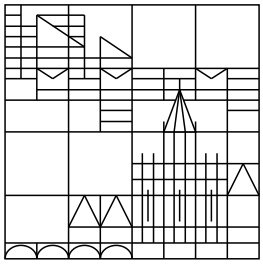

Universität Konstanz

A Software Engineering Perspective on Algorithmics

Karsten Weihe

Konstanzer Schriften in Mathematik und Informatik
Nr. 50, Januar 1998

ISSN 1430–3558

Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften

A Software Engineering Perspective on Algorithmics

Karsten Weihe*

Lehrstuhl für praktische Informatik I
(Algorithmen und Datenstrukturen)

50/1998
Universität Konstanz
Fakultät für Mathematik und Informatik

Abstract

An *algorithm component* is an implementation of an algorithm which is not intended to be a stand-alone module, but to perform a specific task within a large software package or even within several distinct software packages. Hence, the design of algorithm components must also incorporate software engineering aspects. A key design goal is adaptability: this goal is important for maintenance throughout a project, prototypical development, and reuse in new, unforeseen contextes. However, efficiency must not be sacrificed. The aim of this paper is to identify concrete requirements which must be fulfilled by an algorithm component to meet these general goals.

To explore what adaptability of an algorithm component *really* means, we will discuss an acid test: Dijkstra's algorithm for shortest paths and its application in various realistic scenarios. From this analysis, we will derive general, language-independent design goals, which are intended to support implementations of efficient, adaptable algorithm components. Finally, we will review selected approaches from the literature in view of these goals (including the author's own work on this topic).

*Universität Konstanz, Fakultät für Mathematik und Informatik, Fach D188, 78457 Konstanz, email: weihe@informatik.uni-konstanz.de, WWW: http://www.informatik.uni-konstanz.de/~weihe

1 Introduction

We will discuss the problem of designing efficient, adaptable implementations of algorithms. In practice, adaptability is important for several reasons:

- Throughout the life time of a project, the requirements on the functionality of a single component may change time and again. Re-implementing this component from scratch time and again is expensive and often infeasible. Extensive modifications of the core of the component are error-prone and also often infeasible. In particular, extensive modifications of sophisticated algorithmic code is error-prone. Hence, ideally, adapting an algorithm component to changing requirements should amount to modifying a few small, non-algorithmic pieces of code (*customization code*).
- Often, software is implemented in a prototypical fashion, that is first a raw prototype is implemented, which is then refined step-by-step throughout several development cycles. An example is the experimental development of algorithms. In larger projects, such a prototypical approach is feasible only if all components are easily customizable to new, revised versions of the software package.
- Implementing complex components (*e.g.* sophisticated algorithms) is expensive and requires special expertise. Hence, it is desirable to have (in-house or third-party) repositories of complex components, which are reusable in many different contextes. However, to be widely reusable, a component must be adaptable to a large range of possible requirements.

Many libraries offer efficient data structures and algorithms for various algorithmic problem domains. However, an implementation of an algorithm in a library need not fit seamlessly into every application. Hence, the integration may enforce significant overhead both in development/maintenance and in run-time efficiency. In some cases, this overhead may be so large that re-implementing an algorithm from scratch might be more practical than reusing an existing implementation.

In summary, the problem of designing efficient, adaptable algorithm components does not seem to be satisfactorily understood.

Intent

The main contribution of this paper is a systematical analysis and discussion of this problem. In that, we will focus on graph algorithms. However, we believe that the key insights carry over to other algorithmic domains as well. Based on this analysis, we will review existing approaches to this problem. We will mainly concentrate on pragmatical, "down-to-earth" aspects, which are rather motivated by practical experience than by theoretical considerations. For example, this also includes documentational issues: if a solution is too sophisticated to allow a concise, understandable documentation, it is of limited practical value, no matter what its merits are from a purely technical viewpoint.

It goes without saying that this paper does not—and cannot—cover all relevant aspects of the problem. The selection and ranking of criteria and the review of the literature in view of these criteria is necessarily subjective and reflects the author's bias.

Several scientific communities and subcommunities have contributed proposals how to overcome the problems addressed in this paper or related problems. In fact, people working in structured programming, object-oriented programming, functional programming, software reuse, and formal specification, just to mention a few, have contributed to this field. Moreover, user communities of various programming languages have attacked these problems in a language-dependent manner. Recently, research on this topic has started in various algorithm communities. All of this work seems to be done more or less in isolation without much interaction between the individual communities. Hence, we do not claim that the presented review of existing approaches will give a true survey of all of the relevant literature. Since each of these communities looks at the problem

from a different perspective, it does not seem possible to compare the general merits of these approaches in a fair manner. Hence, we will restrict our attention to comparisons in view of the presented analysis of the problem.

Overview

In Section 2, we will systematically discuss the obstacles to efficient, adaptable implementations of graph algorithms. We will identify two key problems: flexible choice of the organization of the underlying data and flexible integration of algorithms into software packages. Then, in Section 3, we will discuss the problem in view of a specific example, namely the problem of finding shortest paths in networks and its application to traffic information systems. In our experience, this is a good "acid test" to evaluate general programming methodologies in view of algorithmic software.

In Section 4, we will review the intent of our work in light of this acid test. Afterwards, in Section 5, we will conclude a couple of concrete, language-independent goals from this acid test. Section 6 is devoted to the review of existing approaches. This includes a systematical review of the author's work in this field, which is summarized in [GKW98, KW97, KNW97, Wei97, Wei98]. See

http://www.informatik.uni-konstanz.de/Research/projects_algo.html#projekt5

for further details of this research. See also

http://www.mpi-sb.mpg.de/LEDA/www/ledaep.html

for an integration of these concepts into the *L*ibrary of *E*fficient *D*ata Structures and *A*lgorithms (*LEDA*, cf. [MN95]). Finally, Section 7 discusses the problem how to document adaptable algorithm components.

Audience and required background

This paper addresses people working in algorithmics, in software engineering, and also in the various application domains of algorithmics, who are interested in the software-engineering aspects of algorithmics.

Reading this paper requires some basic background knowledge in algorithmics, notably in graph algorithmics. Introductory chapters of textbooks on graph algorithms and Dijkstra's algorithms (*e.g.* Chapters 23 and 25 in [CLR94]) might cover all necessary pre-knowledge, except for a few examples, which illustrate or motivate specific aspects. In any case, we will give references to further details in the literature, and important aspects will be explained in detail in this paper. Moreover, a basic background knowledge in programming is assumed. Besides that, the paper should be self-contained. In particular, all details of non-algorithmic aspects (*e.g.* features of programming languages) are briefly introduced when needed. A few central aspects are deferred to an appendix and systematically treated there.

Since most of the practical work underlying this paper was done in one language (C++), a certain bias towards this language might be unavoidable. Nonetheless, the presentation is abstract and does not require C++ literacy.

To some extent, the discussion of concrete details will be based on LEDA [MN95][1] and the *S*tandard *T*emplate *L*ibrary (*STL* [MS95]; see Section D in the appendix for a brief introduction into the design of the STL). The STL greatly influenced the definition of the C++ standard library[2] and the design of many other recent libraries. However, we merely use these two libraries as mature examples and representatives for many other libraries. Nonetheless, no pre-knowledge of these two libraries is required either.

[1] See also http://www.mpi-sb.mpg.de/LEDA.

[2] C++ Final Draft International Standard, ISO/IEC 14882.

Domain, application, context

To avoid ambiguities, we need some basic common vocabulary and a common intuition of what this vocabulary means. The following vocabulary is essential for the rest of the paper:

- A *domain* (or *application domain*) is a broad field of research such as VLSI design, traffic logistics, or computer-aided design, which is not algorithmic in nature, but offers various algorithmic problems.
- An *application* is a specific algorithmic problem occurring in some domain. Thus, an application defines the required *algorithmic functionality* of the desired algorithm component in abstract, mathematical or informal, terms. For example, the problem of finding a shortest path from some *source node* to some *target node* in a network embedded in the plane with respect to the Euclidean lengths of the edges is an application occurring in traffic logistics. The desired algorithmic functionality is described by the input (a directed graph, a pair of coordinates for each node in the plane, the source node, and the target node), the output (a path from the source node to the target node), and optional restrictions on resource requirements such as the run time and the maximum peak of space usage throughout the execution.
- A *context* is more specific than an application in that it also describes the required *non-algorithmic functionality*. For example, the description of an application does not define the exact organization of the input and the output. In a context in which conversions of data structures are infeasible, the algorithm component must be able to process the input and deliver the output exactly in the form dictated by the context. Another example concerns algorithms that have exorbitant run times and shall be used in interactive systems. Such an interactive context might require that the algorithm is frequently interruptable to allow the end-user to interact with the system without significant delay times.[3]

  A context may also include non-technical restrictions. For example, it may be necessary to have a simple, "idiot-proof" documentation of the algorithm components. Such a requirement restricts the design options significantly: apparently, not every design can be documented in such a way.

Using this terminology, we can state our goal more precisely:

> An algorithm A should be implemented as an algorithm component AC such that AC is adaptable to every context whose underlying application is solvable by A from a theoretical viewpoint.

2 The Problem

Many implementations of algorithms are not written for a specific application, but are intended to be used in many different contextes, ideally in all contextes that are covered by the algorithm from a theoretical perspective. However, this goal is hard to achieve. In the following two subsections, we will consider two general problems, which we regard as essential. In principle, the first problem means that an algorithm component should be adaptable to the lower level, and the second problem, that it should be adaptable to the higher level. "Lower level" means the underlying data structures, and "higher level" the *client code* which calls the algorithm.[4] A third case may be of interest: an algorithm A calls an algorithm B, in which case one algorithm is the lower level of the other one and vice versa. However, this case is sufficiently covered by the second case: the client of an algorithm component may also be an algorithm component.

[3] This particular requirement was presented to us by the representatives of a software company in a recent meeting about a potential cooperation, in which we would implement certain algorithm components, and the company would integrate them into their interactive system.

[4] This distinction will reappear in several later sections.

2.1 Adaptation to the Lower Level: Underlying Data Structures

Problem: *The algorithm is implemented on top of data structures which are different from the data structures dictated by the context.*

In general, this problem can be solved by converting all data structures back and forth. However, the induced overhead might not always be acceptable. If the complexity of an algorithm is sublinear in the size of the input and the output, even the asymptotic complexity is increased. For example, an algorithm that accesses only a small area in a large two-dimensional geometric data structure could be sublinear. Even more, if algorithms are not implemented monolithically, but hierarchically composed of existing implementations of simpler algorithms, many conversions may be necessary to plug all of these algorithms together. Moreover, a large number of conversion routines takes a lot of programming effort: in the worst case, a quadratic number of conversion routines is necessary.[5]

A common attempt to overcome these problems is to define a set of basic data structures and to implement all algorithms and all contextes on top of these data structures. In other words, these data structures are intended to set a standard, on which all algorithms are to be implemented. Clearly, at first glance, this would solve the problem. Many libraries offer such a set of data structures. For example, LEDA provides basic data structures such as lists, stacks, and various tree types, data structures for directed and undirected graphs, and geometric data structures.

However, in turn, this strategy causes two problems.

Problem I: In general, no particular implementation of a data structure satisfies all potential requirements in an efficient manner.

For instance, many graph algorithms rely on an efficient way of answering the following question: given two nodes, is there an edge connecting these nodes, and if so, return one. Roughly speaking, this requires either a matrix, whose rows and columns are nodes and whose entries are edges, or a dictionary type (hash, search tree, grid file, ...), whose key type is "pair of nodes" and whose information type is "edge." On the other hand, the induced overhead may not be acceptable for algorithms which do *not* require this feature: in case of a matrix solution, the asymptotic space requirement might be increased; in case of a dictionary solution, inserting and deleting edges puts an additional non-constant factor on the run time.

A more basic example, which appeared quite frequently in the author's own practical work, is the question whether or not a graph should support associative copies. This means that a new graph object is created as a logical copy of an existing graph object, and the two of the graph objects record and maintain a mapping between corresponding nodes and edges. Many algorithms require such a mapping. If the graph type does not support such a mapping, this mapping must be implemented in the algorithm itself. This is inconvenient, makes the complex algorithmic stuff even more complex, and may be a permanent source of bugs. Moreover, the algorithm cannot use an existing subroutine for copying a graph, because the mapping between corresponding nodes and edges cannot be reconstructed efficiently afterwards. However, inserting such a feature in the underlying data structure not only affects the performance. For example, some algorithms shrink the copy of the graph by identifying nodes with each other. It is not clear in which way the mapping between the nodes and edges of the graph objects should be maintained throughout modifications like these. There are several reasonable options for maintaining the correspondence throughout non-simultaneous modifications of two corresponding graph objects. It is not clear at all whether there exists a scheme which satisfies all concerned algorithms.

Problem II: Basic data structures must be *generic* to be independent of particular applications.

[5] For example, effects like these occurred in a project in which the author was involved a few years ago and which used conversions to integrate various algorithms for scheduling problems into one package; see http://www.informatik.uni-konstanz.de/~weihe/manuscripts.html/paper17.

For example, for an all-purposes graph data structure, this means that node and edge attributes such as lengths, capacities, and costs, must be freely choosable, because every algorithmic (and non-algorithmic) problem requires another set of attributes.

To give a concrete example, LEDA provides two ways of associating attributes with nodes and edges in a graph:

- The graph classes themselves are generic[6] and parameterized by information types for nodes and edges. Typically, the generic type parameters are instantiated by records, which comprise all node and edge attributes, respectively.
- In addition, LEDA provides so-called *node* and *edge array* classes. Basically, a node or edge array is a normal or associative array and contains the values of a single node (or edge) attribute for all nodes (edges). Such an array can be instantiated at every stage to introduce a new attribute for nodes or edges.

These two possibilities seem to be the two fundamental ways of attaching generic attributes to nodes and edges without compromising efficiency.[7] Often, the first alternative is strongly preferable: a normal node (resp. edge) array causes severe problems when nodes are inserted and removed by the algorithm; on the other hand, an associative array causes a significant loss of performance. These problems do not occur in the first alternative. Nonetheless, all algorithms in LEDA receive and return attribute settings as node and edge arrays. This comes as no surprise: otherwise, collaborating algorithms had to agree on a common naming convention for all attributes. However, there are two prohibitive obstacles to such a convention:

- *The same attribute may have different meanings in collaborating algorithms.*

  For example, a minimum-cost flow algorithm may use a shortest-path algorithm and a maximum-flow algorithm as subroutines, and the edge attribute that appears as the *residual capacity* inside the minimum-cost flow algorithm and the maximum-flow algorithm is also the *length* inside the shortest-path algorithm.

  When all of these algorithms are implemented within the same project, it may be possible to stick to a common name for this attribute (there might not be a name which is intuitive in all of these algorithms, but this might be a minor nuisance). However, when the shortest-path algorithm and the maximum-flow algorithm have been implemented before as general-purpose components which are now reused for the minimum-cost flow algorithm, it is unlikely that the developers of these two components anticipated this specific application in which the length is just the residual capacity. Hence, it is unlikely that these implementations agree on a common name for this attribute.

- *Two different attributes may have the same meaning.*

  For example, shortest paths in a traffic network shall be determined either according to the number of kilometers or the total time of traveling. In such a case, both attributes—the length in kilometers and the total time of traveling—must be assigned the same identifying name, which is impossible.

Some languages (*e.g.* Ada) provide means of renaming record components. Such a mechanism may solve the problem to some extent. However, C++ and many other languages do not provide such a feature.

Moreover, in some cases it is not even reasonable to explicitly store an attribute at all. For example, consider a graph whose nodes are points in the plane, and assume that the length of an edge is the Euclidean distance of its incident nodes. Since the number of edges may be quadratic in the number of nodes, it may be reasonable not to allocate space for the edge lengths, but to compute an edge's length from the node coordinates when needed.

In such a case, no renaming mechanism or any other feature of mainstream languages will solve the problem.

[6] To be precise: C++ class templates. See Section C in the appendix.

[7] [Wei97], Section 2.2, discusses several variations of these two alternatives.

2.2 Adaptation to the Higher Level: Client Code

Problem: *The algorithmic and non-algorithmic functionality of the implementation of an algorithm should be extensible and modifiable.*

We introduced the notions of algorithmic and non-algorithmic functionality in the section on domains, applications, and contextes, in the Introduction. Many algorithms in the literature—notably the fundamental ones—are generic in the sense that the exact details of the algorithmic functionality are variable within certain limits. In other words, such a "theoretical" algorithm only serves as a pattern for concrete algorithms. Essentially, drawing a concrete algorithm from such a pattern means four things:

1. *For each subproblem, an algorithm is chosen from the set of all algorithms which solve this subproblem.*

   Example: Typical maximum-flow algorithms apply solvers for subproblems such as graph search and path search [AMO93]. These solvers should be exchangeable inside an implementation of a maximum-flow algorithm.

2. *The output is varied or extended.*

   Example: 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.

   For instance, many libraries (incl. LEDA) provide an implementation of depth-first search which only returns a node labeling or the traversal order of all nodes. Constructing a depth-first tree from this information amounts to performing another depth-first search.[8] 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.

3. *Heuristical algorithmic speed-up techniques are added.*

   Examples: see Section 3.2.

4. *Non-algorithmic functionality is incorporated.*

   Examples: see Section 3.1, Item 6.

3 An Acid Test: Dijkstra's Algorithm for Shortest Paths

We now describe an acid test for general programming methodologies in view of algorithmics. A component which implements Dijkstra's algorithm passes this test, if it is adaptable to all variations of the application without sacrificing efficiency. If such a component has been implemented according to some general programming methodology, this programming methodology is also said to pass this test. However, we will not use this acid test directly to evaluate programming methodologies. As announced in the Introduction, we will condense this test to a few general goals in Section 5. This will allow us to evaluate programming methodologies more concisely and systematically.

The acid test consists of two parts: general variations of Dijkstra's algorithm in Section 3.1 and various speed-up techniques in a concrete, realistic application in Section 3.2. These two

[8] Unless the underlying graph data structure supports access to the in-going edges of a given node. In this case, the predecessor of a node $v \in V$ in the depth-first tree can be determined by a scan over the node labels of all heads of its in-going edges: the head with the maximum label less than the label of v is the predecessor. However, in general, such a support cannot be assumed.

subsections will further illustrate the problems raised in Section 2. Some of the variations will not be disjoint, but share common aspects. This is an unavoidable consequence of our systematic, "keyword-oriented" presentation.

3.1 General Requirements on Flexibility

1. Variations of the graph data structure:

   (a) *Graph abstractions*: Although Dijkstra's algorithm is naturally formulated as an algorithm for directed graphs, it can be applied to undirected graphs, bidirected graphs [Der88], and hypergraphs [Len90], just to mention a few. Each of these graph abstractions may be interpreted as a directed graph:

   - The usual interpretation of *undirected* graphs as directed graphs regards each undirected edge as two antiparallel directed edges with identical attribute values.
   - In principle, a *bidirected* graph is an undirected graph such that for each node the edges incident to this node are partitioned into two sets. For bidirected graphs there are two natural interpretations as directed graphs: first, for each node the bipartition of all incident edges is dropped, which reduces this case to the undirected case. Second, only a certain sort of paths is considered, namely paths such that for each intermediate node the two incident edges of the path belong to different partition sets in the partition of the edges incident to this node.
   - The usual interpretation of an undirected hypergraph H is equivalent to a normal, bipartite graph $G = (V \cup W, E)$, where the nodes of H correspond to V, the hyperedges of H to W, and a node of V is connected with a node of W by an edge in E if and only if their corresponding node and edge in H are incident. The lengths of the edges in G must be defined such that for $v_1, v_2 \in V$, $w \in W$, $\{v_1, w\}, \{v_2, w\} \in E$, the length of the path $v_1 \to w \to v_2$ in G equals the length of the hyperedge of H corresponding to w.

   (b) *Special graph classes*: Many applications only need solutions for special graph classes such as bipartite graphs, interval graphs [Gol91], planar graphs and grid graphs [Len90, NC88], and many others. In such a case, it may be reasonable not to use a general graph data structure, but to implement a special data structure, which only represents graphs of this class and whose functionality is somewhat different: on one hand, the functionality is usually extended in order to exhibit the rich structure of the special graph class. On the other hand, basic functionality such as inserting and removing nodes or edges must be restricted or even disabled to prevent manipulations which would drive the graph out of this special class. In the extreme case, such a graph data structure is not at all implemented as a graph. For example, an interval graph may be implemented as a set of intervals.

   Since Dijkstra's algorithm does not change the structure of the graph, it may be useful for virtually every data structure that represents such a special class of graphs.

   (c) *Different views on the same graph*: For example, consider the case that the graph contains first-class and second-class edges, and in some situations only first-class edges shall be considered for shortest paths, whereas in other situations all edges are to be considered.[9] The edges belonging to different classes may be implemented by two separate lists of incident edges for each node, or they may be implemented by a single list, where first-class and second-class edges are merely distinguished by a flag for each edge.

   (d) *Implicit representation*: The underlying graph is only given implicitly. For example, consider the problem of finding shortest paths on the line graph H of an undirected graph G, that is H contains a node for every edge in G, and two nodes are connected

[9] This example was taken from [Sou94], Sect. 2.6.

in H by an undirected edge if and only if the corresponding two edges in G are incident to a common node. Since the size of H may be quadratic in the size of G, it might not be reasonable to materialize H, but to run Dijkstra on an implicit representation of H.

(e) *Hierarchical decomposition*: The underlying graph is hierarchically decomposed into smaller graphs. For example, this technique is applied in geographic information systems. Here we only sketch one concrete realization of this general technique:

The graph (*e.g.* a traffic network) is decomposed into overlapping connected subgraphs such that every pair of nodes in such a subgraph has a shortest connecting path which is completely inside this subgraph. The shortest paths between all pairs inside such a subgraph are determined in a preprocessing phase and stored permanently. The nodes in such a subgraph which also belong to other subgraphs are the *boundary nodes* of this subgraph. To find a shortest path from some node s to some node t, first the subgraphs G_s and G_t to which s and t belong are identified. If $G_s = G_t$, one table-lookup suffices. Otherwise, the shortest paths from s to all boundary nodes of G_s and the shortest paths from all boundary nodes of G_t to t are looked up.

To find the missing segment of the shortest path between G_s and G_t, it suffices to find a shortest path from the set of boundary nodes of G_s to every boundary node of G_t, where the distances of the boundary nodes of G_s are not 0 as in the usual definition of the shortest-path problem, but equal to their distances from s in G_s. This path is determined in an auxiliary graph, whose node set is the union of the boundary nodes of all subgraphs, and two nodes are connected by an edge if and only if they belong to a common subgraph (the length of this edge is the shortest path in the corresponding subgraph and again determined by a table-lookup).

2. Variations of the numeric types: The numeric types of edge lengths and node distances may be different in different contextes (both types may even be different within the same context). In fact, every type that fulfills a certain algebraic specification may be used [LT91].

   In the beginning of Dijkstra's algorithm, the distances of all nodes except for the root are initialized as "infinity." For example, infinity may be simulated by a very large value of the type of node distances, by a reserved, "impossible" value of this type (*e.g.* a negative value), or by an additional Boolean node attribute. See Item 2 in Section 3.2 for yet another, more sophisticated example.

3. Variations of the priority queue: From a theoretical point of view, a *Fibonacci heap* is the best implementation [CLR94]. However, if the number of edges is linear in the number of nodes (*e.g.* in planar graphs), a simple d-heap achieves the same asymptotic run time and may have a better "big-oh factor." On the other hand, if all edge lengths are small integral numbers, the dial implemention might be preferable [AMO93]. Basically, this means that the priority queue is a bounded first-in-first-out queue.

4. Variations of the organization of the node and edge attributes (node distances and edge lengths):

   (a) *Attached to items*: The attributes of nodes (resp., edges) may be organized as a record, which is attached to every node (edge), and a single attribute such as the node distance (edge length) is a slot in this record. The type of the record and the name of this slot may be dictated by the context and thus cannot be chosen freely by the developer of the algorithm component.

   (b) *Separate from the graph structure*: The distances of all nodes (resp., lengths of all edges) are stored in an additional associative array, and the ID of a node (edge) serves as the key to access the corresponding item in this array.

   (c) *Cache*: The data is cached in a buffer and retrieved when needed.

(d) *Implicit organization*: A node or edge attribute is not explicitly stored at all, but is computed when needed. For example, if all first-class edges in Item 1(c) have the same length value, and likewise all second-class edges, third-class edges, and so on, it is not necessary to store the edge lengths explicitly: it suffices to store the class of every edge and an additional, small table outside the graph, in which the lengths of all classes (and possibly many further class-specific attributes) are listed.[10]

5. Variations of the algorithmic functionality:

   (a) Shortest paths from one root to all other nodes. For example, Dijkstra's algorithm may return an assignment of distances to all nodes or a tree of shortest paths rooted at this node.

   (b) Shortest paths from a set of roots to all other nodes. The roots may be initialized with different distance values, in particular, not necessarily with zero as in the usual definition of the algorithm (see Item 1(e) for a concrete example).

   (c) A shortest path from one node s to another node t. For reasons of efficiency, the algorithm should terminate immediately when the distance value of t is determined. Usually, Dijkstra's algorithm shall return a list of all nodes or edges on the path, which is ordered from s to t or the other way round.

6. Various non-algorithmic requirements on the functionality, for example:[11]

   (a) *On-line checkers*: A simple example of an on-line checker for Dijkstra's algorithm is to check that the sequence of updating values of node distances is monotonously increasing.

   (b) *Operation counting* ([AMO93], Ch. 18): The most interesting example might be the number of nodes and edges visited to compute a shortest path from some node s to some other node t.

   (c) Animation of the algorithm running in slow motion, for instance highlighting the currently processed node or the whole frontier line of the search (see Figure 1). It is important to note that animation is also useful for algorithm components which are implemented for other purposes. In fact, a careful animation allows visual debugging of the algorithmic code. If available, this greatly supports the debugging of complex algorithmic code. After all, some sorts of bugs only appear in very large instances—too large to be grasped other than by visual aids.

   (d) Possibly the concrete organization of the input and output data is not known at compile time, but is determined only at run time, immediately before the algorithm is invoked (*e.g.* by user interaction or by a configuration file). For instance, this may happen if the meaning of the edge lengths is not known at compile time. To give a concrete example: the length of an edge in a traffic network may be a combination of two or more criteria such as the estimated run time and the cost for traveling along this edge. This combination may reflect the preferences of an end-user, which are determined interactively and thus not known at compile time.

   (e) Interactive control of the running algorithm through a user interface.

   (f) Saving the current state of the algorithm frequently on an external device to recover after a crash.

   (g) Client code which calls an algorithm should be able to suspend or terminate the algorithm at frequent stop points in order to react on real-time requirements (otherwise, a complex, expensive multi-process solution is necessary).

[10] This is an example of the design pattern *flyweight* [GHJV95]. This scenario may appear in contextes in which the length of an edge models non-geometric aspects such as fixed edge costs.

[11] Items 6(e)-(g) might not be important tasks especially for Dijkstra's algorithm, since this algorithm is very fast even on very large instances. However, these items are listed here for completeness, because they may be important for many other algorithms with higher run times.

(h) *Robustness*: Roughly speaking, a component is called *robust* with respect to a specification of its input and output, if (i) it applies run-time checkers to test its (given) input and its (self-produced) output for conformance with this specification, and (ii) in case of a failure, invokes some exception-handling mechanism in a well-specified way. *Complete robustness* is achieved, when success of the run-time checkers is necessary and sufficient for the input and output to satisfy all aspects of the specification. On the other hand, we speak of *partial robustness*, when only necessary conditions are checked.

For example, Dijkstra's algorithm only requires that all edge lengths are nonnegative. Hence, a complete run-time check of the input is straightforward and efficient. In the scenario of Item 5(a), there is also a simple, efficient complete run-time checker for the output: for each node w except for the root node, there must be an edge (v, w) such that the distance of v is strictly smaller than the distance of w, and the absolute value of the difference equals the length of (v, w).[12] However, in the scenario of Item 5(c), this run-time checker only guarantees partial robustness, because some of the nodes in the graph were not visited. In fact, this test can only guarantee that there is no shorter path which solely contains nodes visited during the search. On the other hand, a complete robustness checker may even increase the expected asymptotic complexity of a single shortest-path computation in the scenario of Item 5(c), which means that such a checker may or may not be acceptable for debugging sessions, but must be turned off in the normal working process for reasons of efficiency.

In summary, the robustness of a component must also be flexibly adaptable to allow a context-specific compromise between robustness, efficiency, and adaptability. See [BW94] for an introduction to partial and complete output checking for implementations of algorithms.

3.2 Concrete Application

As mentioned above, we next discuss a concrete scenario: a traffic information system. Here 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. The core of the system is an algorithm component for shortest paths. For each pair, the system has to activate this component (see Figures 1 and 2). This is a special case within the wide range of algorithmic problems that are best solved by Dijkstra's algorithm ([CLR94], Sect. 25). Our concrete application offers a wealth of algorithmic speed-up techniques. Unless stated otherwise, s is the node from which the search starts.

1. 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, the usual, "textbook" version of Dijkstra's algorithm computes shortest paths from s to *all* other nodes. Unfortunately, the typical implementations of Dijkstra's algorithm in libraries [MN95] and in the literature [Fla95] follow the textbook version and do not even allow this simple speed-up technique.

2. In combination with the first technique, the following trick can be applied to achieve an expected run time that is sublinear in the size of the graph for a single shortest-path computation: the node distances are not initialized with a value representing infinity 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).

3. In *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

[12] In the presence of zero-length edges, the test is more complicated, but still efficient.

Figure 1: a snapshot of Dijkstra's algorithm from root Hannover in the German railroad network with respect to Euclidean edge lengths. The dark edges have already been processed.[13]

Figure 2: a shortest path from $s =$ *Hannover* to $t =$ *Leipzig*. Whenever s and t are relatively close, restricting the search horizon to a small ellipse reduces the run time dramatically.[13]

path from x to y: for example, the Euclidean distance of x and y ([Len90], Sect. 3.8.5.1). It is easy to see that this modification maintains the relative lengths of (s,t)-paths compared to each other. Hence, goal-directed search computes a shortest path with respect to the original edge lengths. However, there is empirical evidence that this modification may direct the search faster towards t.

4. In *bidirectional search* ([LLP83]; cf. [Len90], Sect. 3.8.5.2), a shortest (s,t)-path is computed from s and from t simultaneously. That is, two 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 on-line. The algorithm terminates when one node has been scanned by both searches. The concatenation of the shortest paths from s and from t to this node yields a shortest (s,t)-path.

5. Variations of the A^* algorithm for shortest (s,t)-paths in the plane also fit into this general scheme [KK88]. The effective length of an edge is a combination of its nominal length and an additional cost function, which punishes large angles between the direction of this edge and the direction towards t. In general, this variant does not guarantee a shortest path anymore. However, there is empirical evidence that the strong trend towards t, which is imposed by the additional cost function, may drastically reduce the number of nodes to be visited until a (usually short) path is found.

[13]These figures stem from a joint project with the *Deutsche Bahn AG* (http://www.informatik.uni-konstanz.de/Research/projects_algo.html#projekt7); permission to visualize the data is granted by the TLC/Deutsche Bahn AG.

To achieve a run time that only depends on the number of visited nodes and edges, not on the size of the whole graph, we must apply the trick introduced in Item 2 above. Moreover, we must compute the effective length of an edge on-line, namely when this edge is accessed during the search from s to t. The reason is the following: since the effective length of an edge depends on t, it must be re-computed for every shortest-path computation. The edges to be considered for a pair (s, t) are not known in advance. Hence, the only alternative to on-line computations would be to compute the effective lengths of *all* edges in the network before starting the search.

6. A shortest path from s to t which does not leave a certain region, for example, an ellipse (see Figure 2). This approach is reasonable whenever it may be safely assumed that the real shortest path does not deviate too much from the straight segment connecting s with t. For reasons of efficiency, it may be necessary that the algorithm disregards the (usually much larger) part of the graph outside the ellipse.

7. Such an ellipse may not capture the shortest path, but again for reasons of efficiency, it may be reasonable to start with this ellipse and to add selected nodes and edges outside this ellipse on-line, that is depending on the current state of the shortest-path algorithm. In other words, two algorithms—a shortest path routine and a routine that extends the search horizon on-line—are "merged" into one loop and affect each other.[14]

4 Discussion of the Goals

As announced in the Introduction, we next discuss the goals of our project in light of the acid test in Section 3.

Like many other basic graph algorithms, Dijkstra's algorithm is not too hard to implement. Hence, the effort needed to adapt a very general implementation may equal or even exceed the effort needed to implement, from scratch, an almost optimal version that is tailored to the specific context. However, we think that, from a software engineering point of view, there are still good reasons in favor of general-purpose algorithm components that are adaptable to various contextes by some additional customization code:

- Customization is a relatively straightforward task (provided a good documentation is available, of course), whereas implementing even a simple graph algorithm requires some basic understanding.[15]

- Usually, algorithmic code is complex and fragile. Hence, maintenance of a software system is much easier if the algorithmic stuff is not affected when changing requirements are to be incorporated. This is achieved when every algorithm component is implemented such that a change of requirements only amounts to changing some customizing code, which is non-algorithmic in nature.

  This aspect is even more important since the people doing the maintenance need not necessarily be the people who developed the package.[16] Hence, it cannot generally be assumed that the people who maintain an algorithm component are algorithmicians themselves.

- It is often argued that basic components which are heavily reused are much more reliable than ad-hoc solutions (for example, this is extensively discussed in [Mey95], Chap. 6).

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

[15] Frequently, one can observe articles in various newsgroups, which are posted by people with little background in graph algorithms and urgently ask for implementations of Dijkstra's algorithm and other "simple" graph algorithms. This shows that do-it-yourself is often not a feasible alternative for programmers of client code.

[16] Some companies even delegate this task to other companies, which specialize in maintenance; see [Pig97].

- In our experience, the code that customizes an algorithm can be checked by the compiler or any other automatic tool very rigorously (at least in statically typed languages), whereas algorithmic correctness cannot be checked at all by the compiler. Hence, if all changes only affect the customizing code, the result of the revision is more reliable.
- When code must be ported which is based on a third-party graph library, chances are high that this library is not available on the target platform for whatever reason (for example, see the discussion in the preface of [MS95]). If the code is implemented from scratch and tailored to the original platform and the original library, customizing the code to the new context is very expensive and error-prone. On the other hand, if all algorithms are implemented from the very beginning in view of customization, the pieces of code that must be adapted should turn out to be small, and the modifications might not affect the rest of the code.
- Even a "simple" algorithm such as Dijkstra's can be speeded up by heuristical techniques which are not found in most textbooks about graph algorithms (see Section 3.2 for a few examples). Hence, chances are high that an implementation from scratch by a non-expert is not as efficient as it could be. If an all-purposes implementation comes with a documentation and an environment which formulate, support, and encourage various algorithmic customizations including speed-up techniques,[17] this gives a programmer of client code an opportunity to make use of these techniques. However, it is important that such heuristic techniques are not "hard-wired" in the algorithm, but can be easily turned on and off, because a heuristical technique may be useless or even increase the run time in contextes for which it was not developed.
- Reused components need not be documented again. This may reduce the effort for documentation significantly. The client code becomes more self-explanatory, and the documentation of client code may be leaner.

It remains the question whether it is reasonable to aim at *one* implementation for *all* contextes. A less ambitious aim would be to implement several variations of Dijkstra's algorithm, each of which only covers a subset of all cases in Section 3. However, the following aspects might make an all-in-one solution desirable.

- A real-world context might consist of a combination of the above contextes, and all of these combinations must be covered. It seems that this can only be guaranteed by a single implementation which covers virtually all contextes.
- When minor details of the context change, chances are high that a variant which only covers a subset of all contextes cannot be adapted to the new context and must be replaced by another variant. This may result in a major revision of the client code.

5 General Goals

As in Section 2, we will distinguish adaptability to the lower level and to the higher level from each other. In both cases, we will formulate, explain, and discuss a few concrete goals, which try to capture the lessons learned from the acid test.

5.1 Adaptation to the Lower Level: Underlying Data Structures

Items 1-4 in Section 3.1 and Items 2, 3, and 5 in Section 3.2 are concrete examples where adaptability to the lower level is required.

[17] Basically, by that we mean a documentation which includes an excerpt from Section 3.2 on a more technical level and a set of auxiliary data structures such as various priority queues, which support the different alternatives. A first attempt to that was made in [NW96], where the environment is composed of the LEDA-specific implementation of the toolbox and of the data structures provided by LEDA itself; see also http://www.mpi-sb.mpg.de/LEDA/www/ledaep.html.

General goal 1: *The underlying data structures should not be hard-wired in an algorithm component, but kept variable using some sort of polymorphism.*

A type is *polymorphic* within some component, if the name of the type in the source code of this component is a formal "placeholder" for an actual type, which can be chosen from a certain set of types. In other words: the actual type is not fixed, but may be chosen from some set of types when the component is used. The set of types from which the actual type may be chosen is usually described by syntactical prerequisites. See the appendix, Section C, for a more extensive discussion of polymorphic features of the mainstream object-based and object-oriented programming languages.

It might be obvious that polymorphism is absolutely mandatory to achieve adaptability to the lower level.

General goal 2: *An algorithm component should not access the complex data structures directly, but through some additional layer of indirection. This layer should decouple the algorithm component from the data structures such that a modification of the data structures only requires a revision of the code in this layer, not of the algorithm component itself.*

In other words, the collaboration of algorithms and data structures is mediated by some additional helper types, which connect algorithms with data structures (and also separate them from each other in some sense). Adapting an algorithm to a data structure amounts to implementing these helper types, and changes of this data structure are only propagated to these helper types. In particular, no sophisticated algorithmic stuff must be changed when the underlying data structures are changed. In Section 4, we noted that this may make maintenance easier and more reliable.

Such an additional layer of indirection is necessary because otherwise one cannot generally assume that an algorithm component and its underlying data structures fit seamlessly together. After all, the details of the underlying data structures may be dictated by the context, and the context may dictate an organization that does not fit into the algorithm component.

General goal 3: *The additional layer of indirection addressed in the second general goal should be implemented in a very modular fashion, that is every aspect of the interface to the lower level should be factored out into a separate, very small component in this layer.*

In principle, this means that the interfaces of algorithms to the lower level may be organized as a "toolbox," which contains a couple of polymorphic components, and from which the interface may be constructed. In the ideal case, customizing an algorithm to the lower level amounts to instantiating these polymorphic helper types in a context-specific manner and to plug these instantiations together.

In particular, the following consequences motivated this goal:

- A minor change in the implementation of the underlying data structures should only cause minor, local changes in the implementation of the indirection. For example, if the organization of a single node or edge parameter is modified (Item 4 in Section 3.1), this should not affect more than one (very small) component in this layer.
- Likewise, if the functionality that an algorithm component requires from the underlying data structure is changed slightly (*e.g.* if an algorithm requires an additional node or edge attribute from an underlying graph), only a few, very small components should be affected.
- Sometimes it is desirable to insert additional functionality in this mediating layer of indirection. For example, it may be desirable to perform some additional bookkeeping inside this layer. Again, integrating such functionality should only cause minor, local changes.

- In some cases, an algorithm shall be applied to a data structure that does not really fit into the algorithm's requirements. Item 4(d) in Section 3.1 is an example of that: the cordinates of nodes are provided by the underlying graph data structure, but the lengths of edges are required by the algorithm. The necessary computation must be performed inside the additional layer of indirection which adapts the data structure to the algorithm. The algorithm to compute the length of an edge from the node coordinates should be factored out into a single component.

The building blocks of such a modular decomposition of the additional layer of indirection must be polymorphic to allow a flexible construction of this layer. In every context in which Item 6(d) in Section 3.1 is not relevant, static polymorphism suffices for that. "In theory," the overhead caused by a modular decomposition may then be negligible, because it may be optimized away by the compiler. It might be worth mentioning that a few C++ compilers already come close to this ideal.

5.2 Adaptation to the Higher Level: Client Code

General goal 4: *An algorithm component should offer its clients frequent occasions to invoke context-specific code during the execution of the algorithm. There should be at least one occasion in every iteration of the core loop(s).*

Probably every non-trivial algorithm essentially consists of one or a few loops (or nested loops), plus some pre- and postprocessing operations for each loop.[18] We call these loops the *core loops* of the algorithm. For example, Dijkstra's algorithm 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.

Obviously, the fourth general goal is a necessary condition for Items 6(a)-(c) and (e)-(f) in Section 3.1.

General goal 5: *An algorithm component should offer its clients frequent* stop points *throughout the execution of the algorithm such that the client is able to terminate the algorithm at any such stop point or suspend it temporarily and continue the execution later on.*

This goal is motivated by Item 5(c) and Items 6(e) and (g) in Section 3.1.

General goal 6: *At each of the stop points addressed in the fourth general goal, an algorithm component should allow full read access to its* logical state (*a restricted, disciplined write access may also be useful*).

Roughly speaking, the *logical state* of a component at some stage of its life time is defined as the bunch of information about its current state which is necessary to predict its behavior in all possible threads of interaction with other components throughout the rest of its life time.

For example, many libraries provide stack classes which offer access to the topmost element only, namely through the methods *push*, *pop*, and *top*. Interacting with such a class means calling its methods. 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. Only this whole bunch of information suffices to predict the outcome of all possible sequences of future *push*, *pop*, and *top* requests.

Items 6(a)-(c) and (f) in Section 3.1 obviously require full access to the logical state of a component implementing Dijkstra's algorithm. Moreover, many speed-up techniques rely on extensive knowledge about the current state of the core algorithm. For example, Item 4 in Section 3.2 is of

[18] Often, algorithms are presented in a recursive style. Clearly, every recursion can be turned into an iteration. For example, [Fla95] provides guidelines for that task especially for C++.

this kind: the on-line choice of the algorithm which shall perform the next step requires knowledge about the current logical states of both algorithms. However, the subalgorithm that performs this choice is not fixed, and hence not necessarily predictable at the time when Dijkstra's algorithm is implemented as a component. In particular, it is not predictable which knowledge about the current logical state is required by this subalgorithm. Hence, there might be no alternative to full inspectability of the logical state.

The stack example shows that the sixth general goal contradicts the idea of abstraction: from an abstract viewpoint, a stack is a type which is only accessible at its front. Type abstraction is also desirable and is often recommended as a principle to achieve clean designs. However, for an algorithm component, the "right" abstraction is not that clear. For example, in Section 2.2, Item 2, we have seen that the right abstraction of the output of an algorithm as simple as depth-first search is not at all clear. This indicates that rigorous abstraction should not be the main goal for the development of algorithm components.

Note that, nonetheless, the sixth general goal does not contradict the encapsulation of implementation details. Hence, the most important argument for abstraction is not an argument against this goal.

General goal 7: *Preprocessing and postprocessing should be separated from the core of an algorithm and regarded as some kind of customization rather than as a part of the algorithm component. The residual core should be the unit of reuse, not the algorithm as a whole.*

Dijkstra's algorithm includes some preprocessing, but no postprocessing. In the preprocessing, the priority queue and the node distances are initialized.

From an algorithmic viewpoint, the only difference between Items 5(a) and 5(b) in Section 3.1 is in the initialization: the roots of the search (along with their distance values) are the initial members of the priority queue. Hence, making an implementation of Dijkstra's algorithm customizable to both cases amounts to extracting the initialization of the queue from the algorithm component and to make it a part of the customizing stuff.

Item 2 in Section 3.2 demonstrates that the initialization of the node distances should not be a hard-wired part of the algorithm component either. On the other hand, Item 2 in Section 2.2 discusses an example where a flexible postprocessing is desirable.

General goal 8: *The core loops of several algorithm executions should be "mergeable" into one loop.*

The bidirectional search technique for Dijkstra's algorithm (Section 3.2, Item 4) is an example. Moreover, Item 7 in Section 3.2 may be viewed as another example, in which two different algorithms are merged into one loop.[19]

General goal 9: *The specific format of the output of an algorithm component should be modifiable with minor effort.*

This goal is motivated by the remarks on variations of the output in Item 5, Section 3.1 (see also Item 2 in Section 2.2).

General goal 10: *Algorithm components which are exchangeable in an application from a theoretical viewpoint should also be easily exchangeable in each context based on this application.*

This goal is not motivated by the acid test in Section 3, but by Item 1 from Section 2.2.

[19] To mention another important example: a component which realizes a search strategy such as *simulated annealing* [Vid93] can be easily applied in a pseudo-parallel fashion, if the component satisfies the eight general goal. This means that several instantiations of this algorithm explore different parts of the search space and every instantiation performs exactly one search step until the next one is invoked in a round-robin style.

6 Review of Existing Approaches

Again, we consider the adaptation to underlying data structures and the adaptation to client code separately. In each case, we will group the discussion by a few keywords, which try to capture the essential ideas behind the various approaches.

6.1 Adaptation to the Lower Level: Underlying Data Structures

Type specialization

Item 1(b) in Section 3.1 suggests an approach based on type specialization. Basically, this means that every special class of graphs is realized by a separate type, but these types are not stand-alones. Instead, if type A represents a graph class which is a special case of the graph class of type B, then A is implemented as a specialized version of B. Every algorithm which is based on B may then also be applied to an object of type A.

To give a concrete example: in LEDA, the type *planar_map* is derived from the general graph type. Hence, every graph algorithm for general graphs may also be applied to a planar graph, even if this graph is not organized as an object of the general graph type, but as a planar map.

Moreover, this kind of type specialization can also be used to attack the other parts of Item 1 in Section 3.1. For example, the LEDA class for general undirected graphs is declared to be a specialization of the class for general directed graphs. Hence, an algorithm that is designed for directed graphs may also run on undirected graphs (see Item 1(a)).

LEDA's design is not based on type specialization, and hence its use in LEDA is rather sporadic. In contrast, for instance, the PlaNet package[20] was an attempt to base the design of a visualization tool for graph algorithms on a hierarchy of type specializations, which was intended to naturally reflect the hierarchy of abstract, mathematical graph classes. At any stage during a session, the user is confronted with a current special graph class, which (s)he can replace by another special graph class at any time (by direct choice or by navigation through the class hierarchy). The system offers the user all objects from the database of graph objects which belong to this graph class or one of its specializations and all algorithms for this graph class or one of its generalizations. Hence, it is guaranteed that every eligible algorithm works fine with every eligible graph.

It has turned out that, in principle, this approach is feasible. However, in this project, we constantly encountered a couple of problems, which suggest that this approach is not a good base for algorithm components:

- *Restricted functionality*:

  As mentioned in Item 1(b) of Section 3.1, a data type A for a specialized graph class usually cannot offer all functionality of the general graph class B from which it was derived through type specialization. In fact, otherwise, it is impossible to safely guarantee that every object of A actually represents a graph from the corresponding special abstract graph class. There are two possible options.

  First option: A provides all functionality promised by B and allows that objects of type A leave the corresponding abstract graph class. This option is often chosen. Typically, such a type A is equipped with an additional method that checks whether the object currently falls into this graph class. This option is not always practical. For example, if B represents general graphs, B might offer means of adding or deleting nodes or edges. On the other hand, if A represents rectangular grid graphs [Len90, NC88], the graph might be represented as some kind of matrix. In this case, realizing such basic modifying operations requires additional internal bookkeeping, which might be more complex (and less reliable) than the implementation of the grid itself and significantly increase the overhead in development/maintenance and run time. However, even if this option is practical: experience shows that a class A that promises, but does not guarantee, that every object fulfills certain properties may be a major source of inconsistencies.

[20] See http://www.informatik.uni-konstanz.de/Forschung/Projekte/PlaNet.

Second option: A only offers restricted functionality compared to B and thereby ensures that no object of type A leaves the corresponding abstract graph class. However, this means that the semantics promised by B are not fulfilled by A, which violates the *Liskov substitution principle* [Lis88, Mar96]. A common attempt to overcome this problem is to offer operations which would drive an object out of its special class, but to catch such requests and invoke an exception-handling mechanism. However, this still means that an object of type A does not behave as promised by B. FAQ #118 in [CL94] characterizes this sort of design drastically: "a design error that creates a mess."

- *Conflicting options for type specialization*:

  This is the classical "circle-ellipse dilemma." The relation between directed and undirected graphs is an example of this dilemma from graph algorithmics.

  Many algorithms for directed graphs are also applicable to undirected graphs. In Section 3.1, Item 1(a), an undirected graph was interpreted as a directed graph by viewing each undirected edge $v - w$ as a pair of antiparallel edges, $v \to w$ and $w \to v$. In other words, undirected graphs may be viewed as the special case of directed graphs in which the graph is symmetric and the attribute values of two antiparallel edges are identical.

  This is not the only possible way of interpreting undirected graphs as a specialization of directed graphs. For instance, in LEDA, undirected graphs are also a specialization of directed graphs. However, here an undirected edge is just a directed edge for which the orientation is meaningless. This relation between directed and undirected graphs is appropriate for many algorithms which are naturally formulated in terms of undirected graphs, for example, algorithms for spanning-tree and matching problems.

  These options seem to be inherently contradictory. Moreover, in the first option (*i.e.* undirected graphs are symmetric graphs with symmetric edge attributes), there are further options in case an algorithm modifies the values of an edge attribute. For example, in algorithms for single-commodity flow problems, a flow augmentation step might change the flow on two antiparallel edges such that the sum of their flows is maintained. On the other hand, algorithms for multi-commodity flow problems usually treat the flows on two antiparallel edges separately.

  In summary, a natural type specialization relation between directed and undirected graphs which is suitable for all algorithms does not seem to exist. Many further examples of this general problem can be found in algorithmics.

- *Proliferating hierarchy*:

  The literature on pure and algorithmic graph theory reveals a huge special-case hierarchy of graph classes. Of course, in no software package can this hierarchy be completely realized. Hence, only a small fraction of this conceptual hierarchy will be realized. However, this fraction will change over time. The crucial point is that this typically means more than merely adding new leaves to the existing hierarchy. In fact, an extensive restructing effort might be necessary to keep the hierarchy manageable after new leaves were added. This is usually called *refactoring* and has been identified as one of the major maintenance problems in object-oriented modeling (for example, see [OJ93]).[21]

Section C in the appendix discusses the fact that (dynamic) polymorphism and type specialization are strongly coupled in various programming languages. In particular, the first general goal in Section 5.1 can be achieved by type spezialization. Moreover, the second general goal can also be achieved: if an algorithm is based on a base class A to represent graphs, adapting the algorithm

[21] Nonetheless, on a conceptual level, it is desirable to lean the abstract specification of the hierarchy of graph classes on the mathematical special-case relationships. The first part of [Wei98] introduces programming techniques which are intended to help translate such a mathematical hierarchy into a manageable software system (using geometric shapes as a running example).

to a graph data structure amounts to implementing a specialized version B of A, which "wraps" an A-compatible interface around the data structure (design pattern *adapter* [GHJV95]).

However, this use of type specialization does not address the third general goal (nonetheless, techniques which do address the third general goal may in turn profit from a well-devised incorporation of type specialization).

Composition of components in layers

This general technique has two different aspects.

- *Extending the functionality through additional layers*:

  This is also known as "vertical design." In the first place, it addresses the third and fourth items in the discussion of the third general goal in Section 5.1. Components are built from primitive building blocks. Every building block is designed to form an intermediate layer, which adds a specific aspect of functionality to the whole component. Building a complex component then amounts to selecting the building blocks whose functionality is desired and piling these blocks on top of each other. For example, [BSST93] demonstrates this technique for normal queues. Every aspect, such as the choice of the underlying container type (linear list, doubly-linked list, array, etc.), the storage management, and an optional garbage management technique, is realized by a building block. Each building block is based on an abstract interface of a queue and in turn provides an abstract interface of a queue. This allows a composition in layers.

  As a prerequisite, these components must be "plug-compatible." Ideally, each building block provides the same polymorphic interface. Then an arbitrary selection of building blocks may be plugged together in an arbitrary order. However, [BSST93] demonstrates that this total compatibility is not even realistic for components as simple as normal queues, not to mention more complex components such as graphs.

  This means that the choices for two different aspects cannot be assumed to be polymorphically exchangeable. Therefore, the plugging of aspects must be organized according to strict rules: every aspect is assigned one layer of the composition, and every choice for this aspect must be based on the interface of the next lower level. In particular, when a component is built from such a layered set of building blocks, none of the predefined layers may be omitted. This seems impractical: whenever a new aspect comes into the game, all existing components which were built from this set must be revised.

- *Modifying the functionality through additional layers*:

  This aspect addresses one of the problems of type specialization which were discussed in the beginning of this subsection: conflicting options. To stick to the example of directed and undirected graphs: a directed graph data structure DG and an undirected graph data structure UG are not made compatible by means of inheritance, but by additional types, DGV and UGV. An object of the type DGV (resp. UGV) keeps an internal UG (DG) object, offers the same interface as DG (UG), and realizes a directed (undirected) view on the internal UG (UGV) object. Hence, DG and DGV may be made polymorphically exchangeable, and likewise UG and UGV. So, a component that implements an algorithm on directed graphs may be applied to a real directed graph of type DG or to an undirected-viewed-as-directed graph of type DGV, and vice versa.

  For instance, this design principle is reflected by the various graph types in the standard library for the Sather programming language.[22] To give a concrete example: there is a graph-view type which keeps an internal directed graph and provides the same interface as this internal graph. However, this type exchanges the meaning of in-going and out-going edges of a node: every in-going edge in the internal graph is presented as an out-going edge

[22] See the Sather homepage; http://www.icsi.berkeley.edu/Sather.

and vice versa. In other words, this type simulates the reverse graph of the internal graph. Another graph-view type realizes a *filter* on directed graphs. This type is associated with two *predicates*, P_n and P_e, where P_n applies to nodes and P_e to edges. The view presented by this type is a subgraph of the internal graph. This subgraph contains all nodes for which P_n is fulfilled and all edges such that P_e is fulfilled for this edge and P_n for both endnodes.

One problem occurs in both aspects of composition in layers: the third general goal formulated in Section 5.1 is not really achieved. In fact, the functionality introduced by a building block in the vertical design of a queue (see above) may concern many methods of the queue, potentially all of them. For example, garbage collection must at least be incorporated in the insert and remove methods of the queue, in the constructor, and in a method for final clean-up. On the other hand, a reverse graph view like in the Sather graph library might affect virtually every method. Consequently, every building block is a "fat" component, which contradicts goal 3.

Aspect-oriented programming

This new concept[23] does not fit into our discussion, because it relies on language features which have been designed just to realize this concept and are not supported by any mainstream programming language. We briefly touch upon it, because it has been gaining much attention throughout the last few years. To classify this approach according to our general goals: it may be viewed as an attempt to combine the advantages of vertical design (see the previous approach, composition of components in layers) with the third general goal from Section 5.1.

Like vertical design, aspect-oriented programming factors out every aspect into a separate set of exchangeable components, which realize this aspect in different ways. However, these components are of a new kind: each of these components realizes a choice for an aspect in its pure, abstract form, which means that it does not depend on the details of the implementations of the other aspects. This is where the new language features are needed: any such component is explicitly defined to serve as a choice for an aspect, and a component is defined in the source code by a set of choices for aspects. It is the compiler's task to build the component by interweaving these aspects into one block of code.

Current implementations of aspect-oriented programming features are still experimental and subject to on-going research. Hence, at this point, the advantages and disadvantages of this approach cannot be seriously discussed.

Linear iterators

The concept of *iterators* is quite common in object-oriented programming and one of the fundamental design patterns [GHJV95]. We will briefly review this concept.

An iterator type is a light-weight type (typically one or two pointers and possibly some bookkeeping information) and associated with a linear container class, for example, linear list, stack, queue, array, or search tree. A valid object of an iterator type is associated with a container object and a single item in this container. At least, an iterator provides methods for three tasks:

1. accessing the data of the associated item;
2. testing the iterator for validity;
3. advancing the iterator to the very next item (once all items have been passed, the iterator becomes invalid).

The iterator approach has been adopted in various previous work. Among all of this work, the design of the STL [MS95] had—and still has—the most practical impact. In fact, the core of the STL has become part of the C++ standard library,[24] and the design principles of the STL have

[23] See the home page of the aspect-oriented programming project, http://www.parc.xerox.com/spl/projects/aop, for further details.

[24] C++ Final Draft International Standard, ISO/IEC 14882.

greatly influenced the other parts of this library. Section D in the appendix briefly describes the design ideas behind the STL.

Iterators may form an additional layer of indirection as required in the second general goal from Section 5.1. The interface of this layer to algorithms may be kept polymorphic, so the first general goal is also achieved. The design of the STL demonstrates this: every linear container type comes with some specific iterator types, and every algorithm solely accesses the underlying containers by means of iterators. Hence, exchanging a data structure amounts to exchanging the representing iterator type. To some extent, an iterator-based design also achieves the third general goal in Section 5.1. The key insight is that composition in layers (see above) may also be applied to such an iterator-based layer of indirection.[25] Experience shows that the problems of this approach, which were discussed above, are much more easily manageable in this case, because only the iterator types are involved. These types are small and not too sophisticated (and thus easy to revise in a maintenance process).

Ref. [GKW98] implements an exemplary building block for composing iterators in layers. An object *FI* of this *filter iterator* type contains an object *I* of another iterator type, which is polymorphic in *FI*, and provides access to a selection of the items over which *I* iterates. More specifically, *FI* is associated with a *predicate*. This predicate applies to the items over which *I* iterates. The advance method of *FI* advances *I* until *I* meets an item that fulfills the predicate (or until *I* becomes invalid, in which case *FI* also becomes invalid).

However, two important questions remain open:

- The concept of iterators has been developed for linear containers. To be useful for algorithmics, it must be extensible to non-linear data structures such as graphs.
- Items 1(a) and (b) in Section 3.1 require that an algorithm is adaptable to different data structures. This is not a real problem in the STL, because there all data structures are linear and hence not too different from each other. In contrast, the differences between various graph abstractions need further consideration.

Adjacency iterators

In principle, it is well understood how the concept of iterators may be extended to complex data structures such as graphs. Several libraries reflect this, for example, the AAI base class library.[26] The following kinds of iterators are needed especially for graphs:

- A *linear node iterator* type, which iterates over the set of all nodes in a graph.
- A *linear edge iterator* type, which iterates over the set of all edges in a graph.
- An *adjacency iterator* type, which is associated with a fixed node and iterates over the nodes adjadent to this node (in other words, over the edges incident to this node).

Navigation through the structure of a graph may be reduced to a sequence of operations on a dynamically changing set of adjacency iterators. This set always identifies the nodes which form the current "frontier line" of the navigation (see Figure 1). The current adjacent node (incident edge) of an adjacency iterator implicitly distinguishes the edges incident to this node which have already been scanned for navigation from the edges incident to this node that were not yet seen. To push the frontier line forward, adjacency iterators must provide an additional clone method, which applies to every valid adjacency iterator and places a new adjacency iterator on the current adjacent node.

For example, a depth-first search could maintain a stack of adjacency iterators. A forward step in this search would then amount to calling the additional clone method for the top element of the stack, pushing the resulting adjacency iterator onto the stack, and advancing the previous

[25] Unfortunately, this is not a central design principle of the STL, and only two building blocks for iterators are provided in the C++ standard; see Section D in the appendix.

[26] http://www.aai.com/AAI/IUE/spec/base/base-classes.html

top element to its next adjacent node. A backtrack step is performed whenever the top element is not valid or is placed on a node that has already been seen before. In contrast, a breadth-first search could maintain a normal queue of adjacency iterators. Here, a forward step would mean to remove the top element I from the queue, traverse the list of the nodes adjacent to I, apply the additional clone method in every iteration step to I, and append the resulting adjacency iterators to the queue.

Linear node and edge iteration are special cases of sequentially iterating over a linear sequence of items, which is well understood. However, the third type of iteration raises two questions:

1. What exactly do the terms "adjacent" and "incident" mean here? What do they mean for directed graphs; what for undirected graphs and other variants?

2. Item 1(a) in Section 3.1 requires that the meaning of "adjacent" and "incident" for one type of graphs be mapped onto the meaning for another type of graphs (see the second open question at the end of the previous approach, linear iterators). How can this mapping be achieved without sacrificing the third general goal in Section 5.1?

To the best of the author's knowledge, these questions have not been addressed rigorously or formally up to now. However, they seem to be too complex to be discussed purely informally. Section E in the appendix introduces a rigorous formal framework, which is intended to answer these questions satisfactorily.

Data accessors

Item 4 in Section 3.1 is not at all addressed in any of the programming methodologies discussed so far. In fact, in typical implementations of graph algorithms, the organization of the node and edge attributes is hard-wired, even if otherwise much care is taken to decouple the algorithm from the organization of the graph structure. This is a severe gap in the encapsulation of the underlying graph data structure. The concept of *data accessors* is intended to fill in this gap. This concept was originally introduced in [KW97] to enhance the flexibility of the algorithms in the STL (see Section D in the appendix).

The key abstraction for Item 4 in Section 3.1 is to regard an attributed graph (*i.e.* a graph with node and edge attributes) as a graph that is equipped with two *tables*: one table of node attributes and one table of edge attributes. By a table, we mean a bunch of data that is organized into *rows* (nodes/edges) and *columns* (attributes). All entries in a column are of the same data type, which may be regarded as the column's type. Every row is a record which contains one slot for every column. In this perspective, a table is merely a conceptual entity; the physical organization of the data in data structures is arbitrary and need not reflect this conceptual viewpoint. Typically, an implementation of a table is based on a linear container type, whose items are associated with the rows of the table. For example, the item type of this container may be a record type comprising one slot for each column. Alternatively, the container may contain a mere ID for each row, and the values of the individual attributes are stored in separate data structures. Note that these two extreme cases correspond to Items 4(a) and (b) in Section 3.1 (see also the discussion of Problem II in Section 2.1).

In any case, iterators can be used to sequentially access the rows of a table. This decouples algorithms from the concrete organization of the sequence of rows. However, the organization of the *columns* is not encapsulated. For instance, suppose a table is organized as a list of a record type which stores the values of all columns, and suppose this table contains a column named `a`. Reading and overwriting the value of this column in the row identified by the STL-style iterator `it` looks like this in C++:

```
x = (*it).a;
(*it).a = y;
```

The expression `(*it)` evaluates to a reference to the current row of `it`. The second line of code is legal only if the attribute `a` is writable. If such a statement occurs in an algorithm, this hard-wires

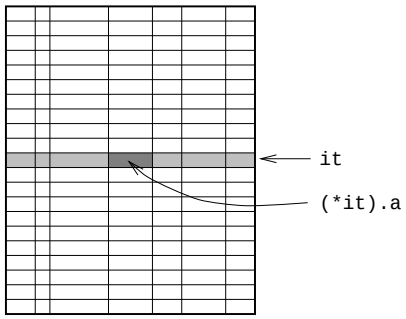

Figure 3: the acess to column `a` in the STL version. A reference to the grey row is returned by the expression `(*it)`. The row must exist as a materialized object of a record type.

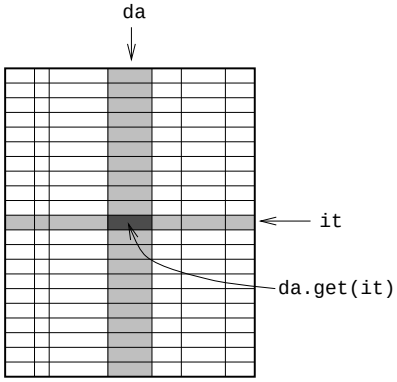

Figure 4: the modified scenario when data accessors are used. The iterator accesses the grey row, and the data accessor, the grey column. The individual columns may be organized arbitrarily (not necessarily materialized).

the organization of each row as a record of column values. To change the organization of the attributes, each such line of code in the algorithm must be modified. This contradicts the second general goal in Section 5.1.

To encapsulate the access to column `a`, this task is delegated to a *data accessor* object `da`, which is firmly associated with its column (in contrast to iterators, which are not firmly associated with individual rows). This object provides a method `get`, which receives an iterator `it` as an argument and returns the value of its associated column in the row identified by `it`. Moreover, if this column is writable, `da` provides an overwriting method `set`. In summary, if a data accessor `a` is used to encapsulate the access to `a`, the above code snippet in STL style would be replaced as follows:

```
x = da.get(it);
da.set(it,y);
```

The syntactical difference is small. However, it bears strong consequences: if such a line of code appears in an algorithm and `da` is polymorphic in this algorithm, then the meaning of `get` may be changed silently by exchanging the actual type of `da`. This closes the gap in the encapsulation of complex data structures, which violated the second general goal in Section 5.1. Figures 3 and 4 illustrate the difference.

Like iterators, data accessors may also be organized by composition in layers. Ref. [KW97] demonstrates this by means of an example from business administration: the rows of a table are the employees of a company, and the total salary of an employee is composed of a fixed and a variable part. These two parts are attributes of employees. A data accessor for the total salary may be a primitive building block based on two further data accessors: one for the fixed part and one for the variable part of the salary. Hence, if the meaning of the total salary in relation to the fixed and the variable part changes (*e.g.* if a third kind of salary is added), only this building block is affected.

Method `get` of this building block calls the method `get` of both underlying data accessors and returns the sum of their results. On the other hand, method `set` overwrites the variable part by the second argument (which is regarded as the new total salary) minus the fixed part.[27]

Items 4(c) and (d) in Section 3.1 are further typical tasks of data accessors composed in layers: a caching mechanism for an attribute is naturally delegated to a building block for data accessors.

[27] This example demonstrates that even non-materialized attributes may be writable.

On the other hand, computing the Euclidean length of an edge from the coordinates of its endnodes is naturally realized as a building block which is based on data accessors for node coordinates.

A combination of iterators and data accessors is especially useful to achieve the third general goal from Section 5.1. In fact, such a combination separates row-oriented aspects such as iteration over selected rows (the filter iterator in the above section about linear iterators) from column-oriented aspects such as computing the Euclidean length of an edge. If such an aspect is modified, only the corresponding component must be changed. It seems that this cannot be satisfactorily achieved unless column-oriented aspects are factored out into components like data accessors, which are only loosely coupled with iterators.

We refer to [GKW98, KW97, Wei97] for all technical details and in-depth discussions.

Integration of algorithms and data structures

Finally, a completely different approach should be mentioned, which does not regard the underlying data structures as a base of algorithms, but as an integral part of them. Roughly speaking, this means that the algorithm plus its underlying data structures are viewed as one, undivisible component.

For example, Ref. [WO94] introduces and discusses such a design for Kruskal's algorithm for minimum spanning trees. In the Sather graph library,[28] integration of algorithms into data structures is a major design concept. The idea is as follows: suppose that graph algorithms $A_1, \ldots, A_k$ are applicable to a graph data structure G. Then a new data structure, G', may be derived from G, such that G' contains k methods, each of which performs one algorithm A_i on the graph object.

Clearly, this aproach does not address the general goals in Section 5.1.

6.2 Adaptation to the Higher Level: Client Code

Tradionally, algorithms are implemented as subroutines. In contrast, in all of the following approaches, algorithms are implemented as abstract data types (see Section A in the appendix). Such a data type provides one or more methods which execute the whole algorithm or parts of the algorithm. The latter kind of methods allows the client to control the execution of the algorithm. If this control is fine-grained, that is if these methods perform small, fast steps of the algorithm, the overhead caused by the additional method calls may not be negligible. However, modern compilers might be able to optimize away these method calls through in-lining techniques (at least if the algorithm component is not dynamically polymorphic in the client).

Strategy pattern

This is one of the fundamental design patterns [GHJV95] and addresses the tenth general goal in Section 5.2. The other general goals in Section 5.2 are not addressed. Every algorithm is implemented as an abstract data type which provides a method to execute the algorithm as a whole. If two algorithms perform the same task, that is if they are exchangeable from a theoretical viewpoint, they are kept exchangeable by means of polymorphism.

In programming languages which allow one to treat subroutines as first-class objects (see Section B in the appendix), polymorphism of algorithms can be expressed even if they are not implemented as abstract data types, but as mere subroutines. Nonetheless, if algorithms are implemented as data structures, the strategy pattern can often be applied more smoothly and conveniently. The reason is the following: assume that algorithms $A_1, \ldots, A_k$ are exchangeable from a theoretical viewpoint. Now, every algorithm A_i may be parametrized by additional control parameters, which appear in the argument list of A_i, but not in that of any other algorithm A_j. However, this destroys exchangeability. On the other hand, if $A_1, \ldots, A_k$ are implemented as abstract data types, these control parameters can be moved to the argument lists of the constructors.

[28] http://www.icsi.berkeley.edu/Sather.

Hence, the methods of $A_1, \ldots, A_k$ to execute the algorithms proper may all have the same syntax, which allows exchangeability.

In the beginning of Section 2, we mentioned that composing algorithms with subalgorithms may be viewed as a special case of the problem to adapt algorithms to clients. This suggests to apply the strategy pattern to keep subalgorithms exchangeable inside an algorithm. In a case study on algorithms for the maximum-flow problem, Gallo and Scutella apply this idea rigorously to compose an algorithm hierarchically from its subalgorithms at run time [GS93]. However, apparently, the subalgorithms in [GS93] were not designed for adaptability to various contextes; they were just designed to fit into the system presented in [GS93], which is entirely devoted to the maximum-flow problem. It is not clear whether this approach is compatible with flexible adaptability. In fact, an algorithm A may be exchangeable with algorithm B in one context and with algorithm C in another context, but B and C are not compatible. A simple example is breadth-first search, which is exchangeable with depth-first search in some sense, and with Dijkstra's algorithm in another, incompatible sense.

One possible approach is to implement all algorithms as stand-alones and to add context-specific polymorphism afterwards. The first part of [Wei98] demonstrates this technique in C++.[29]

Algorithmic generators and loop kernels

Algorithmic generators have been introduced by Flamig [Fla95]. Roughly speaking, an algorithmic generator is an algorithm component which computes and delivers its output step-by-step on request. The running example in [Fla95] is an algorithm that computes all prime numbers within a specified interval of integral numbers in increasing order. At any stage, the logical state of an instance of the algorithmic generator is defined by this interval and the maximum prime number already delivered. The component provides a method to deliver the very next prime number. To fetch all prime numbers, a client must contain a loop, in which this method is invoked until the algorithm generator indicates that all prime numbers have been delivered (*e.g.* this can be indicated by returning an impossible value).

The main intent of algorithmic generators is to achieve the ninth general goal in Section 5.2. The idea is the following: if the client is able to fetch the output of an algorithm piece-by-piece, the client itself may format the output according to its own needs.[30]

As a by-product, algorithmic generators achieve the fourth general from Section 5.2. Moreover, the general goals nos. 5 and 8 are achieved, because the client controls the core loop: suspension and termination are controlled through the break condition of the loop; on the other hand, to merge the core loops of several algorithms into one algorithm, it suffices to implement a loop whose body performs one iteration of each of these core loops. The general goals nos. 6 and 7 are not addressed and not achieved. This is a pity, because it seems that the full value of the general goals nos. 4, 5, 8, and 9 highly depends on the sixth general goal.

The loop-kernel concept has been developed independently and is extensively discussed in [Wei97]. In retrospect, it may be viewed as a variation of algorithmic generators, which also realizes the general goals nos. 6 and 7 in Section 5.2.

Algorithms as iterators

This may be viewed as a special case of algorithmic generators. For example, in the *AAI* base class library,[31] depth-first search and breadth-first search are realized as linear node iterators. In

[29] Ref. [Wei98] uses geometric data structures as a running example. However, the presented techniques are fairly general and carry over to algorithm components.

[30] Dijkstra's algorithm and other graph algorithms are not presented as algorithmic generators in [Fla95], but in a traditional style as subroutines. This is not surprising, since the ninth general goal is not especially crucial for Dijkstra's algorithm. For example, if an algorithm merely delivers the distance of every node from the root, a shortest-path tree can be constructed from this output as follows (see also Item 6(h) in Section 3.1): for every node w except for the root, one incoming edge (v, w) is identified, such that the difference between the distances of v and w from the root equals the length of (v, w). Special care must be taken if there are zero-length edges.

[31] See http://www.aai.com/AAI/IUE/spec/base/base-classes.html.

other words, the output is the set of all nodes that are visited during the search. Hence, from a client's perspective, there are two differences to a normal node iterator:

- The order in which the nodes are visited reflects the search strategy.
- Only the nodes reachable from the root(s) of the search are visited.

Clearly, only a few algorithms may be interpreted as node or edge iterators. For these algorithms, the general goals 5 and 8 from Section 5.2 are achieved; goals 6, 7, and 9 are not addressed and not achieved. Again, this might reduce the value of the other goals.

Iteration in Sather

Iterators in Sather (called *iteration methods* in the following to avoid ambiguities) generalize typical loop constructs in mainstream programming languages [MOSS96].[32] In Sather, iteration methods are specific methods of abstract data types, which are not really subroutines as described in Section A in the appendix below. Such an iteration method may only be called within a loop. Unlike a normal subroutine, it may have two different ways of terminating the methods: a *yield statement* returns the flow of execution to the point immediately after the call to the method, and a *quit statement* breaks the loop, which means that execution is resumed at the point immediately after the loop.

Hence, a yield statement is much like a normal return statement in subroutines, whereas a quit statement is similar to a combination of such a return statement and an unconditional loop break immediately afterwards. However, there is a subtle difference: throughout the loop, an iteration method may have its own, private, internal state. This state may only be modified inside this iteration method, and the state is lost once the loop terminates. In other words, the iteration method realizes an anonymous object, which behaves like an algorithm iterator (see the previous approach above) and whose life time is bound to the execution of the loop. This anonymous object is initialized in the call to the iteration method in the very first iteration of the loop. In each further iteration, this call advances the iterator to its next state. When the flow of control leaves this loop and re-enters it later on, a new anonymous object is created, which is completely independent of the object in the previous execution of the loop.

An algorithm could be implemented as an abstract data type such that the execution of its core loops is delegated to iteration methods. Such an iteration method would receive an argument of the Boolean type, which decides whether a call to the iteration method shall end with a yield or quit statement. Such a design is quite similar to loop kernels, and the general goals 4 and 8 are clearly achieved. Goal 6 is also straightforwardly achievable. However, goal 7 might not be easy to achieve, because the preprocessing (*i.e.* the initialization) and the iteration are performed by the same method. There are certainly work-arounds, but this might not conform to the philosophy of iteration in Sather.

Moreover, a temporary suspension of an algorithm (general goal 5) might cause design problems. As usual, consider the example that an algorithm component in an interactive system is temporarily suspended to allow a completely unrelated user interaction. All of this stuff must be mixed into the algorithm loop. However, a clean design might not always be possible without a strict separation of these mutually unrelated tasks.

Algorithm frameworks

Frameworks are a general object-oriented design methodology [L+95]. See Holland [Hol92] for a concrete example of an algorithm framework. In 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

[32] See also the Sather home page; http://www.icsi.berkeley.edu/Sather. Section 4.5 in[MOSS96] gives an example (in-order iteration over binary trees) which demonstrates that iteration methods allow an impressingly easy reformulation of inherently recursive algorithms in an iterative style. See also Footnote 18 for this aspect.

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 contextes.

If designed carefully, an algorithm framework actually achieves the fourth general goal from Section 5.2. The sixth general goal is not addressed explicitly, but also easily achievable: for this, the base class must export its logical state to all derived classes. The general goals nos. 7, 9, and 10 are not addressed.

In principle, the general goals nos. 5 and 8 may also be achieved. This is a trivial and natural task if algorithms are implemented as, say, algorithmic generators or loop kernels (see above). However, in case of algorithm frameworks, this task might not at all be trivial, and the design of the whole software system may suffer. In fact, frameworks are intended to control the execution, which means that the core loops are implemented inside the framework (like in [Hol92]). However, goals 5 and 8 require that this control can be taken over by the client. Like in the previous approach, iterators in Sather, this may lead to an undesired interdependence of the algorithm and unrelated stuff.

Robustness-oriented design

In Item 6(h) in Section 3.1, we introduced the notion of *robustness*. This is a primary design goal of the *Karla* library.[33] Ideally, every algorithm component is based on a specification of its functionality and comes with integrated robustness checkers, which completely check the component against this specification.

In Section 3.1, we saw that a complete robustness checker that is acceptable for all variants of Dijkstra's algorithm is not likely to exist. Hence, the question whether incorporated robustness checkers contradict neither efficiency nor adaptability remains open and might require further practical research.

7 Documentation

It is fairly easy to specify the functionality of an algorithm component which is implemented in a tradional style, that is as a single, monolithic subroutine with fixed, hard-wired data structures for the input and the output. In fact, it suffices to specify the set of feasible inputs (*preconditions*) and the output as a function of the input (*postconditions*). Optionally, such a specification could also include an estimation of resource requirements (notably run time and space) as a function of the input and the output. The problem of formulating this kind of specification is well understood, and sufficient formal methods are available for that.

However, the problem is much more complex if an algorithm component is designed according to the general goals 1-10. Basically, there are three reasons for that:

- Users might expect that an algorithm component is designed in the traditional style. Hence, another design may cause confusion. This confusion must be taken into account by the author of the documentation.
- Experience shows that a high degree of flexibility may be a source of confusion in itself.
- The fourth, fifth, and sixth general goals from Section 5.2 introduce a new aspect: state. A traditional implementation can be documented in a purely declarative, state-less style. However, if a component offers stop points and exports its logical state at these stop points, the behavior of the component depends on its current state, which in turn depends on the "history" of the component.

  State-oriented documentation is usually based on *representation invariants*. See [Mey94] for a thorough introduction.

[33] See the Karla home page; http://i44www.info.uni-karlsruhe.de/~zimmer/karla.

In the adaptation of our concepts to LEDA, we tried to overcome these problems. For this aim, we additionally implemented a couple of subroutines, which are mere partial or complete customizations of the loop-kernel class to typical classes of contextes (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 annotated 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.

Empirical expirience must show whether this kind of documentation will be broadly accepted by users.

8 Conclusion and Outlook

In this paper, we analysed the problem what flexible adaptability of an algorithm component really means. However, this problem has many facets, and we only could touch upon some of them. Here are a few questions which remain open and should be addressed by further research:

- Dijkstra's algorithm, on which the acid test is built, does not change the structure of the graph. Many graph algorithms apply the basic modifying operations: inserting and removing nodes and edges. Other graph algorithms apply more complex modifying operations such as shrinking subgraphs or decomposing graphs into their biconnected components. It is not clear what modifying methods should be available to an algorithm. In fact, every complex modifying operation is reducible to these four basic modifying operations. However, this is usually not the most efficient and most elegant way. On the other hand, to obtain a manageable system of requirements (like in the STL; see Figure 5), it might be necessary to identify a relatively small set of basic modifying operations and to reduce all potential modifying operations to these basic ones. The question is whether an acceptable compromise exists.

- This paper is mainly based on practical experiences with C++. Although most parts of the paper are completely language-independent, they might be influenced by this background. Further experiences using other programming languages may be interesting.

- The sixth general goal in Section 5.2 also mentions optional write access to the logical state of an algorithm component. Whereas full access to the logical state (and full encapsulation of the implementation details) seems to be a good compromise between flexibility and abstraction in case of read access, it might be much more difficult to find a good compromise in case of write access.

- Although many insights in this paper are formulated in terms of graph algorithms, they might immediately carry over to other algorithmic domains such as numerical analysis and image processing. However, the priorities of goals may vary from domain to domain, and additional goals, which are not crucial for graph algorithms, may have a high priority in other domains (*e.g.* numerical stability). Acid tests for other algorithmic domains are necessary to investigate the consequences.

Acknowledgements

I would like to thank the LEDA people—Kurt Mehlhorn, Stefan Näher, and Christian Uhrig—for fruitful discussions and for a lot of constructive criticism. Special thanks to my co-workers, Dietmar Kühl and Marco Nissen.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows.* Prentice Hall, 1993.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkun, and Jeff Thomas. Scalable software libraries. *Proc. Symposium on the Foundations of Software Engineering*, 1993.

[BW94] Manuel Blum and Hal Wasserman. Program result-checking: a theory of testing meets a test of theory. *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS '94)* pages 382–392, 1994.

[CL94] Marshall P. Cline and Greg A. Lomow. *C++ FAQs.* Addison-Wesley, 1994.

[CLR94] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 1994.

[Der88] Ulrich Derigs. *Programming in Networks and Graphs: on the Combinatorial Background and Near–Equivalence of Network Flow and Matching Algorithms.* Springer, 1988.

[Fla95] Bryan Flamig. *Practical Algorithms in C++.* Coriolis Group Book, 1995.

[GS93] 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.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison–Wesley Publishing Company, 1995.

[GKW98] Dieter Gluche, Dietmar Kühl, and Karsten Weihe. Iterators evaluate table queries. To appear in the *ACM Sigplan Notices*, 1998.

[Gol91] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1991.

[GT87] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory.* Wiley, 1987.

[GS85] Leonidas J. Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[Hol92] Ian Holland. Specifying reusable components using contracts. *Proc. European Conference on Object-Oriented Programming (ECOOP '92)*, pages 287–308, 1992.

[KK88] Laveen Kanal and Vipin Kumar. *Search in Artificial Intelligence.* Springer, 1988.

[KL96] K. Kreft and A. Langer. Iterators in the Standard Template Library. *C++ Report*, 8(7):30-37, 1996.

[KL97] K. Kreft and A. Langer. Building an iterator for the STL and the Standard Template Library. *C++ Report*, 9(2):20-27, 1997.

[KNW97] Dietmar Kühl, Marco Nissen, and Karsten Weihe. Efficient, adaptable implementations of graph algorithms. Available from the on-line proceedings of the First Workshop on Algorithms Engineering (WAE '97) at http://www.dsi.unive.it/~wae97/proceedings.

[KW97] Dietmar Kühl and Karsten Weihe. Data access templates. *C++ Report*, 9(7):15&18–21, 1997.

[LLP83] Eugene L. Lawler, Michael Luby, and B. Parker. Finding shortest paths in very large networks. In Manfred Nagl and Jürgen Perl, editors, *Proc. Ninth International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 184–199, 1983. Trauner Verlag Linz Austria.

[Len90] Thomas Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley, 1990.

[LT91] Thomas Lengauer and Dirk Theune. Efficient algorithms for path problems with general cost criteria. *Proc. Eighteenth International Colloquium on Automata, Languages and Programming* (*ICALP '91*).

[L+95] Ted Lewis et al. *Object Oriented Aplication Frameworks.* Prentice Hall, 1995.

[Lis88] Barbara Liskov. Data abstraction and hierarchies. *ACM SIGPLAN Notices*, 23(5), 1998.

[Mar96] Robert C. Martin. The Liskov substitution principle. *C++ Report*, March 1996.

[Mey94] Bertrand Meyer. *Object–Oriented Software Construction.* Prentice Hall, 1994.

[Mey95] Bertrand Meyer. *Object Success: a Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process.* Prentice Hall, 1995.

[MN95] Kurt Mehlhorn and Stefan Näher. LEDA: a library of efficient data structures and algorithms. *Communications of the ACM*, 38:96–102, 1995.

[MOSS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemans Szyperski. Iteration abstraction in Sather. *Transactions on Programming Languages and Systems*, 18(1):1–15.

[MS95] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison–Wesley Publishing Company, 1995.

[NC88] Takao Nishizeki and Norishige Chiba. *Planar Graphs: Theory and Algorithms.* Annals of Discrete Mathematics, volume 32, North-Holland, 1988.

[NW96] Marco Nissen and Karsten Weihe. Combining LEDA with customizable implementations of graph algorithms. http://www.informatik.uni-konstanz.de/Preprints, 1996.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. *ACM Computer Science Conference*, pages 66–73, 1993.

[Pig97] Thomas M. Pigorski. *Practical Software Maintenance.* Wiley, 1997.

[Sou94] Jiri Soukop. *Taming C++.* Addison-Wesley, 1994.

[Vid93] René V. V. Vidal. *Applied Simulating Annealing.* Lecture Notes in Economics and Mathematical Systems 396, Springer, 1993.

[Wei97] Karsten Weihe. Reuse of algorithms: still a challenge to object-oriented programming. *Proc. Twelth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA '97*), pages 34–48, 1997.

[Wei98] Karsten Weihe. Using templates to improve C++ designs. To appear in the *C++ Report.*

[WO94] Bruce Weide and William Ogden. Recasting algorithms to encourage reuse. *IEEE Software*, 80–88, 1994.

Appendix

A Abstract Data Types

Formally, an *abstract data type* A is formed by a *ground set*, optional *auxiliary sets*, and *methods*. An object of A in a program has a certain life time, and throughout its life time, it is assigned an element of the ground set. This is the *state* of the object. Methods are subroutines whose input, output, and in/out arguments are elements of the ground set or elements of the auxiliary sets. A method of A may change the state of objects of type A. However, no subroutine except for the methods of A is allowed to change the state of an object of type A (this principle is usually called *encapsulation*). An abstract data type must come with one or more methods of a special kind, which are called the *constructors* of A and initialize the state of an object of type A. One of these constructors must be invoked at the beginning of the life time of every object.

For example, a stack of integral numbers is an abstract data type whose ground set is the set of all finite ordered sequences of integral numbers (incl. the empty sequence). Many implementations of the abstract data type "stack" provide exactly one constructor, which takes no additional arguments and assigns the empty sequence to the new stack. Moreover, methods like *push*, *pop*, *top*, and *empty* are common. The set of binary truth values (*true*,*false*) is an auxiliary set and used as the output of method *empty*.

B Subroutines as First-Class Objects

Some languages offer built-in data types which allow one to store the addresses of subroutines in variables. The subroutine may be called through this variable. *Pointers-to-functions* in C and C++ are a low-level example of that; higher-order functions in functional languages are an example on a more abstract level. *Pointers-to-methods* in C++, closures in Sather,[34] and higher-order methods in Pizza[35] store the addresses of methods of abstract data types. All of these examples are strongly typed, that is every variable of such a built-in data type is bound to a specific number, a specific order, and specific types of the arguments of the subroutine.[36] This argument list must be specified in the declaration of the variable, and the variable can only store the addresses of subroutines whose argument lists match this declaration. Moreover, if a variable shall store the addresses of methods, not subroutines, it is further bound to a specific abstract data type and cannot store the address of a method of any other abstract data type.

C Polymorphism

Polymorphic features in programming languages allow the programmer to leave open certain types in the implementation of components, so that this component may be used with different types. Inside the implementation of this component, these types are represented by "placeholders." Existing polymorphic features may be classified according to two schemes:

- *Static/dynamic polymorphism*: whether the concrete type must be chosen at compile time or this decision can be deferred until the component is actually invoked at run time. For instance, the latter sort of polymorphism allows the definition of inhomogeneous sequences.

  To give a concrete example of an inhomogeneous sequence: different geometric shapes, such as points, rectangles, and ellipses, are usually represented by different types. Nonetheless, a list of objects representing geometric shapes may contain a mixture of shapes of different types. From the viewpoint of a client of this list, all of these objects appear as objects of the placeholder type, which represents general shapes. If conformant to the *Liskov substitution*

[34] An extension to Eiffel; Sather home page: http://www.icsi.berkeley.edu/~sather.
[35] An extension to Java; Pizza home page: http://cm.bell-labs.com/cm/cs/who/wadler/pizza.
[36] For convenience, we regard the return value of a function as a member of the argument list.

principle [Lis88, Mar96], this placeholder type merely offers methods such as rotating or translating shapes, which are applicable to all kinds of geometric shapes (in contrast, for example, to stretching in one dimension, which does not apply to shapes like circles and squares).

- *Syntax-based/declaration-based polymorphism*: this concerns the set of types from which a concrete type can be chosen to replace the placeholder in the component. A necessary condition for a type to be eligible is that it must support the functionality which is used within the component to work with objects of the formal placeholder type. This conformance requirement is purely syntactical.[37] Syntax-based polymorphism means that this requirement is also sufficient. In contrast, declaration-based polymorphism additionally requires that a concrete type is explicitly declared to be a candidate for replacing the placeholder.

Most common object-oriented languages offer type specialization (*inheritance*), which allows a declaration-based dynamic polymorphism: the base class is the placeholder, and the derived classes are eligible to replace the placeholder. A few languages provide additional features especially for this sort of polymorphism (*e.g. interfaces* in Java).

Generics in various programming languages (also known as *templates*) realize static polymorphism. For example, this kind of polymorphism is syntax-based in Ada and C++, and declaration-based in Eiffel and in the most prominent proposals for adding genericity to Java. In the latter languages, these declarations are expressed by the corresponding mechanisms for dynamic declaration-based polymorphism: a concrete type is eligible to replace a certain placeholder in a static setting if and only if it is eligible to replace the same placeholder in a dynamic setting.

Dynamic, syntax-based polymorphism is realized in SmallTalk: when it turns out at run time that the chosen concrete type does not provide the (syntactic) functionality assumed in the component, an exception is raised, which means immediate termination of the whole program, unless this exception is caught and handled appropriately.

Many languages allow explicit overloading, that is different subroutines may be given the same name, provided the argument lists are sufficiently different, so no call to a subroutine may be ambiguous. This feature may be regarded as a simple version of static, syntax-based polymorphism and is also known as *ad-hoc polymorphism*.

D Standard Template Library (STL)

The STL [MS95] is completely based on static, syntax-based polymorphism. The core functionality offered by the STL is a set of basic containers such as vectors, stacks, lists, and priority queues, that is data structures which represent linearly ordered collections of items of a fixed type, the *item type*. The key abstraction in the STL is the concept of *iterators* (see the introduction of linear iterators in Section 6.1). In the STL, basic container algorithms such as copying and sorting do not access the containers directly, but indirectly, by means of iterators. The concrete types of the iterators are template parameters of the algorithm. Hence, bringing a new data structure into the game amounts to implementing appropriate iterator classes. This is not too much work, recipes are available [KL96, KL97], and the algorithmic stuff is not affected.

Inside an algorithm, a container is specified by two iterators, which refer to its first and last item.[38] The benefit is a high degree of flexibility: for example, restricting an algorithm to a subsequence of a container simply amounts to calling the algorithm with iterator objects referring to the begin and the end of the subsequence. Moreover, STL comes with two iterator *adapters*, which allow one to seamlessly change the meaning of iteration in the spirit of composition in layers (Section 6.1): one adapter reverts the direction of iteration, which means that the algorithm is

[37] Clearly, semantical conformance is not decidable and hence neither checkable at compile time nor at run time. A few languages (*e.g.* Ada95) provide renaming features. Such a feature allows the use of concrete types whose syntactical functionality does not exactly conform to the usage of the placeholder type inside the component.

[38] To be precise, the second iterator refers to a sentinel behind the last item. However, this minor technical detail is irrelevant in our context.

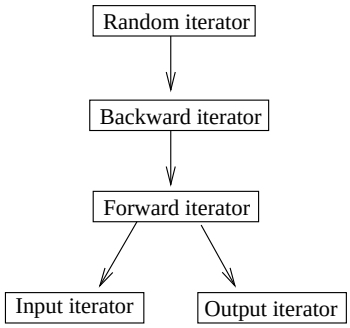

Figure 5: the iterator hierarchy in the STL.

applied to the reverse container (although the container itself is not changed at all); the other one realizes each write access by inserting a new item at the end of the container (the default behavior of write access is overwriting the value of the item to which the iterator currently refers).

In the STL, iterators are classified into *input*, *output*, *forward*, *bidirectional*, and *random* iterators (see Figure 5). Roughly speaking, input iterators are only required to allow read access to an item's data, whereas output iterators are only required to allow write access. Forward iterators must allow read and write access. Bidirectional iterators are forward iterators that also provide a means of advancing the iterator in backward direction. Finally, random iterators are bidirectional iterators that can also be placed on a random item, which is specified by its position in the container.

This classification has been adopted for the STL (and for the C++ standard), because it reflects the functionality of the underlying containers: for a file that is open for reading (resp., writing), an input (output) iterator is appropriate; on the other hand, for a singly-linked list (resp., doubly-linked list, array), a forward (bidirectional, random) iterator is appropriate. This is a classification of *requirements*: a C++ class is one of these iterator classes if and only if its interface provides the required functionality. A hierarchy of requirements as in Figure 5 only appears on the conceptual, documentational level, not in the implementation, and does not induce an inheritance hierarchy or any other relation between the involved classes.

The following design rule allows maximal flexibility in combining algorithms with data structures:

> Data structures come with iterator classes of the highest type in the hierarchy in Figure 5 whose required functionality can be easily and efficiently mapped onto the functionality provided by the underlying container. On the other hand, algorithms work on the lowest type in the hierarchy that fulfills the algorithm's requirements on the underlying container.

E Formal Foundation of Adjacency Iterators

An *adjacency iterator* is associated with a fixed node and iterates over certain nodes which are regarded as the ones *adjacent* to this node. See the discussion of adjacency iterators in Section 6.1. The meaning of adjacency depends on the underlying graph abstraction, that is directed graph, undirected graph, bidirected graph, etc. Hence, we first develop a generic scheme for different graph abstractions and then apply this scheme to various concrete graph abstractions.

Graph views

An implementation of an algorithm must be based on a fixed abstract view of the underlying graph data structure, which may be different from the graph abstraction realized by this data structure. For example, in Section 3.1, Item 1, we noticed that Dijkstra's algorithm may be applied to directed graphs, undirected graphs, etc. However, even for an implementation that covers all of these cases, a fixed abstract view must be chosen, on which the implementation is built. It seems that the abstract view "directed graph" is most appropriate for Dijkstra's algorithm (in contrast, for many other problems, incl. spanning-tree and matching problems, the abstract view "undirected graph" might be most appropriate). To adapt Dijkstra's algorithm to an undirected graph data structure, the abstract view "directed graph" must be expressed in terms of an undirected graph. We will say that the view "undirected graph" is *adapted* to the view "directed graph."

It is not reasonable to define a specific abstract view for every algorithm, but to define a couple of standardized abstract views (much like the small set of standardized abstract views on containers in Figure 5) and to base the implementation of every algorithm on one of them. Basically, there are two reasons for that:

- Since the number of abstract views in such a standard set is small, the number of adaptations of one abstract view to another one is small. In particular, it does not depend on the number of algorithms.
- The documentation for implementations of algorithms is much leaner, because such a documentation may simply refer to the standard set to define the abstract view on which the algorithm is built.

Terminology

Throughout this section of the appendix, we will need the following basic definitions. A *multiset* is a finite set in which each element may occur more than once. The *multiplicity* of an element x of a multiset M is the number of occurrences of x in M and denoted by $\mathit{mult}(M, x)$. This value is zero if x does not occur in M. The cardinality of a multiset is the sum of all multiplicities. The union $M_1 \cup M_2$ of two multisets is defined by addition of the multiplicities: $\mathit{mult}(M_1 \cup M_2, x) = \mathit{mult}(M_1, x) + \mathit{mult}(M_2, x)$. For a finite set S, $\mathcal{M}(S)$ denotes the set of all multisets whose elements belong to S. For $M \in \mathcal{M}(S)$, $\mathit{ord}(M)$ denotes the ordinary set of elements of M, which is defined by

- $x \notin \mathit{ord}(M)$ for $x \in S$ and $\mathit{mult}(M, x) = 0$,
- $x \in \mathit{ord}(M)$ for $x \in S$ and $\mathit{mult}(M, x) > 0$.

Moreover, for M_1, $M_2 \in \mathcal{M}(S)$, $M_1 = M_2$ is used to indicate that $\mathit{mult}(M_1, x) = \mathit{mult}(M_2, x)$ for all $x \in S$. An *injection* from a multiset $M_1 \in \mathcal{M}(S_1)$ onto a multiset $M_2 \in \mathcal{M}(S_2)$ is a multiset $I \in \mathcal{M}(S_1 \times S_2)$ such that

- $\sum_{x \in S} \mathit{mult}(I, (x, y)) = 0$ for $y \in S_2$ and $\mathit{mult}(M_2, y) = 0$,
- $\sum_{y \in S} \mathit{mult}(I, (x, y)) = \mathit{mult}(M_1, x)$ for $x \in S_1$.

Note that all definitions except multiset union naturally extend the corresponding definitions for ordinary sets.

Incidence structure

From Section 6.1 recall that an adjacency iterator *iter* is associated with a fixed node v and iterates over the nodes "adjacent" to v (in other words, over the edges "incident" to v). Moreover, recall that an adjacency iterator shall provide a clone method, which is applicable to valid objects and creates a new object referring to the current adjacent node.

We will not exclude loop edges (*i.e.* edges that connect a node with itself) and multiple edges. In fact, such a restriction would not make sense for most graph algorithms (although many textbooks and articles of original work assume this for notational convenience). Moreover, in many realizations of graphs (*e.g.* in realizations by adjacency lists), such a restriction cannot be efficiently maintained when edges are inserted. Hence, we drop this restriction.

Formally, we define an adjacency iterator type as an *incidence structure*. Such an incidence structure consists of four mappings, *inc*, *adj*, *corr*, and *succ*. Informally, $inc(v) \in \mathcal{M}(E)$ is the set of all edges incident to $v \in V$, which means the edges over which an adjacency iterator *iter* iterates if associated with v. Since we do not exclude loop edges, $inc(v)$ may contain an edge twice.

On the other hand, $adj(v)$ is the set of all nodes adjacent to $v \in V$, which are the nodes traversed by *iter*. If k edges point from v to some node $w \in V$, we have $mult(adj(v), w) = k$. Moreover, *corr* establishes the correspondence between the sequence of adjacent nodes and the sequence of incident edges over which *iter* iterates. Since this sequence depends on the node v with which *iter* is associated, *corr* maps pairs of nodes and edges onto nodes. More specifically, $corr(v, e) = w$ means that v and w are the endnodes of e. Hence, we further require that *corr* is injective (in the sense of multisets).

The intent of *succ* is to identify the potential successor edges of every edge $e \in E$ in graph algorithms that navigate through the graph structure. Hence, *succ* is closely related to the additional clone method of *iter*, which places a new iterator on the current adjacent node: *succ* is the set of all edges over which this new object iterates. For example, in a directed graph this is the set of all out-going edges of the node to which the current incident edge e of *iter* points. Hence, the successors of e only depend on e. In contrast, in an undirected graph, the set of successors of e also depends on the orientation in which e is passed by the algorithm. In the formal definition of *succ* below, we will indicate this orientation by one of the endnodes of every edge. For notational convenience, we choose the head of the edge, that is the node on which the clone of *iter* is placed (the "link" between e and its successors).

Definition 1 (incidence structure) *Let V and E be two finite sets. An* incidence structure *on (V, E) is a quadruple $(inc, adj, corr, succ)$, where*

- $inc : V \to \mathcal{M}(E)$,
- $adj : V \to \mathcal{M}(V)$,
- $corr : \{(v, e) : v \in V, e \in inc(v)\} \to V$,
- $succ : \{(w, e) : w \in V, e \in E, (\exists v \in V : corr(v, e) = w)\} \to \mathcal{M}(E)$.

such that corr is an injection (in the sense of multisets), $\bigcup_{e \in inc(v)} corr(v, e) = adj(v)$ for $v \in V$, $E = ord(\bigcup_{v \in V} inc(v))$, and $\sum_{v \in V} mult(inc(v), e) \leq 2$ for $e \in E$.

Graph views

Equipped with this terminology, we next turn our attention to the first question raised in the description of adjacency iterators in Section 6.1. More specifically, we will give a few examples of specific graph views, which might demonstrate the flexibility of incidence structures as a formal base (and the need for such a complex definition of incidence structures).

Directed graph: Informally, $inc(v)$ is the set of edges leaving node v. Hence, the incidence structure must also fulfill $\sum_{v \in V} mult(inc(v), e) = 1$ for $e \in E$, which means that the sets

$inc(\cdot)$ form a partition of E. Moreover, $succ(w, e) = inc(w)$ for $corr(v, e) = w$, where $v, w \in V$ and $e \in E$. To say it informally: the out-going edges of a node w are the successors of the in-going edges of w.

Two-way directed graph: This means that, from every node, both its in-going and out-going edges are accessible, and that both sets of edges are clearly distinguished from each other. Therefore, we superimpose two incidence structures, $(inc_1, adj_1, corr_1, succ_1)$ and $(inc_2, adj_2, corr_2, succ_2)$, such that

$$\begin{array}{rcll} w \in adj_1(v) & \iff & v \in adj_2(w) & \text{for } v, w \in V; \\ corr_1(v, e) = w & \iff & corr_2(w, e) = v & \text{for } v, w \in V, e \in E; \\ e_1 \in succ_1(v, e_2) & \iff & e_2 \in succ_2(v, e_1) & \text{for } v \in V, e_1, e_2 \in E. \end{array}$$

Undirected graph: To model undirected graphs, we impose the following restrictions on the incidence structure:

- $mult(\bigcup_{v \in V} inc(v), e) = 2$ for $e \in E$;
- $corr(v, e) = w \Rightarrow e \in inc(w)$ for $v, w \in V$, $e \in E$;
- $succ(v, e) = inc(v)$ for $v \in V$, $e \in inc(v)$.

In other words, every edge is incident to exactly two nodes, these two incidence relations are undistinguishable, and the successors of an edge e at node v are all edges incident to v.

Bidirected graph: From Section 3.1, Item 1(a), recall the definition of bidirected graphs. The bidirected graph view is a variation of the undirected graph view. In a bidirected graph, the edges incident to a node are divided into two partition sets [Der88]. Hence, in our incidence structure, every set $inc(v)$ is the union of two sets: $inc(v) = inc_1(v) \cup inc_2(v)$. Note that a global definition of mappings inc_1, $inc_2 : V \to \mathcal{M}(E)$ does not make any sense. In fact, for every node, the numbering of these two sets is arbitrary, and the common numbering of $inc_1(v)$ and $inc_1(w)$ for $v \neq w$ bears no meaning.

The sets $inc_1(v)$ and $inc_2(v)$ are not necessarily disjoint: a loop edge may cross the partition line. This partition of $inc(v)$ naturally induces partitions $adj(v) = adj_1(v) \cup adj_2(v)$ and $corr(v, \cdot) = (corr(v, \cdot)|_{inc_1(v)}) \cup (corr(v, \cdot)|_{inc_2(v)})$. The first two conditions for undirected graphs are also valid for bidirected graphs. The difference between undirected and bidirected graphs is expressed in terms of the mapping $succ$: we require $e \in inc_1(v) \Rightarrow succ(v, e) = inc_2(v)$ and $e \in inc_2(v) \Rightarrow succ(v, e) = inc_1(v)$ for $v \in V$, $e \in inc(v)$. This condition replaces the third condition for undirected graphs above.

Bipartite graph: Clearly, undirected and directed bipartite graphs $G = (V \dot\cup W, E)$ are special cases of general undirected and directed graphs. However, often it is necessary to have different adjacency iterator types for nodes of V and nodes of W. For example, often the nodes in V are assigned another set of attributes than the nodes in W. In this case, the nodes in V are of another type than the nodes in W. Hence, two different adjacency iterator types are mandatory.[39] To capture this difference, we again superimpose two incidence structures, $(inc_1, adj_1, corr_1, succ_1)$ and $(inc_2, adj_2, corr_2, succ_2)$. In that, inc_1 and adj_1 are defined on V and inc_2 and adj_2 on W, and we require that $\bigcup_{v \in V} inc_1(v) = \bigcup_{v \in W} inc_2(v) = E$. Moreover, we have $adj_1(v) \in \mathcal{M}(W)$ for $v \in V$, $adj_2(v) \in \mathcal{M}(V)$ for $v \in W$, $succ_1(v, e) = inc_2(v)$ for $v \in W$, $e \in inc_2(v)$, and $succ_2(v, e) = inc_1(v)$ for $v \in V$, $e \in inc_1(v)$.

Crossing-free embedded graph with faces: This is an example of a special graph class that exhibits more structure than is defined for general graphs. A crossing-free embedding of a graph in a surface assigns each node a point on this surface and each edge a Jordan curve

[39] Unless the node type is dynamically polymorphic in the adjacency iterator; however, this is usually not desired.

such that the endpoints of the image of an edge are the images of the endnodes and the images of any two edges are disjoint except at the image of a common endnode [GT87]. The image of a graph divides the rest of the surface into disjoint *faces*. Many algorithms on this class of graphs assume that the graph structure is enhanced by a representation of these faces: $G = (V \dot\cup F, E)$, where F is the set of faces.

The relation between V, F, and E may be represented in different ways, which induce different superimpositions of incidence structures. For example, the *quad-edge representation* [GS85] may be modeled by four incidence structures, where $adj_1 : V \to \mathcal{M}(V)$, $adj_2 : V \to \mathcal{M}(F)$, $adj_3 : F \to \mathcal{M}(V)$, and $adj_4 : F \to \mathcal{M}(F)$. Roughly speaking, every edge $e \in E$ is incident to two nodes, v_1 and v_2, and two faces, f_1 and f_2, such that

- $corr_1(v_1, e) = v_2$ and $corr_1(v_2, e) = v_1$,
- $corr_2(v_1, e) = f_1$ and $corr_2(v_2, e) = f_2$,
- $corr_3(f_1, e) = v_1$ and $corr_3(f_2, e) = v_2$,
- $corr_4(f_1, e) = f_2$ and $corr_4(f_2, e) = f_1$.

Hypergraph: Hypergraphs of any kind (directed, undirected, ...) are not regarded as graph views of their own. Instead, a hypergraph $H = (\tilde V, \tilde E)$ is modeled as a bipartite graph $G = (V, E)$ with bipartite node set $V = \tilde V \dot\cup \tilde E$. A node $v \in \tilde V$ is connected with a node $e \in \tilde E$ by an edge in E if and only if e is incident to v in H.

Mixed graph: For example, consider a graph $G = (V, E_1 \dot\cup E_2)$, where E_1 is a set of undirected edges and E_2 a set of directed edges. Of course, this case can be modeled by a superimposition of a directed and an undirected incidence structure.

Adaptation of graph views

Finally, we consider the second question raised in the description of adjacency iterators in Section 6.1. We only give a few examples how one graph view A may be adapted to another graph view B. From this, it might be straightforward to implement an adjacency iterator it_B which is based on an adjacency iterator it_A for A and offers view B. Adaptations of other graph views might also be obvious.

- *Two-way directed graph $\longrightarrow$ undirected graph*

  Recall that a two-way directed graph is modeled by two incidence structures, $(inc_1, adj_1, corr_1, succ_1)$ and $(inc_2, adj_2, corr_2, succ_2)$. The undirected graph view, which simply drops the orientations of edges, is modeled by $(inc_1 \cup inc_2, adj_1 \cup adj_2, corr_1 \cup corr_2, succ_1 \cup succ_2)$.

- *Undirected graph $\longrightarrow$ directed graph*

  By that we mean that an undirected graph is viewed as a symmetric directed graph. The incidence structure is identical (and the implementation of an adapting iterator is trivial).

- *Bidirected graph $\longrightarrow$ directed graph*

  An adaptation of a bidirected graph view to a directed graph view has the following consequence: an algorithm that navigates over the incidence structure of a directed graph view regards one vertex v as two different vertices, v_1 and v_2. More specifically, whenever it enters v via an edge in $inc_1(v)$, it regards v as v_1, and otherwise as v_2. The incidence structure $(inc', adj', corr', succ')$ seen by this algorithm is defined by the rule $inc'(v_1) = inc_2(v)$ and $inc'(v_2) = inc_1(v)$. This naturally induces adj', $corr'$, and $succ'$.

- *Crossing-free embedded graph with faces $\longrightarrow$ bipartite graph*

  A graph $G = (V \dot\cup F, E)$ with faces as described above naturally induces an undirected bipartite graph $G' = (V \dot\cup F, E')$, where $\{v, f\} \in E'$ if and only if node v is on the boundary

of face f. For example, if G is represented by a quad-edge structure (*i.e.* four superimposed incidence structures as introduced above), the incidence structures $(inc'_1, adj'_1, corr'_1, succ'_1)$ and $(inc'_2, adj'_2, corr'_2, succ'_2)$ for the bipartite graph view may be straightforwardly defined such that the following rules are fulfilled for $v \in V$ and $f \in F$:

$$\begin{aligned} mult(inc'_1(v), \{v, f\}) &= mult(adj_1(v) \cup adj_2(v), \{v, f\}) , \\ mult(inc'_2(f), \{v, f\}) &= mult(adj_3(f) \cup adj_4(f), \{v, f\}) . \end{aligned}$$