Reuse of Algorithms: Still a Challenge to Object-Oriented Programming

Karsten Weihe
Universität Konstanz, Fak. Mathematik und Informatik
Fach D188, D-78457 Konstanz, Germany
karsten.weihe@uni-konstanz.de

Abstract

This paper is about reusable, efficient implementations of complex algorithms and their integration into software packages. It seems that this problem is not yet well understood, and that it is not at all clear how object-oriented and other approaches may contribute to a solution. We analyze the problem and try to reduce it to a few key design goals. Moreover, we discuss various existing approaches in light of these goals, and we briefly report experiences with experimental case studies, in which these goals were rigorously addressed.

1 Introduction

Many sophisticated algorithms for solving various problems have been proposed in the literature. These include, for example, algorithms for numerical computations, for graph and network problems, for computer graphics, and for symbolic computations.

Implementing such an algorithm is time consuming and prone to error and requires expert knowledge in algorithmics. Since algorithmic problems appear quite frequently, libraries of algorithms for solving various problems might be useful. However, to be widely usable, the components of such a library must meet two requirements: efficiency and flexible adaptability.

Efficiency must in no way be disregarded, because algorithmic software is often time-critical. For example, efficient algorithms are indispensable for real-time programming. In the other extreme, algorithmic software is often applied to large-scale problems, where a single run of an algorithm may take hours, days, or weeks (even when executed massively in parallel). In engineering and operations research, the run time of the algorithmic software is sometimes crucial for the time of the whole developmental or decision process. Many algorithmic problems cannot be solved exactly, for example, numerical and geometric problems on real numbers and $\mathcal{NP}$-hard discrete problems. If such an algorithm is critical for a development or decision process, which is usually subject to strict deadlines, a loss of efficiency results inevitably in a loss of accuracy. Lack of algorithmic efficiency also has an impact on user interfaces: often the response time is dominated by complex graphical algorithms. Hence, efficiency of algorithms is necessary to overcome the "temporal usability problem [16]."

On the other hand, flexible adaptability is necessary to customize the implementation of an algorithm to various specific applications. It is the author's subjective impression that this problem is not yet well understood and often underestimated. The following design goals are an attempt to summarize the properties that make implementations of algorithms truly reusable.

Key design goals:

1. An algorithm should be implemented such that it can be easily adapted to existing and application-specific realizations of the underlying abstract data types without sacrificing efficiency.

2. The interfaces of data structures to algorithms should be designed such that a small set of generic or polymorphic classes makes the implementation of such interfaces easy, convenient, and straightforward.

3. An algorithm should be implemented such that its *algorithmic functionality* is flexibly adaptable.

4. An algorithm should be implemented such that it is possible to inspect all potentially relevant details during its execution.

1.1 Overview

Each of the following sections is devoted to one of these goals. All goals are explained and discussed in detail. This includes a discussion of various existing design methodologies that may be relevant for the respective goal. In that, the focus is on object-oriented methodologies. Each section is concluded by a discussion of the experiences that we gained from practical case studies in C++. Ref. [23] and the technical reports [21, 22, 30] describe selected details of the implementations.

We focus on a specific area, graph algorithms, because a discussion that covers various algorithmic domains and is still sufficiently concrete would exceed the limits of this

OOPSLA '97 10/97 GA, USA

paper. However, analogous arguments apply to other algorithmic areas as well.

A few concrete examples presented in the following sections are taken from projects in which the author of this paper has been involved during the last years [13, 20, 29, 31]. A certain number of further examples are taken from the *L*ibrary of *E*fficient *D*ata types and *A*lgorithms (*LEDA* [25, 27]), which is written in C++. Additional examples are taken from the *S*tandard *T*emplate *L*ibrary (*STL* [28]), which greatly influenced the prospective definition of the C++ standard library and the design of many other libraries. We also refer the reader to the libraries CGAL [6], ffGraph [9], Karla [18], and the AAI base class library [1].

We focus on LEDA to some extent, because LEDA matches the topic of this paper quite well: it is especially devoted to graph algorithms, and maximum efficiency and ease of use were the primary design goals (and have essentially been achieved). The design of LEDA uses various object-oriented and generic features of C++ and might be representative for many other packages. LEDA is mature and used by many groups both in academics and in the software industry. It is applied in various domains, and in some areas it has become a de-facto standard.

1.2 Further Reading

Several books (*e.g.* [5, 14, 33]) present implementations of basic and advanced algorithms in object-oriented and hybrid programming languages, most of them in C++. However, only a few publications address the specific design problems inherent in algorithmic software. Barton and Nackman [3] demonstrate how the object-oriented and generic features of C++ may be used to achieve a better structure for algorithmic code in scientific and engineering computing. Flamig [10] implements algorithms as classes to let them act like input streams (which he calls *algorithmic generators*). Soukop [35] incorporates a few advanced algorithmic examples in his general design discussions (notably section 2.6).

In a concrete case study in Object Pascal (*maximum-flow problem* [2]), Gallo and Scutella [11] apply the *strategy* pattern [12] to keep subalgorithms interchangeable.

Holland [15] implements algorithms as frameworks, which can be adapted to a specific application by subclassing. VanHilst and Notkin [37] re-implement this design using C++ templates.

Several articles discuss extensions to existing mainstream languages, which are intended to support design and implementation of efficient, reusable library components. For example, Biggerstaff [4] discusses the problem to design efficient, reusable libraries such that the number of components remains manageable. He argues that the features of "conventional, mainstream programming languages" are not sufficient for that, and he surveys and discusses software development systems which add extra-linguistic expressibility to the language.

Yu and Zhuang [39] concentrate on efficient, reusable algorithms and propose an extension to C++ (named *kinds*). In Yu and Zhuang's own words, this extension is intended to "realize algorithmic abstraction" and to close the "abstraction gap between algorithms and procedures."

Further publications are concerned with implementations of graph algorithms in functional languages [8, 19, 24].

We will discuss selected approaches from the literature in greater detail in the relevant sections. Moreover, we will discuss the relevance of various design patterns [12] to the individual goals. Here we are interested in the general ideas of these patterns, not in the concrete realization of the interplay between the collaborators. For example, this realization may be based on inheritance as in [12], but also on generic features.

2 Adaptation to Data Structures

> **Goal 1**
>
> An algorithm should be implemented such that it can be easily adapted to existing and application-specific realizations of the underlying abstract data types without sacrificing efficiency.

The discussion will be grouped into two variations of this goal: *different implementations and variations of one abstract type* and *different implementations of the item parameters.*

In the following, the term *abstract type* refers to general concepts such as *directed* or *undirected* graphs. For instance, figures 1(a) and 1(d)–(e) are examples of the abstract type *undirected graph*, and figures 1(b)–(c) of the abstract type *directed graph*. More specifically, figures 1(c)–(e) show examples of specializations of these abstract types: symmetric directed graph, plane graph (a graph is *plane* if it can be drawn such that no two edges cross each other), and grid graph. In turn, grid graphs are a specialization of the specialized abstract type *plane graph*.

2.1 Different Implementations and Variations of One Abstract Type

Different implementations

Typically, an abstract data type such as a directed graph may be implemented in many different ways. Goal 1 means that an implementation of an algorithm should be independent of this concrete implementation.

On one hand, it is crucial for time-critical applications that an algorithm can be adapted to data structures which are tuned for efficiency. This tuning usually varies from application to application, and thus requires a high degree of flexibility. On the other hand, an implementation of an algorithm can be integrated into existing software only if it can be adapted to the data structures on which this software is built. These data structures might have been developed a long time ago and designed to meet absolutely different, non-algorithmic, needs. Clearly, in such a case, the algorithm cannot achieve its best possible run time, because the data structures have not been designed for that. However, the algorithm's interface should be flexible enough so that—within the bounds of possibility—a reasonable degree of efficiency can be achieved.

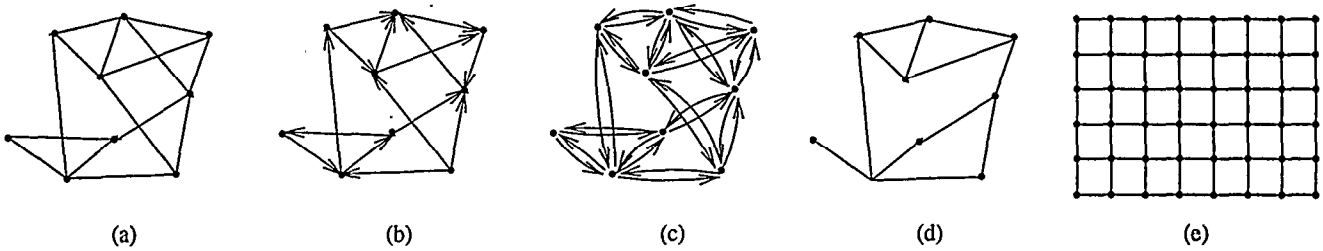

Figure 1: (a) an undirected graph, that is, edges are not oriented; (b) a directed graph constructed from (a) by assigning arbitrary orientations; (c) the directed graph constructed from (a) by substituting a pair of mutually opposing directed edges for each undirected edge; (d) a plane undirected graph, embedded without crossings; (e) a grid graph.

Different abstract types

Moreover, many graph algorithms apply to a broad range of abstract graph types. For example, for many algorithms, it does not matter whether the graph is directed or undirected. Likewise, the same algorithm may apply to *hypergraphs* (*i.e.* an edge may connect more than two nodes) and other, similar variations. Goal 1 also means that an implementation of such an algorithm should work with every abstract graph type to which the algorithm applies "in theory."

Alternatives

There are two alternatives to goal 1: either each algorithm is implemented on its own, specific interface, and the data is converted back and forth for each call of an algorithm; or a standard is defined on which the implementations of all algorithms are based.

Alternative I (*conversions*): very often, the first alternative is simply infeasible in algorithmic software design, because the amount of data to be converted is too large. This problem is aggravated by the fact that algorithms are usually not (and should not be) implemented monolithically, but composed of other, more basic algorithms, which are, in turn, composed of even more basic algorithms, and so on. Such a decomposition would result in a large number of expensive conversions. Quite often, the run times of the most basic algorithms are sublinear in the size of the data structures (at least in an empirical sense). In this case, the run time is not even proportional to the theoretical efficiency.[1]

Alternative II (*standardized abstractions*): Many existing libraries for graph algorithms (incl. LEDA) adopt the second alternative, which is to define a class for graphs and to implement all algorithms on top of this interface. However, experience suggests that the following undesired consequence might inevitably occur when a library is actually used in real applications in different fields: from release to release, the class is equipped with more and more functionality to fulfill the users' demands; the implementation thus becomes less and less efficient, but without the ideal state ever being reached.

It is essential to note that this missing functionality is not only a problem of "exotic" applications, which means that all "mainstream" applications would be satisfied by a well-chosen subset. In fact, even the requirements imposed by mainstream applications differ in many details.

For example, many algorithms require an access method that gets two nodes as arguments and returns the edge connecting these nodes. In principle, this access can be realized efficiently in two ways: either as a matrix whose columns and rows are nodes and whose entries are edges, or as an associative array with pairs of nodes as keys and edges as the associated units of information. However, the overhead in space or run time caused by such a solution may not be acceptable for an algorithm that does *not* require this particular functionality.

The potential present and future variations might not be predictable. Hence, a common generalization of all variations is not possible (even if it were possible, it might not be efficient). In summary, all of these variations should be realized by different classes with different interfaces.

> **Experience 1**
>
> The alternatives to goal 1 do not suffice to achieve a high degree of flexibility without sacrificing efficiency.

2.2 Different Implementations of the Item Parameters

A graph may be viewed as a composition of containers. For example, a directed graph class that is intended for various purposes often consists of a container of nodes, a container of edges, and for each node, a container of references to the edges leaving this node. Further containers may be included, for instance, for each node, a container of references to all edges entering this node.

The items of a container may be assigned *item parameters*. For example, the nodes of a graph are usually assigned parameters such as names, flags, and coordinates, and likewise, the edges are assigned parameters such as lengths, costs, and capacities. Goal 1 states that the concrete organization of these item parameters shall be hidden from each algorithm.

[1] For example, effects like these occurred in a project in which the author of this paper was involved a few years ago and which used conversions to a common format to integrate various algorithms for scheduling problems into one package [20].

From an abstract viewpoint, the values of all item parameters for all items form a table: there is a row for each item and a column for each parameter, and the value of a parameter for an item is the entry at the table slot specified by this row and this column. Many algorithms insert and remove items, which means that the set of rows is usually dynamic.

Dynamic column set

It is essential to note that the set of columns should also be dynamic to some extent. For example, many algorithms rely on auxiliary parameters, which are meaningless for all other parts of the program. A concrete example is the "*seen*" label of nodes inside an algorithm that constructs a path from some node s to some node t using a depth-first or breadth-first search: s and t are the input, the path is the output, and the label is internally used to ensure that no subgraph is searched more than once.

In some cases, it may be preferable to instantiate such an auxiliary parameter only temporarily, immediately before invoking the algorithm, and to drop the parameter after termination of the algorithm. For the sake of argument, suppose that a program applies algorithms $A_1, \ldots, A_k$ one after another to the same graph and each algorithm A_i requires a different set of auxiliary parameters. If the set of node and edge parameters is organized statically (*i.e.* fixed once and for all at compile time), we have to maintain the union of these parameter sets throughout the program, which causes a waste of space and additional run-time overhead. The same effect occurs when $A_1, \ldots, A_k$ are not called after each other, but are made exchangeable using the *strategy* pattern [12]: here only one of $A_1, \ldots, A_k$ is invoked, and the algorithm A_i to be invoked is chosen only at run time. Hence, if all node and edge parameters are static, all parameters of all algorithms must be unnecessarily maintained.

Non-materialized parameters

Even more, in some applications, it might not even be reasonable to store certain node and edge parameters in any way. For example, consider the case that the nodes of a graph are points in the plane and the length of an edge is the normal (*i.e. Euclidean*) distance of its nodes. The number of edges is potentially quadratic. Hence, it may be preferable not to store the lengths of the edges explicitly, but to compute the length of an edge from the node coordinates when needed.

Data organization

The abstract view of item parameters as a table suggests two ways of organizing all data: row-wise and column-wise.

Row-wise organization. in principle, this means that every item is attached a tuple which comprises all item parameters. This organization is supported by many graph libraries.

For example, in LEDA [25] and in the AAI base class library [1], certain node and edge types are generic.[2] Hence, a static set of node and edge parameters can be realized by instantiating the node and the edge type with records which comprise all item parameters.

In contrast, the graph class in the ffGraph library [9] provides a means of adding and removing node and edge parameters (called *labels*) at run time. Each parameter gets a unique ID. Roughly speaking, each node (resp. edge) maintains an associative array of parameters, and the set of IDs in this associative array is the same for all nodes. To allow parameters of different types, each parameter must be a subclass of a prescribed base class for node (edge) labels. This row-wise approach is fully dynamic, however, at the cost of efficiency and type safety.

In addition to the generic features mentioned above, the AAI graph also supports user-defined extensions of the node and edge types: the node and the edge type are designed to serve as base classes, and the graph class only handles pointers and references to nodes and edges. Thus, it is possible to create graphs which handle extended node and edge classes. In particular, this allows the integration of a user-defined row-wise implementation of node and edge parameters.

Column-wise organization. For each item parameter, an associative array is instantiated, which contains the values of this parameter for all items. Each item has a unique ID, which can be used to access its value in this array. Instantiating a new item parameter at run time amounts to instantiating a new array. In contrast to the row-wise case, this concept is dynamic *and* fully type-safe.

For example, the LEDA graph class comes with so-called *node* and *edge arrays*. A node or edge array is an associative array with a static index set, which is the set of nodes or edges at the moment when the array is instantiated. Thus, LEDA allows an arbitrary mixture of a row-wise and a column-wise organization.

However, a normal array or an associative array with a static index set becomes invalid once items are inserted or removed. Hence, such an array is of limited use. On the other hand, an associative array with a dynamic index set might increase the run time significantly, even by more than a constant factor. Moreover, when items are inserted or removed, the corresponding associative arrays should be updated automatically, which requires an expensive solution based on the *observer* pattern [12].

Discussion

In summary, neither a row-wise nor a column-wise organization (nor any other puristic organization) is sufficient for all applications. Hence, to achieve goal 1, an implementation of an algorithm must be able to cope with different ways of organizing the item parameters in the underlying containers.

Very often, a mixed solution may be suitable, in which the records attached to items hold the permanent data, and for temporary data, additional arrays are instantiated when needed. However, whether a certain parameter is permanent or temporary depends on the context, and hence an algorithm cannot assume a specific constellation. Moreover, even when such a mixed solution is applied, the question of non-materialized data remains open.

It seems that the problem of organizing the item parameters in a container is often underestimated or even overlooked. For example, the design of the STL does not address this problem at all [28]. The items of an STL-style container are accessed through *iterators* [12]. However, the

[2] To be precise, certain graph classes in LEDA are generic, and the generic type parameters are the information types of nodes and edges.

syntactic requirements for iterator classes in STL contain only one method for accessing an item's data (the C++ dereferencing operator, `operator*`).

Obviously, the design of the STL only considers containers such that each item is the (only) item parameter itself. In fact, all algorithms in STL assume this scenario. Hence, if there is more than one item parameter, the STL algorithms are not easily applicable (see the example below; STL-function `replace`).

As mentioned above, a typical graph algorithm accesses several node and edge parameters, which may be permanent or temporary. Hence, the problem of organizing the item parameters is addressed by designers of graph libraries. We have analyzed some of the solutions in the above discussion of row-wise and column-wise schemes.

The solutions adopted for LEDA are certainly the most efficient implementations of row-wise and column-wise data organization. This allows one to choose the most efficient intermixed solution for a specific application. Hence, to achieve goal 1, an implementation of an algorithm must be able to cope with both approaches and with mixed solutions.

However, all graph algorithms in LEDA assume that the node and edge parameters are organized column-wise, although this is often not the method of choice. The fact that the organization of node and edge parameters is hard-wired in these algorithms shows that data abstraction has still not been achieved. However, it is not surprising that a column-wise organization is preferred over a row-wise one: C++ does not provide any feature for renaming the members of a record. Therefore, in each algorithm, the names of all accessed parameters had to be hard-wired, when a row-wise organization was adopted. This is impractical because of potential name conflicts.

Experience 2

The concrete organization of the item parameters may be completely hidden from the algorithm by *data accessors* without significant loss of efficiency.

We introduced the concept of *data accessors* in [23]. Roughly speaking, a data accessor is a class that is responsible for the access to a single item parameter. The type of the item parameter is a type tag `vtype` of the data accessor.

A data accessor class provides one or more methods named `get`, which gets a handle or iterator type for items as an argument and returns the parameter value for the item identified by the argument. The method `get` is overloaded for every relevant handle and iterator type. More specifically, if `It_1...It_n` are the relevant iterator classes, the data accessor class must conform[3] to the following C++ class definition:

```
class AnyDataAccessor
  {
  public:
    typedef ... vtype;
    vtype get (It_1) const;
    ...
    vtype get (It_n) const;
  };
```

Since `get` is overloaded, an algorithm only needs one data accessor for each item parameter, no matter how complicated the underlying graph data structure is. The concrete examples below will demonstrate that `get` is usually not overloaded "by hand," but by genericity.

We distinguish between *read* and *read/write* accessors. A *read/write* accessor provides an additional method `set` for each relevant handle and iterator type `It`:

```
void set (It, vtype);
```

This method overwrites the current parameter value by the value of the second argument.

Example

This is a simplified version of the running example in [23]. Consider the STL-function `replace` [28]. This function runs over a linear sequence of items of the template type `T` and replaces all occurrences of the value `old_value` by `new_value`. The linear sequence is given by two iterators, `first` and `sentinel`. Iterator `first` identifies the beginning of the sequence, and `sentinel` is a past-the-end marker (*e.g.* the Null value if the sequence is a linked list and the iterator is a mere pointer).

```
template <class It, class T>
void replace
       (It first, It sentinel,
        T old_value, T new_value)

  {
    for (It it=first; it!=sentinel; ++it)
       if ( *it==old_value )
          *it = new_value;
  }
```

For instance, this function `replace` is applicable if we have a sequence `salaries` of STL-type `list<int>`, which contains the salaries of all employees, and we want to replace every occurrence of a $1,000 salary by $2,000:

```
replace (salaries.first(),
         salaries.last(),
         1000, 2000);
```

[3] For conformance, we make no distinction as to whether the arguments and the return value of a method are values or constant references. In the following code fragments, we will generally disregard this difference.

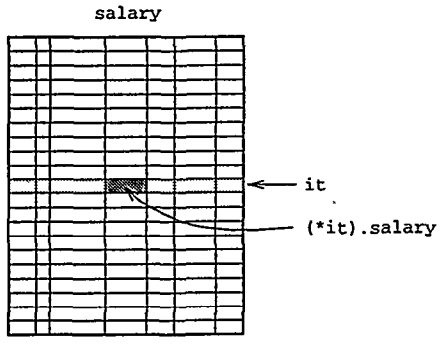

Figure 2: the scenario in the STL version of replace. A reference to the grey row is returned by the method operator* of the iterator. The row must exist as a materialized object of a record type.

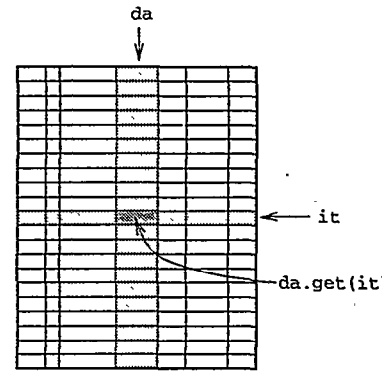

Figure 3: the modified scenario when data accessors are used. The iterator accesses the grey row, and the data accessor, the grey column. The individual columns may be organized arbitrarily (not necessarily materialized).

However, now assume that we have a struct type `Employee`, which contains an employee's salary as a member `salary`, and we want to perform the same update of salaries on a `list<Employee>`. The `operator*` of a list iterator returns the whole struct (see figure 2). To apply the above implementation of `replace`, `It` must be an adapter [12] for iterators, which overwrites `operator*` to drop all parameters but one. However, this is inconvenient when implementing algorithms which access more than one item parameter, because usually, what one wants is to access *all* parameters of an item through *one* iterator. If this is not possible, an algorithm must handle one iterator for each item parameter, which induces consistency problems.

A revised version of `replace`, which relies on a data accessor `DA` for accessing the salary, is flexible enough. We use the type tag `vtype` to drop the template parameter `T`.

```
template <class It, class Da>
void replace
      (It first, It sentinel, DA da,
       typename DA::vtype old_value,
       typename DA::vtype new_value)

  {
    for (It it=first; it!=sentinel; ++it)
       if ( da.get(it)==old_value )
          da.set (it, new_value);
  }
```

Figures 2 and 3 illustrate the difference.

The following class template `MemberAccessor` solves our exemplary problem. The type of the member `salary` appears as template argument `T`, the struct type `Employee` as template argument `Str`, and the concrete member `salary` as an argument of the constructor (namely as a pointer-to-member). Hence, this template covers all scenarios in which all relevant iterator types are STL-compliant, `operator*` returns a (reference to a) struct type, and the item parameter is a member of this struct type.

```
template <class Str, class T>
class MemberAccessor
  {
  public:
    typedef T vtype;
    MemberAccessor (T Str::*ptr)
          : i_ptr(ptr) { }
    template <class It>
       vtype get (It it) const
          { return (*it).*i_ptr; }
    template <class It>
       void set (It it, vtype value)
          { (*it).*i_ptr = value; }
  private:
    T Str::*i_ptr;
  };
```

Using this class template, we can update all salaries as follows:

```
MemberAccessor<Employees,int>
          acc (&Employees::salary);

replace (our_employees.first(),
         our_employees.last(),
         acc, 1000, 2000);
```

This concludes the example.

Pure read accessors may be used to encapsulate non-materialized parameters. The following class template, `ConstAccessor`, is a simple example: it uniformly returns the value received through the constructor.

```
template <class T>
class ConstAccessor
  {
  public:
     typedef T vtype;
     ConstAccessor (T t)
           : i_t(t) { }
     template <class It>
        T get (It) const
           { return i_it; }
  private:
     T i_t;
  };
```

3 Toolboxes for Interfaces

> **Goal 2**
>
> The interfaces of data structures to algorithms should be designed such that a small set of generic or polymorphic classes makes the implementation of such interfaces easy, convenient, and straightforward.

For a simple exposition, assume that there are n graph algorithms and m data structures for graphs, and each algorithm shall work on each data structure. This requires $n \cdot m$ customizations of algorithms to data structures. Hence, it might be useful to have a toolbox of generic or polymorphic classes from which these adapters may be constructed with a relatively modest programming effort. In the ideal case, tailoring an algorithm to a data structure would then amount to instantiating these polymorphic or generic types with the appropriate parameters.

However, this introduces a tradeoff: on one hand, the toolbox should be small and coherent, which significantly restricts the freedom in designing interfaces. On the other hand, a large degree of freedom is necessary to adapt an algorithm to various applications without sacrificing efficiency.

STL viewed as a toolbox

Goal 2 is one of the main design goals of the STL. The most important sort of generic classes in the STL are iterators: each container class comes with a couple of specific iterator classes, and each algorithm accesses the underlying containers solely through these iterators. Containers and algorithms can be easily combined, because the iterator classes are template parameters of the algorithms, the interfaces of all iterator classes conform to a set of *requirements*, and the algorithms access iterators only according to these requirements.

Of course, not every combination of an algorithm and a container makes sense. For example, it does not make sense to sort a singly-linked list by the usual random-access variant of quicksort. Therefore, containers are classified into five groups according to their potential functionality: input stream, output stream, singly-linked (forward) sequence, doubly-linked (bidirectional) sequence, and random-access container. For each of these groups, there is a specific set of requirements, and these sets form a hierarchy as shown in figure 4.

> **Experience 3**
>
> At least in C++, it is possible to implement toolboxes which realize goal 2 and do not sacrifice efficiency. As a prerequisite, the interface of a complex data structure to algorithms is not realized by a single, large class, but by a couple of *light-weight classes.*

We have implemented an experimental toolbox for graph algorithms. This toolbox is written in C++. [23] and the technical reports [21, 22] address the basic ideas and crucial details of the implementation. A restricted version, which is less flexible but specifically adapted to the LEDA data types, is described in [30]. This version will be integrated into LEDA.

Our toolbox consists solely of small, light-weight types. Each of these types is responsible for a specific aspect of the data structure. The most important types in our toolbox are iterators and data accessors.

Handle/iterator hierarchy

Iterators are provided for all relevant ways of iterating over a graph: over the whole set of nodes, the whole set of edges, and the set of edges incident to a single node. The essential details are summarized in [21]. In principle, we follow the guidelines for iterators in STL [28]. However, for reasons discussed in [21], our iterators deviate from the STL conventions in several technical details, which are beyond the scope of this paper.

As a result of our practical case studies, the toolbox offers each of these iterator types in a *normal* and a *reduced* version,[4] and in addition node and edge *handle* types (figure 5). The main difference between normal and reduced iterators is that a normal iterator must provide certain methods which require that the iterator object has some "global knowledge" about the underlying graph. In contrast, an object of a reduced iterator type is only assumed to have knowledge about its specific type of iteration. Finally, a node or edge handle type refers to a single, fixed node or edge and is not assumed to "know" anything about the underlying graph or about any way of iteration.

Often, the reduced version of an iterator class is smaller than the normal version, and the corresponding handler class is even smaller. Hence, if an algorithm manages large containers of iterators, using the reduced version or even

[4] Called the *heavy* and the *light* version in [21].

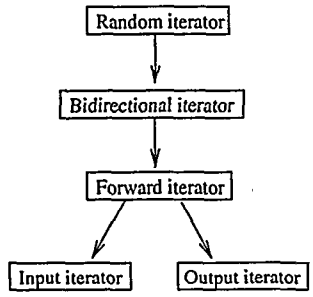

Figure 4: The STL hierarchy of iterator requirements. An arrow indicates that the requirements on the tail side are a superset of the requirements on the head side.

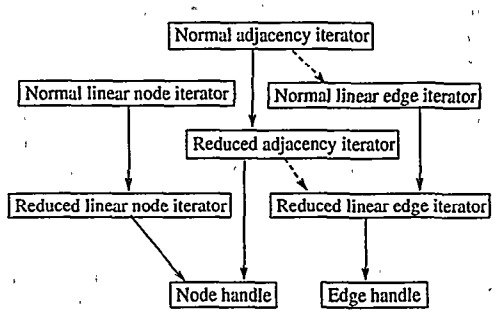

Figure 5: The hierarchy of requirements for iterators on graphs in our toolbox. A dashed line indicates that the relation between the requirements on the head and the tail side are *only syntactical* in nature, but the semantics may be different (to be concrete: iterating over all edges of the graph vs. over all edges leaving a single node).

the handle class may save space and decrease the run-time overhead induced by copy operations.[5]

On the other hand, the reduced version often gets the global knowledge required for the normal version "for free" from the underlying graph data structure. For example, each node and edge in the graph data structure in LEDA and AAI holds a pointer to the graph to which it belongs. Hence, a mere pointer to the node or edge type suffices as a normal iterator, and the normal and the reduced versions collapse into one class. Nonetheless, an algorithm may distinguish between the normal and the reduced version of an iterator class. If the algorithm is generic with these two classes being type parameters, both type parameters are then instantiated with the same type.

Proxies

Moreover, the toolbox provides several generic proxies [12] for iterators, data accessors, and other types.

Example. the following class template, named `Filter`, is a simple, small proxy for iterators.[6] It constructs a *filter iterator* from an object `it` of iterator type `It` and from an object `pred` of predicate type `Pred`. This iterator skips all items which do not fulfill the predicate. As in STL, the predicate is evaluated in function notation (`operator()`). For notational convenience, we do not follow the STL guidelines for iterators: the sentinel is dropped, and instead the Boolean method `valid` returns `false` if and only if the end of the sequence has been passed. Moreover, `advance` performs one forward step, and `get_data` returns the data associated with the current item.

```
template <class It, class Pred>
class Filter
  {
  public:
     typedef typename It::vtype vtype;
     Filter (It it, Pred pred)
        : i_it(it), i_pred(pred)
        { advance_if_false(); }
     bool valid () const
        { return i_it.valid(); }
     vtype& get_data ()
        { return i_it.get_data(); }
     void advance ()
        {
          i_it.advance();
          advance_if_false();
        }
  private:
     It    i_it;
     Pred  i_pred;
     void advance_if_false ()
        {
          while (i_it.valid() &&
                 !i_pred(i_it) )
            i_it.advance();
        }
};
```

Programming languages

We apply many advanced features of C++ to achieve this degree of flexibility and convenience. For example, we use many features of the template mechanism in C++, which is much more flexible than the generic mechanisms in many other languages. Moreover, pointer-to-members [36] are used in our toolbox; see the example `MemberAccessor` in section 2.2.

Of course, the design of our toolbox can be realized in other programming languages, because a flexible generic mechanism and pointer-to-members may be simulated by

[5] To our surprise, it has turned out that simple copy operations may cause a significant overhead. Applying reference semantics to reduce this effect is often problematic in a language such as C++, because there is no built-in garbage collector, and techniques such as reference counting may cancel out the savings.

[6] Recently, we have successfully applied generic iterator proxies like this to the internal design of a database query engine [13].

inheritance. In fact, an effort is currently being undertaken by another team in our group to implement a variant of our toolbox in Java and to apply it to the visualization of social networks [38]. As a nice by-product, our concept allows the adaptation of graph algorithms to a graphic-oriented data structure, which supports editing and layout facilities. This removes the need for maintaining redundant information in *two* data structures: one for graphics and one for graph algorithms.

However, it seems that the features offered by the current definition of Java inevitably result in weaker static type checks and less convenience. For example, we did not find a surrogate for the class template `MemberAccessor` (section 2.2) which is equally general and type-safe (not to mention efficient).

4 Algorithmic Functionality

> **Goal 3**
>
> An algorithm should be implemented such that its *algorithmic functionality* is flexibly adaptable.

Goal 3 addresses many aspects of the design of algorithms. First we will briefly illustrate two aspects: *exchangeable subalgorithms* and *variations of the output*. Afterwards, we will start a more detailed discussion of yet another aspect: *heuristic speed-up techniques*.

Exchangeable subalgorithms

To give a concrete example, many graph algorithms rely on a subalgorithm which determines a path from some specified node s to some other node t. This subalgorithm might start one of various kinds of graph searches at s, for instance, a depth-first search or breadth-first search, or a shortest-path algorithm, which determines the cheapest path with respect to an arbitrary definition of edge costs. Goal 3 states that the concrete strategy for determining this path should be left open in the implementation of the main algorithm. This is necessary to achieve both overall goals—efficiency and reusability—simultaneously, because none of these path algorithms is efficient in all possible applications. Furthermore, some applications even rely on specific properties of the path, which are only guaranteed by specific subalgorithms.

The *strategy* pattern [12, 11] seems to be appropriate to meet this goal. However, the situation is even more complex, and it does not suffice to base the design solely on the *strategy* pattern. The problem is that the usual algorithms for depth-first search, breadth-first search, and shortest paths determine spanning trees, not paths. For reasons of efficiency, it does not suffice to run one of these algorithms up to its regular termination and to extract the path afterwards from the tree. Rather, the algorithm should terminate once the path from s to t is determined. Therefore, these algorithms must be adapted to the problem of finding a single path.

Variations of the output

A depth-first search or breadth-first search algorithm may either return a mere labeling of all visited nodes or additional information such as the traversal order. Moreover, it may also return the tree induced by the search, which comprises all visited nodes and all traversed edges. This tree may be encoded as a 0/1-labeling of the edges, as an (acyclic) object of the graph class, as an object of a specific tree class, or whatsoever.

Many libraries (incl. LEDA) provide implementations of depth-first search and similar algorithms which only return node labelings or the traversal order of all nodes. Constructing a depth-first tree from this information amounts to performing another complete depth-first search. This simple example demonstrates that the concrete variation of the output cannot always be delegated to postprocessing routines. Consequently, this flexibility should be inherent in the implementation of the algorithm itself.

Speed-up techniques

As mentioned above, we discuss this aspect in greater detail. We illustrate this task by means of an example: we are given a traffic network, that is, the nodes are cities and towns, and the edges are traffic connections. The system receives a potentially infinite number of pairs (s, t) of nodes as run-time events. For each pair, it has to compute a shortest path from s to t in the network (see figure 6). This is a special case within the wide range of algorithmic problems that are best solved by Dijkstra's famous algorithm ([7], sect. 25). Our concrete scenario offers a wealth of algorithmic speed-up techniques. If t is often in the vicinity of s, the speed-up factor gained from these techniques may be in the order of 1,000 to 10,000. Unless stated otherwise, s is the node from which the search starts.

Early termination. The first, most basic speed-up technique is to let the algorithm terminate immediately when the shortest path from s to t is found. Actually, Dijkstra's algorithm computes shortest paths from s to *all* other nodes. Unfortunately, the typical implementations of Dijkstra's algorithm in libraries [25] and in the literature [10] do not even allow this simple speed-up technique.

Version stamps. In combination with the technique *early termination*, the following trick can be applied to achieve expected sublinear run time for a single shortest-path computation: the node distances are not initialized with a value representing $+\infty$ in every shortest path computation. Instead, a version stamp is maintained for every node, which is the number of the last shortest path computation in which the distance of this node was updated. If the version stamp of a node is not up to date, this node is regarded as having infinite distance (clearly, to avoid an integer overflow, all version stamps must be reset to the initial value after a—huge—number of steps).

Goal-directed search. The length $\ell(v, w)$ of an edge $(v, w) \in E$ is replaced by the reduced length $\ell'(v, w) := \ell(v, w) - \Delta(v, t) + \Delta(w, t)$, where $\Delta(x, y)$ is a lower bound on the shortest path from x to y: for example, the normal distance of x and y ([26], sect. 3.8.5.1). There is empirical evidence that this modification may direct the search towards t.

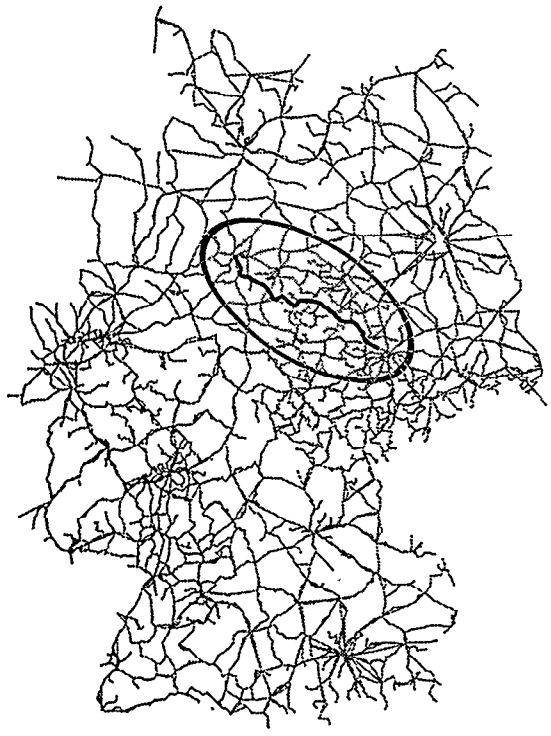

Figure 6: a shortest path from s = *Hannover* to t = *Leipzig* in the German rail network. Whenever s and t are relatively close, restricting the search horizon to a small ellipsis reduces the run time dramatically.[7]

Bidirectional search. The shortest (s,t)-path is computed from two nodes, s and t, simultaneously, from s in a forward direction and from t in a backward direction ([26], sect. 3.8.5.2). That is, two slightly different executions of Dijkstra's algorithm are "merged" into one loop, in each step one of these two algorithms is chosen to scan one single node, and this choice is made by an on-line heuristic. The algorithm terminates once a node u has been reached by both searches (the concatenation of the shortest paths from s and from t to u is the solution).

Restricted search horizon. Often it may be safely assumed (though not provable) that the real shortest path does not deviate too much from the straight line segment connecting s with t. In this case, the search may be restricted to, say, an ellipsis around s and t (see figure 6). This ellipsis may even be changed dynamically during the algorithm, depending on intermediate results. In other words, two algorithms—the shortest path routine and a routine that extends the search horizon on-line—are "merged" into one loop and affect each other.[8]

[7]This figure stems from a joint project with the *Deutsche Bahn AG* [34]; permission to visualize the railway data is granted by the TLC/Deutsche Bahn AG.

[8]For example, heuristical strategies of this kind have been implemented in the on-line information system of the *Deutsche Bahn AG*, the central German train and railway company.

Variations of the A^* algorithm. This sort of algorithm for shortest (s,t)-paths in the plane also fits into this general scheme [17]. In general, the effective length of an edge is a combination of its nominal length and an additional cost function, which penalizes large angles between this edge and the direction towards t. The effective cost of an edge must not be initialized in advance, but computed when needed, because otherwise, the initialization destroys the advantage of the A^* heuristic: that the amortized run time is highly sublinear in the size of the graph.

Concept

Our concept to achieve goal 3 consists of two principles: *loop kernels* and *full logical inspectability.*

Loop kernels. We introduced this principle in [22]. The crucial observation is that probably every non-trivial algorithm essentially consists of one or more loops (or nested loops), plus some pre- and postprocessing operations for each loop. We call these loops the *core loops* of the algorithm. For example, Dijkstra's algorithm for shortest paths, which we discussed above, has one core loop. In each iteration of this loop, the current distance estimation of exactly one node is decreased, provided that the currently available information allows that.

More systematically, implementing an algorithm as a loop kernel means the following:

1. The core loops of an algorithm are the basic units of reuse, not the algorithm as a whole.
2. A core loop is implemented as a class (the *loop-kernel class*), not as a subroutine.
3. Subalgorithms are polymorphic members of the class (design pattern *strategy* [12]).
4. The loop-kernel class provides a method which executes exactly one iteration of the core loop.[9]
5. Pre- and postprocessing operations are regarded as customizations of the core loop and left to the user of the class.
6. In particular, the algorithm does not initialize the data on which it works; it expects that this data has been initialized before the first iteration (not necessarily before instantiating the loop-kernel object).
7. Only the basic functionality of the algorithm, which is common to all applications, is provided by the loop-kernel class. Extensions are regarded as customizations of the core loop and left to the user of the class.

The fourth and fifth items enable the user to extend the basic functionality: the core loop is implemented by the user "around the loop kernel," and the user can freely insert additional stuff in the loop:

[9]For some algorithms, it may even be reasonable to divide ta single iteration into several steps. In other words, a couple of methods is provided such that one iteration is performed when all of these methods are called once in a specific order.

```
Algorithm A (...);  // Loop-kernel object
do_some_stuff_1 ();
while ( ! break_condition_fulfilled(A) )
 {
   do_some_stuff_2 ();
   A.next ();   // one iteration
   do_some_stuff_3 ();
 }
 do_some_stuff_4 ();
```

We are aware of the problem that algorithms which are implemented as classes and leave all pre- and postprocessing to the client are much harder to understand and to apply than algorithms which are implemented as subroutines. Hence, it might be useful to additionally implement a couple of subroutines, which are mere partial or complete customizations of the algorithm class to typical scenarios (incl. pre- and postprocessing). Clearly, these subroutines are much more convenient to use than the loop kernel itself.

Typically, the implementation of such a customizing subroutine is very short. Thus, if the sources of these subroutines are freely distributed as a part of the documentation, they may also serve as tutorials for using the full power of the underlying loop kernel. Moreover, it might be reasonable to implement various speed-up techniques in advance and to describe their potential applications and their usage in the documentation. All of these tasks form an integral part of [30].

Full logical inspectability. We call a class *fully logically inspectable*, if it provides methods to read all details of the current logical state of an object. In that, the *logical state* of an object of some class at some stage is defined as the bunch of information about its state which is necessary to predict the results of all possible sequences of calls to methods to this object after this stage.

For example, many libraries provide stack classes which offer access to the topmost element only (by methods *push*, *pop*, and *top*). However, the logical state of a stack object is not fully determined by the topmost element. Rather, it is described by the ordered sequence of items in the stack object. The principle of full logical inspectability requires methods which allow one to iterate over this sequence and to read the values of all items in this order.

Note that full logical inspectability does not contradict the encapsulation of implementation details, because no details of the implementation are exhibited and the access is read-only.[10] However, a stack class that is fully logically inspectable contradicts the abstract idea of a stack being a sequence that is only accessible at its front. In the discussion below, we will argue that for algorithm classes, the advantages of inspectability outweigh the resulting violation of abstraction.

> **Experience 4**
>
> If an algorithm is implemented as a fully logically inspectable loop kernel class, its algorithmic functionality is flexibly and efficiently adaptable.

[10] In some cases, it might even be reasonable to offer a disciplined *write* access to selected details.

Example

We continue with our example of speed-up techniques: Dijkstra's algorithm for shortest paths. Although it is only a simplified version, the following class template, `Dijkstra`, may illustrate our two design principles. The template parameters `Length` and `Distance` are data accessors and responsible for edge lengths and node distances, respectively. Template parameter `AdjIt` is the adjacency iterator type according to the hierarchy in section 3 (figure 5). According to the terms of section 3, this may be a reduced iterator, because the core loop of the algorithm works purely locally and does not require any "global knowledge" about the graph. The algorithm needs a container `PriorQ`, which serves as a priority queue. To simplify notation, we assume that this class is not a template parameter of `Dijkstra`, and that it is compliant to the list class in STL [28].

Method `next` is the heart of the algorithm. This method performs one step of the iteration. For lack of space, we leave out its implementation. Method `current_node` returns the node currently processed, and `begin` and `end` allow read access to the current frontier line of the search, which is exactly the contents of the priority queue. These three methods implement full logical inspectability.

```
template <class Length, class Distance,
          class AdjIt>
class Dijkstra
  {
  public:
     Dijkstra (Length len, Distance dist)
       : i_len(len), i_dist(dist) { }
     void make_root (AdjIt it)
        { i_pq.push_back(it); }
     void next ();
     bool finished () const
        { return i_pq.empty(); }
     AdjIt current_node () const
        { return i_pq.front(); }
     PriorQ::const_iterator begin ()
        { return i_pq.begin(); }
     PriorQ::const_iterator end ()
        { return i_pq.end(); }
  private:
     PriorQ    i_pq;
     Length    i_len;
     Distance  i_dist;
     // ...
  };
```

The "classical" variant of Dijkstra's algorithm, in which we want to compute shortest paths from root s to every other node, can be realized as follows:

```
AdjIt s;
/* init. s */
/* init. length accessor 'len' */
/* init. distance accessor 'dist' */
Dijkstra<Length,Distance,AdjIt>
   dijk (len, dist);
dijk.make_root (s);
while (!dijk.finished())
   dijk.next();
```

In the following variant, we assume that the function `wait_for_request` waits until a user wants to know the shortest connection from node s to t, which are specified in the call to `wait_for_request`. The speed-up technique "*early termination*" in our traffic network example can simply be realized as follows:

```
while (true)
 {
   AdjIt s, t;
   wait_for_request (s, t);
   /* init. length accessor 'len' */
   /* init. distance accessor 'dist' */
   Dijkstra<Length,Distance,AdjIt>
      dijk (len, dist);
   dijk.make_root (s);
   while (dijk.current_node() != t)
      dijk.next();
 }
```

Recall the speed-up technique "*version stamps*" defined above. To realize this variant, we write a data accessor class `VersionAccessor`, whose get methods return $+\infty$ if the version stamp of the node does not equal the global version number. The access to the node distance and to the version number are delegated to further data accessor classes: `DA` and `VA`. Hence, it is an example of proxies for data accessors. Since `DA` and `VA` are template arguments, `VersionAccessor` is broadly applicable. To simplify notation, we assume that all version numbers are of type `int`, and that `INT_MAX` represents $+\infty$.

```
template <class DA, class VA>
class VersionAccessor
  {
  public:
    typedef typename DA::vtype vtype;
    VersionAccessor
       (DA da, VA va, int global)
       : i_da(da), i_va(va),
         i_global(global) { }
    template <class It>
       vtype get (It it) const
          {
            return i_va.get(it)==i_global
                 ? i_da.get(it) : INT_MAX;
          }
    template <class It>
       vtype set (It it, vtype const& value)
          {
            i_va.set (it, i_global);
            i_da.set (it, value);
          }
  private:
    DA   i_da;
    VA   i_va;
    int  i_global;
  }
```

Here is the code to incorporate the speed-up technique "*version stamps*:"

```
int global_version = 0;
/* init. all node stamps to 0 */
while (true)
 {
   AdjIt s, t;
   wait_for_request (s, t);
   global_version++;
   /* init. length accessor 'len' */
   /* init. distance accessor 'dist' */
   /* init. data accessor 'stamp' for node
      version stamps */
   typedef
        VersionAccessor</*...*/,/*...*/> VA;
   VA stamped_dist
        (dist, stamp, global_version);
   Dijkstra<Length,VA,AdjIt>
        dijk (len, stamped_dist);
   dijk.make_root (s);
   while (dijk.current_node() != t)
      dijk.next();
 }
```

This adaptation is only possible because `Dijkstra` is a loop kernel. Full logical inspectability is not required for this speed-up technique. However, it is required for the speed-up technique *bidirectional search*. We only give a high-level description of the adaptation:

1. Reserve or establish two distance labels for each node and inititialize them.
2. Inititialize shortest-path loop kernel A with the first distance label and make s the (unique) root of A.
3. Inititialize shortest-path loop kernel B with the second distance label and make t the (unique) root of B.
4. WHILE no node is seen by both A and B:
   (a) let C be either A or B, depending on a heuristic on-line choice;
   (b) apply C.next().

The full logical inspectability is necessary in step 4(a): a sophisticated on-line choice must access the logical states of both loop kernels.

The speed-up techniques "*goal-directed search*" and "*variations of the A^* algorithm*" can be easily realized through specific data accessor classes (cf. section 2.2 and 3). Finally, the restriction of the search horizon to an ellipsis can be realized by a filter iterator as defined in section 3. This iterator must be instantiated with a predicate that returns true for a node if and only if this node is inside the search horizon. To increase the search horizon on-line, the heuristic which computes the new search horizon must be invoked in every iteration of the core loop. Because of the loop kernel concept, this loop is "inside-out," and hence this is easily done. Because of full logical inspectability, this heuristic is well informed.

Comparison to algorithmic generators

Loop kernels are a generalization of the *algorithmic generators* introduced by Flamig [10], which are similar to graph search iterators [1] and to built-in concepts for streams and

iterations in various object-oriented programming languages (notably the *iterator* concept in Sather [32]). Like a loop kernel, an algorithmic generator encapsulates an algorithm in a class and offers a method to perform exactly one iteration of the core loop. The main differences between loop kernels and algorithmic generators are the following:

- Algorithmic generators are only intended to implement algorithms which consist of one core loop, and in each iteration of the core loop, a single piece of the algorithm's output is constructed. For example, a generator for a sequence of prime numbers falls into this category. However, Dijkstra's algorithm does not, and many other algorithms do not either. Consequently, Dijkstra's and many other algorithms are not formulated as algorithmic generators in [10], but in a conventional style as subroutines.
- An algorithmic generator does all pre- and postprocessing itself, namely in its constructors and destructor, whereas a loop kernel delegates this to the client. Hence, a speed-up technique such as *"version stamps"* cannot be incorporated afterwards in an existing algorithmic generator, unless this technique was antipicated by the designer of the algorithmic generator.
- After an iteration of the core loop, an algorithmic generator provides a means of accessing the piece of the output generated in this iteration. This is far from full logical inspectability. Consequently, a speed-up technique such as *"bidirectional search"* cannot be incorporated either.

Comparison to algorithm frameworks

Goal 3 can alternatively be approached through a framework-like design (see Holland [15] for a concrete example). In such an algorithm framework, the algorithm is designed as a base class, and this base class provides a method which executes the algorithm as a whole. The crucial design task is to identify the "skeleton" of the algorithm, which is common to all applications, and to determine all points in the algorithm where a possibility to change or extend the behavior is necessary to achieve flexible adaptability. Each of these points is realized as a method of the base class. Derived classes may overwrite these methods to customize the algorithm to concrete applications.

As demonstrated above, variations of the break condition, extensions of the algorithmic functionality, and intermixed algorithms (*e.g.* bidirectional search) are straightforward if the loop-kernel concept is applied. In contrast, in an algorithm framework, any of these modifications requires implementing a derived class. Moreover, these derived classes might often be tricky to implement. For instance, intermixed algorithms can indeed be realized in an algorithmic framework, but the task is not at all trivial, and the design of the whole algorithmic package might suffer. To give another example: suspending an algorithm temporarily to perform other, unrelated tasks is trivial in the loop-kernel concept, but might cause severe design problems in an algorithm framework.

5 Inspectability

Goal 4

An algorithm should be implemented such that it is possible to inspect all potentially relevant details during its execution.

Experience 5

Goal 4 may also be achieved through fully logically inspectable loop kernels.

Goal 4 addresses various tasks. We discuss two different tasks to illustrate experience 5.

- **Animation:** To give a concrete example, a possible visualization of Dijkstra's algorithm in slow motion would be to distinguish unseen nodes, nodes in the current frontier line of the search, and finished nodes from each other by color. It is not hard to write iterator adapters which deliver all necessary information to the graphical display (using the *observer* pattern [12]). Sometimes it might be necessary to refresh the whole display. This requires information about the current logical state of the algorithm. Because of the principle of full logical inspectability, no redundant bookkeeping outside the algorithm is necessary.
- **Snapshots:** Another point, which might not be interesting for Dijkstra's algorithm, but may be important for algorithms which take significant more time per run: if the loop kernel concept is applied, the algorithm may be "snapshot" at frequent occasions to recover after a crash.

6 Conclusion

We have implemented several algorithms based on the toolbox in section 3 and on the loop kernel concept and the principle of full logical inspectability.[11]

In particular, we compared the shortest-path algorithm in LEDA with our adaptable implementation. To achieve a realistic comparison, we customized our implementation exactly to the same algorithmic problem and the same data structures.

Experience 6

The overhead of the techniques discussed in this paper compared to traditional implementations of algorithms might be acceptable for most applications.

[11] Using the GNU C++ compiler, version 2.7.2, on Sparc Stations.

Figures 7-9 show the results of computational studies in which we compared the performance of LEDA's implementation of Dijkstra's algorithm (labeled "Direct") with the performance of our implementation ("Adaptable"). Figures 7 and 9 show the results of two studies in which the "classical" variant of Dijkstra's algorithm was applied to determine the shortest paths from a designated root s to all other nodes. On the other hand, to obtain figure 8, we computed the shortest paths for all pairs of nodes: the classical variant is called once for every node as the root node. Figure 8 shows the accumulated times.

This computational study is based on random graphs. In figures 7 and 8, each graph was constructed such that it is highly connected and highly cyclic. Figure 9 is based on random triangulated graphs: a triangulated graph is a plane graph (cf. figure 1) such that each internal area is a triangle. In addition, we studied all-pairs shortest-path computations on several national and international train networks such as the one shown in figure 6:

Train network	Overhead factor
Austria	1.48
Europe	1.53
France	1.68
Germany	1.60
Germany (local trains only)	1.62
Switzerland	1.57

This overhead is certainly acceptable for the overwhelming majority of all applications. "In theory," the overhead may be even smaller in all situations in which the run-time flexibility of dynamic binding is not required, because all function calls could be inlined and optimized as if they were hand-coded. Hence, there is hope that progress in compiler technology will reduce the overhead even further.[12]

Acknowledgements

I would like to thank Ulrik Brandes, Dieter Gluche, Dietmar Kühl, Kurt Mehlhorn, Stefan Näher, Marco Nissen, Wolfgang Pree, and Christian Uhrig for fruitful discussions. Special thanks to Dietmar Kühl and Marco Nissen for their strong engagement in the project and to Dorothea Wagner for her overall support and encouragement. Moreover, I would like to thank the anonymous reviewers for their valuable comments and for their hard, constructive criticism. In particular, I am very grateful to my "shepherdess," Lougie Anderson. Finally, I would like to thank Tina Deveny for proof-reading.

References

[1] AAI base class library home page. http://www.aai.com/AAI/IUE/spec/base/base-classes.html.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows.* Prentice Hall, 1993.

[3] Robert Barton and Lee R. Nackman. *Scientific and Engineering C++.* Addison-Wesley Publish. Comp., 1994.

[4] Ted J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings 3rd International Conference on Software Reuse*, 1994. http://www.research.microsoft.com/research/ip/tedb/limits/limits.htm.

[5] T. Budd. *Classic Data Structures in C++.* Addison-Wesley Publishing Company, 1994.

[6] CGAL home page. http://www.cs.ruu.nl/CGAL/.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1994.

[8] Martin Erwig. Graph algorithms = iteration + data structures? In *Proceedings 18th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'92, Wiesbaden-Naurod, Germany, June 18-20, 1992*, pages 277–292. Springer-Verlag, Lecture Notes in Computer Science, vol. 657, 1992.

[9] ffGraph home page. http://www.fmi.uni-passau.de/~friedric/ffgraph/main.shtml.

[10] Bryan Flamig. *Practical algorithms in C++.* Coriolis Group Book, 1995.

[11] Giorgio Gallo and Maria G. Scutella. Toward a programming environment for combinatorial optimization: a case study oriented to max–flow computations. *ORSA J. Computing*, 5:120–133, 1993.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns, elements of reusable object-oriented software.* Addison–Wesley Publishing Company, 1994.

[13] Dieter Gluche, Dietmar Kühl, and Karsten Weihe. Evaluation of database queries through iterator proxies, 1997.[13]

[14] Mark R. Headington and David D. Riley. *Data abstractions and structures using C++.* D.C. Heath and Company, 1994.

[15] Ian M. Holland. Specifying reusable components using contracts. In *Proceedings Europoean Conference on Object-Oriented Programming (ECOOP)*, pages 287–308, 1992.

[16] Chris Johnson and Philip Gray. Assessing the impact of time on user interface design. *SIGCHI Bulletin*, 28(2):33–35, 1996.

[17] Laveen Kanal and Vipin Kumar. *Search in Artificial Intelligence.* Springer, 1988.

[18] Karla home page. http://i44www.info.uni-karlsruhe.de/~zimmer/karla/index.html.

[19] Yugo Kashiwagi and David S. Wise. Graph algorithms in a lazy functional programming language. In *Proceedings 4th International Symposium on Lucid and Intensional Programming*, pages 35–46, 1991.

[12] The overhead was essentially caused by failure of method inlining. In principle, zero overhead is possible. In fact, in additional experiments, we found that at least one commercial compiler already comes close to this ideal. We could not use this compiler for the above studies, because LEDA has not been ported to it.

[13] http://www.informatik.uni-konstanz.de/Preprints

[20] Dietmar Kühl, Arfst Ludwig, Rolf H. Möhring, Rudolf Müller, Jörn Schulze, and Karsten Weihe. ADLIPS user manual, 1993.[13]

[21] Dietmar Kühl and Karsten Weihe. Iterators and handles for nodes and edges in graphs.[13]

[22] Dietmar Kühl and Karsten Weihe. Using design patterns for reusable, efficient implementations of graph algorithms - working paper.[13]

[23] Dietmar Kühl and Karsten Weihe. Data access templates. *C++ Report*, 9(July/August), 1997.

[24] John Launchbury. Graph algorithms with a functional flavour. In *First International Spring School on Advanced Functional Programming Techniques*, pages 308–331. Springer-Verlag, Lecture Notes in Computer Science, vol. 925, 1995.

[25] LEDA home page. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[26] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout.* Wiley, 1990.

[27] Kurt Mehlhorn and Stefan Näher. LEDA: a library of efficient data structures and algorithms. *Communications of the ACM*, 38:96–102, 1995.

[28] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison–Wesley Publishing Company, 1995.

[29] Gabriele Neyer, Wolfram Schlickenrieder, Dorothea Wagner, and Karsten Weihe. PlaNet — a demonstration package for algorithms on planar networks.[13]

[30] Marco Nissen and Karsten Weihe. Combining LEDA with customizable implementations of graph algorithms.[13]

[31] PlaNet home page. http://www.informatik.uni-konstanz.de/Forschung/Projekte/PlaNet/.

[32] Sather home page. http://www.icsi.berkeley.edu/~sather.

[33] Robert Sedgewick. *Algorithms in C++.* Addison–Wesley Publishing Company, 1992.

[34] SIMI home page. http://www.informatik.uni-konstanz.de/Research/projects_algo.html#projekt7.

[35] Jiri Soukop. *Taming C++.* Addison–Wesley Publishing Company, 1994.

[36] Bjarne Stroustrup. *The C++ programming language (2nd edition).* Addison–Wesley Publishing Company, 1991.

[37] Michael VanHilst and David Notkin. Using C++ templates to implement role-based design. http://www.cs.washington.edu/homes/vanhilst/research.html.

[38] visone home page. http://www.informatik.uni-konstanz.de/Research/projects_algo.html#projekt8.

[39] Sheng Yu and Qingyu Zhuang. Software reuse via algorithmic abstraction. In *Conference Series Technology of Object-Oriented Languages and Systems (Tools USA '95)*, pages 277–292, 1995.

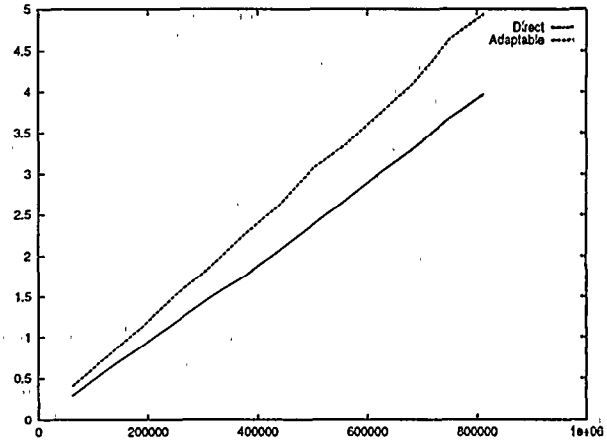

Figure 7: single–source shortest paths on random graphs with 1000 nodes (x: number of edges, y: run time in seconds).

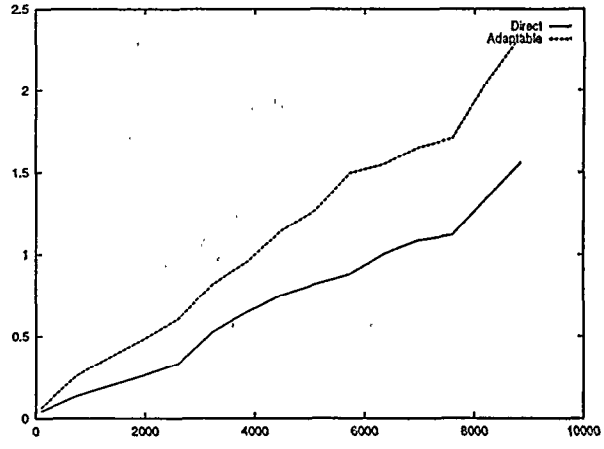

Figure 8: all–pairs shortest paths on random graphs with 100 nodes (x: number of edges, y: run time in seconds).

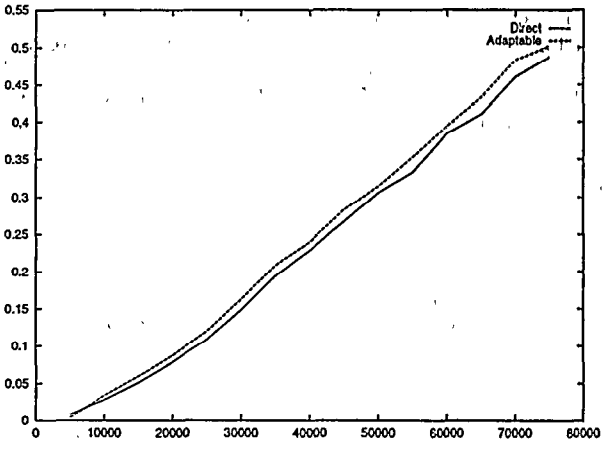

Figure 9: single–source shortest paths on random triangulated planar graphs (x: number of nodes, y: run time in seconds).