Design Patterns for the Implementation of Graph Algorithms

Dietmar Kühl
Technische Universität Berlin

Berlin, the 19th July 1996

Abstract

While the theoretical aspects of many graph algorithms are well understood, the practical application of these algorithms imposes some problems: Typically, the implementation is bound to a specific data structure, the results and their representation are predefined etc. On the other hand, since many graph algorithms use other algorithms to solve subproblems, it is necessary to be able to freely choose the input and the output and/or to modify the behavior of the subalgorithms.

Since the necessary freedom is normally missing from the implementation of graph algorithms, a programmer of a complex algorithm is forced to implement algorithms in an appropriated way to use them as subalgorithms. Thus, implementing complex algorithms becomes even harder resulting in relatively erroneous implementations if complex algorithms are implemented at all. It would desirable to have implementation of algorithms available which can be used as subalgorithms in a flexible way and which can be applied to arbitrary graph representations.

This work introduces and discusses concepts to implement graph algorithms in a reusable fashion. With reusable it is meant that an algorithm can be used with different graph data structures and/or with modified behavior. To achieve this, components for an abstraction from the data structure are introduced which allow using many different data structures as graphs. In addition it is shown how to implement algorithms in a way which allows easy and flexible modification of their behavior.

Contents

Chapter 1

Introduction

It is observed in [1] that in software design the wheel is continually reinvented. Thus it is proposed to implement an "Algorithmic Tool Kit". This Algorithmic Tool Kit is supposed to include graph algorithms. Although the aim to create such a tool kit is clearly stated, the paper gives no insight how this tool kit looks like: The exact approach how to implement such a tool kit is left open.

When writing software there are some important objectives to be achieved: Efficiency, correctness, and reusability are the most common objectives. None of them can be completely ignored and correctness is even essential. This work will provide a concept how to write reusable but still efficient implementations of graph algorithms. The topic of provable correctness of the implementation is not covered although the concepts given should still be applicable where provable correctness is required.

This chapter introduces the reasons for concentrating on reusability. It briefly reviews existing approaches to implement graph algorithms and gives an overview of the concepts proposed. At the end of this chapter an outline for the remainder of the work is given.

1.1 Reuse of Implementations

Often, different programs need similar algorithms and data structures to solve some problems. To avoid implementing an algorithm multiply times, algorithms are implemented once and made available in a library which can then be used by any program needing an algorithm from the library. Examples for algorithms found in libraries are sorting algorithms, algorithms to solve linear equations, or algorithms to compute an optimal solution of a linear program. There are several reasons to reuse an algorithms rather than reimplementing it:

- For most algorithms it takes some expert knowledge to implement them. This already starts with relatively simple algorithms like sorting algorithms: Although sorting algorithms are very basic and topic of most beginner courses in computer science, they are sufficient complex to make a lookup in a book necessary to find out the details. More complex algorithms will require more knowledge about the domain, e.g. to guarantee numerical correctness or to be as efficient as possible.
- Implementing any algorithm takes time. To get an algorithm implemented takes a programmer which has to spend some time he/she has to be paid for. Thus, using an existent implementation of an algorithm may save money if it is easier to use the implementation than reimplementing the algorithm. For sufficient complex algorithms, e.g. finding an optimal solution of a linear program, reuse of an existent implementation will always pay off. For simple algorithms it

may depend on the interface provided to the implementation: If the implementation uses an inconvenient or hard to use interface, it may be easier and faster to reimplement the algorithm. In any case, using the same implementation wherever appropriate amortizes the costs of the implementation between different projects reducing the costs of the individual projects.

- Initial implementations of algorithms are often erroneous. The description of the algorithm, whether it is found in the literature or is developed by the programmer, may contain errors. If the description of the algorithm is correct, the translation to a programming language may introduce errors. Of course, an implementation which is reused may also contain errors but because the implementation is (hopefully) often used, it is more likely that errors are detected and can be corrected. An implementation which is only created for one application is more likely to contain undetected errors as the implementation is not that often applied.

- When programming an algorithm for a library, this algorithm is the main concern. To make it a good candidate for inclusion into a library it is not only tested thoroughly but also optimized to be as efficient as possible. If some algorithm is implemented just to solve a problem arising in some bigger context, this implementation is typically not optimized at all and often uses a simple but not efficient algorithm. Thus, an algorithm taken from a library is often more efficient than a special implementation solving the problem.

If this were the whole story, clearly every algorithm would be taken from some library. Unfortunately, there are also problems with algorithms taken from libraries. An obvious problem are the data structures used: The more complex the data structures are to represent the problem, the more choices are available to design the data structure. To use an algorithm from a library the data structure of the library has to match the data structure of the application somehow. An approach often taken is to implement an algorithm for a certain data structure which has to be used to apply this algorithm. In areas where there is a canonical representation, e.g. because the representation is efficiently supported by the programming language, this approach is sufficient. However, this not the case for the representation of graphs which can be represented in many different ways with different trade–offs.

Another problem arising with implementations from libraries is the fact that the algorithm can normally not be modified: Graph algorithms often extend the behavior of an existent algorithm by making additional computations in each iteration of an algorithm, by adding another termination condition, or by using a special initialization and so on. This requires that the behavior of an algorithm can be modified making the design of the implementation hard: Often, additional flexibility is traded for reduced efficiency e.g. because an additional function has to be called or additional conditions have to be checked.

Despite these problems it would be highly desirable to have reusable implementations of graph algorithms available: Most of the algorithms working on graphs are very complex and take a considerable amount of insight into the problem domain to understand and implement them. Thus, it seems to be logical to search for an approach to the implementation of graph algorithms alleviating the problems encountered.

1.2 Existing Approaches

This section briefly reviews two existing packages: LEDA and SCOPE. The reason to review these implementations is to identify common coding practice which hinders or enhances reusability.

Library of Efficient Data types and Algorithms (LEDA)

LEDA [18] is a library implemented in C++ [23] providing several data structures and some combinatorial algorithms working on these data structures. The algorithms include graph algorithms and algorithms from computational geometry. Here, only the graph algorithms are reviewed.

The data structures provided by LEDA include basic data structures like lists, arrays, stacks, etc. as well as advanced data structures like dictionaries, hashes, and priority queues. The more complex containers like lists and maps allow iteration over the elements stored in the container using **items** corresponding to the container. An item identifies a current element in the container and can be used for efficient access to the corresponding element. The items can also be used to find out the next or the previous element etc. by passing them to a corresponding member function of the container.

Fundamental to the implementation of the graph algorithms is a graph data structure representing a directed graph which serves as a base class for several other graph data structures to create e.g. undirected graphs. For all flavors of graphs there is a "parameterized" version which allows to use arbitrary classes as node or edge weights. For the directed graph it is not possible to access the set of edges incoming to a node efficiently: Only the outgoing edges are stored in the adjacency list. Undirected graphs are modeled by using directed graphs where for every edge (v, w) in the graph the edge (w, v) is also part of the graph.

The graph data structure is a class called `graph` used as base class for all variations of graphs. The graph algorithms in LEDA take a `graph` as argument together with some additional arguments. To access the set of nodes or the set of edges of a graph, objects of the classes `node` and `edge`, respectively, are used. The use of these classes is similar to the use of items for the containers. The `edge`s are also used to access the edges incident to a node.

To allow different data to be associated with the nodes, **node arrays** are used for additional data. These arrays are indexed using `node`s. Likewise, additional data for edges is stored in edge arrays indexed with `edge`s.

One problem immediately arising when using LEDA is the fact that the algorithms only work with the class `graph` or a class derived from `graph`. Due to the style the class `graph` is implemented it is not sufficient to derive from this class to use a completely different representation, e.g. a grid graph. To use a representation different from the graph data structure provided by LEDA, it would be necessary to implement a new class `graph` with a different representation.

When applying an algorithm which takes some node or edge data as input, it is necessary to represent this data explicitly as a node or an edge array, respectively. It is not possible e.g. to calculate the data on the fly if the data is accessed. It is also not possible to apply an algorithm to the parameterized version of the graph and use a member of the node or edge parameters: If the input data is represented as a member of the node or the edge data, it is still necessary to copy the data into a corresponding array. Likewise, it is not possible to choose the representation of the output: If the output involves data stored with every node or every edge a corresponding array has to be passed to store the data.

It is also worth to note that the algorithms cannot be modified: The algorithms are implemented in a monolithic fashion calculating the data which was considered important when the algorithm was implemented. Depending on the algorithm, this may be to few data, rendering the algorithm useless for some applications, or too much data resulting in wasted time for the calculation of data not used.

Looking through the sources reveals that the LEDA algorithms rather implement an algorithm tailored to a specific use than reusing the corresponding algorithm made available to the user of LEDA. This especially holds for graph search which is reimplemented in many of the algorithms. Although it can be argued that graph search is a very simple algorithm not worth reusing, it also takes time to implement graph search and even for such simple algorithms it is possible to introduce errors.

All together it appears that there are three factors which hinder reuse of LEDA graph algorithms:

1. It is hard to apply LEDA's algorithms to different graph representation than the graph representation provided by LEDA.
2. There is no choice for the representation of additional data for the nodes or the edge: It is always necessary to use the corresponding array class.
3. There is no way to modify the behavior of most of the algorithms.

It should be noted that the main concern of LEDA was to create efficient implementations of data structures and implementation. Thus, the main concern was not to allow application of LEDA algorithms to arbitrary data structures or to allow modification of the algorithms. LEDA is currently used in several projects (see http://www.mpi-sb.mpg.de/LEDA/www/projects.html for further information). This indicates that LEDA promotes some reuse.

Simple Combinatorial Optimization Programming Environment (SCOPE)

SCOPE [11] is an environment programmed in Object Pascal to solve maximum–flow problems. The basic idea of SCOPE is to implement algorithms in terms of abstract data types and abstract operators. An abstract data type is a data structure supporting a predefined set of operations modifying the data structure without defining the implementation of the data structure. A well known example of an abstract data type is a stack: The operations and the semantic of the operations are defined but the implementation of this data structure is left open.

Abstract operations are operations applied to a data structure without defining the implementation but only the semantics of the operation. This allows to implement several different operations performing the same logical operation in different ways and using different data structures. The abstract operations in SCOPE are used to create a unified framework for the different algorithms solving the maximum–flow problem: By instantiating the framework for the maximum–flow algorithm with different abstract data types and abstract operations, the different algorithms are created.

Since algorithms in SCOPE are implemented in terms of abstract data types and abstract operations it is possible to use an arbitrary representation of a graph and to add behavior to an algorithm (by placing the additional behavior into an abstract operation). The major drawback seems to be the relative high–level definition of the abstract operations. The abstract operations mentioned in the cited paper are something like AUGMENTING_SUBGRAPH which creates a graph with edges having free capacity relative to some flow or PATH which looks for a path between to nodes. Unfortunately, the only information about SCOPE available to me is the paper cited above.

1.3 Outline of the Approach

The approach proposed uses two major, complementary concepts: First, to allow application of algorithms to different representations, an abstraction from the representations fulfilling some fundamental requirements of the algorithms is used. The idea is to implement algorithms completely independent from any data structure concentrating on the properties of the algorithm. When algorithms are implemented, certain requirements on the data structure will arise. To make a data structure eligible for an algorithm, the data structure has to fulfill the requirements.

Instead of accessing the graph representation directly, an indirection forming an abstraction is used to access the graph representation. The abstraction for graphs uses two kinds of iterators (i.e.

object which can be used to successively access each element from a collection) to provide access to the set of nodes and the set of edges. In addition, to find out which nodes are adjacent or which edges are incident to a given node, another kind of iterators, call an adjacency iterator, is used. To allow use of data associated with the nodes or the edges of a graph, an entity called a data accessor is used.

These abstraction allow non-modifying access to graphs. There are some algorithms using auxiliary modifications. Such modifications can be simulated using the four abstraction for non-modifying accesses without really modifying the graph representation using a concept called a mutator. To allow modifications of the graph representation the design pattern "builder" can be used [12]: This allows modification of a structure without hard–coding the methods to perform the modifications.

The first part of the approach focusses on the abstraction from the representation. The second part is concerned with a flexible implementation of algorithms. To make the implementation of algorithms more flexible, algorithms are implemented as classes (which is similar to SCOPE where the abstract operations are also classes). This allows the creation of abstract algorithms which can be used as different phases of an algorithm and replaced by different implementations potentially tailor to a specific problem or graph representation.

This is still not sufficient to allow changed behavior of algorithm. To allow easy modification of algorithms, loop, algorithms which include a are implemented in a fashing which lets apear them as being turned inside out. The algorithm is implemented as a class similar to an iterator: It identifies a current state of the algorithm and allows to stepwise move over the different states encountered during the excution of the algorithm. Such an implementation of an algorithm is called a loop kernal and allows to perform additional action in each iteration of the algorithm and to use a modified termination condition (plus several other features).

Before discussing the approach to the implementation of graph algorithms, the basics of the problem domain considered are defined in the next chapter. In Chapter 3 a brief outline of general approaches to reusable software is given. A thorough discussion of the various concepts for the implementation of reusable graph algorithms is given in Chapter 4.

Chapter 2

Problem Domain

Although the concepts used to approach the problem of writing reusable software can be applied to other areas, too, this work is mainly concerned with algorithms working on (finite) graphs. In particular implementations of algorithms solving the maximum–flow problem, and related algorithms, are investigated. This problem area is well understood from a theoretical point of view and thus makes up a good candidate to investigate practical problems. This chapter provides some theoretical background used in later chapters.

2.1 Basic Definitions

This section defines and discusses the basic objects from algorithmic graph theory considered in this work: graphs, paths, trees, etc. The objects defined are well known. However, the definitions are discussed with a view on practical aspects: The theoretical point of view is sometimes quite impractical.

Graphs

The main focus is directed towards algorithms working on graphs. An **undirected graph** $G = (V, E)$ is a pair consisting of a finite set V of **nodes** and a set $E \subseteq \{\{v, w\} | v, w \in V, v \neq w\}$ of **edges**. A **directed graph** $G_{dir} = (V_{dir}, E_{dir})$ is a pair consisting of a finite set V_{dir} of **nodes** and a set $E_{dir} \subseteq V \times V$ of edges. Of course, E and E_{dir} are also finite because they contain at most $O(|V|^2)$ and $O(|V_{dir}|^2)$ objects, respectively. Here only finite graphs are considered, i.e. graphs where the number of nodes and the number of edges are finite. This is no real restriction from a practical point of view: It would be impossible to explore an infinite graph completely in finite time anyway.

Two nodes $v, w \in V$ are said to be **adjacent**, if there is an edge $e = \{v, w\} \in E$. The edge e is said to be **incident** to the nodes v and w. Often algorithms refer to all nodes adjacent to a certain node $v \in V$: The set $adj(v) = \{w \in V | \exists \{v, w\} \in E\}$ is called the **adjacency list**[1] of v. The set $inc(v) = \{\{v, y\} \in E\}$ is made of the edges incident to the node v. The **degree** $deg(v) = |inc(v)|$ of a node $v \in V$ is the number of edges incident to v.

Likewise for directed graphs: The two nodes $v, w \in V_{dir}$ are said to be **adjacent**, if there is an edge $e = (v, w) \in E_{dir}$ or an edge $e = (w, v) \in E_{dir}$. The edge e is said to be **incident** to the nodes v and w. The set $adj(v) = \{w \in V_{dir} | \exists (v, w) \in E_{dir}\}$ is called the **adjacency list** of v. The directed edge $e = (v, w)$ is thought to be directed from node v to node w and is depicted in drawings

[1] The term adjacency list actually refers to a certain graph representation. However, it is convenient to think of the set $adj(v)$ of nodes adjacent to a node v as a list of nodes even if a graph is represented differently.

of graphs as an arrow leading from the point representing the node v to the point representing the node w. The node v is called the **tail** and the node w the **head** of the directed edge e. An edge $e = (v, w)$ is an **outgoing** or **leaving** edge for node v and an **incoming** or **entering** edge for node w. It is useful to name the sets of incoming and outgoing edges: The set $in(v) = \{(v, y) \in E_{dir}\}$ is the set of directed edges leading towards v, and the **in–degree** $indeg(v) = |in(v)|$ is the number of these edges. Analogously, the set $out(v) = \{(v, y) \in E_{dir}\}$ is the set of directed edges leaving v, and the **out–degree** $outdeg(v) = |out(v)|$ is the number of these edges. The set $inc(v) = out(v) \cup in(v)$ are the edges **incident** to the node v and the **degree** $deg(v) = |inc(v)|$ is the number of edges incident to v.

The two definitions for directed and undirected graphs are similar to the ones widely used. Unfortunately, the definitions of graphs are quite restricted, and some situations arising in practical problems are not covered:

- For many applications it makes no difference whether the graphs are directed or undirected, and sometimes it even makes sense to have graphs with a mixture of directed and undirected edges in the same graph, e.g. when modeling a system of streets where some streets are one–ways.
- It is impossible with the given definitions to have **loops**: Edges which are incident to just one node. Although the definition of the directed graph can be easily modified to allow loops (just allow edges (v, v) for $v \in V$), this is not that easy for the undirected case: The edges could not be sets with exactly two elements and denote a loop. Making them pairs would impose an order. Using a set with just one element to denote a loop would require a conditional notation whenever an edge is selected. Any choice makes the handling of edges more complex.
- The definitions do not allow **parallel edges**, i.e. two or more distince edges incident to the same two nodes and which have the orientation in the case of directed graphs. Apart from additional modeling freedom, parallel edges are simply present in many graphs encountered in practice or are introduced by algorithmic treatment of the graph.

A graph (whether directed or undirected) is called **simple** if it does not contain loops or parallel edges. With the definitions given above, all graphs are simple. However, representations of graph often provide more freedom such that it is e.g. possible to have parallel edges. Restricting the discussion to simple graphs may be suitable for theoretical work. This allows for example to make use of the fact that $|E| \in O(|V|^2)$, which results in less complicated estimations on the required resources. However, an implementation of an algorithm should be able to cope with non-simple directed graph and undirected graphs as well as with graphs with mixed directed and undirected edges. Only in situations where a restriction is really necessary to make the algorithm work, the implementation should assume the corresponding restriction to be fulfilled.

The following definitions will use directed graphs only. The corresponding definitions for undirected graphs are essentially equivalent differing only in the way individual edges are handled: The corresponding definitions are omitted because it would basically be a repetition of the definitions for directed graphs with only minor modifications.

It is convenient to speak of modifications made to graphs instead of using the more rigorous abstract notation. The difference is basically to see a graph as a concrete object which is modified instead of an abstract entity. The view of a graph as an object is also closer to the view in implementations where the representation is normally really changed instead of creating a new graph differing in some aspect from another one. **Inserting** the node v into a graph $G = (V, E)$ means to replace the graph G by the graph $(V \cup \{v\}, E)$. Correspondingly, **inserting** the edge e where $e = (v, w)$

with $v, w \in V$, means to replace G by the graph $(V, E \cup \{e\})$. **Removing** the node $v \in V$ from the graph $G = (V, E)$ means to replace the graph G with the graph (V', E') where $V' = V \setminus \{v\}$ and $E' = \{(x, y) \in E | x, y \neq v\}$. **Removing** the edge $e \in E$ from the graph G means to replace G by the graph $(V, E \setminus \{e\})$. Inserting or removing sets of nodes or edges means to apply the corresponding operation with all elements from the set.

An **elementary contraction** removes two adjacent nodes v and w and inserts a new node v_{con} which is adjacent to all nodes which were adjacent to either v or w: Formally, the graph $G = (V, E)$ with $v, w \in V$ and $\{(v, w), (w, v)\} \cap E \neq \emptyset$ is replaced by the graph (V', E') where $V' = \{v_{con}\} \cup V \setminus \{v, w\}$ and $E = \{(x, y) \in E | x, y \in V'\} \cup \{(v_{con}, y) | (v, y) \in E \vee (w, y) \in E\} \cup \{(x, v_{con}) | (x, v) \in E \vee (x, w) \in E\}$. This modification is fairly complex and causes yet another problem: Most often, it is an auxiliary modification applied by some algorithm which has to be undone at the end of the algorithm. As there are normally multiple contractions used in such an algorithm (fortunately, there are only few algorithms needing such a modification), keeping the trace of all modifications can become rather complex.

Substructures of Graphs

A **subgraph** of a directed graph $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq \{e = (v, w) \in E | v, w \in V'\}$. The graph G' is called the subgraph of G **induced** by V', if $E' = \{e = (v, w) \in E | v, w \in V'\}$. A subgraph $G' = (V', E')$ is a **spanning subgraph** if $V' = V$ and $E' \subseteq E$.

A typical application of subgraphs is to construct an auxiliary subgraph from a graph to solve a problem on a graph with some known properties: Either the solution on the subgraph provides a solution on the whole graph or multiple subgraphs are constructed until the original problem is solved. Dinits' algorithm presented in Section 2.2 (on page 24) is an example of such an algorithm.

Let $G = (V, E)$ be a directed graph and $v, w \in V$ two nodes. A **path** from the start node v to the end node w in G is a finite sequence $P = (e_1, ..., e_k)$, $e_i \in E$ for $i = 1, ..., k$, of edges with $e_i = (v_{k-1}, v_k)$ or $e_i = (v_k, v_{k-1})$ for $i = 1, ..., k$ where $v = v_0$ and $w = v_k$: Two successive edges are incident to the same node, the first edges is incident ot v_0, and the last edge is incident to v_k. A **directed path** is the same as a path except that the directions of the edges are taken into account, i.e. $e_i = (v_{k-1}, v_k)$ for $i = 1, ..., k$. Note, that the definition of a path does not take the direction of the edges into account. This is only the case for a directed path.

Often paths are defined and represented as sequences of adjacent nodes. This is insufficient to uniquely identify a path if a graph is allowed to have parallel edges. This is no problem with the definitions of graphs given above but is a problem for some graph representations used in practice. In addition, even if there are no parallel edges, it is not suitable to represent a path as a sequence of nodes in an application where the edges are to be accessed: It would be necessary to find the edge between two successive nodes v_{i-1} and v_i which is not efficient for some graph representations. From a theoretical point of view, the definition of a path as a finite sequence of adjacent nodes is equivalent to the definition given above if the graph has no parallel edges.

A path P using each node only once, i.e. $v_i \neq v_j$ for $i, j \in \{0, ..., k\}, i \neq j$, is called a **simple path**. If the start node and the end node of a path C are identical, the path C is called a **circle**. A **simple circle** $C = (e_1, e_2, ..., e_k)$ is a circle where $(e_2, ..., e_k)$ is a simple path.

Using the concept of paths, it is possible to define the connectivity of graphs: A graph $G = (V, E)$ is said to be **connected** (**disconnected** otherwise) if for every pair of nodes $v, w \in V$ there is a path from v to w in G. A connected subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is called a **connected component** if G' is induced by V' and the graph induced by $V' \cup \{v\}$ is disconnected for every $v \in \{V \setminus V'\}$. Obviously, a connected graph is the only connected component of itself. For

most graph algorithms, the graph must be decomposed into connected components before using the algorithm. The algorithm is then applied to each of the connected components.

Some algorithms make use of even stronger knowledge of the connectivity: A graph $G = (V, E)$ is k-connected, if at least k or more nodes have to be removed from G to make the graph disconnected. The graph search algorithm from Section 2.2 can be used to find a connected component. On page 17 an algorithm finding 2-connected or **biconnected** components is given.

A **tree** $T = (V, E)$ is a connected graph without cycles. Thus, in T there is exactly one path between every two nodes $v, w \in V$. If a spanning subgraph $T = (V, E')$ of $G = (V, E)$ is a tree, T is called a **spanning tree** of G. The spanning tree T partitions the set of edges E of G into the set E' of **tree edges** and the set $E \setminus E'$ of **non–tree edges**.

A **rooted tree** (T, v_{root}) is a tree $T = (V, E)$ with a specially designated node $v_{root} \in V$, called the **root** of T. Using the root, a direction can be defined for all edges in the tree: If the unique path from the root v_{root} to every other node $v \in V \setminus \{v_{root}\}$ is a directed path, the tree is called an **out–tree**. If the unique paths from each node $v \in V \setminus \{v_{root}\}$ to the root v_{root} are directed, the tree is called an **in–tree**. When using rooted trees, often a parent/child relationship is assumed: Let $P = (e_1, ..., e_i)$ with $e_1 = (v_{root}, x)$ and with $e_i = (v, w)$ be the unique path from v_{root} to the node w. The node v is called the **parent** of w and w is a **child** of v. All nodes incident to an edge in P except w are called **ancestors** of w. All nodes which have the node $y \in V$ as ancestor are called the **descendents** of y. A node without a child is called a **leaf** while the other nodes are called **internal nodes**.

Spanning in–trees share the same problem as paths: They can easily be represented by storing for each node the parent node, at least if it is sufficient to traverse through the tree from leaves towards the root. However, this representation does not uniquely define the connecting edge in the corresponding graph if there are parallel edges. Thus, for every node different from the root, an edge leading to the parent has to be stored.

In general, a graph without any cycle is called a **forest**: Each of the connected components forms a tree, i.e. the graph is a set of trees.

Special Classes of Graphs

Graphs as defined above are too general for some applications. Instead specific subsets like planar graphs (see below) may be used in applications because they are adequate or sufficient models for the application. Some problems formulated with graphs can be solved more efficiently when the graph has a special structure. In addition, there are some special representations which make use of the properties of the graph classes or which allow efficient access to the special properties. Thus, it is suitable to handle classes of graphs differently when appropriate.

A very simple special class are the **complete graphs**: The undirected graph $K_n = (V, E)$ is the graph with $n = |V|$ nodes where each node is adjacent to every other node, i.e. $E = \{\{v, w\} | v \neq w \wedge v, w \in V\}$. The K_n is obviously the undirected graph with n nodes which has the most edges (if there are no parallel edges allowed). More precisely, the K_n has $\frac{n(n-1)}{2}$ edges.

A **bipartite** graph $G = (V, E)$ is a graph whose nodes can be partitioned into two sets V_1 and V_2 such that $V_1 \cup V_2 = V$ and there is no $e = (v, w) \in E$ such that $v, w \in V_1$ or $v, w \in V_2$, i.e. a bipartite graph is a graph on two sets of nodes where there are only edges connecting these two sets of nodes. The **complete bipartite** graph $K_{n,m} = (V, E) = (V_1 \cup V_2, E)$ is the graph with $n + m$ nodes partitioned into the set V_1 with $|V_1| = n$ and V_2 with $|V_2| = m$ where each node from V_1 is adjacent to every node in V_2. Thus, $E = \{\{v, w\} | v \in V_1, w \in V_2\}$.

A generalization of bipartite graphs are layered graphs: A graph $G = (V, E)$ is a **layered graph** if the following two conditions hold:

1. $V = \bigcup_{i=1,...,k} V_k$
2. $\forall e = (v, w) \in E$: $v \in V_i$ and $w \in V_{i+1}$

In other words, there exist k **layers**, the sets V_i, $i = 1, ..., k$, and there are only edges between two successive layers. Layered graphs are used e.g. by Dinits' algorithm (see Section 2.2, page 24).

An important class of graphs are planar graphs. An informal definition of a planar graph is this: A graph is called **planar** if it can be drawn in a two–dimensional (Euclidean) plane such that no two edges cross (or intersect each other). This (topological) definition is not well suited for algorithmic treatment. Instead, a definition using a combinatorial embedding is used for algorithmic treatment.

A **combinatorial embedding** of an undirected graph $G = (V, E)$ is a cyclic order of the edges incident to the node v for every node $v \in V$. A **face** in the combinatorial embedding is formed by a cycle C which is constructed such that the edge $e = \{v, w\}$ is immediately followed by the edge $f = \{w, u\}$ ($u, v, w \in V, e, f, \in E$) in the cycle if and only if $\{v, w\}$ is followed immediately by $\{w, u\}$ in the cyclic order of node w. A combinatorial embedding is a **planar combinatorial embedding** if the formula $|V| - |E| + f = 2c$ holds, where f is the number of faces in the embedding and c is the number of connected components of the graph. A **planar graph** is a graph for which a planar combinatorial embedding exists. It was shown that this combinatorial definition of planar graphs is equivalent to the topological definition above [7].

2.2 Algorithms on Graphs

This section presents some algorithms working on graphs. The algorithms presented are well known and solve problems which frequently arise as subproblems when solving more complex problems. The goal is not to develop new algorithms but rather to present algorithms in a way which is close to an implementation.

The algorithms presented in this section are used to identify the access methods which have to be provided for a graph representation to allow an efficient implementation of an algorithm. In addition, the reuse of algorithms as subalgorithms for more complex algorithms is demonstrated. This yields an insight in which way implementations need to be flexible: The necessity to have a flexible control on initialization and termination for an algorithm will be shown as well as the use of algorithms solving a relaxed problem. It will also be shown that it is useful to be able to augment the behavior of an algorithm with additional actions.

A Basic Algorithm: Graph Search

Graph search is a generic term for an algorithm accessing every node in a connected component of a graph. Its definition already yields many degrees of freedom: See Algorithm 2.1 for a definition of graph search as found in [21]. The authors claim that this algorithm runs in $O(|E|)$ time.

In this formulation, Q is a set used to maintain a list of nodes. This algorithm and its proof rely on the basic property that a set can contain every element just once, i.e. if an element already present in a set is added to the set, there is still only one copy stored in the set after the addition. If Q is implemented to maintain this set property, this algorithm performs with the claimed time bounds (provided that the basic operations on Q perform in constant time). Note that it is necessary to couple the implementation of the set with something like a queue or stack to gain additional properties of the search (see below).

Input: A graph G, represented by its adjacency lists; a node v_1
Output: The same graph with the nodes reachable by paths from the node v_1 "marked"

```
Q := {v_1}
while Q ≠ ∅ do
    let v be any element of Q
    remove v from Q
    mark v
    for all v' ∈ adj(v) do
        if v' is not marked then
            add v' to Q
        end if
    end for
end while
```

Algorithm 2.1: Graph Search: A basic outline

The problem with the set property leads towards an issue of correctness of implementations: In [21] on page 195 the set Q is implemented as a queue using an array where "add v to Q" is implemented as "$last := last + 1; Q[last] := v$" and "remove v from Q" is implemented as "first := first + 1; v := Q[first]". This does not yield the claimed time bound! Instead, with this implementation, the algorithm takes $\Omega(|V|^3)$ time when the algorithm is applied to a K_n, the complete graph on n nodes:

In the first iteration, all nodes except v_1, the initial node, are inserted in the queue. The only node marked is v_1. Thus, at the end of the first iteration, the queue contains the nodes $v_2, ..., v_n$. In the second iteration, v_2 is extracted from the queue and marked. In the inner loop all nodes adjacent to v_2, i.e. $n-1$ nodes, are investigated and all but the nodes v_1 and v_2 are added to the queue: None of the nodes $v_3, ..., v_n$ is marked. Thus, at the end of the second iteration the queue contains the nodes $v_3, ..., v_n, v_3, ..., v_n$: The nodes $v_3, ..., v_n$ are present twice in the queue: The implementation of Q given has no way to avoid duplicates in the queue. In the i-th iteration, for $i < n$, there are $n-1$ nodes examined and the nodes $v_{i+1}, ..., v_n$ are added to the queue: Only the nodes $v_1, ..., v_i$ are marked at this point. If $i \geq n$ no node will be added to the queue because all nodes are marked. But all adjacent nodes, i.e. as many as $n-1$ nodes, will be examined. Thus, the total amount of time required can be calculated as the number of nodes put in the queue times the number of nodes adjacent to the respective node, which is always $n-1$ for the nodes of K_n:

$$T(n) = \sum_{i=n-1}^{1} i(n-1) = \frac{(n-2)+1}{2}(n-1)(n-1) = \frac{1}{2}(n-1)^3$$

But $T(n) \notin O(n^2) = O(|V|^2) = O(|E|)$.[2] Thus, the time bound given in [21] is not achieved for the implementation of Q given. □

It is no problem to correct the algorithm to perform with the desired time bound. All what is necessary is to make the execution of the inner loop dependent on the mark of the node v: If this loop is only executed if v is not yet marked, the algorithm has the desired bounds. The major problem with this error is that it only yields a worse execution time than the expected one: The results calculated are correct. In a normal testing procedure for the algorithm the error will probably not be detected if no profile of the execution time is made.

[2] The equality $O(|V|^2) = O(|E|)$ holds because the number of edges is quadratic in the number of nodes for the K_n.

This little digression is not taken to blame the authors. Instead it demonstrates that it is possible to make errors even when implementing a simple algorithm like graph search. Thus, it is worth reusing an implementation of such a simple algorithm, too: One of the reasons for reusing code is to avoid errors introduced by a new implementation.

But let's get back to the graph search. In every iteration one node, in the above formulation node v, is treated special. This node is said to be the **current node**. The current node is used to determine which nodes are to be investigated and potentially put in the set Q. If the edge (v, w) of the current node v and an adjacent node w is put in the set instead of just the node w, a graph search can be used to create a spanning tree of the connected component of v_1: Instead of a single node v, the edge (p, v) is selected from the set Q. The node p becomes the parent of v in the spanning tree if v is not yet marked. The edge (p, v) can e.g. be used to record the parent of a spanning tree with v_1 as root. Whether or not the tree is actually created, this tree is henceforth referred to as the **implicit tree** of the graph search.

With this approach there is just a minor problem: Q is initialized with a node, not with an edge. An option to avoid this problem in an implementation is to use special edges: There can be a special representation for non-existing edges e.g. a pair which consists of a node and an indicator for an invalid node (like "*NIL*"). Thus, initially an invalid edge can be stored to indicate that the node is the root of the implicit tree. The notation (NIL, v) will be used in the description below to indicate an invalid edge used to store a node v.

The algorithm as presented above (if corrected) provides some flexibility in the output created:

- The implementation of Q is left open. Using different implementations for Q, it is possible to get different characteristics of the implicit tree. Two well known choices for Q are a **queue** and a **stack** resulting in a breadth first search (BFS) or a depth first search (DFS), respectively (see below).

- The marks used are not specified. For a graph search used to check whether the graph is connected, boolean marks would be sufficient. It is also possible to use ascending numbers to label the nodes with numbers indicating the order in which they became current. Yet another choice for the marks is to use "version numbers" to use the same marks for multiple searches without having to initialize the marks: Each individual search uses a number bigger than the previous number used to mark the nodes. Nodes marked with a smaller number than the currently used number are considered to be unmarked.

The set Q is used to store the nodes which are eligible for investigation. From a theoretical point of view this can be a set where the next node to become current is chosen randomly. From a practical point of view, it often makes more sense to use a different strategy than random selection. Thus, the set Q is actually a container defining an order on the nodes inserted.

One approach already mentioned is to use a queue to implement this container: The nodes are extracted from the queue in the same order in which they are inserted into the queue. A queue is sometimes called a **FIFO**: "First In First Out". When using a queue for Q, the implicit tree grows apparently into the breadth such that a graph search using a queue is called a **breadth first search** or **BFS** for short. It can be shown that a path from the root v_1 to a node v in the implicit tree generated by a BFS is a shortest path with respect to the number of edges from v_1 to v in the input graph.

The other approach mentioned is to use a stack to implement the container: The nodes are extracted in the reverse order in which the nodes currently stored in the container are inserted. Thus, a stack is also called a **LIFO**: "Last In First Out". The implicit tree when using a stack grows apparently into the depth and the corresponding graph search is called **depth first search** or **DFS**.

Some of the potential flexibility of graph search is not explicit in the formulation above. Here are some of the things which could not be made with Algorithm 2.1:

- There is no reason why there should not be more than one root node. For example, if the shortest path from one of multiple start nodes $s_1, ..., s_k$ to a given end node t has to be determined, it could be useful to put the nodes $s_1, ..., s_k$ as roots of the graph search in the container Q.
- Often it is useful to perform additional actions in each iteration of the algorithm: For example, the parent in the implicit tree has to be recorded to explicitly construct this tree or the node has to be labeled with some label.
- In some applications of the graph search algorithm, it is important to access all edges. This is e.g. necessary for the biconnectivity algorithm presented below. Algorithm 2.1 only marks the nodes although it accesses all edges. Like for the nodes, it is useful to allow an arbitrary action for every edge.
- The condition when the algorithm terminates is fixed: The graph search terminates once all nodes are removed from the container. A different termination condition useful e.g. for a path search algorithm (see below) is to terminate once a certain node became current.
- The algorithm asks for a graph represented as a set of adjacency lists. This is not really necessary. All what is necessary is that the nodes adjacent to a given node can be accessed efficiently (this restriction can be further relaxed: Efficient access is only necessary to make the graph search algorithm run as efficient as possible).

Thus, it makes sense to reformulate the algorithm to make use of this additional flexibility. See Algorithm 2.2 for a more flexible formulation of graph search.

Input: A graph $G = (V, E)$, with efficient sequential access to all elements of $inc(v)$ for $v \in V$; a set of nodes $v_1, ..., v_k \in V$
Output: The graph G with the nodes "marked" which are reachable from at least one of the nodes $v_1, ..., v_k$.

```
Add the invalid edges (NIL, v_1), ..., (NIL, v_k) to Q
while Q is not empty and termination condition not yet reached do
  let (p, v) be the next edge from Q
  remove edge (p, v) from Q
  if v is not yet marked then
    optionally perform some additional action for (p, v)
    mark v
    for all (v, v') ∈ inc(v) do
      optionally perform some action for edge (v, v')
      if v' is not marked then
        add edge (v, v') to Q
      end if
    end for
  end if
end while
```

Algorithm 2.2: Graph Search: Full flexibility

Q is a container of edges (including the illegal edges). Let (p, v) be the object most recently extracted from Q. The node v is the node which becomes the new current node. The object p is either *NIL*, or some node: If $p = NIL$, the node v is one of the root nodes of the graph search. Otherwise, i.e. $p \neq NIL$, p identifies the parent node of v in the implicit forest (there are, potentially, multiple root nodes, thus there is no connected spanning tree but rather a set of implicit trees) of the graph search. The edge (p, v) is the tree edge leading from the parent node p to the node v.

This formulation is a little bit more complex but much more flexible. Using the concepts from Chapter 4 the interface to an algorithm implemented corresponding to the formulation in 2.2 can be made easy while making the implementation itself even more flexible: As formulated in Algorithm 2.2 there may be $\Omega(|E|)$ edges stored in the container. With a slight modification it is possible to reduce this amount to $O(|V|)$, at least in some situations: This can be essential if $|V|$ is large and the graph is nearly complete. The idea is this: The inner loop inserts each edge incident to the current node which leads to a node not yet marked. This is not always necessary: If not all edges have to be known to the container to determine the next edge, it can be sufficient to store just the information to find all edges incident to a node. This information can then be used when an edge is extracted to determine the next edge. Instead of removing the edge, the information stored is updated to indicate that the current edge is removed and only subsequent edges are to be considered. This is done until all edges incident to a node are processed. At this time the information stored is removed. Thus, the inner loop is moved into the container Q: If the container needs to know about all edges (as is e.g. the case for a priority queue) the information passed to the container is used to insert all edges incident to the current node. Otherwise, the information is used to obtain the next edge whenever an edge is removed from the container.

With the graph search algorithm as given in Algorithm 2.2, Dijkstra's algorithm is a simple specialization using a priority queue as container. To make this possible, the priority queue has to be extended to record the distance of a node if an edge to a not yet marked node is extracted from the container: The length of the edge and the distance of the parent node is known (for *NIL* it can be defined to be zero) such that the distance of the not yet marked node becomes the length of the edge plus the distance of the parent node. This value has to be calculated anyway because it is used for the sorting criterion of the priority queue.

The graph search algorithm accesses all edges incident to a given node. Thus, it is necessary for a graph representation allowing the application of the graph search algorithm given above to provide (efficient) access to the adjacency list of a node.[3] In addition to this access, the algorithm marks the nodes and checks whether a node is marked. In other words, an implementation of this algorithm needs to store and retrieve auxiliary data associated with the nodes.

Finding Paths

A problem arising in some situations is to find a path between two nodes in a graph: Given a graph $G = (V, E)$ and two nodes $s, t \in V$, find a path from the node s to the node t in G.

A general approach to solve this problem is to use a graph search to find the path. The search is started at node s and proceeds by recording for the current node the parent node in the (partial) implicit tree until the node t is marked. The graph search can terminate as soon as t is marked or when the first edge (v, t), is put in the container.[4] Algorithm 2.3 shows an algorithm extending graph

[3] If the graph contains parallel edges, it is not sufficient to have access to the nodes adjacent to a node: In this case it is necessary to provide access to the edges incident to a node.

[4] The reason to wait until the node t is marked instead of terminating the algorithm if the first edge (v, t) is put in the container is to make sure that the last edge has the properties obtained for the edge by the specific choice of the container.

search (Algorithm 2.2) to find a path in a graph.

The extension mainly consists of three parts:

1. Whenever a node is marked, the edge leading to the parent node in the (partial) implicit tree is recorded.

2. If the target node t is reached, the algorithm constructs the path by traversing the implicit tree from the node t to the root s and adding the edges used to the path. When the path is constructed, the algorithm terminates by returning the path.

3. If the graph search terminates without reaching the node t, an empty path is returned to indicate an error: There is no path from the node s to the node t in G.

Input: A graph $G = (V, E)$, with efficient sequential access to all elements of $inc(v)$ for $v \in V$; a start node $s \in V$; a target node $t \in V$
Output: A path $P = (e_1, ..., e_k)$ from s to t in G or an error indication

```
add the invalid edge (NIL, s) to Q
while Q is not empty do
  let (p, v) be the next edge from Q
  remove edge (p, v) from Q
  if v is not yet marked then
    set parent(v) := (p, v)
    mark v
    if v = t then
      set P = ()
      while v ≠ s do
        let (p, v) be parent(v)
        set P := ((p, v), P)
        set v := p
      end while
      return P
    end if
    for all (v, v') ∈ inc(v) do
      if v' is not marked then
        add edge (v, v') to Q
      end if
    end for
  end if
end while
return ()
```

Algorithm 2.3: Path finding using graph search

This algorithm does not cover all situations as efficiently as possible (from a practical point of view): For example, the edges to the parent nodes are always stored although this is not necessary if a stack is used as container. If a stack is used, the stack contains the path from the root node to the

This can be an issue e.g. if a priority queue is used as container.

current node in reverse order. Thus, in this case there is no need to store the edge to the parent with every node.

Another issue is the construction of the path: The construction of the path returned requires memory for data already stored. Whether the stored data, i.e. the edges of the in–tree which can be used to traverse the path in reverse order (from node t to node s), is sufficient for an application using the path finding algorithm, depends on what is done with the path. For some algorithms it is sufficient to know which edges are in the path ignoring how they are ordered (e.g. the algorithm of Ford–Fulkerson, see Algorithm 2.8). If the data is sufficient, there is no need to create a new sequence of edges wasting time and space to create a representation of the path. Instead, it is sufficient to somehow provide access to the edges of the path.

This path finding algorithm does not take advantage of properties known of the graph, resulting in an algorithm which always runs in $O(|E|)$ time. In some situations the graph has some known properties which might result in a more efficient algorithm: For example, with Dinits' algorithm a layered graph is constructed where every node lies on a path from the start node s to the target node t using only one node of every layer: In this graph, it is possible to construct a path in $O(l)$ time, if l is the number of layers in the graph. In algorithms using a path finding algorithm as subalgorithm, the use of an algorithm taking advantage of some known properties might be indicated to get a more efficient implementation.

As mentioned above, there are different possibilities to find a path between two nodes in a graph. Thus, the algorithm used to find a path should not be hard–coded into an algorithm using a find path algorithm. Instead, the implementation should be free to use a path finding algorithm appropriate in the situation which might use additional knowledge resulting in specific properties of the path and/or a more efficient implementation.

The algorithm given to find a path provides an example of an extension of the graph search algorithm: It adds statements in the body of the graph search algorithm. In addition to the marks used by the graph search, the algorithm needs to store additional data for every node, namely the edge to the parent of the node in the implicit tree. In the formulation above, the mark and the edge are not related but an implementation can choose to represent the marks together with the edges: If the node is marked (and the mark is accessed), the edge to the parent node is also determined. Thus, the representation of the mark can be avoided if it is possible to figure out for a node whether the edge to the parent node has already been determined.

Connected Components

The graph search algorithm marks every node which becomes current to avoid an endless recursion. These marks can be used to determine the connected component of a node v_{root}: With v_{root} as start node, the nodes marked after termination of the graph search are the nodes in the connected component of v_{root}. Using different numbers for multiple applications of the graph search, it is possible to mark every node with a number identifying the connected component (see Algorithm 2.4): The list of nodes is searched for a node not yet marked. If such a node is found, it is used as the root for a graph search using the current number of connected components as marks (which is initially 1 and incremented after every graph search).

This reuse of graph search does not extend the internals of the graph search (apart from defining what marks are to be used) but applies the algorithm directly. A more complex problem is to determine biconnected components. Algorithm 2.5 determines the biconnected components of a connected graph. To determine the biconnected components of a disconnected graph, the algorithm has to be applied to all connected components. The algorithm presented is a sequential version of a recursive

Input: A graph $G = (V, E)$, with efficient sequential access to all elements of V and $inc(v)$ for $v \in V$

Output: The graph G with every node "marked" with the number of the respective connected component.

set `current_component := 1`
for all $v \in V$ **do**
  if v is not yet marked **then**
    do a graph search with v as root and `current_component` as mark
    set `current_component := current_component+1`
  end if
end for

Algorithm 2.4: Finding connected components

algorithm given by Tarjan [24].

By definition a connected graph is not biconnected, if there is a node v whose removal leaves a disconnected graph. Such a node is called a **cutting node**. Algorithm 2.5 uses the implicit tree of a depth first search to find the cutting nodes in a graph $G = (V, E)$: Let v_{root} be the root of the DFS. A node $v \neq v_{root}$ is a cutting node, if there exists a child w of v such that there is no edge $e \in E$ from w or a descendent of w to an ancestor of v in the implicit tree of the DFS. The node v_{root} is a cutting node if it has more than one child [24].

To determine efficiently whether a node is a cutting node, the nodes are labeled with increasing numbers in the order they are marked during the search i.e. every node v is labeled with its **DFS–number** $dfs\_no(v)$. The DFS–number is used to determine the **low–point number** $low(v)$ for every node v: $low(v)$ is the smallest DFS–number of a node incident to a non–tree edge which is incident to v or a descendent of v. If the low–point numbers and the DFS–numbers of the nodes are given, it is easy to determine whether a node s cutting node: If the low–point number $low(v)$ of a node $v \neq v_{root}$ is greater or equal than the DFS–number of its parent node then is its parent node a cutting node.

If it is known which nodes are the cutting nodes, a DFS can be used to determine the edges forming a biconnected component (note that a node can be part of multiple biconnected components): The edges encountered during the DFS are recorded on a stack. If a cutting node is found, all edges on the stack with a low–point number bigger or equal than the DFS–number of the cutting are removed from the stack and labeled with the current component number. These edges form a biconnected component. In Algorithm 2.5, the low–point numbers are determined during the same DFS as the connected components. It would also be possible to split the algorithm into two phases: One to determine the low–point numbers and one to construct the biconnected components.

This algorithm is only useful to compute the biconnected components of a graph although it would also be reasonable to just compute e.g. the low–point number separately as it can be used for different purposes, too (e.g. to compute an "st–numbering" [8]). In addition, it is not clear what the output of the algorithm should be: In the formulation above, the algorithm computes edges labels. It may also be desired to construct several graphs each of which represents a biconnected component.

The algorithm basically reuses the graph search algorithm although in some extended from. During the search, the container maintained includes a little bit more data than in the graph search algorithm. Instead of putting all edges incident to the current node in the container used by the search, the algorithm stores an edge to the parent node and the set of edges to be considered. The main difference between this approach and the approach in the formulation of Algorithm 2.2 is the possibility to know when the last edge incident to a node was considered: This information is used to update the

low–point number of the parent node only once.

Input: An undirected, connected graph $G = (V, E)$, with efficient sequential access to all elements of $inc(v)$ for $v \in V$
Output: The graph G with every edge "marked" with the number of the respective biconnected component.
choose some node $v_1 \in V$
mark v_1 and let $dfs(v_1) := 1$ and $low(v_1) := 1$
let $dfs\_no := 2$ and $comp\_no := 1$
push $((NIL, v_1), inc(v_1))$ on the search stack
while the search stack is not empty **do**
  let $(\{p, v\}, INC)$, with $\{p, v\} \in E$ and $INC \subseteq inc(v)$ be the top of the stack
  while $INC \neq \emptyset$ and there is an edge $\{v, w\} \in INC$ with w marked **do**
    let $low(v) = \min\{low(v), dfs(w)\}$; remove $\{v, w\}$ from INC.
  end while
  if $p = NIL$ or $(INC = \emptyset \wedge low(v) \geq dfs(p))$ **then**
    while top edge of component stack $\{x, y\}$ has $low(y) \geq low(v)$ **do**
      let $comp(\{x, y\}) = comp\_no$; remove $\{x, y\}$ from component stack
    end while
    let $comp\_no := comp\_no + 1$
  end if
  if $INC \neq \emptyset$ **then**
    let $\{v, w\} \in INC$ any edge from INC
    mark w and let $dfs(w) := dfs\_no$, $low(w) := dfs\_no$, and $dfs\_no := dfs\_no + 1$
    push $\{v, w\}$ on the component stack and $(\{v, w\}, inc(w))$ on the search stack
  else
    if $p \neq NIL$ **then**
      let $low(p) := \min\{low(p), low(v)\}$
    end if
    remove $(\{p, v\}, INC$ from search stack
  end if
end while

Algorithm 2.5: Finding biconnected components

During the algorithm, the nodes marked by the graph search are actually marked with the corresponding DFS–number: If the DFS–number the of the nodes are initialized with is some illegal value, e.g. -1, it can be determined whether a node is marked by checking whether the current DFS–number is different from the initial value. In addition to the DFS–number for the nodes, the algorithm uses another label for the nodes (the low–point number) and a label for the edge (the component number). It also uses two different stacks: One stack to guide the graph search and another one to keep track of the edges encountered. The latter stack is used to determine which edges form a biconnected component.

Solving the Maximum–Flow Problem

Before defining the general max–flow problem, a **flow** is defined: Given a directed graph $G = (V, E)$, a set $S \subset V, S \neq \emptyset$ of sources, a set $T \subset V, T \neq \emptyset$ of targets, $S \cap T = \emptyset$, a function $lower : E \mapsto \mathbb{R}_0^+$ of lower bounds, and a function $upper : E \mapsto \mathbb{R}_0^+$ of upper bounds, a **feasible flow** is a function

$flow : E \mapsto \mathbb{R}_0^+$ with

$$lower(e) \leq flow(e) \leq upper(e) \quad \text{for all} \quad e \in E, \tag{2.1}$$

$$\sum_{e \in inc(v)} flow(e) - \sum_{e \in out(v)} flow(e) \geq 0 \quad \text{for all} \quad v \in S, \tag{2.2}$$

$$\sum_{e \in inc(v)} flow(e) - \sum_{e \in out(v)} flow(e) \leq 0 \quad \text{for all} \quad v \in T, \quad \text{and} \tag{2.3}$$

$$\sum_{e \in inc(v)} flow(e) - \sum_{e \in out(v)} flow(e) = 0 \quad \text{for all} \quad v \in V \setminus (S \cup T) \tag{2.4}$$

If the lower bounds are all zero, a feasible flow is often called just a **flow**. A (feasible) flow in a graph without sources and targets, i.e. a flow where 2.4 holds for all nodes, is called a **circulation**. Circulations are used to determine a feasible flow (see below).

An illustrative interpretation is to assume the function $flow$ defines the amount of fluid flowing through some pipes, the edges of the graph. The amount of fluid flowing through the pipe e is at least $lower(e)$ and at most $upper(e)$. The fluid is poured in the sources (the nodes from S) and is vanishing in the targets (the nodes from T). In the remaining nodes, fluid is neither lost nor gained. The **flow value** $flow\_val_{flow}(G)$ of the graph G with a certain flow $flow$ is the amount of fluid flowing through the system of pipes, i.e.

$$flow\_val_{flow}(G) = \sum_{v \in S} \left(\sum_{e \in out(v)} flow(e) - \sum_{e \in in(v)} flow(e) \right) \tag{2.5}$$

$$= -\sum_{v \in T} \left(\sum_{e \in out(v)} flow(e) - \sum_{e \in in(v)} flow(e) \right) \tag{2.6}$$

The **general max–flow problem** is to find a (feasible) flow $flow$ such that $flow\_val_{flow}(G)$ is maximal. The **max–flow problem** is to find a flow such that $flow\_val_{flow}(G)$ is maximal for a problem where $lower(e) = 0 \quad \forall e \in E$ and $|S| = |T| = 1$. Algorithms solving the max–flow problem can be used to solve the general max–flow problem involving some modifications of the original graph. The approach used to solve the general max–flow problem involves several steps [4]:

1. The multi–source/multi–target problem is converted into an equivalent single–source/single–target problem by introducing a super–source s and a super–target t incident to all source or all targets, respectively, in the original problam (i.e. $S = s$ and $T = t$ in the modified problem).

2. A feasible solution, i.e. one fulfilling the lower bounds, is constructed by solving a single–source/ single–target max–flow problem without lower bounds (i.e. $lower(e) = 0 \quad \forall e \in E$).

3. The feasible solution, if there exists such a solution, is augmented to become a maximal solution solving another single–source/single–target max–flow problem without lower bounds.

The first step modifies the graph to get an easier description of the problem (see Algorithm 2.6): Two nodes, s and t are inserted into the graph. s becomes a "super–source", which is connected by an edge (s, v) to each node $v \in S$, and t becomes a "super–target", which is connected by an edge (v, t) to each node $v \in T$. The lower bounds for the flow are all zero for these edges and the upper bounds are N where $N \in \mathbb{R}_0^+$ is sufficiently large, e.g. $N > \sum_{e \in E} upper(e)$.

Input: A directed graph $G = (V, E)$, a set $S \subset V$ of sources, a set $T \subset V$ of targets, $N \in \mathbb{R}_0^+$
Output: The graph G converted to a single–source/single–target problem

```
insert a new node s in G
for all v ∈ S do
   insert a new edge e = (s, v) into G
   set lower(e) := 0 and upper(e) := N
end for
insert a new node t in G
for all v ∈ T do
   insert a new edge e = (v, t) into G
   set lower(e) := 0 and upper(e) := N
end for
```

Algorithm 2.6: Construct a single–source/single–target max–flow problem

If the edges of the resulting graph have no lower bounds (i.e. $lower(e) = 0 \quad \forall e \in E$), the problem can be solved by the algorithms presented below (Algorithms 2.8, 2.9, or 2.10). Otherwise, it is necessary to establish a feasible flow (if possible) before these algorithms can be used. The algorithm to establish a feasible flow (Algorithm 2.7) modifies the graph again and uses an algorithm solving the max–flow problem: First, an auxiliary edge from the target to the source is added. This way the flow is converted into a circulation. Then, for every edge with non-zero lower bounds sufficient flow is send over the edge to satisfy the lower bounds. This creates an overflow in some nodes and a demand for flow in other nodes. The problem can then be seen as a multi–source/multi–target problem: All nodes with overflow are considered to be sources and thus connected with edges to a new source. Since the overflow available at a node v is limited, the upper capacity of the edge between the new source and v is set to the overflow available. The nodes with some demand are handled correspondingly by connecting them with edges to a new target. A feasible flow is then established by solving a maximum flow problem on the resulting graph and adding this flow to the flow send over the edges with lower bounds. If the flow over an edge leaving the auxiliary source does not use the whole capacity available, there is no feasible flow possible with the lower and upper bounds (the corresponding holds for an edge entering the auxiliary target but it is sufficient to test the edges incident to one of the nodes): The overflow produced in a node by satisfying the lower bounds can not be transported away (for edges leading to the auxiliary target the demand could not be satisfied in a node). In this case, an error is indicated.

The main work of the algorithm to solve the general max–flow problem is done by an algorithm to solve the max–flow problem: It is used both to establish a feasible flow, if this is necessary, and to perform the maximization of the flow.

For the description of the algorithms maximizing the flow, it is convenient to assume, that the graph $G = (V, E)$ is transformed into a graph $G' = (V, E')$ by replacing each directed edge with two anti-parallel edges: The edge $e = (v, w)$ with $upper(e) = u$, $lower(e) = l$ (where l is set to 0 if there are no lower bounds), and $flow(e) = f$ is substituted by the two edges $e' = (v, w)$ with $res\_cap(e') = u - f$ and $e'' = (w, v)$ with $res\_cap(e'') = f - l$: The function $res\_cap : E' \mapsto \mathbb{R}_0^+$ defines the **residual capacity** of the edges, i.e. $res\_cap(e), e = (v, w) \in E'$, specifies the amount of flow which can be send from v to w using the edge e. The function $reverse(e)$ is used to obtain the reverse edge for the edge e, i.e. the edges e and $reverse(e)$ are the edges substituted for an edge in the original graph.

The algorithms maximizing a flow only know about residual capacity and nothing about a flow.

The flow of an edge can be calculated by undoing the substitution. If the edge $e = (v, w) \in E$ was substituted by $e' = (v, w), e'' = (w, v) \in E'$, the flow of $\mathit{flow}(e)$ is the residual capacity $res\_cap(e'')$ of e'' plus the lower bound $lower(e)$ (if there is a lower bound). Note, that the substitution is only the view of the graph taken by the algorithm: It not necessary to really substitute every edge in the representation by a pair of edges. It is only necessary for the algorithms to make the graph appear as if this substitution was made.

Input: A directed graph $G = (V, E)$, a source $s \in V$, a target $t \in V$, lower bounds on the flow $lower : E \mapsto \mathbb{R}_0^+$, upper bounds on the flow $upper : E \mapsto \mathbb{R}_0^+$, a sufficient large $N \in \mathbb{R}_0^+$
Output: A flow $\mathit{flow} : E \mapsto \mathbb{R}_0^+$ satisfying the lower bounds on the flow or an error indication if no such flow exists

add an edge $e = (t, s)$ to E with $lower(e) = 0$ and $upper(E) = N$
for all $e \in E$ **do**
  set $\mathit{flow}(e) := lower(e)$ and $upper(e) := upper(e) - lower(e)$
end for
add a new source s' and a new target t' to V
for all $v \in V \setminus \{s', t'\}$ **do**
  let $balance := \sum_{e \in in(v)} lower(e) - \sum_{e \in out(v)} upper(e)$
  if $balance < 0$ **then**
    add an edge $e = (v, t')$ with $upper(e) := -b$ and $\mathit{flow}(e) := 0$ to E
  else if $balance > 0$ **then**
    add an edge $e = (s', v)$ with $upper(e) := b$ and $\mathit{flow}(e) := 0$ to E
  end if
end for
solve the max–flow problem on G with upper bounds $upper$; let the result be $\mathit{flow}_{mathitaux} : E \mapsto \mathbb{R}_0^+$
if $\exists e \in \{(s', v) \in E\}$ with $\mathit{flow}_{aux}(e) < upper(e)$ **then**
  indicate an error: There is no valid flow
else
  for all $e \in E$ **do**
    if $e = (s, t)$ or $e \in out(s')$ or $e \in in(t')$ **then**
      remove e
    else
      let $\mathit{flow}(e) := \mathit{flow}(e) + \mathit{flow}_{aux}(e)$ and $upper(e) := upper(e) + lower(e)$
    end if
  end for
  return the flow $\mathit{flow}(e)$
end if

Algorithm 2.7: Establish a feasible flow

A simple algorithm to maximize the flow in a graph is to find directed paths from s to t where every edge e has a positive residual capacity ($res\_cap(e) > 0$) and to augment flow along these paths (see Algorithm 2.8): Let $P = (e_1, ..., e_k)$ be a path from s to t. The residual capacity $res\_cap(P) = min_{e \in P} res\_cap(e)$ of the path P is the minimum of the residual capacities of the edges in P. It is possible to increase the flow by an amount of $res\_cap(P)$ on the edges along the path without violating the bounds on the flow. In addition, the flow condition 2.4 is also not violated: The additional amount of flow sent to a node is immediately send to the next edge until the target is reached.

Input: Directed graph $G = (V, E)$, a source $s \in V$, a target $t \in V$, residual capacities $res\_cap : E \mapsto \mathbb{R}_0^+$
Output: residual capacities implying a maximal flow
find a path $P = (e_1, ..., e_k)$ from s to t with $res\_cap(P) > 0$
while $P \neq ()$ **do**
for all $e \in e_1, ..., e_k$ **do**
set $res\_cap(e) := res\_cap(e) - res\_cap(P)$
set $res\_cap(reverse(e)) := res\_cap(reverse(e)) + res\_cap(P)$
end for
find a path $P = (e_1, ..., e_k)$ from s to t with $res\_cap(P) > 0$
end while

Algorithm 2.8: Augmenting–Path algorithm to solve the max–flow problem

The flow constructed with this algorithm is maximal due to the Max–Flow/Min–Cut theorem [10]. The algorithm is called **augmenting path algorithm** or **Algorithm of Ford/Fulkerson**. This algorithm runs, in general, in pseudo–polynomial time. If the paths are constructed with certain limitations, e.g. using only shortest paths or the path with with the maximal residual capacity to increase the flow, yields polynomial time algorithms [6].

Input: Directed graph $G = (V, E)$, a source $s \in V$, a target $t \in V$, residual capacities $res\_cap : E \mapsto \mathbb{R}_0^+$
Output: residual capacities implying a maximal flow
compute exact distance labels $dist(v)$ for all $v \in V$
set $dist(s) := |V|$
for all $e = (s, v) \in out(s)$ **do**
push $res\_cap(e)$ flow from s to node v using e
end for
while there is a node $v \in V$ with $excess(v) > 0$ **do**
select a node v with $excess(v) > 0$
if $\exists e = (v, w) \in out(v) : res\_cap(e) > 0 \wedge dist(v) = dist(w) + 1$ **then**
push $res\_cap(e)$ flow from v to node w using e
else
set $dist(v) := min\{dist(w) + 1 | (v, w) \in out(v)\}$
end if
end while

Algorithm 2.9: Preflow–Push algorithm to solve the max–flow problem

The augmenting path retains the property that the flow conforms to the flow condition 2.4 throughout the execution of the algorithm.[5] A different approach to solve the problem, the **Preflow–Push Algorithm**, first attempts to push as much flow towards the target and later corrects violations of the flow condition (see Algorithm 2.9). To track the violation of the flow condition, in each node the amount of **excess** is stored: $excess(v), v \in V$ is the amount of flow which has to be pushed to an adjacent node. In the formulation of this algorithm a **push** of some flow f along and edge $e = (v, w)$ is the following

[5]More precisely, the algorithm maintains the fbw balance in the nodes (except the source and the target) throughout the algorithm: If the fbw balance is initially violated in some node, the fbw balance will be violated by the same amount at the end of the algorithm.

operation: The flow on the edge e is increased by f, i.e. $res\_cap(e)$ is replaced by $res\_cap(e) - f$ and $res\_cap(reverse(e))$ is replaced by $res\_cap(reverse(e) + f)$. At the same time, the excess $excess(v)$ of node v is decreased by f while the excess $excess(w)$ is increased by w. Initially, the excess of all nodes is zero. The algorithm uses a distance from the target for its initialization: The distance used is the length with respect to the number of edges of a shortest path from the respective node to the target node t. This distance can e.g. be computed using the graph search algorithm with a queue as container.

This preflow–push algorithm [15] leaves some details unspecified such as the choice of the node with positive excess and the choice for the edge used to push the flow. In addition, there are several improvements which can be embedded into this framework: Heuristics to recalculate the exact distance from time to time and an early detection of a saturated cut (i.e. the situation that there is no directed path P from the source to the target with positive residual capacity on all edges) can yield a better run–time.

While the augmenting path and the preflow push algorithms solved the max–flow problem immediately, the problem is transformed into a sequence of problems by **Dinits' Algorithm** [5] shown in Algorithm 2.10 . The idea is to create a max–flow problem on layered graphs: The node s is the only node in the first layer L_1 and the node t it the only node in the last layer L_{k+1}, where k is the length of a directed, shortest path (with respect to the number of edges) from s to t using only edges with a positive residual capacity. The layered graph $G' = (V, E')$ is constructed from the original graph $G = (V, E)$ by ignoring all edges not laying on a shortest path P from s to t with $res\_cap(P) > 0$. A layered graph $G' = (V, E)$ with $k + 1$ layers $L_1, ..., L_k$ with source $s \in L_1$, target $t \in L_{k+1}$, and residual capacity $res\_cap : E \mapsto \mathbb{R}_0^+$ is called a **layered network** if the following properties hold:

1. The source s is the only node in L_1 and the target t is the only node in layer L_{k+1}.
2. Every edge $e \in E'$ has a residual capacity $res\_cap(e) > 0$.
3. Every edge $e \in E'$ lies on a directed path P from s to t in G'.

A layered network can be constructed using a BFS starting at the source s and terminating when the target node t is reached. During the search the edges between the layers (and the layers) are recorded and after the search the edges not laying on a path between the source and the target are removed. When a layered graph is constructed, a max–flow problem on this graph is solved. If this algorithm takes no advantage of the layered structure of the graph, it is no real benefit to use Dinits' algorithm but e.g. the augmenting path algorithm can be improved by the use of a path finding algorithm using the layered structure.

Input: Directed graph $G = (V, E)$, a source $s \in V$, a target $t \in V$, residual capacities $res\_cap : E \mapsto \mathbb{R}_0^+$
Output: residual capacities implying a maximal flow
while $\exists P : res\_cap(P) > 0$ **do**
construct a layered graph G' where each edge e has $res\_cap(e) > 0$ and is part of a shortest path P with $res\_cap(P) > 0$
solve the max–flow problem on G'
end while

Algorithm 2.10: Dinits' algorithm to solve the max–flow problem

A max–flow problem on a layered network can be solved using a simple algorithm. For a node $v \neq s, t$ in the layer L_i the **through–put** is defined as

$$throughput(v) = min\{\sum_{e \in \{(w,v) \in E' : w \in L_{i-1}\}} res\_cap(e), \sum_{e \in \{(v,w) \in E' : w \in L_{i+1}\}} res\_cap(e)\}$$

This is the amount of fluid which can pass through this node without violating the bounds on the flow. The algorithm finds the node v with the minimal through–put. Let v be in the layer L_i. The algorithm pushes $throughput(v)$ units of flow to the nodes in the layer L_{i+1} using as many edges as necessary to transport the flow. From there, the flow is pushed to the next layer and so on until the flow arrives at the target. Likewise, flow is pushed from the layer L_{i-1} to the node v creating demands in some or all nodes of the layer L_{i-1}. The missing flow is pushed from the previous layer and so on until the flow is pushed from the source.

The approach described to solve the general max–flow problem involves several steps. Each of these steps can be performed using a different algorithms: Depending on the representation or known properties of the graph, an application of these algorithms might choose a suitable approache. The algorithm mainly consists of two phases: First, a legal flow is established, then the legal flow is turned into a maximal one. Both phases consist of an application of a max–flow algorithm which solve a problem similar to the general max–flow problem but without lower bounds and only a single source and a single target. There exist several major approaches to solve the max–flow problem (augmenting path, preflow–push, Dinits') each of which can be modified itself.

The first phase of the algorithm establishes a certain flow (if there exists a valid flow): It maximizes a flow on a certain modified network where the initial flow is the zero flow. Common descriptions of max–flow algorithm start always with initializing the flow to zero. However, this is not be reasonable for the second phase of the algorithm as this would destroy the existing flow. In addition, some heuristic might be used to establish a flow which is already closer to the final flow than the zero flow.

As already noted, the algorithm solving the max–flow problem modifies the graph to create a single–source/single–target problem. If Dinits' algorithm is used to solve the problem, several subgraphs are created during the execution of the graph e.g. by removing the edges not present in the subgraph. Thus, to make the algorithm work, it is necessary to modify the graph somehow. All these modifications are auxiliary and do not appear in the final result of the algorithm: The auxiliary modification have to be undone at the end of the algorithm or when they are no longer necessary (the auxiliary modifications used to create a legal flow have to be undone before the final maximization is performed).

Like the other algorithms, the max–flow algorithms need to access data associated with the nodes and the edges of the graph. One peculiarity is introduced by the two phases of the general max–flow algorithm where the same algorithm may be applied twice to solve different problems: The exact meaning, and maybe the representation, of the upper bounds for the flow of an edge changes!

The description of the algorithms to solve the max–flow problem above assumes the existence of closely related pairs of edges. This is necessary to solve the max–flow problem. However, if the update of the residual capacity of the reverse edge is made implicit (i.e. if there exist pairs, modifying the residual capacity of one edge automatically updates the residual capacity of the other edge), there is no need for the pairs of edges. Depending on the algorithm used, this can be used solve slightly modified problems (e.g. to determine a blocking flow).

2.3 Graph Representations

This section introduces some possible representations of graphs and discusses some of the advantages and disadvantages of the respective representations. The intent of this discussion is to show that there is no single representation which is suitable for all purposes. This results in the requirement for the implementation of algorithms to be applicable to different representations.

Here, the discussion is centered on the structure of the graph, i.e. how to provide efficient access to the adjacency lists of the nodes. One reason for accessing a node or an edge is to access data such as weights, capacities or distances associated with the corresponding object. Here are only some possibilities of the representation of data associated with the nodes and edges of a graph discussed.

The definition of graph suggests to represent a graph as a set of nodes and a set of edges: Some container, e.g. a **linked list** or an array (see [4] for a description of linked lists and arrays), is used to store the objects representing the nodes and another container, potentially a linked list or an array, stores the objects representing the edges. The edges store references (e.g. indices or pointers) to the nodes they are incident to. If there is no additional property maintained for the elements of the container, for example that the directed edges are ordered such that all edges leaving a certain node are consecutive in the container, this representation is normally insufficient to get an efficient implementation of an algorithm: Often it is necessary to have efficient access to the edges incident to a certain node or to have efficient access to the edge incident to two nodes.

There are many different representations of graphs. Here are some common representations: A graph can be represented as an adjacency matrix, an incidence matrix, a set (i.e. a linked list, an array, etc.) of adjacency lists, or one of various representations which make use of some specialized properties of a graph. For example, a rectangular grid graph can be represented as a matrix, an interval graph can be represented by corresponding intervals, a planar graph may be represented in a way which allows easy access to the dual graph and so on.

The characteristics of the various representations differ in support for efficient accesses and modifications. Depending on the application, different representations are suitable: An application needing fast access to the edge connecting two nodes but making no modifications of the graph will prefer a different representation than an application making many modifications but with weaker requirements on the accesses supported.

General Requirements

Independent of how the structure of the graph is represented, often it is necessary to have access to the set of nodes and the set of edges. The need to access all nodes or all edges arises in many situations, notably to do some initialization: In the algorithms presented above it is assumed that the data associated with the nodes and the edges is initialized appropriately. Thus, an efficient possibility to access all nodes and all edges of a graph is desirable.

If the set of nodes can be accessed and if there is a method to access all edges incident to a node, it is possible to access the set of edges, too: Just access all nodes and for each node access every edge. This might access every edge at most twice such that this way the edges can be accessed in an asymptotically efficient manner. For directed graphs it is easy to make sure that each edge is accessed exactly once: Before accessing the edge, it is checked that the node from which it is accessed is the head (or the tail). In many cases this method also works for undirected graphs,[6] e.g. because the two nodes incident to an edge are stored somehow in the edge which has to be done in a particular order

[6] It does not work for all cases as it is possible that the edges are not really stored. In Section 4.2.1 an example of a graph with implicitly stored edges will be shown.

on a computer: The two nodes can be stored in a structure with two distinct names for the two nodes or in an array with two elements etc. In any case, the two nodes incident to an edge are accessed by two (probably only slightly) different methods. This difference can be used to make sure that each edge is accessed just once.

Let $v \in V$ be a node of a directed graph $G = (V, E)$. Normally, it is necessary to have efficient access to both sets $in(v)$ and $out(v)$. In some implementations of graphs there is apparently some confusion about the meaning "directed": The fact that a graph is directed does not necessarily mean that it is sufficient to provide efficient access to the the set $out(v)$ of a node v. The reason why there is a temptation to provide access to $out(v)$ only, is that the representation is more efficient and the methods modifying and accessing the representation are easier to implement. A simple approach to store a directed edge visible to one node only could be a pointer referencing the adjacent node. If the edge has to be visible from both incident nodes (efficiently) there are at least two pointers necessary plus some information about the direction of the edge. This conforms to the theoretical view where the edges of a directed graph contain more, not less, information than the edges of an undirected graph: In addition to the nodes incident to the edge, the edge contains the information about the direction. As there are some algorithms for which it is sufficient to have access only to the edges leaving a node (e.g. an algorithm determining a topological order of the nodes [16]) such a representation will be presented below.

The algorithm to solve the general max–flow problem modifies the graph: Some auxiliary nodes and edges are added to transform the problem into one which is easier to solve. Auxiliary modifications are often used for graph algorithms. In addition, some algorithms create a graph (a special case of a modification) or transform the graph in some way: There are modifications which are part of the result, too.

In general, it should not be required that a graph representation allows modifications to avoid restrictions for the representations: Certain graph representations are highly efficient for special classes of graph (e.g. the representation of grid graphs presented below) but do not allow modifications or only some very restricted modifications. On the other hand algorithms which apply auxiliary modifications should be applicable to the specialized representations, too. In section 4.2.3 it will be shown how modifications can be applied to a graph even if the actual representation does not allow such modification. It is still desirable to have the possibility to modify the actual representation for two reasons:

1. Making the modifications without modifying the actual representation imposes some additional overhead which is best to be avoided.

2. Sometimes, the modifications made are not just auxiliary modifications but the main output of the algorithm.

A discussion of the exact approach used to allow modifications and the involved advantages and disadvantages of this approach is deferred until section 4.2.3 where it is discussed in depth. For now it is sufficient to remember that it is desirable for a graph representation to allow modification but it is not necessary.

Adjacency Lists

A flexible representation of a graph is called **adjacency lists**: The graph is represented as a container of nodes and for every node a container of edges. The name is derived from a specific choice for the containers of edges: If a linked list [4] is used, every node stores an adjacency list. Actually, the list is

an incidence list as it stores the edges incident to a node and not only a list of the adjacent nodes but this is basically the same if the graph does not contain parallel edges. The distinction is only important if there are parallel edges between two nodes.

This representation allows to store additional data like marks or weights in the respective objects: If it is known what data will be accessed for the nodes and the edges when the data structure for the representation of the graph is created, the corresponding data can be stored in the lists of the nodes and the edges. If this is not known, the representation of additional data becomes more complex: The nodes and edges would need some suitable ID which can be used by some sort of map (like a hash, a search tree, a dynamic table, or an array; see [4]) to identify the data associated with the corresponding object.

There are many choices to implement a representation as a set of adjacency lists. There are several choices for the containers, e.g. singly linked lists, doubly linked lists, arrays, or trees. The choice of the container for the nodes is independent from the choice of the containers for the edges and the containers for the edges can potentially even vary: For some nodes an array is used, for others a linked list etc. Of course, the efficiency and possibility of modifications and accesses depends on the choice of the containers but it is possible to create a representation using a set of adjacency lists which provides efficient access to the set of nodes, the set of edges, the edges incident to a node, and which allows efficient removal and insertion of nodes:

If both the container for the nodes and the containers for the edges are chosen to be doubly linked lists, it is possible to implement modification and access operations efficiently. However, this does not mean that this representation is sufficient for all purposes. One drawback is the size of the representation: For every node and every edge two pointers are necessary to represent the doubly linked list. Additional pointers for every edge are necessary to allow both efficient access to the edges from both nodes and efficient removal of the edge. This overhead can be too expensive for large graphs with only few data stored with the nodes or the edges. Different choices for the containers can reduce the size of the representation at the expense of increased run–time.

The main idea for the representation can be varied in many ways. Using a container for the nodes and for each node a container for incident edges results in a representation with efficient access to the nodes of the graph and the incidence list of each node but without a direct access to the edges of the graph. To avoid this problem and at same time saving the overhead needed for the representation of some lists (often there is some administrative data stored for a list) the incidence lists can be joined to become just one list: The edges incident to a node are stored sequentially in the list and every node stores a reference to it's first incident edge and it's last incident edge.[7] For more information on adjacency lists and variations thereof see [2].

A special case of adjacency lists to represent directed graphs is a representation where each edge only appears in one adjacency list: The edge is basically invisible from one of the nodes. This representation avoids some overhead imposed otherwise by this representation. The drawback of this representation is the fact that it is insufficient for many applications.

The representation as a set of adjacency lists is suitable for most problems and allows efficient modification and access as well as parallel edges and loops. However, there are some algorithms which require accesses being inefficient with a representation as adjacency lists. For example, it is normally inefficient to find an edge incident to two given nodes: This operation involves finding of one of the nodes and an edge incident to the other node in the incidence list of this node. Normally,

[7] To allow empty incident lists for nodes, it is convenient to store a reference to the first edge which is not incident to the node instead of the a reference to the last incident edge: An empty list would be represented by two identical references. This technique is used intensively in STL [19].

this operation is not in $O(1)$ as required by some algorithms to be efficient.

Adjacency Matrix

A directed graph $G = (V, E)$ can be represented as a $|V| \times |V|$ matrix $A = \{a_{ij}\}$ called **adjacency matrix**. The node v_i corresponds to the i-th row and column: The element a_{ij} in the i-th row and the j-th column is 1 if there is an edge (v_i, v_j) and 0 otherwise. For undirected graphs it suffices to store an upper triangular matrix to store the information about edges incident to the nodes if it is made sure that only elements a_{ij} are accessed where $i < j$.

This representation allows efficient insertion and removal of edges: Only the corresponding element has to be changed. Insertion of new nodes involves increasing the size of the matrix to get an unused column and an unused row. Insertion of a nodes it thus, in general, not efficient for an adjacency matrix. It can be efficient, if there is at least one removed node: Removing a node can be done by just ignoring the corresponding row and column which would yield an efficient possibility to remove a node. If a node is inserted after removing some other node, the corresponding row and column can be used for the new node.

While the access to the set of nodes is efficient, the efficiency of access to the set of edges depends on the number of edges present in the graph: Access to the set of edges is only efficient if there are many edges (i.e. $|E| \in \Omega(|V|^2)$) because the matrix has to be searched for edges taking $O(|V|^2)$ time. If there are fewer edges, it is necessary to maintain additional information, e.g. a linked list, to access the set of edges efficiently. Likewise, the efficiency of accesses to the set of edges incident to a node depends on the number of edges. The advantage of this representation is the possibility to access an edge incident to two given nodes in $O(1)$ time: All what has to be done is to access the corresponding element in the matrix.

Planar Graphs

As representative for special representations for graphs with a known property, planar graphs will be used. Planar graphs have some nice properties making it possible to solve some problems efficiently for a planar graph which would otherwise be NP-complete (for example the Max-Cut Problem; see [14, 20]):

- The number of edges is at maximum linear in the number of edges, i.e. the equation $O(|V|) = O(|E|)$ holds for every planar graph $G = (V, E)$.
- For every planar graph $G = (V, E)$ a planar dual graph $G_{dual} = (F, E')$ can be constructed: For every face in G insert a node into G_{dual}. Two nodes in G_{dual} are adjacent, if the corresponding faces in G are adjacent: Thus, for every edge $e \in E$ there is an edge $e' \in E'$.
- For planar graphs an orientation (respective to the embedding) can be used. This orientation is defined by the cyclic order of the edges incident to a node.

To make use of these properties, it is necessary to have efficient access to the necessary information. The corresponding information is normally not present in the general representations for graphs and special data structures are used for planar graph. The use of special data structure is also indicated to maintain planarity if the graph is modified: In a planar graph, insertion of an edge can potentially violate planarity. A special representation for planar graph would refuse the insertion of an edge violating planarity. Thus, insertion of an edge in a planar graph may fail.

A data structure maintaining the planarity and providing efficient access to order of the edges and the dual graph is the **Quad Edge structure** [13]. This structure maintains cyclic lists of edges where every edge is in a total of four lists: As every edge is incident to two nodes and two faces it is stored in two lists storing the edges incident to a node and in two lists storing the edges incident to a face. If the graph is viewed as the dual graph, the roles of nodes and faces are exchanged while the rest remains identical. Thus, a Quad Edge structure store simultaneously a planar graph G and its dual graph G_{dual}. In addition, each cyclic list of edges defines the cyclic order of the edges incident to a node or the circle defining a face (again with reversed meaning in the dual graph).

When inserting an edge, the structure checks whether the incident nodes are laying on a common face: If this is the case, the edge is inserted by splitting the cyclic list defining the common face into two cyclic lists. The new edge is inserted into both lists where they are split. In addition, the edge is inserted into the cyclic lists of edges incident to the nodes: It is inserted between the two edges incident to the face (remember that a face is defined by successive edges in the cyclic orders around the nodes).

If the nodes of the planar graph are to be accessed, it may be necessary to maintain an additional container of nodes: The nodes are only given implicitly by the QuadEdge structure. Likewise, the edges may be also stored in a container to access them directly.

Although this structure can store arbitrary planar graphs, it is not appropriated for all situations where planar graphs are involved: An example are grid graphs used in VLSI design [17] which can be represented much more efficient. A **grid graph** is an undirected graph with all pairs from $\{1, ..., n\} \times \{1, ..., m\}$ as nodes and with edges connecting each node $(i, j), i = 1, ..., n-1, j = 1, ..., m-1$ with the nodes $(i + 1, j)$ and $(i, j + 1)$. An efficient representation of this graph consists of two integers, more precisely the pair (n, m). If the nodes should be represented explicitly, e.g. because some data is associated with the nodes, this can be done using a $n \times m$ matrix. Likewise, the edges can be represented by a $n - 1 \times m$ matrix for the horizontal edges (i.e. edges of the form $\{(i, j), (i + 1, j)\}$), and a $n \times m - 1$ matrix for the vertical edges (i.e. edges of the form $\{(i, j), (i, j + 1)\}$).

The Quad Edge structure and the representation of grid graphs demonstrate specialized representation providing additional information not found in representations of general graphs and a representation storing a graph much more efficiently than the representation of a general graph. Planar graphs are also a typical example for non–simple graphs: The dual graph of planar graph, even if the original graph was simple, is in general not simple. A loop is introduced by a node with degree 1 and a node with degree 2 introduces parallel edges.

Chapter 3

General Approaches

Reuse of software is becoming more important with more complex systems to be implemented: The costs of implementation should be amortized over multiple products. Thus it comes as no surprise that some general technology has evolved over the years which has reusability as its main concern. One of the most promising approaches is object-oriented programming. Since there exists already a wealth of literature covering object-oriented programming, only some of the key issues are introduced in Section 3.1.

Apart from reuse of an implementation there is another issue where reuse can be exercised: The design of an implementation can also partially be reused. There are some recurring patterns found in how implementations are designed. These can be identified and described as "Design Patterns." Section 3.2 features an introduction to the general concept of design patterns.

Finally, Section 3.3 gives a brief discussion of the "Standard Template Library" (STL), which will be part of the Standard C++ Library. This library provides highly generic algorithms for "linear" objects: lists, arrays, sets, etc. There are concepts used to implement this library which are very similar to those proposed in this work. In particular the implementation of the algorithms is made independent of the data structure used.

3.1 Object-Oriented Software Design

The intention of this section is to briefly review the basic concepts of object oriented programming as these are the basics of the concepts presented in the next chapter. A complete discussion of this topic is found in the books of Booch [3] and Rumbough et al. [22] (which also introduce the graphical notation used).

The focus of object–oriented programming is on decomposing the problem into objects: An **object** is an entity with some state, some behavior, and an identity. Objects sharing the same structure and the same behavior are grouped into **classes**. The aim of object–oriented programming is to decompose the problem into cooperating objects. The property new in object–oriented programming is to use objects as the important abstractions and to decompose the problem into object rather than using the traditional algorithmic decomposition.

One of the key features of object–oriented programming is **abstraction**: An abstractions denotes the essential characteristics of an object distinguishing it from all other kinds of objects relative to the perspective of the viewer. By abstraction it is possible to concentrate on the properties of an object important to the system build, i.e. its interface, while ignoring irrelevant details, e.g. the implementation of the object's behaivor.

The details of the objects are hidden from the outside world: They are **encapsulated** by the object's class. Encapsulation serves to separate the contractual interface of an abstraction and its implementation. Proper encapsulations guarantees that an object's state can only be modified through the interface making the behavior of an object predictable. It also allows to change the implementation of a class without affecting the objects using an object of this class. The interface of an object in object–oriented programming consists of a set of **methods** implemented for the class of the object. A method is basically a function which is given special permission to access the internals of an object: In general, only methods of a class can access the state of an object of this class directly. All other functions have to use the methods provided for the corresponding class to use an object.

To make the use of objects and classes more powerful, another concept employed in object–oriented programming are **hierarchies**: A hierarchy is a ranking or an ordering of abstractions. The two most important hierarchies in object–oriented systems are the **class structure** and the **object structure**. The object structure is already known from other programming paradigms where objects are created from other objects by means of **aggregation**: An object may be made up from multiple objects of different classes. These objects become a part of of a bigger object and are said to be **members** of the bigger object.

The class structure orders classes in a fashion which makes it possible to relate objects of different classes by means of **inheritance**: The objects of a class D may share the structure and the behavior of the objets of some other class B and add some additional structure and behavior, i.e. they **inherit** from the class B which is called a **base class** in this context. The class D inheriting properties of some base class B is called a **derived** class. This relation among classes forms an inheritance hierarchy. In general, a class may inherit from multiple classes such that the objects share the structure and behavior of all classes inherited from. By inheritance a form of **polymorphism** is created: Since the structure and the behavior of derived class is shared with the base class, an object of a derived class can be used wherever an object of the base class can be used.

The mechanism used to create this polymorphism is to define the methods of a class to be **virtual functions**. These function can be overridden in a derived class to perform special actions depending on the actual class the method is invoked with. This allows to dynamically decide what is done if a method is invoked. Sometimes it is useful to force derived classes to override the function. In this case a virtual function is called an **abstract function**: Only the semantic of a call to this function is defined by the base class but the implementation to create the desired behavior has to be implemented by a derived class.

Static Polymorphism vs. Dynamic Polymorphism

Some strongly typed object–oriented languages like C++ and Ada 95 provide two styles of polymorphism: inheritance and templates.[1] Both approaches have advantages and disadvantages and it is often argued that one approach is superior to the other. However, as described below, a combination of both approaches yields a mechanism more powerful than any of the approaches used alone.

Normally, the object–oriented approach uses inheritance to realize **dynamic polymorphism**: A method taking a (reference to a) base object can be given a (reference to a) derived object. The exact type of the argument need not be specified at compile–time but can vary while the program is running. Still, the method can invoke functions depending on the type of the argument. This works by defining (probably abstract) functions in the base class which can be overridden by derived classes: These functions are called **virtual functions**. To make this work, reference semantics for the arguments has

[1] The term templates is quite C++ specific. The same concepts is called **generic package** in Ada. However, to avoid confusion, the C++ terminology will be used.

to be used because the type is not known at compile–time. If value semantics is used instead, there would be no dynamic dispatch because the information about the type of the object is lost when the argument is created. In addition, the object of the derived class would be **sliced** e.g. when it is copied, i.e. everything not part of the base class is not present in the copy: Even if the original object were of a derived class, the copy would be an object of the base class and additional information is cut off.

Polymorphism using templates does not depend on inheritance: Any type conforming to the interface required by a templatized implementation of a method can be used to instantiate the corresponding method. This has the disadvantage that it must be fixed at compile–time: The corresponding code has to be generated. That is why this approach is called **static polymorphism**. However, there are also some advantages: The classes used to instantiate the method need not be derived from a base class. This can be used to avoid strongly coupled inheritance hierarchies. In addition, value semantics can be used for templates unless static polymorphism is mixed with dynamic polymorphism:

It turns out that the template approach can be used together with the inheritance based approach: It is easily possible to use a template implementation with a hierarchy of classes. This is done using a simple hierarchy where the abstract base class provides all relevant methods (see Figure 3.1). A reference to an object of a class derived from the base is stored in a handle class, which is used to instantiate the templatized method. An object of the handle class uses this reference to delegate requests to the object referenced.

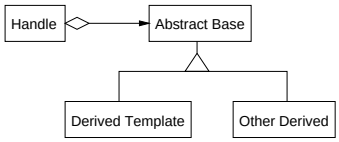


Figure 3.1: Using inheritance with templates

The key idea behind this structure is the fact that in both forms of polymorphism, the interface of the argument to a method has to match exactly: For templates the interface is fulfilled by conformance to the required syntax; for inheritance the interface is fulfilled by inheriting and possibly overriding the required functions. In the hierarchy, the class `Handle` conforming to the syntactic requirements of the template interface is implemented by delegating the requests using virtual functions: The class `Handle` has all methods needed to conform to the interface of the templatized method. The implementation of the methods in the class `Handle` is rather simple because all `Handle` has to do is to delegate the requests to the object it is handling. The object handled by an object of type `Handle` is an object of a class derived from `AbstractBase`. The class `Handle` does not bother about the exact type of the object handled as it only needs the methods declared by `AbstractBase`. There are two approaches depicted to derive from `AbstractBase`. The first, depicted as `OtherDerived`, is the normal manual derivation and there is nothing tricky about it. The other method is more interesting as it demonstrates how to fit a class into a hierarchy: This one uses a template derived from `AbstractBase`, called `DerivedTemplate`. Thus, any class conforming to interface to instantiate `DerivedTemplate` can be used and becomes a part of the hierarchy of classes manageable by a `Handle`. This structure demonstrates the benefit to use both dynamic and static polymorphism together.

3.2 Design Patterns

A relatively new tool to help with the creation of object–oriented systems are **Design Patterns** [12]: A design pattern describes a solution to a recurring problem arising when designing object–oriented systems. To do so, it systematically names, motivates and explains a general design in a language independent manner.

There are four essential parts of a design pattern: The **pattern name**, the **problem description**,

the **solution**, and the **consequences** of the application of a certain pattern. These parts are described below.

The pattern name conveys the essence of the pattern. The name is used as part of the design vocabulary to describe solutions to certain problems. The name of the pattern makes it easier to talk about a design, to document a design, and even to think about a design. Thus, it can be used as an abstraction at a higher level.

The problem description motivates the general problem solved by the design pattern. This defines when to apply the pattern. This part of the pattern describes a general situation which has to be solved. Sometimes a concrete example is used to do so. The problem description might include a list of conditions which must hold before the application of the pattern makes sense.

The solution describes the elements of the pattern together with their interaction, their responsibility, and their relationships. The description of the solution is not dependent on a concrete problem and there is no implementation given which is used to always solve the problem: The exact layout of the objects involved in a pattern depends on the actual problem to be solved. The design pattern only gives a general arrangement of objects and classes involved in the solution of a general problem. This solution has to be tailored and adapted to the problem actually solved.

The consequences section lists the trade–offs made when applying a design pattern. This is used basically to decide whether the approach of the design pattern is feasible to solve a certain problem: There may be other design patterns solving a problem which is quite similar to the problem solved by the design pattern at hand but with different trade–offs. In addition, the costs involved in the application of a design pattern may be too high to solve a certain problem such that a different solution has to be sought. The consequences of design patterns help to evaluate the impact on a system's extensibility, flexibility, and portability but also list the costs and limitations.

The notation of design pattern follows a consistent format which makes it possible to quickly find the relevant details. This format as well as some of the design patterns used in the next chapter are described in the book about design pattern by Gamma et al. [12].

3.3 The C++ Standard Template Library

The Standard Template Library, STL for short, is a library for very generic algorithms working on sequences [19]. Being part of the (prospective) C++ Standard Library, STL was faced with a simple problem: The algorithms should work with any implementation of a sequence (providing sufficient accesses) storing an arbitrary data type. If there are d different data types, s different implementations of sequences, and a different algorithms, apparently $d * s * a$ different implementations are necessary. Fortunately, this amount can be reduced to $s * a$ by assuming that all sequences and all algorithms take the data type as a template argument. However, this is still to much: s is unknown because a user can implement his/her own sequence. The key feature of STL is to reduce the amount of implementations necessary to a algorithms and s interface classes.

The key idea of STL is to access sequences only through iterators: All algorithms work with iterator classes which form the interface to the sequence. Since the interface to the iterators is itself standardized, each algorithm needs only be implemented once. The only thing necessary to apply an STL algorithm to a self–written sequence is to implement a corresponding iterator.

Actually it is not really true that every algorithm is implemented only once: There are several different implementations of some algorithms using different iterators. The reason for this is to get an implementation as efficient with a certain sequence as possible. For example, an array provides random access to all its elements in $O(1)$ time. This is not true for a (normal) list: There, access to

an arbitrary element takes $O(n)$ time (if the list stores n elements). Some algorithms, like a search in a sorted sequence, take advantage of the arbitrary access facilities: For an array, the search can be a binary search while the search in a list is probably linear.[2] To enable different implementations, some iterator categories are defined in STL.

The main idea to provide implementations of algorithms working on arbitrary sequences is to implement the algorithms using an indirection, the iterators, instead of implementing the algorithms directly on the sequences. This way, only the entities used for the indirection have to be adapted to the sequence (once for each sequence) without touching the algorithm. For the implementation of STL algorithms, the attention is focussed on the algorithms only: It is completely irrelevant to the implementation of the algorithm what sequences it is applied to. The implementation imposes certain requirements (which define the iterator category the algorithm gets as argument) and can then be applied to any sequence which fulfills these requirements.

Like the name STL suggests, the algorithms are implemented as template functions, i.e. static polymorphism is used. The functions are normally at least templates in the iterators used. Additional arguments, like are compare functor (i.e. an object which serves to compare two objects of a certain class) or a predicate (i.e. an object which maps some arguments to a boolean value), are also template arguments. This creates very fast algorithms at the expense of increased size of the executable program. Of course, using the class hierarchy shown in Section 3.1, it is possible to apply STL algorithms in cases where dynamic polymorphism is preferred, too.

[2] It makes also sense to apply a binary search in a linear list if comparisons of the elements are expensive compared with iterating over some elements.

Chapter 4

Concepts

The various techniques presented in chapter 3 suggest the general direction how to approach the problem: Object oriented design provides a way to decouple the implementation from its interface. The indirection used by STL shows a convenient way of applying (static) polymorphism to decouple algorithms from the data structures using indirection for the accesses. This chapter presents a strategy how to implement graph algorithms using mainly the techniques from the previous chapter such that the implementation is reusable to a high degree. As far as possible this strategy is formalized into (domain specific) design patterns. Before going into the details of the individual building blocks, a sketch of the overall structure of the concept is given.

4.1 The Big Picture

For a reusable implementation of an algorithm it is necessary to be applicable to different data structures and in (at least slightly) different situations. Also, the behavior of an algorithm can often be modified by using different subalgorithms, different data structures to store additional information, or different conditions to terminate. A truly reusable implementation of an algorithm would allow to modify these parameters in some way instead of using a hard–wired method which appeared to be suitable when the algorithm was implemented. This also makes the implementation easier because there is no need to determine the best choice for all situations (which is impossible in most cases anyway). Instead, only a default may have to be established for the cases where the user of the implementation doesn't want to bother about the parameter.

The strategy given here to address the problems with flexible implementations of algorithms is basically made up of two major parts:

- An indirection is introduced to access the representation. This is used to abstract from the actual representation of the graph and the data associated with the nodes and edges.
- Algorithms are made first class citizens, i.e. they become classes, and the focus of attention is shifted from the data structure used to the algorithms: Instead of implementing algorithms for one specific data structure, algorithms are implemented for all data structures satisfying a (minimal) set of requirements.

The first thing being discussed is the abstraction from the data structure used to represent the graph's structure. There are many possible data structures which might be used to represent the graph. Apart from different representations for general graphs, there are specialized representations

for certain classes of graphs, and the possibility to view data structures as graphs although they were not intended to be viewed as graphs.

The implementation should be applicable to all representations that are theoretically suitable. The trivial approach, i.e. converting the input data structure to some data structure used by the implementation of the algorithm, applying the algorithm and then converting the result back, if necessary, has some disadvantages: This approach is likely to be inefficient, e.g. if the algorithm can perform in sublinear time. In addition the conversion might lose knowledge about the structure of the graph which could be exploited by a specialized subalgorithm otherwise or it might be necessary to maintain a mapping between the two representations to allow a conversion back to the original representation.

Another alternative is to use the common denominator of all suitable graph representations as a base class for the graph representations. This approach also appears to be too restrictive: Not all representations support the same operations, e.g. insertion or removal of an edge is not possible with all graph representation (it might be impossible to add an edge to a planar graph or to remove an edge from an interval graph while still maintaining the intrinsic properties of the respective graphs).

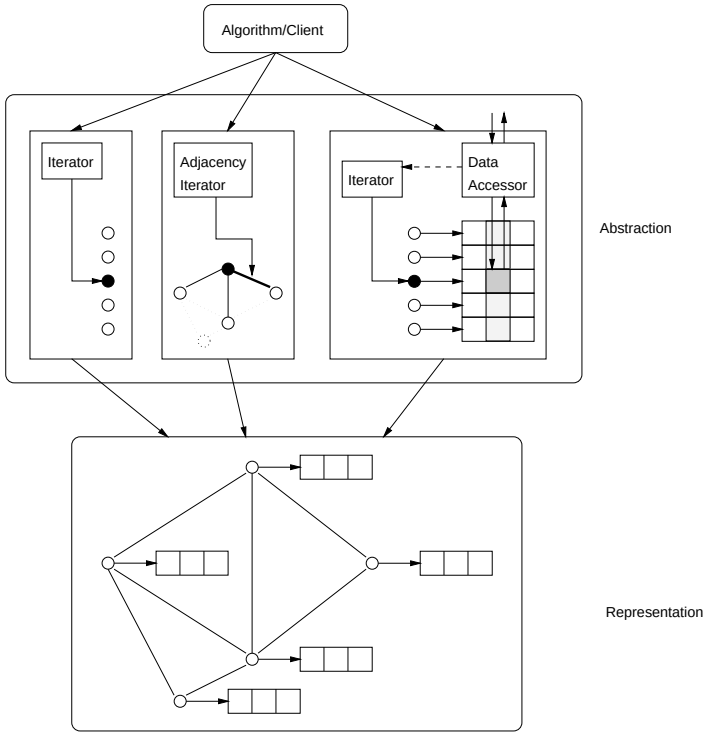


Figure 4.1: Components of the abstraction from the representation

The approach taken here is to separate the abstraction into multiple components which might interact with each other where it makes sense (see Figure 4.1). An implementation of an algorithm just uses the services provided by the components it requires. Here is an overview of the main services provided by the abstraction:

- A specific node from the set of nodes or a specific edge from the set of edges is identified using an **iterator**: An iterator identifies one element from a set and can be advanced to identify another element. Of course, different iterators are used to access the set of nodes and the set of edges.

- Access to the edges incident to some given node v is provided by an **adjacency iterator**: An adjacency iterator is used to explore the neighbourhood of the node v. It can be advanced to identify the next node in the adjacency list of v and the corresponding edge.

- The data associated with the nodes or the edges is accessed using **data accessors**: An iterator identifying the object whose data is to be accessed is passed to a data accessor which in turn extracts or stores one data item associated with the corresponding object.

Other needs for non-mutating accesses are either bound to graphs with a special structure (e.g. the dual edge in planar graphs) or can be created using a combination of the accesses above. However, if necessary or advantageous additional facilities can be required by the algorithm (e.g. because the algorithm makes use of a special structure of the graph) such that only graph representations can be used which fulfill these requirements.

Merely reading the structure of a graph is not always sufficient. Instead, algorithms often need to modify the graph's structure. If the modified structure is part of the result, these algorithms require additional capabilities of the graph representation: If the purpose of the algorithm is to modify the structure, the corresponding operations have to be supported for the representation. Thus the implementation can require some **graph traits** which provide a generalized interface to the necessary mutating operations and whose implementation is tailored to the corresponding representation. If the modifications are only auxiliary, a **decorator** using the normal accesses (iterators, adjacency iterators, and data accessors) and providing the necessary mutating operations can be used which captures the temporary changes and acts like the actual graph representation to the implementation of the algorithm. However, in cases where the original graph representation supports the modifications, it is also possible to omit the intermediate decorator to gain efficiency.

Whatever the requirements imposed on the representation are, the actual accesses are done using an indirection: The iterators, the data accessors, the graph traits, and the decorators. This decouples the implementation from the representation of the graph. However, this is not sufficient to make an implementation of an algorithm as reusable as possible: The behavior of the algorithm itself often has to be augmented and/or modified. An idea leading towards algorithms which are themselves more flexible is to make algorithms first class citizens: Every algorithm becomes a class instead of a function. One advantage is the control provided over the various stages of the life-time of an algorithm: Instead of having a monolithic function which performs the whole task of the algorithm, the behavior can be controlled at least between the different phases of the algorithm without forcing the client to manage the data used by the algorithm. The main phases common to all algorithms are:

- The construction, establishing the general framework in which the algorithm is applied: The initial subalgorithms, data structures, and access methods for the graph representation used are determined. If it is possible that the algorithm is preempted, it might even be possible to replace these components while the algorithm is applied.

- The initialization, specifying the actual problem instance and establishing the necessary invariants of the algorithm. Note that it is not necessary to couple the complete initialization with the implementation of a certain graph algorithm: Instead, it is possible to factor out the initialization into a separate **initializer** algorithm, being usable as initialization for different algorithms with similar preconditions.

- The core of the algorithm, solving the problem at hand. This phase may be further subdivided into the sequential application of several different subalgorithm if it is possible to subdivide the algorithm into **algorithm steps** and/or into equal sequential steps resulting in a **loop kernel**.

- If necessary or advantageous some post processing is done after the main part of the algorithm. The same post processing might be used for different algorithms like the same initialization can be used for different algorithms.

The two main techniques used to obtain flexible implementations are abstraction from the data structures using an indirection and implementing algorithms as objects. Using these two techniques, it is possible to focus on the needs of the algorithm, while ignoring the details of used subalgorithms and data structures. This makes it easy to replace a subalgorithm or a data structure used: The algorithm depends neither on a certain data structure nor on a certain subalgorithm. Instead, it only depends on an intermediate interface which can be much more general than the interface of a data structure or an algorithm. In addition, both techniques have additional effects, like preemptive algorithms and implicitly stored data, which are explained in this chapter along the detailed description of these techniques.

Before going on to a more detailed description of the techniques proposed, some goals achieved by an implementation of a graph algorithm using the proposed techniques are listed.[1] This list includes brief comments how the respective goal is achieved:

- The implementation of an algorithm is based on a graph abstraction which can be realized by potentially all graph representations that are suitable for this algorithm. In particular it is not assumed that a certain representation (e.g. an adjacency list or an adjacency matrix) is used. Such an assumption would hard–wire the graph data structure or at least seriously limit the choices of the representation used for a graph to which the algorithm is to be applied. If specific characteristics of the representation are necessary for an algorithm to perform correctly or efficiently, these should be required in a fashion which does not determine the representation and which is usable in other situations too.

- Data associated with the nodes or the edges is accessed through an indirection. This allows to choose a representation which is suitable for a specific application. For example, in one application it may be impossible or unnecessary to store all data associated with the edges because the graph is too large and the data can be calculated from data associated with the nodes (e.g. the "length" of an edge may the Eucledian distance between the nodes). In another situation, it may be appropriate to store the data with the edges because it is impossible or too expensive to calculate the data. However, both representation as well as many others can be chosen.

- It is possible to augment the behavior of the algorithm by additional actions performed during the algorithm. This can be used to compute additional data depending on the state of the algorithm. For example, the biconnectivity algorithm (see 2.5 on page 19) uses the state of the graph search to compute the low–point numbers.

- Due to the separation of initialization from the main algorithm, it is possible to establish only the preconditions used and not yet established. This allows the reuse of an already established state and to violate assumptions in a controlled fashion: e.g. the augmenting–path algorithm (see chapter 2.8 on page 23) to solve the maximum flow problem does not make any use of the fact that it is modifying a valid flow. It maintains the balances between the incoming and outgoing flows of the nodes independent of a valid flow. There is no point in forcing the flow

[1] Most of the goals apply to different problem domains directly or in a modified variant too. However, this work is focused on graph algorithms such that this restriction seemed to be suitable and made the description easier.

to be a valid one. If an invalid flow is suitable for the application, such a flow can also be maximized by the algorithm.

- During construction of an algorithm the data structures used are established. This can be used to keep the data structures used to store auxiliary data flexible e.g. to allow modifications of the behavior or to use data structures better suited to the data actually stored than some default choice.

- Used subalgorithms are exchangeable. This is e.g. advantageous if the algorithm is applied to a graph with a known structure which can be used by a specialized version of the subalgorithm.

The approaches used create some additional overhead due to additional function calls or additional indirections. Whether this additional overhead is acceptable depends on the alternatives and the overhead really imposed.

4.2 Abstraction from the Representation

The primary purpose of the abstraction is to allow support for different representations. The aim is to have just one implementation of an algorithm being applicable to different representations. There is another use of the abstraction involved: It is possible to wrap the abstraction (or parts of the abstraction) into another layer providing a different view of the same object. For example a subgraph can be modeled by the access facilities of the actual representation and a boolean mark for every node and/or edge indicating whether it is part of the subgraph: A decorator for the original representation checks the boolean mark and simply ignores the nodes and edges which are not part of the subgraph. Another example would be a decorator which (virtually) adds some edges to the representation e.g. by storing additional information for every node indicating whether there are additional edges (and which edges). When asked for the adjacent edges of a node the decorator returns the “normal” edges plus the ones stored with the node. An example of this technique is shown in section 4.2.3.

To make both the actual abstraction and the decorators as flexible as possible, the abstraction is divided into several more or less related components:

- node iterators for a “set view” of the nodes
- edge iterators for a “set view” of the edges
- adjacency iterators for a “structural view” of the graph
- data accessors to access data associated with nodes or edges
- mutators to allow modification of the structure

Iterators are used to sequentially access all objects from a set. For example a node iterator is used to access the nodes of a graph. The iterator identifies one element from the set and can be moved to the next element until all elements have been seen. The operations possible on the specific iterator are very limited: Apart from advancing the iterator there may be a possibility to convert a given iterator into another iterator (e.g. a node iterator into an adjacency iterator for the indicated node) or a possibility to get another iterator (e.g. an adjacency iterator for the current adjacent node of an adjacency iterator). Algorithms do not require the iterator to provide direct access to data associated with the identified object, although an iterator might be implemented in a way which immediately

allows access to associated data. Instead, data accessors are used by the algorithms for accesses to associated data: A data accessor uses a given iterator to provide access to some data associated with the indicated object. The data accessor "knows" how the association between the object identified by the iterator and the required data has to be handled and also "knows" how the required data is represented: The implementation of the algorithm can completely ignore these details of the representation.

Algorithms modifying a graph can do so using a **builder** [12]: The builder is an object which supports the modifying operations used by the algorithm. The builder itself delegates the modification request to the graph representations. In doing so, it can perform additional processing or it can delay modifications, e.g. to avoid temporary violations of a certain property of a specialized graph representations. In addition, this way the interface of different representations can be unified: The builder maps the used interface to the respective interface used by the representation.

This approach provides some flexibility but does not help if the representation does not allow the required modifications. In this case, a **modifier** can be used: The access methods provided by the representation are used to provide a modified view to the algorithm: The algorithm gets passed objects conforming to the same requirements as the original object but which behave (slightly) differently to take the applied modifications into account. For example a modifier can consist of an adjacency iterator which just skips edges registered with the modifier as deleted edges.

To allow a maximum of flexibility for the representation, an algorithm should only require the necessary components: For an algorithm with low demands a simple data structure providing only some very specific accesses might suffice for the representation. Algorithms with a lot of requirements will need more complex, more specialized accesses to the graph which might impose some restrictions on the representation.

4.2.1 Adjacency Iterator

Intent

Make the access to different graph data structures uniform.

Motivation

Section 2.3 beginning on page 26 introduced several fairly different computer representations of the structure of graphs. It is desirable to apply an implementation of an algorithm to all or at least most graph representations, including existing representations provided by some packages.

As an example for typical access to a graph's structure a simple graph search, e.g. BFS (see Section 2.2 on page 13), can be used: The search operates by finding all adjacent nodes not yet labeled from the current node. Thus, BFS needs to access the adjacency list of a given node. If the graph is represented using adjacency lists, an iterator passing this list is used. If the graph is represented using an adjacency matrix an index for every row of the column corresponding to the nodes is used. For a grid graph the direction indices $North, South, East,$ and $West$ are used. And so on. The point here is: All those graph data structures have their own interface to the structure of the graph. An implementation of an algorithm should be free from the details how the structure is accessed to make it applicable to all structures.

To be able to apply an implementation of an algorithm to all these different representations, an indirection is introduced, the **adjacency iterator**, which provides access to the adjacency list of the used representation: The structure of the graph. While the adjacency iterator only provides (direct) access to the structure, Data Accessors (see 4.2.2) are used to access data which is associated with the objects present in this structure, i.e. the nodes and the edges.

An individual adjacency iterator is used to explore the neighbourhood of a node, i.e. the nodes adjacent to this node. First of all, it is associated with a node. In addition, it is in one of two states:

valid	it identifies a current adjacent node and optionally a current edge
invalid	there is no current node (and thus no current edge) to be identified

The valid state of an adjacency iterator indicates that there is still some structure to explore. Correspondingly, the invalid state indicates that the whole neighbourhood of the node has been examined. An adjacency iterator is actually an iterator with special support for access to the adjacent node: Basically, it is used to traverse the set of adjacent nodes and, implicitly, the incident edges.

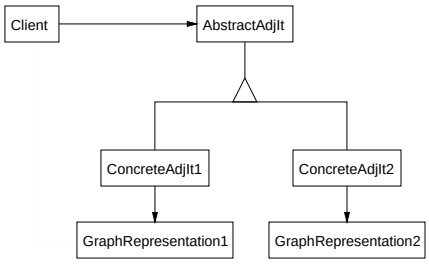
Instead of defining a standard interface to all graph data structures, only the interface for the adjacency iterator is defined. Since the adjacency iterator is an entity of its own, it can be added to existing graph representations implementing it in terms of the corresponding representation. Thus, using an adjacency iterator decouples the algorithm from the details of the graph representation. In fact, an adjacency iterator allows to view entities as a graph which were not intended to be accessed as graphs when they were designed and implemented: It must be possible to implement an adjacency iterator for the data structure.

Applicability

Use an adjacency iterator when

- you want an implementation of an algorithm to be applicable to a variety of different graph representations.
- you want to provide a restricted view of a graph's structure to an algorithm: You can use a decorator (see [12] for a description of the pattern **decorator** for an example) to hide parts of the structure.
- you want to temporarily augment the structure of the graph. Again, this can be done by a decorator (see 4.2.3 for an example).

Structure



Participants

- **AbstractAdjIt**
  - declares the interface to access the adjacency list of a node.
- **ConcreteAdjIt**

  - identifies a node v whose neighbourhood, i.e. the nodes in $adj(v)$, is investigated.
  - defines an access method for the adjacency list $adj(v)$ for node v.
  - identifies a current node $w \in adj(v)$ adjacent to v.
  - provides a mechanism to create a new ConcreteAdjIt to access the adjacency list of the current node w.
  - implements the interface of AbstractAdjIt.

- **GraphRepresentation**
  - represents the graph accessed using ConcreteAdjIt
  - might be some data structure which does not represent a graph but which is interpreted as a graph.
  - is used by ConcreteAdjIt to implement the interface defined by AbstractAdjIt.

- **Client**
  - uses AbstractAdjIt to access the structure of a graph.

Collaborations

- A ConcreteAdjIt uses entities from the corresponding GraphRepresentation for the representation of its state. The operations on a ConcreteAdjIt are partially delegated to operations on entities of the GraphRepresentation.
- A ConcreteAdjIt is used to identify a node or an edge incident to a certain node. It is used together with another abstraction, e.g. a data accessor or the graph representation, to perform operations on the node or the edge identified.

Consequences

Here are some of the consequences when using a Adjacency Iterator:

- The adjacency list $adj(v)$ of a node v is accessed through the adjacency iterator. Thus, there is no knowledge about the representation of the adjacency list coded into the implementation of an algorithm: There is only one implementation of an algorithm necessary to work on all graph representations which support an adjacency iterator.
- Because the graph is accessed exclusively through the adjacency iterator, there is no access to the graph as a whole, e.g. to determine the number of nodes in the graph. If such access is necessary, a different abstraction has to be used to support access not provided by the adjacency iterator. Such a need might impose serious restrictions on the possibility to virtually modify the structure of the graph using decorators.
- The algorithm does not work immediately on the representation. Instead, the adjacency iterator introduces an indirection. This additional indirection might cause some overhead resulting in a less efficient implementation of an algorithm. However, if the representation is designed to support adjacency iterators, the performance penalty should be acceptable.
- It has to be ensured that the entity referred to by the adjacency iterator is still existent when it is accessed. This is due to the separation of the interface and the representation: The object constituting the interface can still exist although the representation is already destructed.

Sample Code

As an example, a linear container representing a dictionary of words is interpreted as a graph: The words of the dictionary are the nodes. Between two nodes there should be an edge if the two words differ in just one position. As a container, a list of strings, i.e. a `list<string>` in C++ notation, is used.

```
#include <list>
#include <string>
#include <algorithm>

int distance(string const &str1, string const &str2)
{
  int dist = 0;

  if (str1.length() == str2.length())
  {
    string::const_iterator s1  = str1.begin();
    string::const_iterator s2  = str2.begin();
    string::const_iterator end = str1.end();

    for (; s1 != end; ++s1, ++s2)
      if (*s1 != *s2)
        ++dist;
  }
  else
    dist = -1;

  return dist;
}
```

The first thing is to make necessary declarations known by including the corresponding header files: `list` for the declaration of the list class, `string` for declarations of the string class, and `algorithm` for the declaration of the copy and find algorithms. Next, a function is defined which computes the Hamming–distance of two strings: The Hamming–distance of two strings with the same length is the number of positions where the strings differ. Thus, there is an edge between two nodes if the distance between the corresponding strings is one. The function `distance()` is used to determine the next node to be considered adjacent in the adjacency iterator `AdjIt`:

```
class AdjIt
{
  typedef list<string>::const_iterator NodeIt;
private:
  NodeIt const begin, end; // the list of words
  NodeIt const node;       // the node
  NodeIt       adj;        // the current adjacent node
```

```
  void find_adjacent()
  {
    while (adj != end && distance(*node, *adj) != 1)
      ++adj;
  }
public:
  AdjIt(NodeIt b, NodeIt e, NodeIt n):
    begin(b),
    end(e),
    node(n),
    adj(b)
  {
    find_adjacent();
  }

  bool   valid() const   { return adj != end; }
  string value() const   { return *node; }
  AdjIt  cur_adj() const { return AdjIt(begin, end, adj); }
  AdjIt  &operator++()   { ++adj;find_adjacent();return *this;}
};
```

The graph representation is in this case a normal linear list containing the nodes. Whether two nodes are adjacent is determined from the values of the nodes: There is an edge between two nodes if a certain condition is true, in this case if the Hamming–distance between the strings corresponding to the nodes is one. The same scheme can be used for arbitrary binary predicates: If the predicate evaluates to true for two given nodes, there is an edge between these two nodes. The adjacency iterator is implemented straight forward: The representation consists of two iterators `begin` and `end` for the start and the end of the list, as is convention for STL, the iterator `node` identifying the node whose neighbourhood is searched, and the iterator `adj` identifying the current adjacent node. The adjacency iterator is invalid, if the current adjacent node is equal to the end of the list. Figure 4.2 shows how an `AdjIt` is represented with the list and nodes used in the example below: The adjacency iterator searches the neighbourhood of the node `root` pointed to by the member `node`. The neighbours of a node are the nodes labeled with a word differing in only one position from `root`, i.e. the nodes labeled with a word having Hamming–distance one from the word `root`. The current adjacent node, identified by the member `adj` is `foot`. Moving the adjacency iterator to the next adjacent node, searches the list between `adj` and `end` for the next word differing in just one position from `root`. This would result in `adj` pointing to `roof` in this case.

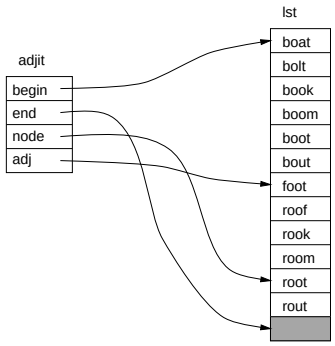


Figure 4.2: Representation of an `AdjIt`

The constructor of the adjacency iterator makes sure that the members are in a consistent state: All but the member `adj` are initialized from the arguments passed to the constructor. The member `adj` is initialized with an iterator identifying the first node in the list having Hamming–distance one from

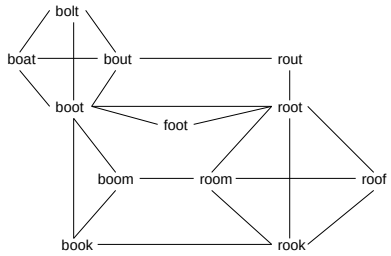


Figure 4.3: The example graph

the adjacency iterator's node. This node is searched using the function `find_adjacent()` which is also used by the increment operator to move the adjacency iterator to the next adjacent node: This function searches through the list until a node `n` is found for which `distance(node, n)` yields one or the end of the list is reached.

All other functions are just used to query the state of the adjacency iterator: `valid()` indicates whether there is a current adjacent node. If there is a current adjacent node, the function `cur_adj()` can be used to obtain an adjacency iterator positioned on the current adjacent node. The function `value()` is not part of the required interface for adjacency iterators: It returns the string associated with the node. Normally, such data is retrieved using a Data Accessor (see 4.2.2). However, this function is convenient in a small program demonstrating the use of an adjacency iterator to access all nodes adjacent to one given node:

```
int main()
{
  char const   *init[] = { "boat", "bolt", "book",
                           "boom", "boot", "bout",
                           "foot", "roof", "rook",
                           "room", "root", "rout"};
  int  size = sizeof(init) / sizeof(char const *);

  typedef list<string> List;
  List lst;

  copy(init, init + size, back_inserter(lst));

  List::iterator root = find(lst.begin(), lst.end(), "root");
  AdjIt adjit(lst.begin(), lst.end(), root);

  for (; adjit.valid(); ++adjit)
    cout << adjit.cur_adj().value() << " ";
  cout << endl;

  return 0;
}
```

The example program first generates a list containing some words. The words used are stored in an array of string literals. These strings are put into a list using the STL algorithm `copy()`. This list is interpreted as a graph using the adjacency iterator class `AdjIt`. The corresponding graph is depicted in Figure 4.3. The STL algorithm `find()` is used to locate the element representing the string "root" within this list. This element is used as the node associated with the adjacency iterator `root`. Finally, all nodes adjacent to the node `root` are accessed with the help of the adjacency iterator. The strings associated with the corresponding nodes are printed: This prints all strings from the list having a Hamming-distance of one from the string "root". Thus, the output of the program is:

```
boot foot roof rook room rout
```

This is a somewhat artificial application of an adjacency iterator. More realistic applications of adjacency iterators require some algorithm which would take to much space to be presented here.

4.2.2 Data Accessor

Intent

Make the use of data independent from the data's representation.

Motivation

Algorithms, including algorithms on graphs, often perform some operations working on or depending on some data with multiple possible representations: The data manipulated can be calculated when needed, it can be stored as needed by the algorithm or, to be more efficient, in a more compact form, etc. In addition, for sets of objects like the nodes or edges of a graph, there has to be decided where the data is stored: As part of each node structure or each edge structure, in a container separated from the graph and indexed by nodes or edges, and so on. Even with one given representation, there are some choices: Consider an edge where some data is immediately stored with the representation of the edge. To access the data, it is necessary to name it. However, different algorithms can have different names for (or interpretations of) the same data: What is a length for one algorithm might be interpreted as a weight or a capacity by another algorithm.

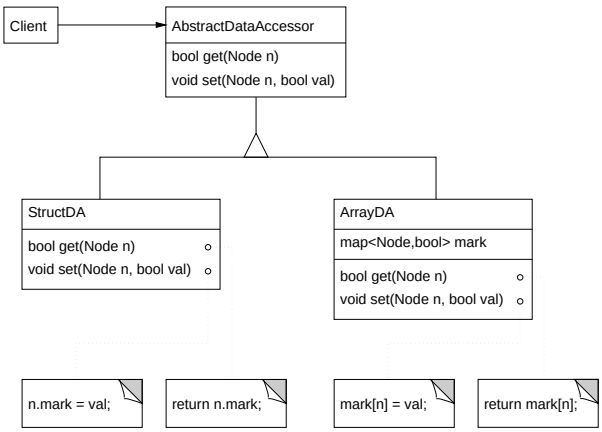


Figure 4.4: Two styles to store some data

A typical use of data associated with the nodes of a graph, is to mark nodes with a boolean label. This is e.g. used to indicate that a node has been processed in a graph search. There are multiple possibilities to represent the mark for a graph. One method is to reserve a member of the node's struct, say `mark`, to be used to mark the nodes.[2] Another approach is to use an external array or map, indexed by the nodes, to store the mark. However, the basic accesses made to either representation are very simple: The current value of the mark of a node is retrieved or a new value is stored for some node. Thus, there are just two operations, `get()` and `set()`, which can be bundled together into an object describing the method used to access the data: A **data accessor** is used for the accesses.

Using the structure in Figure 5.4. it is possible for the client to ignore the details of how the data is represented: The client uses the interface defined by `AbstractDataAccessor` to set and get the mark associated with a node. Two possible representations are shown: If a `StructDA` is used, the mark is directly stored in the node, in the depicted situation in the member `mark` of a `Node`. The number of marks possible with this approach is limited by the number of members reserved to be used as marks in the structure `Node`. Another possibility is to use an `ArrayDA`, which uses a map to associate a boolean label with the nodes. Obviously, there can be multiple instances of the `ArrayDA` allowing as many marks as needed.

These two data accessors only store or retrieve the mark. It would also be easy to provide data accessors which do operation counting, update another data structure, or do some animation. Still, all this would be transparent to the client.

The use of data accessors is not limited to graphs: They are useful whenever some data is to be associated with the members of a set. In the case of graphs the relevant sets are the nodes and the edges.

Applicability

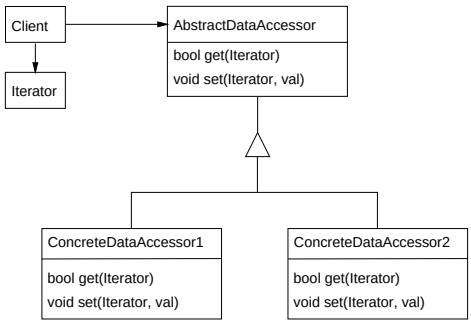
Use a Data Accessor when

- different representations of the data are reasonable. In this case, the implementation should not be limited to use just one representation. For example, the augmenting path algorithm (see 2.8) uses the residual capacities of the edges, which can be represented directly or as the difference between the total capacity and the current flow.
- different names are used for the same data in different algorithms: If the data would be accessed directly, e.g. as the member of a struct, the algorithms (actually the people who implement the algorithms) would have to agree on a common name to use.
- multiple instances of similar data, e.g. multiple marks, are needed. This applies especially in contexts where algorithms are to be combined in a flexible way such that an algorithm might be used multiple times on the same structure.
- the data accessed is not stored as it is viewed by the algorithm: The actual data can be calculated from other data yielding the possibility to store the data only implicitly. For example the distance between to adjacent nodes can be stored with the edge but it can also be calculated from the coordinates of the two nodes, if the distance is the Eucledian distance between the nodes.

[2] It is immediately obvious that one mark is not sufficient for all purposes: More than one mark is necessary. Thus, the name of the struct's member has to be varied or an array has to be used. This is just another example of why the name might change although the interpretation remains the same.

- accesses to the data need additional processing like accounting, operation counting, some sort of animation, or updates of other data structures.
- the data accessed is stored in a data base, should be persistent, or is accessed through some cache mechanism: Processing such data often needs special procedures to access the data which can be implemented by the data accessor.

Structure



Participants

- Iterator
  - Identifies an object to be accessed. The important part is hereby to provide enough information to the ConcreteDataAccessor to locate the data associated with the object.
  - The iterator is not necessarily a “real” iterator but can be any other object identifier, like an index into an array.
- Client
  - Uses an iterator to identify an object.
  - Wants to access (store or retrieve) data associated with the object identified by the iterator.
- AbstractDataAccessor
  - AbstractDataAccessor defines the interface how to access the data: The type(s) of the supported iterator(s), the type of the data returned from `get()` and passed to `set()`, and the access functions.
  - AbstractDataAccessor is either a base class with pure virtual functions such that dynamic polymorphism has to be used to define a concrete data accessor or a set of requirements such that static polymorphism can be used.
- ConcreteDataAccessor
  - Implements the interface defined by AbstractDataAccessor.
  - Given an iterator, the class ConcreteDataAccessor “knows” how to access the associated data.
  - Encapsulates the details of the access.

Collaborations

The client passes an iterator to the data accessor. The iterator provides enough information to the data accessor such the data accessed can be determined. Depending on the used data accessor this can e.g. mean that the iterator provides a pointer to a structure, which is used by the data accessor just to selects a member. Another data accessor might retrieve a unique integer identifying the object whose associated data is to be accessed from the iterator. This integer can then be used to lookup the accessed data in a map.

Consequences

Here are some of the consequences when using a Data Accessor:

- The data accessor introduces an indirection which is otherwise not present. Depending on the implementation this can result in a performance penalty.
- It is possible that an iterator referring to one set is passed to a data accessor used to access the data associated with the objects from another set. This would lead to inconsistencies or serious program errors.
- An algorithm need not to care about the details of the representation of used data. It can evan be applied to different representations without being changed.
- It becomes possible to do additional actions like some animation or access counting by just modifying the data accessor: It is completely transparent to the algorithm that this is done.

Implementation

- Boolean marks are used for many purposes. After each use, the mark has to be reset to an initial value $init$ before it can be reused. In cases where an algorithm normally explores only a part of the whole graph, e.g. a path search algorithm, it may be useful to avoid touching all nodes or all edges. Instead of explicitly initializing the data, a "versioning" mechanism can be used: The marks are not represented by a boolean value but rather by an integer. An integer value $current$ not used before for that purpose marks the nodes which are already explicitly labeled since the last resetting and which have the opposite value of the intial value $init$. When the marks are reset, this is done simply by using a new value for $current$: $current$ is replaced by $current+1$ and in addition a new value for $init$ is saved which might be different from the previous initial value. This technique can be extended to arbitrary data: In this case a combination of an integer used for the ageing and the actual data is stored: The data is valid, if the stored age corresponds to the current age. Otherwise, the value is some old data and the initial value will be returned for retrieving operations.
- If the data stored is not a simple value like an integer or a boolean value, but rather a structure consisting of multiple fields, it is useful to distinguish between data accessors really storing the data somehow, called **lvalue accessors** henceforth, and data accessors returning a calculated value called **rvalue accessors**: Modifications to only a part of the structure need not touch the whole structure. However, to modify just a part of a structure, direct access to the representation has to be provided. This access cannot be provided for rvalue accessors because there is no representation which can directly be modified: The whole structure has to be retrieved to modify only the part without loosing the remaining data and the whole structure has to be stored after

the modification. For lvalue accessors this is not necessary: It would be possible to store only the modified part without touching the other members of the structure. Thus, the distinction can be used to gain efficiency.

Sample Code

The sample data accessor merely adds the possibility to associate data with strings: The data accessor simply stores the data associated with a string in a map and retrieves it from there. Actually, this is precisely the behavior of a map. The key difference is that the user of the data accessor is not aware of the use of a map. Some other data structure, e.g. a hash, could also be used.

The core of the data accessor is object containing the map: The data accessor object, in this case `StringDA`. In this implementation, the type of the data stored is left as template argument. Apart from the the map storing the data associated with the strings called `i_data`, the object also stores a default value for strings without associated data: `i_def`.

```
#include <string>
#include <list>
#include <map>
#include <algorithm>

template <class T>
class StringDA
{
private:
  typedef map<string, T, less<string> > DAMap;

  DAMap i_data;
  T     i_def;

public:
  typedef T value_type;
  StringDA(T const &d): i_def(d) {}

  DAMap const &data() const { return i_data; }
  DAMap       &data()       { return i_data; }
  T     const &def() const  { return i_def; }
};
```

The only member function which should really be part of the data accessor class is the constructor: It takes the default value as argument used to initialize the member `i_def`. All other member function are necessary to work around insufficiencies of the current compiler: The two template functions `get()` and `set()` defined below, should actually be member functions. Unfortunately, there are only few compilers supporting member templates which would be necessary... Hence, the class `StringDA` provides access to its internal representation through some member functions. These are used by the functions `get()` and `set()` to retrieve or store data for a string defined by some iterator:

```
template <class T, class Iterator>
T get(StringDA<T> const &da, Iterator const &it)
```

```
{
  map<string, T, less<string> >::const_iterator mit;
  mit = da.data().find(*it);
  return mit == da.data().end()? da.def(): (*mit).second;
}

template <class T, class Iterator>
void set(StringDA<T> &da, Iterator const &it, T const &val)
{
  da.data()[*it] = val;
}
```

This data accessor can basically be used to store some data for the nodes of the graph from the sample code for adjacency iterators: In this example, the nodes are uniquely identified by a string. The only modification necessary would be to use the member function 'value()' of the adjacency iterators to obtain the string of the node identified by the iterator. This sample code does not use this member to allow a simple demonstration. The following program demonstrates a simple application of this data accessor: The set of strings identified by two iterators and a data accessor is passed to the function `process()`. Each string stored in the list is associated with a value calculated from the string. After this is done, the value is retrieved and printed.

```
template <class Iterator, class DA>
void process(Iterator const &beg, Iterator const &end, DA &da)
{
  for (Iterator it = beg; it != end; ++it)
    set(da, it, (*it)[0] + 256 * (*it)[3]);

  for (Iterator it = beg; it != end; ++it)
    cout << *it << ": " << get(da, it) << endl;
}

int main()
{
  char const   *init[] = { "boat", "bolt", "book",
                           "boom", "boot", "bout",
                           "foot", "roof", "rook",
                           "room", "root", "rout"};
  unsigned int size = sizeof(init) / sizeof(char const *);

  typedef list<string> List;
  List          lst;
  StringDA<int> da(0);

  copy(init, init + size, back_inserter(lst));

  process(lst.begin(), lst.end(), da);
```

```
  return 0;
}
```

The same effect can be obtained with a much simpler data accessor, too: If the `set()` function does not do anything and the `get()` function calculates the corresponding value, the behavior would be the same. Below is a corresponding data accessor. Of course, this data accessor would not be suitable for any other purposes.

```
class SimpleDA {};

template <class Iterator>
int get(SimpleDA const &, Iterator const &it)
{
  return (*it)[0] + 256 * (*it)[3];
}

template <class Iterator>
void set(SimpleDA const &, Iterator const &, int) {}
```

If an object of this data accessor is passed to the function `process()`, the data stored for each string will be ignored. Instead, the data is calculated immediately from the string: This is an example for implicitly stored data.

4.2.3 Modifier

Intent

Allow auxiliary modifications of a graph's structure without really changing the graph.

Motivation

To get a problem instance passed to an algorithm into some normalized form, algorithms often apply some auxiliary transformations to the input. For example, algorithms solving a multi-source/multi-target maximum flow problem add a "super source" and a "super target" to the problem instance (see 2.6). These two new nodes are then connected with the sources and targets of the original input resulting in a single source/single target maximum flow problem which can be solved with the corresponding algorithms. When the solution is computed, the auxiliary super source and super target are removed to get a solution to the original problem.

Although this approach causes no problems in theory it does so in practice:

- The required operations for the transformations are not possible with some graph representations. For example it is impossible to add a node or an edge to a grid graph unless there is special support to add some nodes and edges to the grid violating the property that the graph forms a grid.
- It is necessary to keep track of the modifications: Otherwise the auxiliary modifications cannot be undone afterwards and would be part of the result. Depending on the modifications made, this can be rather complex. For example when parallel edges are removed if an edge is contracted, the implementation has to keep track of the removed edges to insert them after

the algorithm is done. It can be impossible to save the attributes associated with an edge when removing it, because the algorithm does not necessarily know about all attributes.

A completely different approach to do auxiliary modification is to apply the modifications to a decorator [12] of the original graph: The decorator provides a modified view of the graph to the algorithm. The use of the decorator is transparent to the algorithm.

Consider the problem noted above: A multi source/multi target flow problem on a grid graph is to be solved. The algorithms to solve this problem convert the problem into a single source/single target problem by adding a super source and a super target. The resulting problem is then solved with a corresponding algorithm. However, a grid graph has no possibility to add two additional nodes and the corresponding edges. [3] There is in fact no possibility to change the structure of a grid graph directly. However, there is also no real need to modify the structure: It is sufficient if the abstract structure seen by the algorithm *appears* to be modified. This view is provided by a **modifier** (see Figure 4.5).

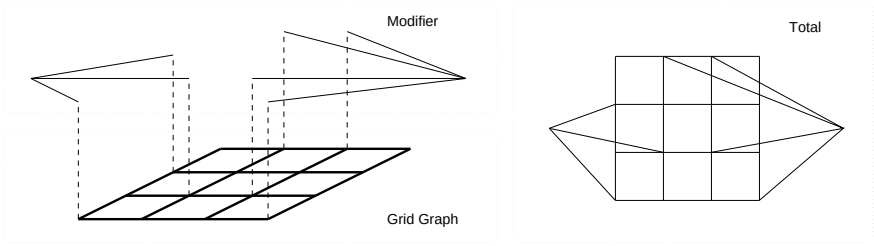


Figure 4.5: A decorator modifying the view of a grid graph

A modifier provides two things: An interface to do some modifications to a graph structure and a new view to the structure of a graph. A modifier combines some aspects of a decorator which provides a different view to an object and some aspects of an adapter which provides a new interface to a given class. It operates by storing information about the modifications and controlling the original interface to the graph's structure based on this information. This is done by providing a complete new interface made up of data accessors and adjacency iterators. The description how this work uses the term "original" to refer to something being part of the interface of the input and the term "adapted" to refer to something being part of the interface provided by the modifier:

In the multi source/multi target example, an adapted adjacency iterator can either be positioned on an inserted node or on a node in the original graph. If it is positioned on an inserted node, all edges incident to this node are not present in the original graph and are thus also inserted. In this case, the adapted adjacency iterator iterates over an adjacency list maintained by the modifier for the two new nodes. If an adapted iterator is positioned on a node which was present in the original graph, it uses an original adjacency iterator to access all edges present in the original graph and an additional partial adjacency list maintained by the modifier for the inserted edges. Such a list is maintained for every node incident to an inserted edge. Accessing the edges incident to a node present in the original graph can e.g. be done by first accessing the inserted edges using the modifier's adjacency list for the corresponding node and then using the original adjacency iterator to access the original edges.

When adding a super source and a super target to a graph, the number of nodes inserted is limited by two. Thus, in the above case the modifier is quite simple: It always has two new nodes, and

[3] Of course, it would be possible to augment the grid graph by implementing a feature for the grid graph which allows certain modifications. Since such a feature is specific to the grid graph and might be insufficient for certain application, this approach is not reasonable.

every node present in the input is incident to at most two inserted edges. Thus, the modifier can be implemented using two boolean marks for the nodes and two lists: One mark indicates whether the node is connected to the super source, the other mark indicates whether it is connected to the super target. For each of the two new nodes, a list is used to store original adjacency iterators identifying the nodes which are adjacent to the corresponding new node.

This nearly suffices to provide a new view to the graph. The only remaining problem is to store and retrieve data associated with the nodes and the edges. This is done by providing adapted data accessors, which use original data accessors to access data associated with the original nodes and edges. For the new objects, the data is maintained by the adapted data accessor e.g. in a map. To do this efficiently, the adapted adjacency iterator collaborates with the new data accessor by telling them whether the identified object is a new one. If data associated with a new object is accessed, the adapted adjacency iterator also provides a unique ID which, can be used to find the data stored in the map.

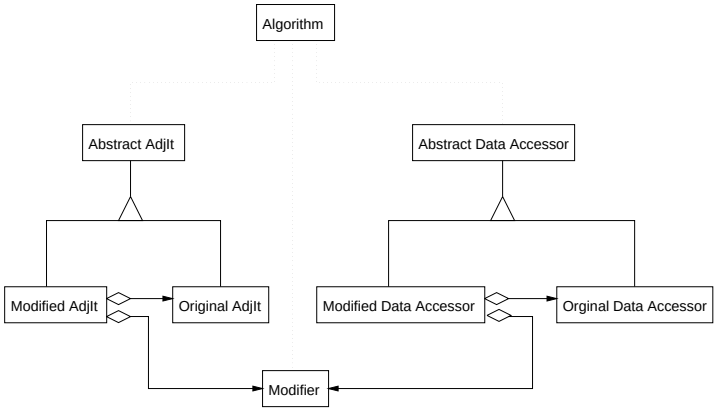
Applicability

Use a modifier when

- auxiliary modifications have to be made to a data structure which does not support the modification operations.
- the exact data structure is not known. A modifier can be used in this case to reduce the requirements imposed on the used data structure.
- modifications have to be made to a structure which does not support the necessary modifications.
- temporary modifications are made which need to be undone: Since the modifications are not really done, there original structure is kept. This is especially useful when the temporary modifications include the removal of objects with additional attributes: It is, in general, not possible to save and restore the attributes as the algorithm might have only access to a subset of all attributes.

  In cases where the original data structure already provides the necessary operations to restore the graph when the temporary modifications are no longer necessary, a modifier can be used to record the modifications and undo them later.
- the algorithm has only a view to a part of the graph but the isomorphism between the part and the whole needs to be maintained: Instead of creating a separate representation of the part, the modifier hides everything which should be invisible to the algorithm.

Structure



Participants

- Algorithm
  - The algorithm uses some data structure(s) to operate on. The data structure is not hard coded but rather defined through an abstract interface, AbstractAdjIt and AbstractDataAccessor in this case (potentially there are more entities).
- Modifier
  - The modifier stores the information about modifications apparently applied to the data structure.
  - The modifier provides the interface used to modify the data structure. This interface may be potentially visible to the algorithm in which case the modifier conforms to the interface of some builder.
- AbstractAdjIt, AbstractDataAccessor
  - The Abstract... classes define the interface of the abstraction used by the algorithm.
  - General modifiers which are not specific to the data structure used, use this interface for their service, too. In this case, it is normally necessary that the interface of the Abstract... classes support a mechanism create objects of the concrete type.
- OriginalAdjIt, OriginalDataAccessor
  - The Original... classes implement the access to the actually used representation.
- ModifiedAdjIt, ModifiedDataAccessor
  - The Modified... classes use objects of the corresponding Original... class for the main part of their behavior.
  - A reference to the modifier object is used to get the information about modifications apparently applied to the data structure and to access global information necessary.
  - Accesses to a Modified... object check whether there is a modification applied: If this is not the case, the request is delegated to the Original... object. Otherwise, the modification is taken into account appropriately. For example, if the modifier allows removal of edges, the method to position on the next edge might use an OriginalAdjIt which is repositioned as long as removed edges are encountered.

Collaborations

The Modifier object stores the information about applied modifications. This information is provided to the objects used by the algorithm (ModifiedAdjIt, ModifiedDataAccessor). The modifier also provides some access to the objects managed, e.g. to inserted objects. For inserted objects, the modifier provides some identification which can be used to get information about these object, to store them in auxiliary data structure (like partial adjacency lists), or to look up associated data in a map.

Consequences

Here are some of the consequences when using a Modifier:

- The modifier introduces an additional indirection to all accesses to the representation. This indirection can normally not be avoided. Thus, it may be reasonable to avoid multiple layers of modifiers, if possible.
- The modifier has often to determine whether there is a modification involved in the access to some object. For example, if data associated with the objects is accessed, the access methods differ for inserted objects and original objects. Apart from this conditional processing the methods to access the data my be less efficient for the data accessor e.g. because data is directly stored with original object but is stored in a map for inserted objects.
- Using a modifier allows to (apparently) modify a structure which does not support modifications. This reduces the requirements on representations used and makes an implementation more general.
- If objects are removed from a data structure, this might result in a loss of information: Data associated with the objects might be removed, too. As the implementation is not necessarily aware of all data stored with the removed object there might even be no possibility to restore the data. Using a modifier, this problem does not occur, as no object is really removed from the representation (unless the modifier is implemented to modify the representation, of course).
- There is no need for the algorithm to keep track of auxiliary modifications to later undo them if a modifier is used: The representation is not changed.
- Neither need representations support complex modification like the contraction of an edge nor need every algorithm using such modification implement them using lower level operations. In addition, there is no ambiguity what the operation really does if a specific modifier is used instead of some operations supported by the representation: For example, contraction of an edge might remove parallel edges and loops or might retain them.

Sample Code

The example shows a modifier used to add two new nodes and some edges between new and original nodes. This modifier can e.g. be used to convert a multiple source/multiple target problem into a single source/single target problem. It works quite simple: For each of the two new nodes, called “source” and “target”, there is a list of adjacent nodes stored using the original adjacency iterators (OA). In addition, two boolean marks are used for the original nodes: One mark indicates whether the node is adjacent to the source node. Likewise the other mark is used for the target node.

```
#include <list>

// OA is the type of the original adjacency iterator
// SM is the type of the data accessor for the source marks
// TM is the type of the data accessor for the target marks
template <class OA, class SM, class TM>
class SingleSourceSingleTarget
{
private:
  typedef list<OA>   AdjList;
  typedef AdjList::const_iterator ListIt;
  typedef SingleSourceSingleTarget SSST;

  // node_desc is a bitmask to allow easy detection of
  // original nodes: 'desc & original_any? true: false'.

  enum node_desc
  {
    invalid         = 0x00, // there is no node
    original_node   = 0x01, // original node with original edge
    original_source = 0x02, // original node with edge to source
    original_target = 0x04, // original node with edge to target
    source_node     = 0x08, // it is the source node
    target_node     = 0x10, // it is the target node
    original_any    = 0x07
  };

public:
  class AdjIt;
  friend class AdjIt;

private:
  SM      source_mark;
  TM      target_mark;
  AdjList source_list;
  AdjList target_list;

public:
  SingleSourceSingleTarget(SM const &sm, TM const &tm);

  AdjIt source() const;
  AdjIt target() const;
  AdjIt convert(OA const &ai) const;

  void add_edge_to_source(OA const &node)
  {
    set(source_mark, node, true);
```

```
    source_list.push_back(node);
  }
  void add_edge_to_target(OA const &node)
  {
    set(target_mark, node, true);
    target_list.push_back(node);
  }
};
```

In the objects of this class the modifications are stored. The structure of the graph can be modified using the functions `add_edge_to_source()` and `add_edge_to_target()`, which add a new edge from an original node to the source or the target node, respectively. To provide access to this new structure, a decorator for the original adjacency iterator is defined:

```
template <class OA, class SM, class TM>
class SingleSourceSingleTarget<OA, SM, TM>::AdjIt
{
  typedef SingleSourceSingleTarget<OA, SM, TM> SSST;
  friend class SSST;
private:
  SSST const *ssst;
  node_desc  desc;
  OA         adjit;
  ListIt     current;

  AdjIt(SSST const *st,node_desc d,OA const &a,ListIt const &c);
  static AdjIt create_adjit(SSST const *ssst, OA const &node);

public:
  AdjIt();
  AdjIt(AdjIt const &);

  bool  valid() const
  {
    switch (desc)
    {
      case original_node:   return adjit.valid();
      case original_source: return true;
      case original_target: return true;
      case source_node: return current!=ssst->source_list.end();
      case target_node: return current!=ssst->target_list.end();
      default:          return false;
    }
  }
  bool is_source() const { return desc == source_node; }
  bool is_target() const { return desc == target_node; }
  bool is_original() const {return desc & original_any!=invalid;}
```

```
    OA original() const { return adjit; }

    AdjIt cur_adj() const;
    AdjIt &operator++ ();
  };
```

This class uses original adjacency iterators augmented with some auxiliary data for the cases where an additional edge is involved. The enumeration `desc` is used to tell how the adjacency iterator is interpreted:

`invalid`	there is no node: the adjacency iterator is invalid
`original_node`	an original node, potentially with an original edge
`original_source`	an original node with an edge to the source
`original_target`	an original node with an edge to the target
`source_node`	the source node with an edge to an original node
`target_node`	the target node with an edge to an original node

The value `original_node` indicates that an original adjacency iterator is used. Whether there is an edge associated with the adapted adjacency iterator depends in this case on the state of the original adjacency iterator: If this one is valid, there is an edge identified. Otherwise, the adapted adjacency iterator only identifies a node.

Depending on this value, some of the operations have to perform differently. This is shown for the member `valid()`:[4] If the adjacency iterator identifies an edge incident to the source or the target node (`original_source` or `original_target`) it is clearly valid. If it identifies an original node, the original adjacency iterator is used for the representation. Whether the adjacency iterator is valid thus depends on the validity of this adjacency iterator. Finally, the adjacency iterator is valid if it identifies the source or the target node and not all edges incident to the corresponding node are processed.

To make the modification complete, it is necessary to make data accessors work with the extended graph. This be done for the nodes by storing the data associated with the source and the target node in the data accessor and delegating the accesses to the data associated with original nodes to a data accessor for the original graph:

```
template <class OA, class SM, class TM, class DA>
class SingleSourceSingleTargetDA
{
public:
  typedef typename DA::value_type value_type;
  typedef value_type VT;

private:
  typedef SingleSourceSingleTarget<OA, SM, TM> SSST;

  DA         original_data;
  value_type source_data;
  value_type target_data;

public:
```

[4] The other member functions are left out for brevity.

```
  SingleSourceSingleTargetDA(DA const &, VT const &, VT const &);

  friend void set(SingleSourceSingleTargetDA const &,
    SSST::AdjIt const &, value_type const &);
  friend value_type get(SingleSourceSingleTargetDA const &da,
    SSST::AdjIt const &adjit)
  {
    if (adjit.is_original())
      return ::get(da.original_data, adjit.original());
    else if (adjit.is_source())
      return da.source_data;
    else
      return da.target_data;
  }
};
```

The implementation of this class is straight forward. The constructor takes a data accessor and two values, used as the initial value associated with the source and the node, respectively, as arguments. The data accessor is used for original nodes to delegate the request. For the other two nodes the data accessor fulfills the requests immediately.

4.3 Algorithm Objects

Most graph algorithms use some subalgorithms to perform common tasks or to solve subproblems. The more basic an algorithm is, the more often it is used by other algorithms. Although it seems to be sufficient to implement these basic algorithms just once, this is not done because there is a catch: The algorithms normally need to be augmented by additional actions or termination conditions to be suitable for specific applications. Consider e.g. graph search (see chapter 2.2 on page 11): It is used by many graph algorithms as a subalgorithm. While the actual behavior of graph search is to visit every node once, this is typically not exactly what is desired. Instead, there are other operations necessary: The nodes have to be numbered (e.g. to determine the connected components) or the structure of the implied spanning tree has to be recorded (e.g. for a path find algorithm using graph search). In addition, it is often desired to modify the condition when to terminate the algorithm: e.g. the graph search used by a path finder must terminate, once the two nodes for which a connecting path is sought are reached.

There are two common techniques, to get around this problem: The most common approach used, is to write specialized versions of the algorithms, often tightly coupled with the main algorithm. For simple algorithms this is easily done. A graph search is implemented in a few lines with the changed behavior taking up only another few lines typically directly weaved into the algorithm. This approach has the disadvantage that it is not easy, if possible at all, to use different versions of the subalgorithm or to use different algorithms achieving the same goal.

To achieve some customization of an algorithm, sometime **callbacks** are used: A callback is a function passed to the algorithm which is called in some well defined situation of the algorithm. To be useful, the callback gets passed some well described arguments defining the current state of the algorithm. See also the Command pattern in [12]. Normally, there is a possibility to pass a reference to callback specific data in addition to the function: This data can be used to parameterize the callback and to keep track of data which should persist between multiple calls of the callback. Using callbacks

to augment the behavior of an algorithm is more flexible but still has its limitations: All callback function possible have to be specified and are called whether they really perform additional actions or use some default callback provided. In addition, the callback is passed the all parameters defining the current state of the algorithm whether they are needed or ignored and it is normally impossible to modify the current state of the algorithm.

The approach to pass a callback together with some data specific to this callback to an algorithm is quite flexible. Actually, this approach is nearly identical to the approach which is suggested here: Instead of using a callback together with some user defined, callback specific data, an object of a class representing the algorithm can be used. Using objects instead of functions is more in the spirit of object orientation and provides some advantages:

- It is more convenient to pass around just one object instead of a pair of a function and some data.
- It is natural to provide several different member functions which manipulate the same data. This can be used to give finer control to the algorithm.
- There is not data exposed which is only needed internally to a function.
- Type safety can be maintained even if subalgorithms that require different auxiliary data are exchangeable.
- The algorithm can be preempted

Algorithm objects make it also easy and natural to assemble complex algorithms from simpler ones: An algorithm is just constructed with the subalgorithms used. Allowing to construct an algorithm from a user specified set of subalgorithms also provides a possibility to use different subalgorithms with an algorithm. The advantage of different subalgorithms is to make use of additional knowledge instead of using the "best" known algorithm working on a general problem. For example a path finding algorithm used as part of the augmenting path algorithm can make use of the fact that it is called several times and maintain some data which will direct the path search for the next call.

4.3.1 Algorithm Step

Intent

Encapsulate an algorithm while making it flexible to be used in different contexts.

Motivation

Algorithms often perform their service using some subalgorithms or auxiliary data structures. To make an algorithm as general as possible the subalgorithms and data structures used have to expect the most general input or have to be replaceable.

Consider e.g. the Augmenting Path Algorithm to solve the max–flow problem: This algorithm uses a path finding algorithm to find an augmenting path in a network. Since there are many different path finding algorithms with different run–time profiles or characteristics on the found path plus algorithms taking advantage of properties known for the specific graph class the algorithm is applied on, it is useful to allow the user to choose the algorithm used. The augmenting path algorithm gets passed an object of a path finding algorithm and doesn't bother any further about this object. In particular, the augmenting path algorithm does not know about additional requirements of the path finding algorithm

like auxiliary data needed to mark nodes or whether the path finding algorithm stores data which is used by the next call to the algorithm.

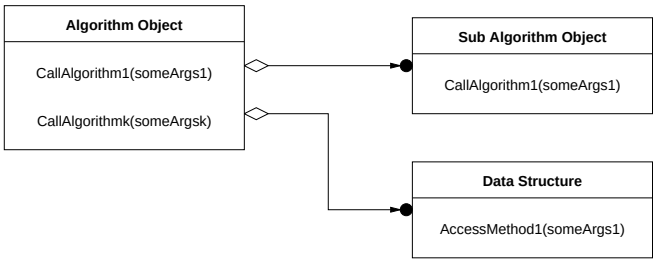
All what is known to the augmenting path algorithm about the path finding object is the behavior of the object, i.e. that it generates a sequence of edges forming a path between two given nodes, and the interface: How to specify the start and the end node of the path, how to start the algorithm and how to retrieve the path.

Applicability

Use the Algorithm Step when

- an algorithm uses one or more subalgorithms for which several implementations can exist.
- several solvers for the same problem can exist.
- the algorithm is used to parameterize another algorithm.
- different interfaces to the same algorithm are useful.

Structure



Participants

- Algorithm Step
  - defines the interface to the algorithm.
  - is created with the subalgorithms and data structures specified which should be used by this object.
- Sub Algorithm Step
  - provides a solver for a partial problem.
  - can itself use arbitrary subalgorithms and data structures.
  - is used through the interface it defines.
- Data Structure
  - provides some service required by the algorithm.

Collaborations

- The algorithm step defers some request to the subalgorithms and use the data structures provided.
- Clients provide subalgorithms and data structures which match the requirements of the algorithm step if a default should be overridden.
- Clients don't modify the subalgorithms or the data structures in a fashion which brings the algorithm step into an inconsistent state.

Consequences

Here are some of the consequences of using the Algorithm Step pattern:

- Encapsulating all data: Nothing has to be known to the client.
- Open for modifications: Subalgorithms and Data Structures can modify the behavior.
- Easy to construct for the user.
- Compared to a normal function, additional programming effort is necessary when creating an algorithm step due to necessary copy and assignment semantics.
- May provide an interface with different levels of control
- Standard applications can be provided e.g by encapsulating the objects into functions.

4.3.2 Initializer

Intent

Decouple the initialization from the actual algorithm.

Motivation

Normally, an algorithm operates in at least two steps: It first initializes the data structures used before it starts the main part of the algorithm. The initialization is typically used to establish a certain precondition. For example a solver for the max–flow problem might first establish a legal flow (normally, the zero flow is used). However, this imposes serious restrictions on the use of the algorithm: It may be totally reasonable to apply an algorithm where a certain precondition not used by the algorithm is not fulfilled.

Apart from a violated invariant, there may be some heuristic which generates a partial solution which should be used as starting point for the algorithm. If the initialization is folded into the algorithm it is impossible to use a partial solution or a violated invariant. Thus it is sometimes useful to separate the initialization from the algorithm and offer an interface to the algorithm which does not initialization (for the convenience of use, it may be useful to offer both interfaces: one which does initialization and one which does no initialization).

Another advantage of the separated initialization is that the same initialization part can be used by different algorithms (if initialization is desired): The augmenting path and the preflow-push algorithm both initialize the network with a zero flow to create a valid flow.

Applicability

Use an Initializer when

- the same initialization can be used by several different algorithms.
- an initialization is not always necessary (i.e. the initialization is used to set up a property which is not used by the algorithm itself).
- there are different possible initializations from which the user of an algorithm may choose.

Consequences

Here are some of the consequences of using the Initializer pattern:

- An initialization algorithm has to be written only once even if it is used as initialization of several different algorithms.
- The initialization is more flexible as the user can choose to use one of the predefined initializations or to write a specialized initialization which exactly suits the purpose.
- It is possible to generate a partial solution by a heuristic which is then used as a starting point by the algorithm for speed ups.
- Properties guaranteed by the original algorithm but neither used by the algorithm nor by the user of the algorithm are not necessarily established. This can safe run–time and make the algorithm useful in situation where it is desired that some property is violated.
- Since there is not control over the initializer, the initializer may fail to establish an invariant necessary for the proper operation of the algorithm. If possible, the algorithm should be implemented to show some well defined behavior even if this case, making the algorithm more complex.

4.3.3 Loop Kernel

Intent

Provide a possibility to insert additional code into the body of an algorithm and make the termination condition of a loop more flexible.

Also Known As

Algorithmic Generators [9]

Motivation

Algorithms often contain central loops which need to be modified slightly by certain applications. In addition it is often useful to have some control over the condition when the algorithm should terminate. Consider e.g. a graph search algorithm: In its pure form it merely traverses through the nodes of a graph but does nothing special with the nodes accessed. To become useful there has to be some additional action: A path finding algorithm might store the node from which the current node was reached and can terminate as soon as the two nodes between which a path has to be found are

reached. In contrast, a connectivity algorithm traverses through all nodes and gives each node a label indicating when it was reached. However, both algorithms can use the same graph search algorithm as subalgorithm.

An algorithm implemented as loop kernel consists of an object used to store the state of the algorithm, some functions modifying the current state (i.e. advancing the algorithm) and some functions used to query the current state. Hence, the algorithm does not solve the problem in one call but rather operates in small steps. After each step the user may decide to terminate or preempt the algorithm. Depending on the algorithm and the objective, it may also be useful to modify the problem while the algorithm is active.

Applicability

Use a loop kernel when

- it makes sense to perform additional operations in each iteration of a loop.
- it should be possible to preempt the algorithm. Preempting an algorithm can be used e.g. to apply multiple algorithms pseudo parallel.
- the condition to terminate the algorithm should be flexible.

Consequences

Here are some of the consequences when using a loop kernel:

- The control over the loop is transferred to the user: It can be terminated or preempted whenever the user wishes to do so.
- In every iteration additional actions can be performed: The algorithm can be augmented or the instance can be modified.
- The state of the algorithm has to be remembered so that the algorithm can continue again after being suspended. However, when implementing the loop kernel the programmer has control over the situations when to pass the control to the user.
- To be more useful, additional functions providing access to the current state of the algorithm have to be provided.

Implementation

A loop kernel is mainly a loop found in an algorithm turned inside out: The loop is removed and the control over the loop is passed to the user of the loop kernel. Thus, it is necessary to store the state of the algorithm. The variables used in the scope of the loop become member variables of the loop kernel. This includes the input of the algorithm and auxiliary data structures as well as the local variables. When implementing the interface of the loop kernel, it has to be decided to which of the members access is provided: To make the loop kernel useful, a user should be able to query and potentially modify the current state the algorithm. For example, for a graph search algorithm it would be reasonable to provide access to the current node and the container. An important part of the current state of the loop kernel which has to be exported to the user, is an indication whether the algorithm is finished or not.

The style used to advance the algorithm is unknown the user. Thus, there has to be an operation which advances the loop kernel. This corresponds to the stepping mechanism of the original loop. Instead of being surrounded by a loop modifying the objects in the scope of the loop, each step is performed in isolation (as seen from the algorithm) and works on member variables.

Apart from deciding on the interface of the loop kernel to query the current state, the transformation of a loop to a loop kernel is straight forward. What was used as initialization in the loop, becomes construction and initialization of the loop kernel. This includes initialization of the members which correspond to variables used by the loop. The body of the loop is what advances the loop kernel: Applying an advance operation continues execution of the algorithm for iteration through the loop before the executation is stopped again. This can be divided into smaller parts than a complete iteration through the loop to allow finer control. The termination condition becomes part of the interface of the loop kernel which can be queried. In addition it may be used to protect the advance operation against erroneous application when the algorithm is already terminated.

Chapter 5

Conclusions

The concepts presented in Chapter 4 provide a rather new style for the implementation of graph algorithms. Instead of implementing an algorithm tailored to a specific graph representation, it is focussed on the implementation of the algorithm ignoring details of the graph representation. This is done by using a set of powerful abstractions to access the graph representation: Linear iterators are used to access the set of nodes and the set of edges, adjacency iterators are used to access the edges incident to a node, and data accessors are used to access additional data associated with the nodes or the edges. Apart from algorithms build upon these abstractions, modifiers can be used to change the graph as it is seen by an algorithm: Without modifying the representation of a graph, the appearance of a graph as seen by an algorithm is changed.

To implement algorithms as flexible as possible, they are implemented as classes and are separated into different steps gathered in algorithm steps: Each of the algorithm used to implement a step can be replaced individually by an algorithm tailored to the problem or the representation. In particular, the initialization is separated from the main part of the algorithm because the same initialization can often be used by different algorithms performing the same task. In addition, a general purpose initialization often destroys already established results when an algorithm is used as subalgorithm by a more a complex algorithm.

Using a loop kernel, it is possible to perform arbitrary additional actions in the body of an algorithm. This can be used to add a condition for the termination of an algorithm, to preempt an algorithm, to calculate some special output, or to modify the algorithm's behavior. There are definitely other uses of a loop kernel not mentioned, too.

Although there is some experience gained from some experimental implementations of algorithms, it still has to be shown that use of the concepts presented in this work promotes reuse. It is also necessary to determine the abstraction overhead imposed by the application of the data abstraction and the splitting of algorithms into small parts.

Bibliography

[1] Alfred V. Aho, David S. Johnson, Richard M. Karp, S.R̃ao Kosaraju, L.A. McGeoch, Christos H. Papadimitriou, and Pavel Pevzner. Theory of computing: Goals and directions. 1996. This report is available electronically at ftp://ftp.cs.washington.edu/tr/1996/03/UW-CSE-96-03-03.PS.Z and will be published in Computing Surveys.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows*. Paramount Publishing International, 1993.

[3] Grady Booch. *Object–oriented Analysis and Design with Applications*. Benjamin/Cummings Publishing Comp., Inc., 1991.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. McGraw–Hill, 1992. 6th printing.

[5] E.A. Dinic. Algorithm for solution of a probem of maximum flow in networks with power estimation". *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[6] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficency for network flow porblems. *J. of the Association for Computing Machinery*, 19:248–264, 1972.

[7] R. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society*, 7:646ff, 1960.

[8] Shimon Even and Robert E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2:436–441, 1976.

[9] Bryan Flamig. *Practical C++ Algorithms and Data Structures*. Wiley, 1993.

[10] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Can. J. Math.*, 8:399–404, 1956.

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

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

[13] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4:75–123, 1985.

[14] F. Hadlock. Finding a maximum cut of a planar graph in polynomial time. *SIAM J. Comput.*, 4:221–225, 1975.

[15] Alexander V. Karzanov. Determining the maximum flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.

[16] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison–Wesley Publishing Company, 2nd edition, 1973.

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

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

[19] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Weslye Professional Computing Series. Addison–Wesley Publishing Company, 1996.

[20] G.I. Orlova and Y.G. Dorfman. Finding the maximum cut in a graph. *Engrg. Cybernetics*, 10:502–506, 1972.

[21] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice Hall, 1982.

[22] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object–Oriented Modeling and Design*. Prentice Hall, 1991.

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

[24] Robert E. Tarjan. Depth–first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.