community-reviewed
Basic Design Goals and Paradigms

CGAL/Exacus paradigms

Goals

- **Make geometric algorithms available for industrial applications**
- Be correct: produce *mathematically* correct results (even for degenerated inputs)
  - Exacus: robust *non-linear* geometry (e.g. for curves)
- Be flexible:
  - work with any user code
  - *user* decides which underlying basic types or/and implementations to use
  - *user* can choose a compromise between efficiency and exactness
- Be easy to use: shortcuts and defaults for novices
- Be efficient: many fast implementations, freedom to choose the best

Quality of code

- Well documented (LEDA-style)
- Community-reviewed
### CGAL/Exacus paradigms

#### Goals

- **Make geometric algorithms available for *industrial* applications**
- Be correct: produce *mathematically* correct results (even for degenerated inputs)
  - Exacus: robust *non-linear* geometry (e.g. for curves)
- Be flexible:
  - work with any user code
  - *user* decides which underlying basic types or/and implementations to use
  - *user* can choose a compromise between efficiency and exactness
- Be easy to use: shortcuts and defaults for novices
- Be efficient: many fast implementations, freedom to choose the best

#### Quality of code

- Well documented (LEDA-style)
- Community-reviewed
### Basic Design Goals and Paradigms

#### CGAL/Exacus paradigms

**Goals**

- **Make geometric algorithms available for industrial applications**
- Be correct: produce *mathematically* correct results (even for degenerated inputs)
  - Exacus: robust *non-linear* geometry (e.g. for curves)
- Be flexible:
  - work with any user code
  - *user* decides which underlying basic types or/and implementations to use
  - *user* can choose a compromise between efficiency and exactness
- Be easy to use: shortcuts and defaults for novices
- Be efficient: many fast implementations, freedom to choose the best

#### Quality of code

- Well documented (LEDA-style)
- Community-reviewed
**Basic Design Goals and Paradigms**

**CGAL/Exacus paradigms**

**Goals**

- **Make geometric algorithms available for industrial applications**
- Be correct: produce *mathematically* correct results (even for degenerated inputs)
  - Exacus: robust *non-linear* geometry (e.g. for curves)
- Be flexible:
  - work with any user code
  - *user* decides which underlying basic types or/and implementations to use
  - *user* can choose a compromise between efficiency and exactness
- Be easy to use: shortcuts and defaults for novices
- Be efficient: many fast implementations, freedom to choose the best

**Quality of code**

- Well documented (LEDA-style)
- Community-reviewed
1 Introduction

2 Library Overview

3 Libraries as Building Blocks

4 Basic Design Goals and Paradigms

5 Fundamental Operations

6 Advanced Number Types

7 Basic Data Structures and Algorithms

8 Graph Data Structures and Algorithms

9 Computational Geometry

10 Conclusion
What are fundamental operations?

- Implicit operations usually not mentioned at all by algorithm descriptions
  - Dynamic memory allocation and deallocation ⇒ Memory management
  - Data structure access
  - Data structures parameterization (specific to libraries)
  - Calling subroutines (only partly mentioned here)

- Only(?) constant factor in running time, no asymptotical problem
**Language Support**

- Language supports basic dynamic memory management with support of compiler and/or run time system/operating system: `malloc, free`
- Usually not very sophisticated implementation
- No heuristics, but maximum parsimony, fragmentation
- Allocation might include (repeated) construction

**LEDAl**

- Own memory manager, particularly useful for many small objects of a common class
- Uses free lists
- Significant performance improvements for certain algorithms shown $\Rightarrow$
- Also easily usable for user-defined types
Language Support

- Language supports basic dynamic memory management with support of compiler and/or run time system/operating system: `malloc`, `free`
- Usually not very sophisticated implementation
- No heuristics, but maximum parsimony, fragmentation
- Allocation might include (repeated) construction

LEDA

- Own memory manager, particularly useful for many small objects of a common class
- Uses free lists
- **Significant performance improvements for certain algorithms shown** $\Rightarrow$
- Also easily usable for user-defined types
STL + Boost

- STL allows integration of user-defined allocators, but “only for adventurers”
- Boost *Pool* allows management for objects of *one certain size*
  - Integration into container classes possible, using above
  - Also uses free lists
  - All object can be made aligned
  - All objects in a pool can be deallocated at a single blow

Summary and Comparison

- LEDA allows choosing its memory manager “per type”
- STL + Boost allow choosing a memory manager “per allocation” or “per container”
- No direct performance comparison available
- Comparison to garbage collection would be interesting
STL + Boost

- STL allows integration of user-defined allocators, but “only for adventurers”
- Boost Pool allows management for objects of one certain size
  - Integration into container classes possible, using above
  - Also uses free lists
  - All object can be made aligned
  - All objects in a pool can be deallocated at a single blow

Summary and Comparison

- LEDA allows choosing its memory manager “per type”
- STL + Boost allow choosing a memory manager “per allocation” or “per container”
- No direct performance comparison available
- Comparison to garbage collection would be interesting
Requirements of Algorithms

- Some kind of generalized pointer inducing an element of a container *in context*
- Enumeration of/iteration over certain elements
- References *into* a data structures can speedup algorithms considerably, e.g. finger search $\Rightarrow \infty$

Available Solutions

- STL *iterator* concept
- LEDA *item* concept

Difference:
- Iterator knows container, supports advancing autonomously
- Item only useful when corresponding container is known

Commonality: limited durability, usually invalidated by updates to the container
Requirements of Algorithms

- Some kind of generalized pointer inducing an element of a container \textit{in context}
- Enumeration of/iteration over certain elements
- References \textit{into} a data structures can speedup algorithms considerably, e.g. finger search \Rightarrow \mathcal{AE}

Available Solutions

- STL \textit{iterator} concept
- LEDA \textit{item} concept

Difference:
- Iterator knows container, supports advancing autonomously
- Item only useful when corresponding container is known

Commonality: limited durability, usually invalidated by updates to the container
Requirements of Algorithms

- Some kind of generalized pointer inducing an element of a container in context
- Enumeration of/iteration over certain elements
- References into a data structures can speedup algorithms considerably, e.g. finger search $\Rightarrow \mathcal{O}$

Available Solutions

- STL iterator concept
- LEDA item concept
- Difference:
  - Iterator knows container, supports advancing autonomously
  - Item only useful when corresponding container is known
- Commonality: limited durability, usually invalidated by updates to the container
Parameterization of Data Types

Problem

- Container Types must be able to contain data of different type
- Explicit implementation for each contained type impossible
- **Type not only carries data, but also semantics**
  - Ordering relation(s)
  - Copy and assigning semantics

Basic Approaches

1. Specific container for each combination of container and data using generic programming
2. Only one container type with weakly typed reference to data
Problem

- Container Types must be able to contain data of different type
- Explicit implementation for each contained type impossible
- **Type not only carries data, but also semantics**
  - Ordering relation(s)
  - Copy and assigning semantics

Basic Approaches

1. Specific container for each combination of container and data using generic programming
2. Only one container type with weakly typed reference to data
Hybrid approach

- Type safety assured by generic wrapper class
- **Actual implementation with weakly typed reference,**(`GenPtr`)
- Shortcut for small types (smaller equal to pointer) \(\mathcal{AE}\)⇒
- Semantics via global function / function pointer

Advantages

- No code duplication, binary distribution possible, faster compilation
- Better compatibility to compilers (historic reason)

Disadvantages

- Twice the (de-)allocation of objects: container object and data object
- Indirection
  - Additional load each time accessing the data
  - Construct/compare/copy/hash: indirect function call (either function pointer or virtual function call)
The LEDA solution

Hybrid approach

- Type safety assured by generic wrapper class
- **Actual implementation with weakly typed reference**, $(\text{GenPtr})$
- Shortcut for small types (smaller equal to pointer) $\land \Rightarrow$
- Semantics via global function / function pointer

Advantages

- No code duplication, binary distribution possible, faster compilation
- Better compatibility to compilers (historic reason)

Disadvantages

- Twice the (de-)allocation of objects: container object and data object
- Indirection
  - Additional load each time accessing the data
  - Construct/compare/copy/hash: indirect function call (either function pointer or virtual function call)
The LEDA solution

Hybrid approach

- Type safety assured by generic wrapper class
- **Actual implementation with weakly typed reference**, (GenPtr)
- Shortcut for small types (smaller equal to pointer) \( \Rightarrow \)
- Semantics via global function / function pointer

Advantages

- No code duplication, binary distribution possible, faster compilation
- Better compatibility to compilers (historic reason)

Disadvantages

- Twice the (de-)allocation of objects: container object and data object
- Indirection
  - Additional load each time accessing the data
  - Construct/compare/copy/hash: indirect function call (either function pointer or virtual function call)
The STL Solution

Generics to the Limit

- Data is integrated into specific instantiation of container item type using the template mechanism
  - Partial specialization avoids unnecessary code duplication, e.g. only one instantiation for all pointer types
- Semantics via functor classes or class members

Advantages

- Full inlining and optimizing of semantic calls possible by the compiler
- Direct access to data
- No cache problems

Disadvantages

- Still some code duplication
- Crude syntax, functors cannot be placed directly at the call site
  - Some relief provided by Boost: Lambda expressions and expression templates
### Generics to the Limit

- **Data is integrated into specific instantiation of container item type using the template mechanism**
  - Partial specialization avoids unnecessary code duplication, e.g. only one instantiation for all pointer types
- Semantics via functor classes or class members

### Advantages

- Full inlining and optimizing of semantic calls possible by the compiler
- Direct access to data
- No cache problems

### Disadvantages

- Still some code duplication
- Crude syntax, functors cannot be placed directly at the call site
  - Some relief provided by Boost: Lambda expressions and expression templates
Generics to the Limit

- Data is integrated into specific instantiation of container item type using the template mechanism
  - Partial specialization avoids unnecessary code duplication, e.g. only one instantiation for all pointer types
- Semantics via functor classes or class members

Advantages

- Full inlining and optimizing of semantic calls possible by the compiler
- Direct access to data
- No cache problems

Disadvantages

- Still some code duplication
- Crude syntax, functors cannot be placed directly at the call site
  - Some relief provided by Boost: Lambda expressions and expression templates
Visualization of the Parameterization Paradigms

LEDAs Container

Generic Wrapper

\textit{virtual compare(...)}

Single Implementation

\textit{n times}

Container Item

Data Structure
Item Internals

GenPtr

Data

\textit{compare(...)}

LEDAs Container (small data)

Generic Wrapper

\textit{virtual compare(...)}

Single Implementation

\textit{n times}

Container Item

Data Structure
Item Internals

\textit{compare(...)}

STL/Boost Container

\textit{n times}

Container Item

Data Structure
Item Internals

Data

\textit{operator<(...)}
1 Introduction
2 Library Overview
3 Libraries as Building Blocks
4 Basic Design Goals and Paradigms
5 Fundamental Operations
6 Advanced Number Types
7 Basic Data Structures and Algorithms
8 Graph Data Structures and Algorithms
9 Computational Geometry
10 Conclusion
Advanced Number Types

**Built-in Arithmetic**

**Theory: Real RAM Model**

- Algorithms operate on numbers with *unlimited* accuracy

**Practice: Limited precision**

- CPUs have 32/64 bit integers
- and IEEE floating point numbers of the form \((-1)^{\text{sign}} \cdot \text{mantissa} \cdot 2^{\text{exponent}}\)
- Programming language (C++) number types are just *mappings* of the CPU number types: `int, long, long long, float, double, long double, ...`

⇒ **Built-in number types can only *approximate* mathematical numbers**
Advanced Number Types

Integer numbers

Problem

- C++ unsigned long overflows reaching the value $2^{32}$
- Results in incorrect results or crashes

Solution

- Maintain a vector of unsigned longs
- Implemented in LEDA (leda::integer)
  - Operations +, -, *, %, +, ...
  - Assembler for critical sections (only for RISC)
  - 30-50 times slower than C++ longs
- Implemented in GNU Multiple Precision Arithmetic Library (GMP) mpz_t
  - C library, has C++ class interface
  - Operations +, -, *, %, +, ... (expression templates for e.g. a=b+c)
  - Assembler for critical loops (almost all CPUs)
  - Optimized for very large number values
  - For medium-sized values worse than leda::integer [Schirra 99]
Integer numbers

Problem

- C++ unsigned long overflows reaching the value $2^{32}$
- Results in incorrect results or crashes

Solution

- **Maintain a vector of unsigned longs**
  - Implemented in LEDA (`leda::integer`)
    - Operations +, -, *, %, ++, ...
    - Assembler for critical sections (only for RISC)
    - 30-50 times slower than C++ longs
  - Implemented in GNU Multiple Precision Arithmetic Library (GMP) `mpz_t`
    - C library, has C++ class interface
    - Operations +, -, *, %, ++, ... (expression templates for e.g. $a=b+c$)
    - Assembler for critical loops (almost all CPUs)
    - Optimized for very large number values
    - For medium-sized values worse than `leda::integer` [Schirra 99]
Problem

- Many algorithms need an exact division of two integers
- The result can not be represented as a single integer type

Solution

- **Store the result as a pair: ** numerator/denominator
  - `leda::rational`
    - keeps two `leda::integer`
    - 30-100 times slower than C++ `double`
  - GMP rational `mpq_t`
    - keeps two `unsigned longs` ⇒ may overflow
- Generic approach: `boost::rational`
  - user is *free* to choose the underlying type
Normalization

- Numerator and denominator have a common factor
- **May be expensive** \((\text{gcd computation})\)
- LEDA: no automatic normalizations (operations on \textit{leda::integer})
- GMP: always normalized (cheap, less overflows)
- Boost: always normalized
Floating Point Numbers

**GMP**\texttt{mpf_t}

- User-defined length of mantissa (fixed length)
- Exponent length depends on the machine word
- Not compatible to IEEE standard (different results on different platforms)

**LEDA**\texttt{bigfloat}

- Mantissa and exponent: \texttt{leda::integer}s
- Rounding to user defined bit-length and mode
Floating Point Numbers

### GMP $\text{mpf}_t$
- User-defined length of mantissa (fixed length)
- Exponent length depends on the machine word
- Not compatible to IEEE standard (different results on different platforms)

### LEDA $\text{bigfloat}$
- Mantissa and exponent: $\text{leda::integers}$
- Rounding to user defined bit-length and mode
**Advanced Number Types**

**Algebraic Numbers**

**Problem**

Exact computations with roots of a polynomial with rational coefficients

**Solution**

**LEDA real**: algebraic numbers for computations with

- $k^{th}$ roots, $k \in \mathbb{N}$
- Rational $+, -, \times, /$, and comparisons

Stores an expression DAG:

\[
(\sqrt{17} + \sqrt{12}) \cdot (\sqrt{17} - \sqrt{12}) - 5
\]

1. Determine the required precision (traverse the graph)
2. Perform computations using `leda::bigfloat` (very slow)
Interval Arithmetic

Applications

- In computer graphics and computational geometry (e.g. in CGAL)
- Quantify propagation of rounding errors in floating point arithmetic

boost::interval

- *Any* type for interval borders
- Works out-of-the-box for all built-in C++ types
- Adapts to work for other types if rounding/exception policies are provided (a helper sublibrary exists)
Interval Arithmetic

Applications

- In computer graphics and computational geometry (e.g. in CGAL)
- Quantify propagation of rounding errors in floating point arithmetic

Boost::Interval

- Any type for interval borders
- Works out-of-the-box for all built-in C++ types
- Adapts to work for other types if rounding/exception policies are provided (a helper sublibrary exists)
Four accuracy levels of CORE library:

- Level 1: Standard IEEE 754 floating point numbers
- Level 2: Specify *arbitrary* accuracy (x bits), overflow after x bits
  - long ⇒ BigInt: GMP integer mpz_t
  - double ⇒ BigFloat: mantissa is BigInt, exponent is unsigned long, interval arithmetic
- Level 3: Specify *guaranteed* accuracy (x bits), first x significant bits are *always* correct
  - double, long ⇒ algebraic number Expr (similar to leda::real)
- Level 4: Intermix accuracy types for individual variables
Four accuracy levels of **CORE library**:

- **Level 1**: Standard IEEE 754 floating point numbers
- **Level 2**: Specify *arbitrary* accuracy \((x \text{ bits})\), overflow after \(x\) bits
  
  - `long` ⇒ `BigInt`: GMP integer `mpz_t`
  - `double` ⇒ `BigFloat`: mantissa is `BigInt`, exponent is `unsigned long`, interval arithmetic

- **Level 3**: Specify *guaranteed* accuracy \((x \text{ bits})\), first \(x\) significant bits are *always* correct
  
  - `double, long` ⇒ algebraic number `Expr` (similar to `leda::real`)

- **Level 4**: Intermix accuracy types for individual variables
Four accuracy levels of CORE library:

- Level 1: Standard IEEE 754 floating point numbers
- Level 2: Specify arbitrary accuracy ($x$ bits), overflow after $x$ bits
  
  long $\Rightarrow$ BigInt: GMP integer mpg
  
  double $\Rightarrow$ BigFloat: mantissa is BigInt, exponent is unsigned long, interval arithmetic

- Level 3: Specify guaranteed accuracy ($x$ bits), first $x$ significant bits are always correct
  
  double, long $\Rightarrow$ algebraic number Expr (similar to leda::real)

- Level 4: Intermix accuracy types for individual variables
Four accuracy levels of CORE library:

- Level 1: Standard IEEE 754 floating point numbers
- Level 2: Specify arbitrary accuracy \((x\) bits), overflow after \(x\) bits
  
  - \(\text{long} \Rightarrow \text{BigInt}: \text{GMP integer mpz_t}\)
  
  - \(\text{double} \Rightarrow \text{BigFloat}: \text{mantissa is BigInt, exponent is unsigned long, interval arithmetic}\)

- Level 3: Specify guaranteed accuracy \((x\) bits), first \(x\) significant bits are always correct

  - \(\text{double, long} \Rightarrow \text{algebraic number Expr (similar to leda::real)}\)

- Level 4: Intermix accuracy types for individual variables
Functionality Overview

Data Structures: “Containers”, e.g.
- lists, arrays, queues
- dictionaries, sets
- priority queues
- (strings)

Algorithms: Executing on the data structures mentioned above, e.g.
- iteration, scanning
- find, search, sort
- splice, concatenate
Data Structures: “Containers”, e.g.

- lists, arrays, queues
- dictionaries, sets
- priority queues
- (strings)

Algorithms: Executing on the data structures mentioned above, e.g.

- iteration, scanning
- find, search, sort
- splice, concatenate
STL + Boost vs. LEDA

- Great part of functionality matches, but different naming schemes and division of functionality
- Certain specialties in both libraries

STL + Boost Particularities

- Surprisingly, no hash data type at all in neither STL specification nor Boost
  - However, common vendor extension

LEDA

- Extra: e.g. dynamic partitions of sets, hashes
- In general: many easily switchable variants for containers like dictionary and priority queue ⇒Æ
  - Different implementations came up because of insufficiencies Æ⇒
  - Extensive test results on different kinds of problems reported, good decision helper ⇒Æ
Functionality — Data Structures

STL + Boost vs. LEDA

- Great part of functionality matches, but different naming schemes and division of functionality
- Certain specialties in both libraries

STL + Boost Particularities

- Surprisingly, no *hash data type* at all in neither STL specification nor Boost
  - However, common vendor extension

LEDA

- Extra: e.g. dynamic partitions of sets, hashes
- In general: many easily switchable variants for containers like dictionary and priority queue ⇒Æ
  - Different implementations came up because of insufficiencies Æ⇒
  - Extensive test results on different kinds of problems reported, good decision helper ⇒Æ
Functionality — Data Structures

STL + Boost vs. LEDA

- Great part of functionality matches, but different naming schemes and division of functionality
- Certain specialties in both libraries

STL + Boost Particularities

- Surprisingly, no *hash data type* at all in neither STL specification nor Boost
  - However, common vendor extension

LEDA

- Extra: e.g. dynamic partitions of sets, hashes
- In general: many easily switchable variants for containers like dictionary and priority queue ⇒Æ
  - Different implementations came up because of insufficiencies Æ⇒
  - Extensive test results on different kinds of problems reported, good decision helper ⇒Æ
Commonalities

- **Basically the same for both library packages**
  - All operations reasonable for the data structures
  - Measures for enumerating elements of a container
  - Basic general algorithms like sorting, string algorithms

Differences

- In LEDA, algorithms operating on one object only are *member functions*
- In STL/Boost, algorithms are *independent* from the data structure
Commonalities

- **Basically the same for both library packages**
  - All operations reasonable for the data structures
  - Measures for enumerating elements of a container
  - Basic general algorithms like sorting, string algorithms

Differences

- In LEDA, algorithms operating on one object only are *member functions*
- In STL/Boost, algorithms are *independent* from the data structure
#### pure STL

```cpp
class print_int
{
public:
  void operator()(const int& i)
  {
    std::cout << i;
  }
};

void print_vector_stl()
{
  std::vector<int> v(10);
  for_each(v.begin(), v.end(), print_int());
}
```

#### Boost Lambda

```cpp
using namespace boost::lambda;

void print_vector_boost()
{
  std::vector<int> v(10);
  for_each(v.begin(), v.end(),
    std::cout << _1);
}
```

#### LEDA

```cpp
void print_vector_leda()
{
  leda::array<int> v(10);
  int i;
  forall(i, v)
    std::cout << i;
}
```

#### Comments

- STL has a lot of syntax overhead, Boost helps
- STL and Boost use functors, LEDA embraces functionality
**pure STL**

```cpp
class print_int
{
public:
    void operator()(const int& i)
    {
        std::cout << i;
    }
};

void print_vector_stl()
{
    std::vector<int> v(10);
    for_each(v.begin(), v.end(), print_int());
}
```

**Boost Lambda**

```cpp
using namespace boost::lambda;

void print_vector_boost()
{
    std::vector<int> v(10);
    for_each(v.begin(), v.end(),
        std::cout << _1);
}
```

**LEDA**

```cpp
void print_vector_leda()
{
    leda::array<int> v(10);
    int i;
    forall(i, v)
        std::cout << i;
}
```

**Comments**

- **STL has a lot of syntax overhead, Boost helps**
- **STL and Boost use functors, LEDA embraces functionality**
Small Own Performance Comparison on Sorting

Results (> is user-defined comparator)

- LEDA twice as fast as STL for integers (integer sorting)
- As expected: slowdown for large types and for user-defined comparators, between factor 2 and 5
### In general

- Very old report [Lauther 1992] says LEDA is four times slower than hand-coded implementations
- Lots of improvements for LEDA in the meantime, report outdated

### Search trees for integers [Dementiev 2004]

- LEDA’s (2,16)-trees win over STL for large inputs not fitting into cache
- LEDA’s (specialized) Van-Emde-Boas tree implementation slowest
- Better implementation presented, using LEDA’s memory manager $\mathcal{M}$

### Linked lists [Frias Moya 2006]

- LEDA’s implementation is a little faster than the STL’s one
- Surprisingly, there is a significant difference even in the most simple one data structure
Other Results Comparing the Performance of LEDA and the STL

In general

- Very old report [Lauther 1992] says LEDA is four times slower than hand-coded implementations
- Lots of improvements for LEDA in the meantime, report outdated

Search trees for integers [Dementiev 2004]

- LEDA’s (2,16)-trees win over STL for large inputs not fitting into cache
- LEDA’s (specialized) Van-Emde-Boas tree implementation slowest
- Better implementation presented, using LEDA’s memory manager $\mathcal{E}$

Linked lists [Frias Moya 2006]

- LEDA’s implementation is a little faster than the STL’s one
- Surprisingly, there is a significant difference even in the most simple one data structure
In general

- Very old report [Lauther 1992] says LEDA is four times slower than hand-coded implementations
- Lots of improvements for LEDA in the meantime, report outdated

Search trees for integers [Dementiev 2004]

- LEDA’s (2,16)-trees win over STL for large inputs not fitting into cache
- LEDA’s (specialized) Van-Emde-Boas tree implementation slowest
- Better implementation presented, using LEDA’s memory manager

Linked lists [Frias Moya 2006]

- LEDA’s implementation is a little faster than the STL’s one
- **Surprisingly, there is a significant difference even in the most simple one data structure**
Summary

- STL and Boost have little constant factors, algorithms independent from data structures $\Rightarrow \mathcal{AE}$
- LEDA has great algorithmic variety, many variants to choose from $\Rightarrow \mathcal{AE}$
- Use STL or Boost for simple/general cases, LEDA for advanced requirements/special cases
1 Introduction

2 Library Overview

3 Libraries as Building Blocks

4 Basic Design Goals and Paradigms

5 Fundamental Operations

6 Advanced Number Types

7 Basic Data Structures and Algorithms

8 Graph Data Structures and Algorithms

9 Computational Geometry

10 Conclusion
Libraries Considered

- Boost Graph Library (BGL)
- LEDA

Overview

- BGL is more flexible and better extensible
  - Generic adjacency matrix and list type
  - Algorithms are independent from data structures again
  - Only rely on concepts which must be modelled
  - Create own graph data structures compatible with algorithms easily \( \Rightarrow \mathbb{AE} \)
  - Adaptation to use important LEDA graph data type \( \Rightarrow \mathbb{AE} \)
  - Possibility to implement static graphs, implicit graphs (e.g. grid graphs)

- LEDA provides only two types, dynamic and static graph
  - Can be parameterized, static graphs also in terms of functionality
  - Algorithms only work on the graph data structures provided

- Main issue: How to associate additional information with nodes and edges: e.g. node color, edge weight
Graph Data Structures and Algorithms

Graph Data Structures Overview

Libraries Considered

- Boost Graph Library (BGL)
- LEDA

Overview

- BGL is more flexible and better extensible
  - Generic adjacency matrix and list type
  - Algorithms are independent from data structures again
  - Only rely on concepts which must be modelled
  - Create own graph data structures compatible with algorithms easily $\Rightarrow \mathcal{A}$
  - Adaptation to use important LEDA graph data type $\Rightarrow \mathcal{A}$
  - Possibility to implement static graphs, implicit graphs (e.g. grid graphs)

- LEDA provides only two types, dynamic and static graph
  - Can be parameterized, static graphs also in terms of functionality
  - Algorithms only work on the graph data structures provided

- Main issue: How to associate additional information with nodes and edges: e.g. node color, edge weight
Dynamic Graph

- Old, ancienct implementation
- "Fat interface": covers all functionality in one type
- Supports even combinatorial embedding of planar graphs
- Based on adjacency list implementation
- Huge memory consumption: $11n + 12m$ words
- Additional information via the "LEDA Solution"

Static Graph

- Quite new, fully generic, structure fixed after construction phase
- Subtypes functionally restricted: directed, bidirectional, opposite
- Much better performance for max-flow algorithms [Näher 2002] ⇒
- Less memory consumption

Summary: Static graph complete opposite of (dynamic) graph
### Dynamic Graph

- Old, ancienct implementation
- "Fat interface": covers all functionality in one type
- Supports even combinatorial embedding of planar graphs
- Based on adjacency list implementation
- Huge memory consumption: $11n + 12m$ words
- Additional information via the “LEDA Solution”

### Static Graph

- Quite new, fully generic, structure fixed after construction phase
- Subtypes functionally restricted: directed, bidirectional, opposite
- Much better performance for max-flow algorithms [Näher 2002] "⇒
- Less memory consumption

Summary: Static graph complete opposite of (dynamic) graph
Dynamic Graph

- Old, ancient implementation
- "Fat interface": covers all functionality in one type
- Supports even combinatorial embedding of planar graphs
- Based on adjacency list implementation
- Huge memory consumption: $11n + 12m$ words
- Additional information via the "LEDA Solution"

Static Graph

- Quite new, fully generic, structure fixed after construction phase
- Subtypes functionally restricted: directed, bidirectional, opposite
- Much better performance for max-flow algorithms [Näher 2002]
- Less memory consumption

Summary: Static graph complete opposite of (dynamic) graph
### Solution 1: Integrate In-Place

- No overhead through indirection
- Good cache locality
- Static: waste of memory when holding data for different algorithms
- Hard to access from generic algorithm

### Solution 2: External Storage

- Associative container (hash/array) allocated at time needed
  - Passable as parameters
  - Memory efficient
  - Runtime overhead through indirection

### Hybrid approach

Reserve *slots* to be filled with data dynamically
Node and Edge Data Solutions

Solution 1: Integrate In-Place

+ No overhead through indirection
+ Good cache locality
- Static: waste of memory when holding data for different algorithms
- Hard to access from generic algorithm

Solution 2: External Storage

- Associative container (hash/array) allocated at time needed
+ Passable as parameters
+ Memory efficient
- Runtime overhead through indirection

Hybrid approach

Reserve slots to be filled with data dynamically
Node and Edge Data Solutions

Solution 1: Integrate In-Place

+ No overhead through indirection
+ Good cache locality
- Static: waste of memory when holding data for different algorithms
- Hard to access from generic algorithm

Solution 2: External Storage

- Associative container (hash/array) allocated at time needed
+ Passable as parameters
+ Memory efficient
- Runtime overhead through indirection

Hybrid approach

Reserve *slots* to be filled with data dynamically
Both competitors support all approaches

- BGL has template metaprogramming solution to data access
  - Allows same algorithm to access either internal or external data
  - Without any overhead, decisions made at compile time
Common Functionality

- DFS, BFS: LEDA returns order or iterates, BGL calls back visitor(s)
- Random graph generation, topological sorting, shortest path, MST, transitive closure, (somehow) connected components, maximum flow, basic graph drawing algorithms, graph isomorphism

Add-Ons

- BGL: A*-search, very special orderings
- LEDA: min-cost flow, many matching variants, Euler tours
- LEDA for planar graphs: check, layout, s-t-numbering, triangulation
- LEDA: Graph isomorphism implementation features much more variants and improved algorithms \( \mathcal{AE} \Rightarrow \)

Summary: Much greater area covered in LEDA
Common Functionality

- DFS, BFS: LEDA returns order or iterates, BGL calls back visitor(s)
- Random graph generation, topological sorting, shortest path, MST, transitive closure, (somehow) connected components, maximum flow, basic graph drawing algorithms, graph isomorphism

Add-Ons

- BGL: A*-search, very special orderings
- LEDA: min-cost flow, many matching variants, Euler tours
- LEDA for planar graphs: check, layout, s-t-numbering, triangulation
- LEDA: Graph isomorphism implementation features much more variants and improved algorithms $\mathcal{A}E\Rightarrow$

Summary: Much greater area covered in LEDA
Graph Algorithms

Common Functionality

- DFS, BFS: LEDA returns order or iterates, BGL calls back visitor(s)
- Random graph generation, topological sorting, shortest path, MST, transitive closure, (somehow) connected components, maximum flow, basic graph drawing algorithms, graph isomorphism

Add-Ons

- BGL: A*-search, very special orderings
- LEDA: min-cost flow, many matching variants, Euler tours
- LEDA for planar graphs: check, layout, s-t-numbering, triangulation
- LEDA: Graph isomorphism implementation features much more variants and improved algorithms $\mathcal{A}\Rightarrow$

Summary: Much greater area covered in LEDA
Syntactic Differences

- Better flexibility in the BGL leads to
  - **Syntax without OO-style, rather procedural**
  - Macros and complicated template constructs needed on user side for customization
  - Calling the algorithms is almost the same

BGL

```cpp
add_edge(A, B, g);

std::vector<vertex_t> topo_order(num_vertices(g));
topological_sort(g, topo_order.rbegin(), color_map);
```

LEDA

```cpp
g.new_edge(A, B);

list<node> ts;
bool acyclic = TOPSORT(g, ts);
```
Graph Data Structures and Algorithms

Graphs: Summary

- LEDA: not extensible, huge collection of algorithms
- BGL: framework to be filled in, complicated syntax
- So far, LEDA static graphs provide excellent performance for specific algorithms
- Performance comparison proposed for the “Google Summer of Code”
What are Geometric Kernels

- Geometric objects: points, segments, lines, rays, planes, etc.
- Predicate functions: two segments intersect (y/n), sign of the triangle defined by three points, etc.
- Construction functions: the intersection point of two lines, etc.
- **Geometric kernel:** geometric objects + primitive functions (predicates and construction)

LEDA and CGAL: geometric kernels for 2D, 3D and higher dimensions
Introduction

What are Geometric Kernels

- Geometric objects: points, segments, lines, rays, planes, etc.
- Predicate functions: two segments intersect (y/n), sign of the triangle defined by three points, etc.
- Construction functions: the intersection point of two lines, etc.

**Geometric kernel:**

geometric objects + primitive functions (predicates and construction)

LEDA and CGAL: geometric kernels for 2D, 3D and higher dimensions
Problems in Computational Geometry

1. Degenerated inputs
2. Number instabilities (floating point rounding)

Possible Solutions

- (1) the code must consider all possible situations
- (2) easy if the underlying geometric kernel is exact
Problems in Computational Geometry

1. Degenerated inputs
2. Number instabilities (floating point rounding)

Possible Solutions

- (1) the code must consider all possible situations
- (2) easy if the underlying geometric kernel is *exact*
LEDAC Kernels

LEDAC History

1991: implementations assume “general position” inputs and rely on floating point arithmetic

⇒ Programs deliver wrong result and crash

- Revise old theory and develop theoretical backgrounds of exact geometry

1994:

- new exact rational kernel
- new robust implementations of geometric algorithms
Advanced Number Types

- LEDA rational kernel: `leda::integer` homogeneous coordinates
- LEDA real kernel: `leda::real` Cartesian coordinates

Floating Point Filtering

- LEDA exact kernels: try to compute correct result using floating point numbers, if fails use `integer/real \Rightarrow`
- Exacus: `integer \Rightarrow rational \Rightarrow real` (needed for curved geometry)
Advanced Number Types

- LEDA rational kernel: `leda::integer` homogeneous coordinates
- LEDA real kernel: `leda::real` Cartesian coordinates

Floating Point Filtering

- LEDA exact kernels: try to compute correct result using floating point numbers, if fails use integer/real $\not\Rightarrow$
- Exacus: integer $\Rightarrow$ rational $\Rightarrow$ real (needed for curved geometry)
Flexible CGAL Kernels $\Rightarrow \mathbb{A}$

- CGAL started in 1996 $\Rightarrow$ could rely on the improved template support
- Coordinates and coefficients: any type can be used (e.g. `leda::real`, GMP `mpz_t`, user type)
- Representation: choose Cartesian or homogeneous
- Floating point filtering: given kernel $K \Rightarrow$
  
  `CGAL::Filtered_kernel<K>` with *exact* and *fast* predicates

Why Floating Point Kernels

- Speed! (real time computer graphics, visualization)
- If not constructing new objects, FP kernels with filtering guarantee the correct flow control
Flexible CGAL Kernels

- CGAL started in 1996 ⇒ could rely on the improved template support
- Coordinates and coefficients: any type can be used (e.g. `leda::real`, `GMP mpz_t`, user type)
- Representation: choose Cartesian or homogeneous
- Floating point filtering: given kernel `K` ⇒ `CGAL::Filtered_kernel<K>` with exact and fast predicates

Why Floating Point Kernels

- Speed! (real time computer graphics, visualization)
- If not constructing new objects, FP kernels with filtering guarantee the correct flow control
Kernel Objects

- Non-modifiability of kernel objects (LEDAs and CGAL)

- **Reference-counting for “fat” kernel objects** (easier with immutable objects)
  - LEDA FP kernel is not ref-counted, others are ref-counted
  - CGAL: 2D and 3D in both variants, \(d\)-dimensional are ref-counted \(\Rightarrow\) \(\&(\)

Memory Management

- Specific pattern of (de-)allocation: many small objects (points, segments, circles, etc.)

- LEDA and CGAL use tuned allocators: free list allocator does a good job \(\&(\)

\(\&(\)
Kernel Objects

- Non-modifiability of kernel objects (LEDA and CGAL)
- **Reference-counting for “fat” kernel objects** (easier with immutable objects)
  - LEDA FP kernel is not ref-counted, others are ref-counted
  - CGAL: 2D and 3D in both variants, $d$-dimensional are ref-counted $\Rightarrow \mathcal{AE}$

Memory Management

- **Specific pattern of (de-)allocation: many small objects (points, segments, circles, etc.)**
- LEDA and CGAL use tuned allocators: free list allocator does a good job $\mathcal{AE}$
Polymorphy in CGAL

- CGAL::intersect can return a point, a segment, or empty
- Derive all geometric objects from one generic object:
  - Bad performance (space penalty and two indirections)
- CGAL uses Runtime Type Information RTTI to determine the type of returned object
## Differences between LEDA and CGAL Kernels

<table>
<thead>
<tr>
<th></th>
<th>LEDA</th>
<th>CGAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>number type</td>
<td>predefined</td>
<td>any</td>
</tr>
<tr>
<td>coupling with algorithms</td>
<td>yes</td>
<td>no (kernel is a template parameter)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⇒ tuning ⇒ ÀE</td>
</tr>
<tr>
<td>user kernels</td>
<td>no</td>
<td>yes (own geometric objects,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>primitive functions = geometric traits)</td>
</tr>
</tbody>
</table>
Algorithm and data structures implementations

- 2- and $d$-dimensional search: range, segment, interval trees, skip lists
- Point sets (Delaunay triangulations): update, search, closest point, range queries
- Polygon and polyhedra, LEDA: arcs are allowed
- 2/3/$d$D convex hull, boolean ops on polygons and polyhedra
- Segment and line intersection, Triangulations, Voronoi diagrams
- They are many! See the manuals
- LEDA and CGAL: several algorithms for a problem $\Rightarrow \mathcal{AE}$
- LEDA: many algorithms exist as a function and as a data structure (online updates)
- LEDA: algorithms return a list of objects
- CGAL: uses output iterator (STL approach)
Algorithm and data structures implementations

- 2- and $d$-dimensional search: range, segment, interval trees, skip lists
- Point sets (Delaunay triangulations): update, search, closest point, range queries
- Polygon and polyhedra, LEDA: arcs are allowed
- 2/3/$d$D convex hull, boolean ops on polygons and polyhedra
- Segment and line intersection, Triangulations, Voronoi diagrams
- They are many! See the manuals
- LEDA and CGAL: several algorithms for a problem $\Rightarrow AE$
- LEDA: many algorithms exist as a function and as a data structure (online updates)
- LEDA: algorithms return a list of objects
- CGAL: uses output iterator (STL approach)
Result Checking/Certifying in LEDA

Checking/Certifying

- Check if the output is correct ⇒ find errors early ⇒ AE
- Must be simple: correctness is evident
- Must be efficient: take less resources than the checked program

LEDA Geometric Checkers

- (Delaunay) triangulation
- Delaunay/Voronoi diagram
- planar graph
- …
Many functionality is common in CGAL/Exacus and LEDA. Benchmark?

[Schirra 99]: run the same implementations of convex hull algorithms
⇒ CGAL can be up to 2 times faster than LEDA (predicate inlining)

[Schirra 99]: Java-style kernels are 4-6 times slower than CGAL kernels (virtual functions)

[Näher 03]: LEDA Delaunay triangulation (Dwyer) faster than CGAL implementation (exact/inexact)

[Näher 03]: CGAL and LEDA 2D convex hull (FP kernel) are equally fast

Other studies wanted!
Many functionality is common in CGAL/Exacus and LEDA ⇒ benchmark?

[Schirra 99]: run the same implementations of convex hull algorithms ⇒ CGAL can be up to 2 times faster than LEDA (predicate inlining)

[Schirra 99]: Java-style kernels are 4-6 times slower than CGAL kernels (virtual functions)

[Näher 03]: LEDA Delaunay triangulation (Dwyer) faster than CGAL implementation (exact/inexact)

[Näher 03]: CGAL and LEDA 2D convex hull (FP kernel) are equally fast

Other studies wanted!
Algorithm libraries fasten the transfer of theoretical results into industry
⇒ greater impact of whole field
- LEDA, CGAL and STL/Boost are already used in industry
- More performance comparisons between libraries wanted

Future Work in the Field of
- Libraries for graph drawing, embedded graphs
- Libraries for bioinformatics
- Libraries for advanced machine models
  - Library for cache-oblivious algorithms (not yet existent)
Acknowledgments
- Thanks to Stefan Schirra and Peter Sanders

Discussion
- Questions?
- Suggestions?
- Distinction between libraries/software systems/frameworks? What is CPLEX, for example?
- Provocative Question:
  Must libraries be free / open / open-source to be respected in science?