Chapter 3. Selected Design Issues

Sabine Helwig, Falk Hüffner*, Ivo Rössling, and Maik Weinard

3.1 Introduction

In the cycle of Algorithm Engineering, the design phase opens after the modeling phase. We may assume that the algorithmic task to be performed is well understood, i. e., that the desired input-output relation is specified, and an agreement has been reached as to what makes a solution to the problem a good solution. These questions must be settled in cooperation with representatives from fields of application.

Once the problem specification has been successfully translated into the language of computer science, we must design an appropriate algorithm. We seek a construction plan for the algorithm to be developed, starting with choices about very fundamental algorithmic concepts and iteratively enhancing this picture, until the plan is sufficiently convincing to move it forward to the implementation phase. If there are several alternatives and a theoretical analysis does not reveal a clear winner, design decision should be based on an experimental evaluation.

This chapter discusses selected aspects of the design phase. However, we do not discuss classical algorithm design paradigms like divide & conquer, dynamic programming, prune & search, or greedy approaches, because textbooks on algorithms like [191, 742, 475, 14, 348, 520, 562] usually provide very instructive examples on how to use and combine these design paradigms.

While classical algorithm design mainly considers asymptotic worst-case performance in a certain model of computation, Algorithm Engineering now deals with algorithms exposed to a real-world environment like real-world data, real-world computers, and real-world requirements as to performance and reliability. In Algorithm Engineering, an algorithm *and* an implementation is sought. Hence, in the big picture of Algorithm Engineering the sublime task of algorithm design is to bridge the gap from the first abstract algorithmic ideas to the implementation by anticipating questions that arise during the implementation phase and providing sufficiently detailed answers to them.

A first important step in that direction is to recognize the inherent limitations of the models used. Abstraction is and will remain one of the fundamental approaches to science and everything from the asymptotical $O(\cdot)$ notation to the simplifying PRAM model has been developed for a good and justified reason. However, awareness is advised. When designing an algorithm for real world applications, it is crucial to recognize the potential dangers, e. g., the imprecision resulting from the finite representation of *real* numbers in a computer. Hence,

* Supported by DFG Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4.

M. Müller-Hannemann and S. Schirra (Eds.): Algorithm Engineering, LNCS 5971, pp. 58–126, 2010.

the task of algorithm design is being extended by a couple of important issues arising from reasonable practical needs.

In order to complement standard textbooks on algorithms with respect to Algorithm Engineering we concentrate on the following design issues, which have turned out to be of quite practical importance:

Simplicity. We explain how simplicity of an algorithm is not just a nice feature, but has wide-ranging effects on the applicability. We give several techniques that can help in developing simple algorithms, among them randomization and the use of general purpose modelers.

Scalability. Algorithm designers have to deal with rapidly growing data sets, large input sizes, and huge networks, and hence, they have to develop algorithms with good *scalability* properties. We will introduce some basic ideas, ranging from the pure definition of scalability to scalability metrics used in parallel algorithm design. Moreover, some fundamental techniques for improving the scalability properties of an implementation are presented. Finally, we will discuss techniques for designing highly scalable systems such as decentralization, content distribution in peer-to-peer networks, and self-organization.

Time-space trade-offs. The time and space requirements of algorithms are key parameters of an algorithms performance. How easily one measure of quality can be improved by moderately sacrificing the other one is the central question in the analysis of time-space trade-offs. We discuss formal methods to analyze the capability to exchange time for space and vice versa. Typical application of time-space trade-offs in the context of storing data or supporting brute force methods are discussed, as well as general techniques like lookup tables and preprocessing.

Robustness. Conventional algorithm design is a development process that is often based on abstraction and simplifying assumptions – covering things like the model of computation, specific properties of the input, correctness of auxiliary algorithms, etc. Such assumptions allow the algorithm designer to focus on the core problem. Yet, resulting implementations and runtime environments are not generally able to meet all of these assumptions, at times leading us to the sobering conclusion: In theory, the algorithm works provably – in practice, the program fails demonstrably. The section on robustness discusses the various aspects of this issue, points out focal problems and explains how to consciously design for robustness, i. e., making algorithms able to deal with and recover from unexpected abnormal situations.

Reusability is another design goal. The benefits of reuse are obvious: using a building block that is already available saves implementation time and one inherits the correctness of the existing implementation. This limits the chances to introduce new bugs during the coding and everything that has been done in terms of testing or proofs of correctness is of immediate use. Furthermore, if at a later time a part of the required functionality needs changes or extensions, it suffices to change the one building block every algorithm is using, rather than making similar changes in similar codes.

The issue of reusability arises at two occasions during the algorithm design stage. At an early stage it arises as an opportunity. It should be checked whether the entire algorithm to be designed or algorithms performing subtasks of the given problem are already available. Using a top down approach in designing algorithms the designer will eventually arrive at building blocks that perform a functionality that has been required before. Public software libraries should be checked as well as components of previously completed projects.

At a later stage reusability arises as a strategic option. Newly developed algorithms should be decomposed into building blocks performing functionalities that are easy to grasp and document. The more thoroughly this decomposition is performed and dependencies between the blocks are minimized, the higher is the chance that some of them will come in handy at a later project.

Design for reusability is supported by functions, procedures and modules in imperative programming languages or to a higher degree by objects and classes in object oriented languages. The smaller the degree of interdependence between the building block, the higher the likelihood that a building block can be reused.

3.2 Simplicity

Simplicity is a highly desirable property of an algorithm; a new algorithm that achieves the same result as a known one, but in a simpler way, is often an important progress. Although the simplicity of an algorithm seems to be an intuitively clear concept, it probably cannot be defined rigorously. A reasonable approximation is "concise to write down and to grasp". However, this clearly depends on "cultural" factors: for example, using sorting as a subroutine would certainly not be considered to make an algorithm complicated nowadays, since library functions and knowledge about their behavior are readily available. This might have been different 50 years ago.

Also, much of the perceived simplicity of an algorithm lies in its presentation. For example, Cormen et al. [191] define red-black trees (a dictionary data structure) based on five invariants, and need about 57 lines of code to implement the *insert* function. In contrast, a presentation by Okasaki [620] uses two invariants and requires 12 lines of code. The reason is that Okasaki focuses on simplicity from the start, chooses a high-level programming language, and omits several optimizations.

Because of these inherent difficulties, and to avoid getting tangled in semantic snares, we will do without a formal definition of "simplicity" and rely on the intuition of the concept.

Advantages of simplicity will be further discussed in Section 3.2.1. Section 3.2.2 shows some general design techniques that can help in keeping algorithms simple. Finally, Section 3.2.3 examines the interplay between simplicity and analyzability of algorithms.

3.2.1 Advantages for Implementation

The most obvious reason to choose a simple algorithm for practical applications is that it is quicker to implement: an algorithm that is more concise to describe will take fewer lines of code, at least when using a sufficiently high-level language. This means simple algorithms can be implemented more quickly. Moreover, since the number of bugs is likely to increase with the number of lines of code, simple algorithms mean fewer bugs. Also, the effort for testing the implementation is reduced.

Another major factor is maintainability. A smaller and simpler code base is easier to understand and debug. Also, if the specification changes, simple methods are more likely to be adaptable without major efforts.

A third factor is employment in resource constrained environments, such as embedded systems, in particular pure hardware implementations. An even moderately complicated algorithm has no chance of being implemented in an application-specific integrated circuit (ASIC) or a field-programmable gate array (FPGA).

As an example for the importance of simplicity, the Advanced Encryption Standard (AES) process, which aimed to find a replacement algorithm for the aging DES block cipher, required "algorithm simplicity" as one of the three major criteria for candidates [238].

Lack of simplicity in an algorithm may not only be a disadvantage, but even make implementation infeasible. A famous example is the algorithm for four-coloring planar graphs by Robertson, Sanders, Seymour, and Thomas [674]. The algorithm works by finding one of 633 "configurations" (subgraphs), and then applying one of 32 "discharging rules" to eliminate them. Even though this is the only known efficient exact algorithm for four-coloring, it has to the best of our knowledge never been implemented.

On the other hand, algorithms initially dismissed as too complicated sometimes still find uses; for example Fibonacci heaps, a priority queue data structure, have been described as "predominantly of theoretical interest" [191], but have still found their way into widely used applications such as the GNU Compiler Collection (gcc) [754].

3.2.2 How to Achieve Simplicity?

Clearly, one cannot give a recipe that will reliably result in a simple algorithm. However, several general principles are helpful in achieving this goal.

Top-Down Design. A standard way of simplifying things is to impose a hierarchical, "top-down" structure. This means that a system is decomposed into several parts with a narrow intersection, which can then independently designed and understood, and be further subdivided. Possibly the most simple example are algorithms that work in phases, each time applying a transformation to the input, or enforcing certain invariants. For example, compilers are usually divided

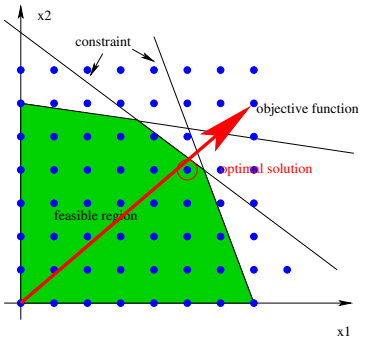

Fig. 3.1. Example Integer Linear Program (ILP)

into a *lexing*, a *parsing*, and a *translation* phase, even though in principle lexing and parsing could be done at the same time. The translation phase is usually broken down further; for example gcc chains more than 100 separate optimization passes.

This concept is well-explored (although typically at the somewhat lower programming language level) in software engineering, for example as *modularity*, but can equally be applied in algorithm design, where one still thinks in terms of pseudo-code. Another benefit of this approach is increased robustness, as explained in Section 3.5.1.

In addition to straightforward phases, there are several more standard algorithm design schemes which can simplify algorithms by reducing them to smaller steps. Examples are divide & conquer, dynamic programming, greedy, and branch & bound. A particular advantage of choosing such a standard scheme is that they are well-known and thus simpler to grasp, and much knowledge about implementing and analyzing them has been accumulated.

General-Purpose Modelers. Often, it is possible to cast a problem in terms of a general problem model. Particularly successful models are linear programs (LPs) and integer linear programs (ILPs) [710,191], constraint satisfaction problems (CSPs) [34], and boolean satisfiability problems (SAT) [522,191]. The chapter on modeling, Section 2.3, gives an extended introduction to this topic; we here focus on an example that demonstrates the simplicity of the approach.

LP solvers optimize a linear function of a real vector under linear constraints. ILPs add the possibility of requiring variables to be integral (see Figure 3.1). This allows to express nonlinear constraints, as will be seen in an example below. CSPs consists of variables that can take a small number of discrete values and *constraints* on these variables, where a constraint forbids certain variable allocations. This generalizes a large number of problems, for example graph coloring. Finally, SAT solvers find assignments to boolean variables that satisfy a boolean expression that contains only AND, OR, and NOT.

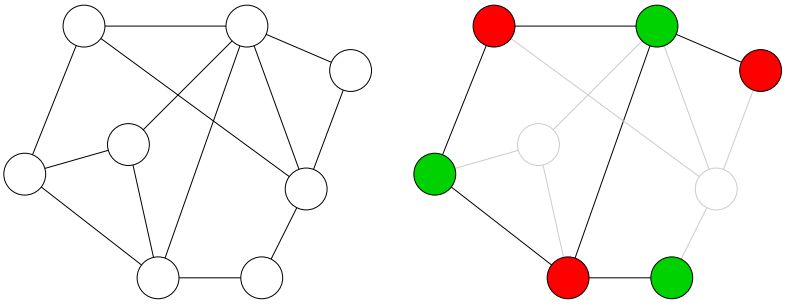

Fig. 3.2. Example Graph Bipartization instance (left) and optimal solution by deleting two vertices (right)

As an example, consider the NP-hard Graph Bipartization, which asks for a minimum set of vertices to delete from a graph to make it bipartite. Given a graph $G = (V, E)$, this problem can be formulated as an ILP with little effort:

$$
\begin{aligned}
& c_1, \ldots, c_n : \text{binary variables } (c_i \in \{0,1\}) \quad \textit{(deletion set)} \\
& s_1, \ldots, s_n : \text{binary variables } (s_i \in \{0,1\}) \quad \textit{(side)} \\
& \text{minimize} \sum_{i=1}^{n} c_i \\
& \text{s. t. } \forall \{v,w\} \in E : (s_v \neq s_w) \vee c_v \vee c_w
\end{aligned}
$$

which can be expressed as an ILP constraint as

$$
\begin{aligned}
\text{s. t. } & \forall \{v,w\} \in E : s_v + s_w + (c_v + c_w) \geq 1 \\
& \forall \{v,w\} \in E : s_v + s_w - (c_v + c_w) \leq 1.
\end{aligned}
$$

Here, $c_v = 1$ models that v is part of the deletion set, and the variables s_v model the side of the bipartite graph that remains after deleting the vertices from the deletion set. The solution space then has $2n$ dimensions (in contrast, the example in Figure 3.1 has only 2 dimensions).

To actually solve an instance, it takes little more than a script containing the above description in a solver-specific syntax. In this way, problem instances could be solved much faster than with a problem-specific branch&bound-algorithm that consists of several thousand lines of code [417], and the size of instance that could be solved within reasonable time was doubled to about 60 vertices.

The power of this approach comes from the many years of Algorithm Engineering that went into the solvers. These solvers are readily available, e. g., GNU GLPK [536] for LPs and ILPs, MINION [326] for CSPs, or MiniSat [270] for SAT, as well several commercial solvers.

When it is possible to formulate a problem in one of these general models without too much overhead, this is usually the quickest way to obtain a solution,

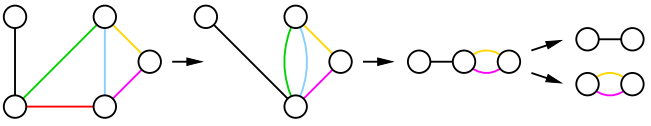

Fig. 3.3. Example for the Monte Carlo algorithm for Min-Cut. In the last step, the top option displayed yields the optimum minimum cut.

and often performance is surprisingly good. Even if it is not satisfactory, there are ways to tune performance and amend the solving process with problem-specific tricks, such as branch&cut for ILPs [185]. Therefore, it is recommendable to try this approach first, if applicable, before thinking about any problem-specific algorithms. An exception are very simple problems that are expected to be solvable in a very good polynomial time bound, since the transformation of the problem representation incurs a noticeable linear-time overhead.

Trade-Off Guaranteed Performance. Sometimes, bad worst-case performance of an algorithm comes from corner case inputs, which would have to be specifically designed by an adversary to thwart the usually good performance of the algorithm. For example, consider quicksort, a sorting algorithm that works by selecting an element as *pivot*, dividing the elements into those smaller than the pivot and those larger than the pivot, and then recursively sorting these subsets. It usually performs very well, except when the choice of the pivot repeatedly divides the subsequence into parts of very unequal size, resulting in a $\Theta(n^2)$ runtime. Even elaborate pivot choice schemes like "median-of-three" cannot eliminate this problem. A very simple solution to this problem is to choose a *random* pivot. In a sense, this thwarts any attempt of an adversary to prepare a particularly adverse input sequence, since the exact behavior of the algorithm cannot be predicted in advance. More formally, one can analyze the *expected* runtime of this algorithm to be $\Theta(n \log n)$. The disadvantage of the approach is that with a small probability, the algorithm takes much longer than expected. An algorithm employing randomness that always produces a correct result, but carries a small probability of using more resources than expected, is called a *Las Vegas algorithm* [589].

A disadvantage of Las Vegas algorithms is that they are often hard to analyze. Still, it is often a good idea to employ randomness to avoid excessive resource usage on corner case inputs, while retaining simplicity.

Trade-Off Guaranteed Correctness. While Las Vegas algorithms gamble with the resources required to solve a problem, *Monte Carlo algorithms* gamble with the quality of the result, that is, they carry some small chance that a solution will not be correct or optimal [589]. Consider for example the Min-Cut problem: given a graph G, find a *min-cut* in G, that is, a minimum size set of edges whose removal results in G being broken into two or more components. We consider the following algorithm: pick a random edge and merge

its two endpoints. Remove all self-loops (but not multiple edges between two vertices) and repeat until only two vertices remain. The edges between these vertices then form a candidate min-cut (see Figure 3.3). The whole process is repeated, and the best min-cut candidate is returned. With some effort, one can calculate how often the procedure has to be repeated to meet any desired error probability. This algorithm is much simpler than deterministic algorithms for Min-Cut, which are mostly based on network flow. In addition, a variant has an expected running time that is significantly smaller than that of the best known deterministic algorithms [589].

Another classical example is the Miller–Rabin primality test [574, 656]. In particular in public-key crypto systems, it is an important task to decide whether an integer is prime. Only recently a deterministic polynomial-time algorithm has been found for this problem [13]. This method is quite complicated and will probably never be implemented except for educational reasons; moreover, it has a runtime bound of about $\tilde{O}(g^{7.5})$, where g is the number of digits of the input [243]. The Miller–Rabin primality test, on the other hand, is quite practical and routinely used in many software packages such as GNU Privacy Guard (GnuPG) [492]. To test a number n for primality, $n-1$ is first rewritten as $2^s \cdot d$ by factoring out 2 repeatedly. One then tries to find a *witness* a for the compositeness of n. With comparably simple math one can show that if for some $a \in \mathbb{Z}/n\mathbb{Z}$

$$a^{2^r d} \not\equiv -1 \pmod{n} \text{ for all } 0 \leq r \leq s-1$$

holds, then n is not prime. By trying many random a's, the probability of failing to detect compositeness can be made arbitrarily small. This algorithm is very simple, can be implemented efficiently, and is the method of choice in practice.

3.2.3 Effects on Analysis

Intuitively, a simpler algorithm should be easier to analyze for performance measures such as worst-case runtime, memory use, or solution quality. As an example, consider the NP-complete Vertex Cover problem: given a graph, find a subset of its vertices such that every edge has at least one endpoint in the subset. This is one of the most well-known NP-complete problems, and it has found many applications, for example in computational biology. A simple greedy algorithm repeatedly chooses some edge, takes *both* endpoints into the cover, and then deletes them from the graph. Clearly, this gives an approximation factor of 2, that is, the solution is always at most twice the size of an optimal solution. The currently best approximation for Vertex Cover [456] is based on semidefinite programming and achieves a factor of $2 - \Theta(1/\sqrt{\log n})$, where n is the number of vertices in the graph. This algorithm is quite complicated, and requires advanced concepts to be analyzed.

However, in fact sometimes simplicity and analyzability seem to be excluding properties, and more complicated algorithms are developed to make them more amenable to analysis tools.

This is illustrated by the NP-complete Shortest Common Superstring problem: given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of strings, find the shortest string that contains each element of $\mathcal{S}$ as a contiguous substring. This problem has important applications in computational biology and in data compression. The standard approach is a simple greedy algorithm that repeatedly merges two strings with the largest overlap, until only one string remains. Here, the *overlap* of two strings A and B is the longest string that is both a suffix of A and a prefix of B. For example:

```
TCAGAGGC GGCAGAAG AAGTTCAG AAGTTGGG
AAGTTCAGAGGC GGCAGAAG AAGTTGGG
AAGTTCAGAGGC GGCAGAAG AAGTTGGG
GGCAGAAGTTCAGAGGC AAGTTGGG
GGCAGAAGTTCAGAGGC AAGTTGGG
AAGTTGGGCAGAAGTTCAGAGGC
```

In the first line, the largest overlap is "TCAG", found at the start of the first string and the end of the third string. Therefore, these strings are merged (second line). After this, the largest overlap is "AAG" (third line), and so on.

One can find an example where the resulting superstring is twice as long as an optimal one, but no worse example is known. This has lead to the conjecture that a factor of 2 is indeed the worst case [828], which is supported by recent smoothed analysis results [532]. However, despite considerable effort, only an upper bound of 3.5 has been proven yet [455].

The currently "best" algorithm for Shortest Common Superstring [769] provides a factor-2.5-approximation. In contrast to the 3-line greedy algorithm, it takes several pages to describe it, and, to the best of our knowledge, has never been implemented. However, its design and features allow to derive the better bound.

Another example for the interplay of simplicity and analysis are recent results on exponential-time algorithms for NP-hard problems. As an example, the Vertex Cover problem can be solved in $O(2^k n)$ time, where n is the number of vertices in the input graph, and k is the size of the cover. For this, one considers an arbitrary edge and branches into two possibilities: the one endpoint is in the cover, or the other is. In a long series of papers, the runtime of this algorithm has been improved to $O(1.274^k n)$ [166]. Most progress was based on an ever increasing number of case distinctions: a list of possible graph substructures, and a corresponding list on how to branch, should they occur. Similar studies were undertaken for other NP-complete problems. The process of finding and verifying such algorithms became somewhat tedious; eventually computer programs were written to automate the task of designing case distinctions [354]. Also, experiments have shown that the numerous distinguished cases do often not lead to a speedup, but in fact to a slowdown, due to the overhead of distinction. Better methods of analyzing the recurrences involved were designed by Eppstein [274]. Using these methods, it was shown that many simple algorithms perform in fact

much better than previously proved; for example, an algorithm for Dominating Set runs in $O(2^{0.598n})$ on n-vertices graphs instead of $O(2^{0.850n})$ [292].

These examples seed the doubt that some "improvements" to algorithm performance in fact may actually be only improvements to their analyzability. There are several ways how this situation could be ameliorated:

- Experimental results can shed some light on the relative performance. For example, one could generate random Shortest Common Superstring instances and see whether the 2.5-approximation fares better than the 3.5-approximation. However, these tests will always be biased by the choice of instances and can never prove superiority of an algorithm.
- Proving lower bounds on the performance of algorithms can give hints on the quality of an upper bound. However, proving good lower bounds can be difficult, and often there remains a large gap between lower and upper bounds. Also, instances used to show the lower bounds are often "artificial" or could easily be handled as special cases in actual implementations.
- Improving the algorithm analysis tool chest. This is clearly the most valuable contribution, as illustrated by the effects of Eppstein's paper [274].

These steps can help to avoid that designers give up simplicity without an actual gain in implementations.

3.3 Scalability

Due to rapid technological advances, system developers have to deal with huge and growing data sets, very large input and output sizes, and vast computer networks. A typical Internet search engine has to find relevant data out of billions of web pages. These large data sets can only be processed by very sophisticated text-matching techniques and ranking algorithms [626]. Car navigation systems have to find shortest paths in graphs with several billions of nodes, ideally with taking traffic jams and road works into account. The graphs used for North America or Western Europe have already about 20,000,000 nodes each [697]. Although the shortest path problem can be solved with the well-known Dijkstra's algorithm [244] in time $O(n^2)$ (where n is the number of nodes), for large road graphs and on mobile hardware with memory constraints, the original Dijkstra algorithm is much too slow. Here, we need new, more specialized, algorithms which can successfully handle real world problems, not only today but also tomorrow, i. e., taking growing data sets into consideration.

Simulation and measurement results of, e. g., car crash tests or computed tomography, produce gigabytes of data which have to be evaluated, analyzed, or visualized. In the "Visible Human Project", the data set representing a human's body is about 40 GB [7]. We certainly expect these data sets to grow larger and larger due to technological progress which allows better and better resolutions of, e. g., computed tomography scans. An algorithm designer must be aware of increasing data sets and larger input and output sizes when approaching a real world problem.

Moreover, not only data sets, but also computer networks are growing. The Internet connects billions of computers which are often sharing the same resources, for instance, a certain web service. The world wide web introduces a lot of new challenges for algorithm designers: Load variations and denial of service attacks must be handled by, e. g., using redundancy and data distribution [247]. The Internet allows its users to share any kind of resources such as data or computational power. Grid computing projects like SETI@home [726] use the idle times of ordinary computers to perform large computational tasks. Coordination between the participants is strongly required in order to solve huge problems jointly in networks whose structure changes permanently. In computer networks, it can be advantageous to spread data over the network in order to decrease space requirements. Distributed data storage has recently become an important research area, and new protocols for efficient data storage and access in large, unreliable, and ever-changing networks have to be invented. The Chord protocol from Stoica et al. [757] proposes a very efficient protocol for distributed data storage.

Summarizing, we have seen that an algorithm designer who wants his or her algorithm to also be used in a few years has to anticipate growing data sets, increasing input and output sizes, and large, permanently changing networks. Such algorithms are said to "scale well". Countless papers propose a "new scalable algorithm" for a certain problem, which suggests that scalability is an important feature of an algorithm. But what is the exact meaning of the term "scalability"? When can we claim our algorithm to "scale well"? The term scalability is used in many different application areas such as data mining, computer graphics, and distributed computing. Thus, giving an overall definition seems to be rather difficult. Nevertheless, in 2006, Duboc et al. [260] presented a "scalability framework", which is a first step towards a formal definition of "scalability". This framework will be presented in Section 3.3.1. In parallel computing, however, the term "scalability" already is widely-used and there exist metrics for evaluating the scalability of a parallel system. Unfortunately, these metrics are too specific to be applied elsewhere. Nevertheless, they give deeper insight into the whole topic, and they might be helpful when an algorithm designer wants to prove his or her system to scale well. Thus, we will show two of these metrics in Section 3.3.2. Afterwards, some basic techniques for designing algorithms with good scaling properties will be presented in Section 3.3.3. Finally, state-of-art strategies for creating highly scalable computer networks such as using decentralization or hierarchies, distributed hash tables, and self-organization, are discussed in Section 3.3.4.

3.3.1 Towards a Definition of Scalability

The term *scalability* is used in many different application areas in order to describe technical systems or algorithms. There exists a variety of different scalability aspects, for example:

– An algorithm should be designed such that it can deal with *small and large input sizes.*

- A database system should be designed such that queries can be answered on *small and large data sets.*
- The running time of a parallel algorithm should decrease in relation to the *number of processing elements.*
- Peer-to-peer networks should be able to deal with a *small and large number of users.*

Usually, a system is said to *scale well* if it can react to modifications (mostly enlargement) of the application or the hardware properties in a way which is acceptable for the system developer as well as for its users. Due to the many aspects of scalability, it is difficult to develop evaluation methods suitable for broad application. But in some research areas the use of specific scalability metrics is well-established, e.g., as already mentioned, in parallel computing. Although the concept of scalability is hard to define, many systems are claimed by their developers to scale well. In 1990, Mark Hill [394] considered the question "What is Scalability?" and concluded with "*I encourage the technical community to either rigorously define scalability or stop using it to describe systems.*" Few studies have been published since then providing more general definitions [368, 117], but most of them correspond to the intuitive definition mentioned above. In a recent study, Duboc et al. [260] still argue:

> *Most uses of the term* scalability *in scientific papers imply a desired goal or completed achievement, whose precise nature is never defined but rather left to the readers' imagination.*

Duboc et al. provide a first step towards expressing scalability more generally. They claim that scalability is about the relationship between cause and effect, i. e., how a system reacts to changes in the environment. Based on this definition, the framework presented in Figure 3.4 has been derived. The application domain and the machine domain are called *independent variables* whereas system requirements such as performance, economics, physical size, security, or reliability are called *dependent variables.* When investigating the scalability properties of a system, single parts of the independent variables, e. g., input size, number of users, size of a database system, or number of processing elements, are changed, and the *effects* on the dependent variables are considered. Accordingly, scalability should never be regarded on its own, but always together with one or more independent and one or more dependent variables. For example, a developer could claim his or her system to scale well in the input size regarding the system's performance. Of course, more precise statements are also possible, e. g., by showing that a parallel system scales well up to 50 processors, but poorly for more than 50 processors. Ideally, the relationship between cause and effect is expressed as a function, but most researchers only present experimental results in order to demonstrate the scalability of their system.

Sometimes, well-scaling systems can be achieved by sacrificing one or more rather unimportant dependent variables for the benefit of one or more other qualities. For example, in many applications it is possible to get faster algorithms by using more space, and vice versa. These so-called *time-space trade-offs* are

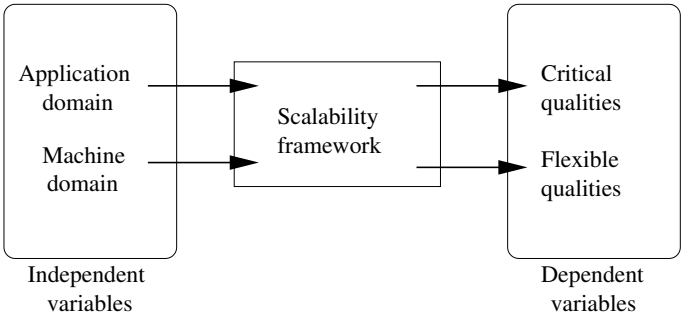

Fig. 3.4. A first step towards a general definition of scalability: the scalability framework [260]

extensively discussed in Section 3.4. Duboc et al. have integrated trade-offs into their framework by dividing the dependent variables into *critical qualities* and *flexible qualities*. Critical qualities are those qualities which are assumed to be very important whereas flexible qualities can be sacrificed. If, for example, performance and quality are critical qualities, and space is a flexible one, we can try to find a time-space trade-off. If performance and space are assumed to be critical qualities, but quality is rather flexible, we might design an approximation algorithm.

Although this framework does not include any concrete scalability metrics, it provides a useful general definition. However, parallel algorithm designers have developed methods for evaluating the scalability of a system. Two of these metrics will be presented in the next section.

3.3.2 Scalability in Parallel Computing

Parallelization can be used to improve the performance of a computer program, and is discussed in detail in Chapter 5. Here, we focus on scalability metrics for parallel systems. The running time of a parallel algorithm does not only depend on the input size, but also on the number of processing elements that are used. Ideally, we would expect a program to run ten times faster if ten processing elements are used instead of a single one. However, for most parallel algorithms, this is not the case due to the following overheads which might occur through parallelization [352]:

Communication overhead. In most parallel systems, the processing elements have to interact with each other to spread intermediate results or to share information.

Idle times. A processing element becomes idle when it must wait for another processor in order to perform a synchronization step, or due to load imbalance.

Poor parallel algorithm. It might be impossible to parallelize the best known sequential algorithm for a given problem. Thus, it might be necessary to use a poorer algorithm, resulting in inherent performance loss.

In the context of parallel algorithm design, scalability is defined as the ability of an algorithm to scale with the number of processing elements, i. e., if more processors are used, the running time should decrease in proportion to the number of the additional processing elements. It is a measure for how efficiently additional processing elements can be used.

There are some broadly applicable techniques to achieve scalability in parallel systems: As communication often is one of the main sources of parallelization overhead, Skiena recommends to design parallel algorithms such that the original problem is split into tasks which can be executed completely independently from each other, and to just collect and put together the results in the end [742]. This strategy is successfully applied in grid computing projects like SETI@home [726]. If communication is necessary for performing the task, Dehne et al. [216] suggest to partition the problem such that only a constant number of global communication steps are required. They show the practicability of this approach on a number of geometric problems such as 2D-nearest neighbor search on a point set, or the calculation of the area of the union of rectangles.

The most challenging part in parallelization is to divide the given problem into appropriate subproblems. Grama et al. [352] identified four decomposition techniques which can serve as a starting point for designing a parallel algorithm:

Recursive decomposition. Problems that can be solved by using a *divide-and-conquer* strategy are qualified for recursive decomposition. The problem is divided into a set of independent subproblems, whose results are then combined to the overall solution. For each of the subproblems, the same algorithm is applied, until they are small enough to be solved efficiently on a single processing element.

Data decomposition. There are several ways for data decomposition: Each processor can compute a single element of the output, if the computation of each output element only depends on the input. Sometimes, the input can be split using a kind of divide-and-conquer strategy: For example, let us assume that the sum of a sequence of numbers has to be computed. It is possible to split the task into summing up the numbers of subsequences and to finally combine the results. If the algorithm is structured such that the output of one step is the input of another, it might be possible to partition the input or the output of one or more such intermediate steps.

Exploratory decomposition. If, for example, the solution of a combinatorial problem is searched for, we might give the problem to an arbitrary number of processors, letting each one apply another search strategy, and finish if a solution has been found.

Speculative decomposition. Some applications are hard to parallelize because a long sequential computation must be performed in order to decide what should be done next. If this is the case, all possible next computation steps

can be executed in parallel, and needless computations will be discarded afterwards.

Evaluating the Scalability of a Parallel Algorithm. Until now, we have defined scalability as a measure for a parallel system's capability to utilize additional processing elements efficiently. Grama et al. [352] described the following model for parallel programs which allows a formal definition of a parallel system's scalability.

When analyzing a parallel algorithm, its performance is usually compared with a sequential algorithm which solves the same problem, and has execution time T_S. To provide fair comparison, a parallel algorithm designer should always use the best known sequential algorithm for the analysis and not, if existing, the sequential algorithm which has been the basis of his or her parallel algorithm.

The execution time T_P of a parallel algorithm is the time elapsed between the beginning of the computation until the last processor has finished.

The *overhead* of a parallel program which is executed on p processing elements is defined as

$$T_O = pT_P - T_S$$

which is the time that would have been required in addition to the sequential running time if the parallel algorithm was processed sequentially.

Speed-up is defined as the ratio of the time required to solve a given problem sequentially to the time required to solve the same problem on p processing elements:

$$S = \frac{T_S}{T_P} \ .$$

Theoretically, this assures that $S \leq p$ is always true, but in practice, speedups greater than p have also been observed, referred to as *superlinear speed-up*. This can be due to, for example, cache effects: If the data is too large to fit in the cache of a single processing element, partitioned for parallel computation it might fit. We need two more definitions to complete the model:

- The *efficiency metric* tells us how efficiently the processing elements are used:
$$E = \frac{S}{p} = \frac{T_S}{pT_P} \ .$$
- The *problem size* W is the number of computation steps that is required by the best known sequential algorithm for solving the problem. For example, for matrix addition, the problem size is $\Theta(n^2)$. The problem size is a function of the input size.

With the previous definitions in mind, scalability is now defined as a parallel system's ability to increase speed-up in proportion to the number of processing elements.

Looking at scalability from another point of view (but based on the same definition), a parallel system is called scalable, if the efficiency can be kept constant as the number of processing elements as well as the input size is increased. If we assume that the problem size W is equal to the sequential running time T_S, we can evaluate W to [352]:

$$W = KT_O(W, p) \ , \tag{1}$$

where T_O is the overhead, which depends on the problem size W and on the number of processing elements p, and $K = E/(1 - E)$ is a constant, as we keep the efficiency constant.

Equation (1) is called the *isoefficiency function* of a parallel system. The isoefficiency function is a measure for the scalability of a system: It specifies the growth rate of the problem size (which is a function of the input size) required to keep the efficiency fixed when adding more processing elements. If its asymptotic growth is slow, additional processing elements can be utilized efficiently. For unscalable parallel systems, the isoefficiency function does not exist, since it is impossible to keep the efficiency fixed when p increases.

3.3.3 Basic Techniques for Designing Scalable Algorithms

After having presented the scalability framework and shown some metrics which can be applied in parallel computing, we will now approach scalability from a more general point of view. In this section, we will present some fundamental techniques for designing algorithms with good scaling properties.

In Algorithm Engineering we are supposed to solve a concrete real world problem. Designing algorithms with good asymptotic worst case running times is of great theoretical interest and leads to valuable insights for practical applications. However, when regarding Algorithm Design in the context of Algorithm Engineering, we have to extend our view and consider the specific application. An algorithm with a bad worst case behavior might be a good choice if the worst case seldom or never happens. Consider the simplex algorithm as an example: It has exponential worst case running time, but it is nevertheless a very popular and successful technique for solving linear programming problems.

One of the most-used methods to achieve scalability in a software system is to apply problem-specific heuristics which have proven to scale well in practice. Since Algorithm Engineering means cycling between design, analysis, implementation and experimentation until an appropriate solution is found, heuristic approaches are very common. Their quality can be checked, e. g., in the experimentation step, and further improvements are possible in the next cycle. The major drawback of applying heuristics is that most of the times, only suboptimal outputs are generated, and often, an analysis of the expected output quality does not exist.

When approaching a concrete problem, we first analyze the problem and make assumptions on the expected inputs and system properties. In order to design an algorithm with good scaling properties, we have to decide which concrete

scaling properties we want to realize. Certainly, if we have to develop a program for a mobile phone, it is not important for us whether our algorithm scales in the number of processors, because we will not use thousands of them in a single mobile phone. Thus, the expected application domain (e. g., input size, number of users) and the system properties (e. g., number of processing elements, memory size) have to be bounded appropriately. Then, we concentrate on designing algorithms which are efficient within the given restrictions, and thus scale well in their application areas. We are not searching for algorithms with good asymptotic behavior because our concrete problems are bounded. A typical example of bound dependent design is the choice of an appropriate data structure which will be explained in the following subsection, and is also discussed in the context of time-space trade-off in Section 3.4.4.

Using Appropriate Data Structures. The choice of the most efficient data structure for a given application depends on the number of elements which have to be stored as well as on the operations which are expected to be most commonly applied to them. Using appropriate data structures can improve an application's performance significantly.

Dictionaries. One of the most important data structure needed in computer science is dictionaries, which can store data identifiable by one or more keys and provide methods for inserting, deleting, and searching for objects. There exists a variety of different dictionaries such as simple arrays, hash tables, and trees. The following dictionary types are widely used (see, e. g., Skiena [742]):

Unsorted arrays. For very small data sets, i. e., less than 20 records, a simple array is most appropriate. A variant which has been proven to be very efficient in practice is a *self-organizing list*: Whenever an element is accessed, it is inserted at the front of the list in order to provide faster access the next time.

Sorted arrays. In a sorted array, elements can be accessed in logarithmic time by performing binary search. However, they only perform well if there are only very few insertions or deletions.

Hash tables. Often, a hash table with bucketing is good choice, when many elements have to be stored. The keys are transformed to integers between 0 and $m - 1$ via a hash function, and then the objects are stored at the respective position of an array of length m. If two or more elements have been mapped to the same position, they can for example be organized as a linked list. The array size m and the hash function have great impact on the performance of a hash table, and should therefore be chosen carefully.

Binary search trees. Binary search trees provide fast insertions, deletions, and access. There are balanced and unbalanced versions. For most applications, balanced trees such as *red-black trees* or *splay trees* are more efficient since an unbalanced tree might degenerate to a linked list, which performs very poorly.

For large data sets which do not fit in main memory, using a *B-tree* might be appropriate. In a B-tree, several levels of a binary tree are moved into one node in order to process them sequentially before requiring another disk access.

Space Partitioning Trees. In many applications, such as computer graphics, statistics, data compression, pattern recognition, and database systems, many objects in low- or high-dimensional spaces have to be stored. Typical questions to such systems are "which object is closest to another given object" (nearest neighbor search), "which region contains the following object" (point localization) or "which objects lie within a given region" (range search).

The nearest neighbor problem is defined as follows: A set S of n elements in k dimensions is given, and we are searching for the closest element in S to a query object q. Obviously, this query can be answered in linear time by comparing all objects with the given one. This simple approach performs very well for a small number (less than 100) of objects. However, for hundreds or thousands of objects, there are better approaches, based on *space partitioning trees*. The idea is to arrange the objects into a tree structure so that the time for answering queries depends on the height of the tree, which ideally is $\log n$. The most-used space partitioning trees are *kd-trees* [91, 307]: The space is recursively divided into two parts according to a splitting strategy, e. g., such that each subregion contains equally many elements. The recursion stops if the number of elements in a region is below a given threshold [590]. These elements can be processed more efficiently using the simple linear time approach. Of course, it is also possible to divide a region not only along one dimension but along every dimension in every split, resulting in *quadtrees* for two-dimensional data sets and *octtrees* for three dimensions [690]. Figure 3.5 shows a kd-tree and a quadtree. The kd-tree should only be used for less than 20 dimensions since it performs very poorly in higher-dimensional spaces. For such applications, searching for an approximate solution of the given problem might be a good approach [49].

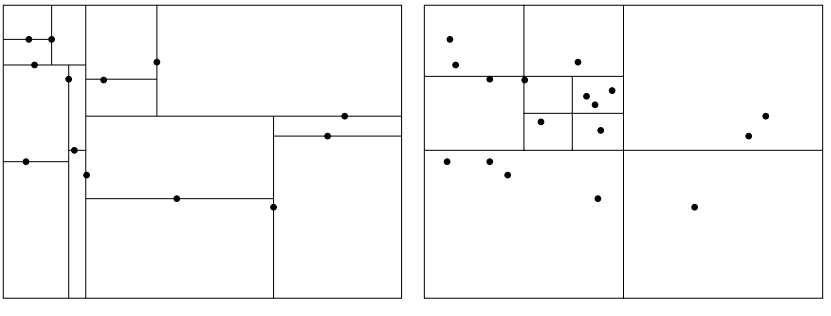

Fig. 3.5. An example of a kd-tree (left) and of a quadtree (right)

The process of analyzing, transforming, and/or reducing data before applying an algorithm is called *preprocessing*, and is discussed in more detail in Section 3.4.4, concentrating on time-space trade-offs. Usually, creating a kd-tree is not a space-critical procedure, but a time-consuming task, and therefore only pays off for large input sizes, or if we expect a large number of queries.

Algorithm Selection. If it is impossible to make any assumptions on the expected application domain, we can design more than one algorithm for a given problem and choose the appropriate one during runtime, when more information is available.

For small input sizes, an algorithm with good asymptotic running time might not be the best choice due to overhead produced by, for example, applying complicated transformations on the input data before solving the problem. These computations (preprocessing operations) are hidden in the O-notation, and might significantly slow down an algorithm for small input sizes. When solving larger problems, however, it pays off to create additional data structures or to do precalculations, since the additional running time is mostly negligible compared to the overall execution time.

An example for algorithm selection is the *introspective sorting algorithm* described by Musser [602]. The median-of-three quicksort algorithm has an average computing time $O(n \log n)$ and is considered to be faster than many other algorithms with equally good asymptotic behavior. However, there are input sequences which lead to quadratic running time. In these cases, a better performance can be achieved by heapsort, with average and worst case running time $O(n \log n)$, but which on average is slower than quicksort. The *introspective sorting algorithm* uses quicksort on most inputs, but switches to heapsort if the partitioning of quicksort has reached a certain depth. The result is an algorithm which works almost exactly like quicksort on most inputs and is thus equally fast, but has a $O(n \log n)$ worst case running time by using heapsort for the critical cases.

After having introduced some fundamental strategies for achieving algorithms with good scaling properties, we will now present some examples of advanced, more modern design techniques.

3.3.4 Scalability in Grid Computing and Peer-to-Peer Networks

In *grid computing* projects, large computational tasks are performed by using many processing elements which can be located geographically far away from each other. Often, they are connected via existing communication infrastructures, mostly the Internet, and try to solve computational problems together by sharing their resources. The number of processing elements can be significantly larger than in traditional parallel applications, while communication can be much slower, and thus, scalability is an important concern here. Special types of grid computing are desktop grids. They have become popular through projects like SETI@home [726], which try to use the idle times of ordinary desktop computers to perform large computations.

Peer-to-peer is a concept which differs from the traditional client-server-approach: Every participant, also called *peer*, acts as both client and server, which means that it provides resources for other peers, but also uses resources of the others. Peer-to-peer networks have become well-known through file sharing, but more generally, each kind of resource can be shared. In peer-to-peer networks, two communicating participants usually establish a direct connection to each other. The scalability question which arises here is whether new participants can be integrated without decreasing the performance of the whole net. For further reading, an overview on peer-to-peer networks and grid computing can be found in [31], and a comparative study has been published by Foster and Iamnitchi [303].

When designing distributed algorithms for grid or peer-to-peer computing, we have to consider two main scalability issues: The system should be able to deal with a large number of participants, and it should use available resources efficiently, even if they are not known beforehand. There exist some techniques to achieve these goals, namely decentralization, making use of hierarchies, and utilizing distributed hash tables, which have been proven to work very well in practice. These techniques will be explained below, in the context of information sharing and content distribution.

Decentralization. Using a central instance which coordinates the whole computation can easily become the bottleneck of a distributed application since the performance of the whole network depends on the performance of this central node. Consider the information sharing application Napster (see, e. g., [31, p. 344ff]): Data is stored in a decentralized manner on the peers, but a central server knows where to find which piece of data. If a peer is searching for something it must ask the central server where to find it. Afterwards, a direct connection to a peer owning the desired information is established. Although data storage takes place decentralized, the existence of a central server slows down the whole application significantly the more users participate.

Gnutella [443] is completely decentralized: A central server does not exist, instead, each peer helps other peers to find information by forwarding incoming requests to its neighbors. The bottleneck caused by centralization vanishes, but Gnutella has another problem, resulting in bad scalability properties: Each request is published randomly in the net, and thus the unintelligent searching strategy can become the bottleneck of this application when too many messages are spread and single nodes become overloaded.

Making Use of Hierarchies. The problem caused by broadcasting requests has been solved by the developers of Kazaa, which is, like Gnutella, fully decentralized, but divides its participants into super nodes and ordinary nodes. This way, Kazaa exploits the heterogeneity of the peers as they can strongly differ in up time, bandwidth connectivity, and CPU power [525]. Each ordinary node is assigned to a super node. Super nodes are fully informed about which information is provided by their children. If a participant searches for data, it asks its

super node where to find it. The super node first checks whether another child node has the desired information, otherwise forwards the request to its adjacent super nodes. If an appropriate node has been found, a direct connection between the peers is established for transferring the data.

Besides Kazaa, there exist many other algorithms which exploit the inherent heterogeneity of a problem in order to achieve better scalability. For example, in car navigation systems, the inherent hierarchical structure of road networks can be used in order to develop extremely efficient, problem-specific solutions [224]. Shortest path algorithms are discussed in detail in Chapter 9. Another example is an algorithm for rendering objects in computer graphics, namely *level of detail* [275]: Objects, which are near the viewer are rendered with high resolution whereas objects which are far away are shown less detailed.

Intelligent Data Distribution. Although Kazaa has helped to eliminate some of the bottlenecks of earlier protocols, it has two main disadvantages: The first one is the lookup strategy. Each request is broadcasted in the net, but after passing a specified number of nodes, it is deleted in order to avoid that unanswered requests are rotating in the network forever. This means that it might happen that a data request is removed even if the data is available. Hence, accessibility of data can not be guaranteed. The second disadvantage is the inhomogeneity of the nodes, although helpful to achieve scalability, this property causes larger vulnerability as, for example, the failure of a super node may cause serious problems for the whole net.

More recent protocols like CAN [664], Chord [757], Pastry [683], Tapastry [391], Viceroy [537], Distance Halving [610], and Koorde [445] overcome these two drawbacks by using *distributed hash tables (DHT)*, which have been introduced as *consistent hashing* in 1997 by Karger et al. [460]. All these protocols assume that data does no longer belong to a certain peer, but can be distributed arbitrarily in the net. This is done by hashing data as well as peers into a predefined space. In order to illustrate this principle, Chord will now be presented in more detail.

Here, each piece of data is represented by a unique key which is hashed to an m-bit identifier. The participating nodes also get an m-bit identifier by hashing, for example, their IP address. Thus, every piece of data and every node has an ID in the interval $[0 \dots 2^m - 1]$. The nodes are arranged into a logical ring structure, sorted by their IDs. Each piece of data is now assigned to the first node whose ID is equal to or follows the data's ID. Figure 3.6 shows a Chord ring with $N = 3$ nodes and $m = 3$, i. e., all IDs are in the interval $[0 \dots 7]$. Let us assume that our nodes have IDs 0, 1, and 3. Data is always stored in its succeeding node, which means that, in our example, in the node with ID 0 all data with ID 0, 4, 5, 6, and 7, is stored, and in the node with ID 3 all data with ID 2 and 3 is stored. This strategy allows efficient leaving and entering of nodes. If a node leaves the net, all its data is transfered to its successor, while when a node is joining, it might get data from its successor.

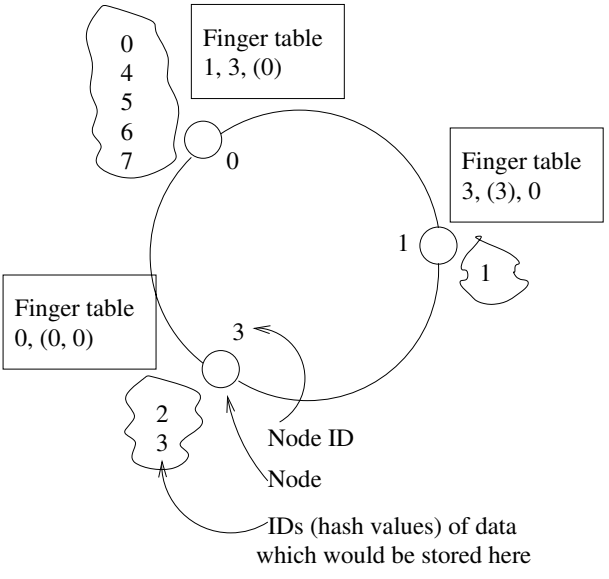

Fig. 3.6. A Chord ring with 3 nodes with IDs 0, 1, and 3. Data is always stored in the node whose ID follows the data's ID, and thus, data with ID 0, 4, 5, 6, 7 is stored in node with ID 0, and data with ID 2 or 3 is stored in node with ID 3.

In order to locate data, each node must have information about other nodes. A Chord node only maintains a very small amount of such routing information; this is the main reason why it scales well in the number of nodes. The routing table of a Chord node, also called *finger table*, consists of at most m entries, which is in $O(\log N)$ where N is the number of nodes. The i-th entry of the finger table of node n contains the ID and the IP address of the first node following $n + 2^{i-1} \bmod 2^m$, for $i = 1 \ldots m$. The finger tables of the nodes in our example are also shown in Figure 3.6. The construction of the finger table assures that each node has more information about the nodes following it than about those located further away in the ring structure, but also has some information about more distant nodes.

If a node wants to look up a piece of data with ID k, it searches its finger table for the node whose ID most closely precedes k and asks this node for more information. By repeating this procedure, the data with ID k will finally be found. Stoica et al. [757] show that with high probability or in steady state, each data can be located using only $O(\log N)$ communication steps. Also with high probability, entering and leaving of nodes only cost $O(\log^2 N)$ messages.

Using Self-Organization as Algorithm Design Strategy. Many biological systems have very good scaling properties, whereby scaling in this context means that the system works well no matter how many individuals are involved. Consider, for example, a fish swarm. There are swarms with only very few fish, but there are also swarms with millions of them. The behavior remains the same which indicates good scalability properties. Reynolds [670] succeeded in visualizing fish swarms by assigning a small set of rules to each fish:

1. Collision Avoidance: Avoid collisions with neighboring fish.
2. Velocity Matching: Try to have the same speed and direction as the neighboring fish.
3. Centering: Try to move towards the center of the swarm, i. e., try to be surrounded by other swarm members.

The reason that this algorithm scales well is that every fish only makes local decisions, and knowledge about the whole swarm is not required.

This is similar to decentralization, but goes one step further: The rules described above are very simple, but nevertheless, complex structures can result [149]. There have been attempts to imitate biological systems in order to develop systems which are simple, scalable, robust, and adaptable, such as ant colony optimization, and genetic algorithms. Only few approaches use self-organization without a concrete natural basis, among them the *organic grid* of Chakravarti et al. [154]. In grid computing, a large number of computers are working together in order to perform expensive calculations. Often, a central instance distributes the tasks among the clients. In the organic grid, however, everyone should be able to use the resources of the whole system by spreading its task over the net. As already mentioned, centralization often becomes the bottleneck of such an application, and thus, Chakravarti et al. developed a fully decentralized system by using self-organized agents which carry the tasks from one node to another. They did not use any biological system as their basis. Instead, they developed their system by first defining the desired goals and by then thinking about the rules each agent must obey in order to achieve these goals. The resulting rules are simple, but provide complex behavior. Thus, they showed that using self-organization as an algorithm design strategy might lead to simple, well-scaling, and robust algorithms.

3.4 Time-Space Trade-Offs

Introductory textbooks about algorithm design usually focus on the time complexity of algorithms and problems, the space requirements are mostly just mentioned in passing. For teaching purposes on an elementary level there is an easy justification for this apparently one-sided approach: using the fundamental model of a Turing machine, it is obvious that an algorithm running for $f(n)$ steps cannot use more than $f(n)$ cells of the working tape. Hence, $DTIME(f) \subseteq DSPACE(f)$ follows and an analysis of time complexity suffices to establish the term of an efficient algorithm.

Space constraints may arise out of system requirements. As nowadays only a vanishing portion of the produced computers are in the shape of multi-purpose computers (e. g., personal computers or laptops) and the major part is embedded in systems as miscellaneous as cell phones, cars, watches or artificial pacemakers, the space requirements of algorithms and problems will likely increase in significance. Technological improvements making it possible to store more and more data at the same cost are usually met by an ever-growing wish to store more and more data creating a permanent shortness.

To pick the most space efficient algorithm among algorithms of the same running time is a first step to include space analysis into algorithm design. When designing algorithms minimizing time and minimizing space may easily be two conflicting goals. Hence, a variety of different algorithmic solutions to one and the same problem may be optimal for different time and space requirements.

The problem of choosing the right algorithm remains relatively simple if one of the resources time or space is by far more crucial than the other one, in a given setting: in an artificial pacemaker, one might be willing to sacrifice orders of magnitude in calculation time to gain a constant factor in space. For a high-end chess computer providing immense extra storing capacities might be acceptable if this enables the system to evaluate 5% more configurations in the time available for a move, because it is able to recognize more configurations which were already evaluated.

The degree to which minimization of time and space are in conflict is the central issue in the discussion of time-space trade-offs. These trade-offs differ immensely in scope: For some problems, the set of time-optimal algorithms and the set of space-optimal algorithms may intersect while in other cases, time-optimality can only be reached by accepting severe space requirements and vice versa. Apart from the scope of the trade-offs also their shape is of interest: For some problems the trade-off between time and space is smooth which means that an increase in time by a factor of $f(x)$ results in a reduction of space by a factor of $g(x)$ with f and g not too different in growth or, in the best case, even asymptotically equivalent. In other cases, the trade-off is rather abrupt which means there is a certain bound of one resource so that even a relatively small step below this bound can only be made by sacrificing the other resource to an extreme extent. We will see examples of these cases in the following discussions.

Time-space trade-offs do not only arise in the comparison of different algorithmic approaches, but may also arise within one algorithm that can be adapted, e. g., by tuning parameters appropriately. A search algorithm can be run with very restricted memory resulting in revisiting the same places over and over again, or it can store its entire search and avoid redundancy completely.

Such an adaption can even be made at runtime if the algorithm itself evaluates parameters like the current processor idle time or the amount of main memory available. The SETI@home [726] project may serve as an example for this line of thought: the system is allowed to use resources (space and time) of participants willing to contribute but it must be ready to clear the resources at all times, should the user require his resources for other purposes.

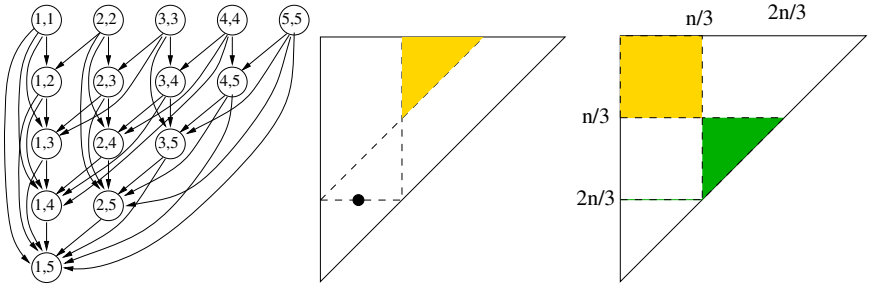

Fig. 3.7. Left: A dependency graph with $G = (V, E)$, $V = \{(i, j)|1 \leq i \leq j \leq n\}$ and $E = \{((a, b), (a, d))|b < d\} \cup \{((a, b), (c, b))|a < c\}$ shown for $n = 5$. **Middle:** An illustration for results no longer needed at a certain point. **Right:** An illustration for the counting argument.

Beyond the application driven motifs to consider time-space trade-offs, there is also a structural insight into the nature of the problem in question and its algorithmic solutions that should not be underestimated. Theory has provided means to establish lower bounds for time-space products and to analyze the important question whether a time-space trade-off for a given algorithm is smooth or abrupt. (See for example [700] for an introduction.)

3.4.1 Formal Methods

Theoretical studies break down into the analysis of *straight line programs* and the analysis of *data-dependent programs*. The latter class is more powerful. In straight line programs, the input does not effect the course of computation – loops (with variable number of executions) and branches are forbidden. Hence, straight line programs for a given input size n may be written as a fixed sequence of input and output steps and operations on previously computed values. A naive bubble sort is a straight line algorithm (it looses this property if a test is added to make the algorithm stop should the array be sorted after some iteration). Prominent straight line algorithms are the Fast Fourier Transformation, computation of convolutions or matrix multiplications.

The dependencies between the different steps of an algorithm can naturally be modeled as a directed acyclic graph. The vertices are the different steps of the algorithm and an edge (u, v) is inserted, if the result of step u is called for in the computation of step v. Input steps have indegree 0, output steps have outdegree 0. Consider for example the dependency graph of Figure 3.7 that we will revisit when discussing dynamic programming in the next section.

The Pebble Game. A fundamental approach to formally studying time-space trade-offs is the so-called *pebble game*, played on these dependency graphs. Pebbles are placed on the vertices and moved from vertex to vertex according to

certain rules. A pebble on a vertex indicates that the result of this node is currently stored by the algorithm. The rules follow naturally:

1. A pebble may be placed on an input vertex at any time.
2. A pebble may be placed on an inner vertex, if there is a pebble on every predecessor of the node. (It is allowed that the pebble placed on the node is one of the pebbles of the predecessors.)
3. A pebble can be removed at any time.
4. If all nodes have once carried a pebble, the game is won.

As every move represents an operation or a reading of an input component, the number of moves needed to win the game corresponds to the running time of the algorithm. The maximum number of pebbles that has been in use at the same time is the algorithm's space requirement. (If a distinction between input space and working space is required, input vertices never carry pebbles and an inner node may be pebbled if all its preceding inner nodes carry a pebble.)

In our example we could just pebble the graph row-wise from top to bottom and within each row from left to right. This results in $\Theta(n^2)$ for time and space. Clearly $\Omega(n^2)$ is a lower bound for time, as every node must be pebbled at least once. But what about space? The center part of Figure 3.7 shows that nodes arise that are no longer needed when computing in this order. If the dot resembles the node currently being pebbled, every result in the shaded area will not be needed again. Hence, we could save pebbles by using them over. It turns out however, that the number of pebbles needed remains $\Theta(n^2)$.

The next approach would be to modify the order in which the nodes are pebbled. This freedom cannot be exploited to yield a lower space requirement: Let (a_1, b_1) and (a_2, b_2) with $1 \leq a_i \leq b_i \leq n$ be two nodes and let $X(a_1, b_1, a_2, b_2)$ be an indicator that is 1 iff the result for node a_2, b_2 is stored at the time the result for node (a_1, b_1) is computed, 0 otherwise. Then $\sum_{(a_2,b_2)} X(a_1, b_1, a_2, b_2)$ is the space in use at the time (a_1, b_1) is evaluated. Due to an averaging argument it suffices to verify that $\sum_{a_1,b_1} \sum_{a_2,b_2} X_{a_1,b_1,a_2,b_2} = \Theta(n^4)$ in order to establish a $\Omega(n^2)$ space bound.

Define $\text{Span}((a_1, b_1), (a_2, b_2)) := (\min\{a_1, a_2\}, \max\{b_1, b_2\})$. Now note that if $\text{Span}((a_1, b_1), (a_2, b_2)) \notin \{(a_1, b_1), (a_2, b_2)\}$, both (a_1, b_1) and (a_2, b_2) are predecessors of the span and hence they are both stored when the node of their span is computed. Consequently, $X(a_1, b_1, a_2, b_2) = 1$ or $X(a_2, b_2, a_1, b_1) = 1$ as we do not delete results until we know that they will not be called for again. A counting argument completes the proof: Pick (a_1, b_1) from the pale shaded area in the right diagram of Figure 3.7 and (a_2, b_2) from the dark shaded area. The span of these combinations is below the lowest dashed line and hence all these $\Theta(n^4)$ combinations contribute a 1 in the above sum.

Hence, we cannot save space (asymptotically) without sacrificing time. If we allow results to be computed, deleted and later recomputed, we are actually able to win the game with $2n - 1$ pebbles. The price turns out to be exponential running time: We inductively verify that a node in the i-th layer (counting top-down starting with layer 1) can be pebbled with $2i - 1$ pebbles. This is obviously

true for level 1. For a node in level i, proceed from $k = i-1$ down to 1 and pebble its two predecessors recursively. This can be done by the induction hypothesis. At the end of each recursive call remove all pebbles placed during this call except the final one. When all the predecessors are pebbled, the node in level i can be pebbled. For the running time, we obtain the recursion $T(n) = 2\sum_{i=1}^{n-1} T(i)$ with $T(1) = 1$ which solves to $T(n) = 2 \cdot 3^{n-2}$.

The time-space trade-offs of other graphs can be much smoother. Several graph classes like binary trees, pyramids, lattices and butterflies have been studied. Hence, an algorithm designer even if he does not want to establish trade-off bounds by analyzing pebble games himself, should at least check whether the specific pattern of dependencies in his task is a prominent one and already analyzed.

The total independence between input and course of computation in straight line programs appears to be a rather severe restriction. However, in algorithms that are *mostly straight line* the same formal methods may still give a hint on time-space trade-offs even though they lose the formal assurance of a mathematical proof. We revisit the above graph and this line of thought when discussing dynamic programming.

3.4.2 Reuse and Lookup Tables

On the conceptual level, an important way to save time is avoiding doing the same things over and over again. Storing and reusing results that have been obtained from a time consuming process is an obvious solution requiring space. Some simple examples:

- In a lookup table values previously obtained by a lengthy calculation are stored for later use.
- In caching, a certain amount of pages is kept in main memory to minimize slow hard disk accesses.
- In distributed databases an object may be stored in more than one location in order to keep the communication time small.

An important question in a given application is to what degree the access pattern to data can be predicted. If the designer is dependent on working with probabilities or relying on heuristics, a more redundant storing must be used to obtain the same performance.

Dynamic Programming. In dynamic programming it is known in advance which previously computed result is needed at a given stage of the algorithm. Hence, the necessary space can be figured out in advance and can be made the subject of a minimization process. A graph, like the one to demonstrate the pebble game, modeling the dependencies between the different subproblems, comes in handy.

Consider the following two basic problems, firstly the problem to decide whether a word w of length n is in a language of a context free grammar

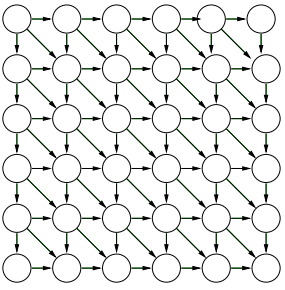

Fig. 3.8. The dependency graph for the longest common subsequence problem

$G = (V, T, P, S)$ given in Chomsky normal form, where V denotes the set of variables, T the set of terminals, P the production rules, and S the start symbol. The well-known Cocke-Younger-Kasami (CYK) method solves $\frac{n\cdot(n+1)}{2}$ problems corresponding to all subwords of w. In fact, the dependency graph in Figure 3.7 is exactly the dependency graph of the subproblems in the CYK-algorithm. Node (i, j) represents the computation of $V_{i,j}$, the set of variables that can produce the subword $x_i, \ldots, x_j$. The dependencies are due to the rule

$$V_{i,j} = \{A \in V | \exists_{i \leq k < j} \exists_{B,C \in V} (A \to BC) \in P, B \in V_{i,k}, C \in V_{k+1,j}\}$$

for $j > i$.

The second problem we discuss is to find the longest common subsequence of two sequences x and y of length n. A standard dynamic programming algorithm computes the longest common subsequences $Max_{i,j}$ of every pair of prefixes $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$ including the empty prefix ϵ for $i = 0$ or $j = 0$. Hence, $(n+1)^2$ subproblems are solved. The dependency graph (Figure 3.8) reflects the rule

$$Max_{i+1,j+1} = \max\{Max_{i,j+1}, Max_{j+1,i}, Max_{i,j} + 1_{x_{i+1}=y_{i+1}}\}$$

for $i, j \geq 0$.

While the common subsequence problem computes roughly twice as many subproblems, a smart implementation only requires storing $\Theta(n)$ solutions at a time. A glance at the structure of the dependency graph of this algorithm reveals this. If the subproblems are solved in a top-down and left to right manner, the algorithm does only need the results of the last $n + 1$ computations. The older ones may be overwritten. We thus have a case where space improvement can be obtained without a loss in computation time.

Observe that our proof for the CYK-algorithm graph needing $\Omega(n^2)$ pebbles when time is constrained to $O(n^2)$, only proves the necessity of storing $\Theta(n^2)$ subsolutions for CYK, *provided* the task of computing one table entry is not interrupted and no partial solutions are stored. We may conclude however, that if we intend to break the $\Omega(n^2)$ space bound, we must do exactly that. Hence,

there is no *easy* way to break this bound, i.e., the bound cannot be broken simply by optimizing the order in which subproblems are solved.

An aspect of dynamic programming that deserves special attention are the two different versions of optimization problems. In many problems which are approached with dynamic programming there is a *value version* and a *construction version*. The value version just asks for the value of the solution, while the construction version also requires producing the solution itself. This solution consists of information gathered during the computation.

In the above example of the longest common subsequence the computation of an internal node consists of picking the maximum among three candidates provided by the predecessors. A constructive solution does not just require the length of the longest common subsequence, but also the sequence itself. This is easily solved by storing after every computation the identity of the predecessor that delivered the maximum value. Then, after the value of the optimal solution is found, one traces these predecessor information back to the origin. This requires storing the whole table even if it is redundant for the value version. In the dependency graph the shift from value- to construction version is reflected in further edges: in the constructive case every internal node has an incoming edge from every node located to its upper left. We loose a factor of n just by switching from the value version to the constructive version.

This problem is addressed in [114] for a variety of interesting cases. It is shown that in the cases they describe, a construction version may be computed with asymptotically the same space as the value version if a slowdown of a logarithmic factor is acceptable. In the cases covered, subproblems are organized in bags and the dependencies are reflected by a constant degree tree with the bags as nodes. A simple case would be the one where the tree is a simply linked list. We can obtain this structure by organizing the different columns of the subsequence graph as a bag. (See Fig. 3.9.)

Bodlaender and Telle point out that this setup arises for many NP-hard problems on graphs that can be solved efficiently, provided the path decomposition or the tree decomposition of the graphs is bounded by a constant. We will only describe the algorithmic idea for the case where the bags constitute a simply linked list. The more general case follows similar ideas.

Assume a sequence of n bags is to be solved. Each subproblem in bag $i+1$ can be solved using only the results of bag i. Furthermore, assume that for every subproblem a single predecessor *delivers* the optimal solution, hence the constructive solution is a path through the dependency graph. (This assumption is met in the subsequence problem, as we only need to remember which predecessor constituted the maximum. The assumption is not met in the CYK graph, as we need to store a pair of predecessors.)

In a first iteration (Fig. 3.9 top) the optimal value as well as a pointer to the subproblem in the first bag, where the path of the optimal solution starts, are computed. Afterwards the algorithm works recursively on problems $P_{l,r}$ with $l < r$, starting with $P(1, n)$. It is assumed that we know which result in bag

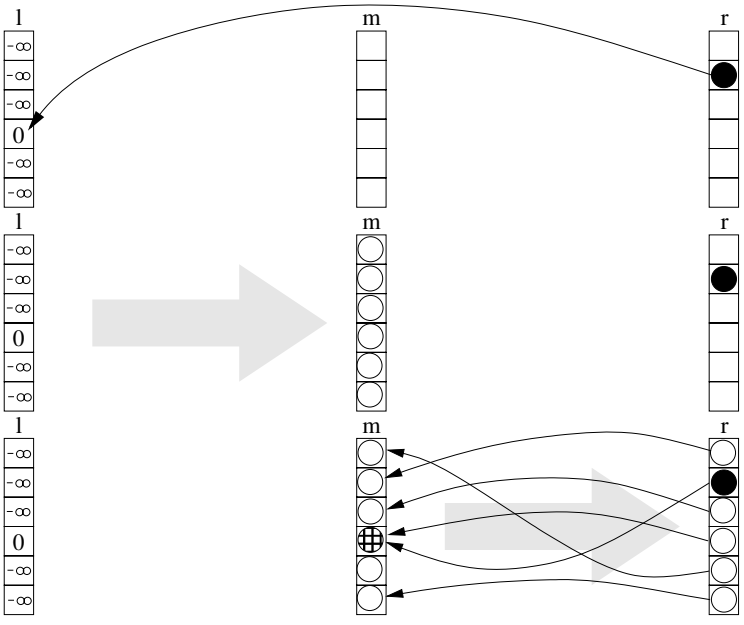

Fig. 3.9. Finding the middle of an optimal path. A method to reconstruct the optimal path in short time, without storing the entire table.

l and which result in bag r is part of the optimal path. We seek to determine which result from bag $m = \lceil \frac{l+r}{2} \rceil$ is on the optimal path.

This is achieved as follows. Assuming a maximization problem, we set the value of the starting point of the optimal path in bag l to 0 and all the other entries of bag l to $-\infty$. That way the optimal path does not change and we do not need to know the real values of bag l. With these fictitious starting values we rerun the dynamic program up to bag m (Fig. 3.9 middle). In the second half of the run from bag m to r we maintain a set of pointers indicating which result in bag m is on the path leading to a specific result. When bag r is reached we follow the pointer from the optimal result to its predecessor in bag m (Fig. 3.9 bottom). The recursion than continues independently for $P(l, m)$ and $P(m, r)$.

We never store more than two bags at a time and have asymptotically the same space requirement as the value version. The time is described by $T(n) = 2T(\frac{n}{2}) + n$ yielding $T(n) = \Theta(n \log n)$.

Online Scenarios. In an online scenario the input and therefore the requests for stored data are revealed at runtime. A web server, for example, has no way of knowing which page a user might request next. The theory of online computation [125] provides formal frameworks for performance guarantees on worst case inputs. We say an approximation algorithm A is *c–competitive* for a minimization

problem, if for every legal instance I the inequality $A(I) \leq c \cdot opt(I)$ holds. The paging heuristic *Least-Recently-Used* for example is k-competitive if k is the number of pages which fit into main memory. It is optimal among all deterministic strategies in the worst case analysis [744]. A marking algorithm using random bits is $O(\log k)$-competitive for *oblivious* input (i. e., if the sequence of page requests is independent of the behavior of the paging algorithm) [4, 286] and optimal in this sense.

However, for real world applications, the idea of an adversary generating the input with the intent to hurt the system as bad as possible is overly pessimistic (*denial of service attacks* being an exception to this rule). A stochastic analysis, assuming every page being requested with the same probability, is equally little appealing. More advanced methods have turned out to resemble experiments with real world data pretty well. The idea is to model legal sequences of requests either deterministically in an access graph or more generally allowing randomness in a Markov chain [461]. By the last one it is assumed that a request for page a is followed by a request for page b with a probability $p_{a,b}$. The sparser the Markov chain, the better it can be exploited to figure out a tailor-made paging strategy for the given application. The probability values themselves may arise out of experimental studies.

In online scenarios we usually have a smooth trade-off: the more space we are able and willing to provide in main memory, the more seldomly page misses will occur.

Interpolation. For arithmetic functions, lookup tables may be used in yet a different manner. Assume a complicated function $f(x)$ is to be evaluated many times throughout an algorithm with the x-values neither being known in advance nor sufficiently predictable. Hence, we are facing an online scenario. But if the function is defined over real-valued variables – even given the usually finite representation of a real value in a computer – the odds of luckily having a requested value in store is negligible. Assuming a certain smoothness of the function however, it might be acceptable to work with interpolation: if function f is called for a specific value x the algorithm determines the biggest x-value smaller than x and the smallest x-value greater than x in the lookup table. The result for x is then obtained for example by linear interpolation or, if the first k derivates of f are also stored, by a more advanced method. In this scenario, building a lookup table for interpolation is also an example of a preprocessing phase yielding a time-space trade-off.

In this case of arithmetic functions the time-space trade-off is a conceptual decision and is not smooth. The lookup table is only constructed if the decision is made to totally ban exact evaluations of f from the computation. Once a table is established, the time saved does not increase with the size of the table, but of course, we do have a clear space precision trade-off.

3.4.3 Time-Space Trade-Offs in Storing Data

Crucial properties of **data structures** are their space requirement and their ability to execute specific operations in a given time. Hence, for data structures the time-space trade-off is *the* measure of performance.

An obvious example are B-trees. They are specifically designed to organize data that does not fit into main memory and the space requirements are only measured in the number of hard disk accesses as the operations in main memory are by orders of magnitudes faster. A B-tree is specified by a parameter $t \geq 2$. Every node except for the root contains at least $t-1$ and at most $2t-1$ keys. Every inner node containing the keys $x_1 < x_2 < \ldots < x_r$ has exactly $r+1$ children that correspond to the intervals

$$[-\infty, x_1], [x_1, x_2], \ldots, [x_{r-1}, x_r], [x_r, \infty].$$

Furthermore, every leaf of a B-tree has the same depth. Insertions and deletions are arranged so that they maintain these invariants.

For reasonably high values of t almost every key will be stored in a leaf. Hence, every unsuccessful and almost every successful search requires d hard disk accesses if d is the depth of the tree. The depth d of a B-tree with n keys is bounded by $d \leq \lceil \log_t n \rceil$. Therefore t should be picked as large as possible, that is, t should reflect the amount of main memory one is able and willing to provide for the search. This example also reminds that different measures of time (main memory operation or external memory access) and space must be used properly in order to achieve a useful performance description. Here the $O(\cdot)$ notation poses a specific danger.

As data structures are designed to support specific operations efficiently while keeping space small, an exhaustive discussion of time-space trade-offs in data structures would actually be an exhaustive discussion about data structures, well beyond the scope of this chapter. We thus restrict ourselves to three aspects that are of specific interest in Algorithm Engineering. First, using the example of resizing in hashing schemes, we describe how a scheme with good amortized performance bounds can be enriched to yield good worst-case performance bounds. We then point out that advanced data structures usually use sub-data structures that need to be well chosen. Hence, both observations deal with bridging the gap from theoretical analyzes to practical necessities. We finish with some remarks about data compression.

Hashing. Hashing is almost a scientific field on its own. The more data is stored in a hash table of a given size, the more often collisions will occur. These collisions either result in longer linked lists in the case of hashing with chaining or in multiple hash table accesses when a form of open addressing is used. Hence, densely filled hash tables require less space per item stored but the price is longer lookup times.

Every concept of hashing provides a system of resizing the hash table. Once a certain load factor is exceeded, a larger hash table is created and the data is rehashed. Should a table become too sparse due to deletions, the table is shrunken.

As the hash function will involve the table size, it becomes necessary to recompute the hash values of every key already stored, whenever a resize measure is undertaken. Usually the change in size of the hash tables will be by a multiplicative factor, as then amortized analysis using the simple accounting method shows constant amortized cost per operation. Hence, even if a single operation, namely the one causing the resizing measure, may take time $\Theta(n)$ where n is the number of elements currently stored, the cost per operation remains constant in the long run.

However, if for a given time crucial operation we cannot afford operations which take linear time in rare peak situations, and therefore an amortized constant time per operation is not good enough, further space can be invested to obtain this goal. One possibility is to setup a new larger hash table in the case of overflow, but to also keep the old one. Lookups are performed on both tables resulting in an increase of a factor 2 in lookup time. Deletions are treated similarly. Whenever a new element is inserted, it is inserted into the larger table and c further elements from the small table are removed from there and hashed into the larger one. The constant c needs to be picked in a manner such that the time needed to rehash c elements is still acceptable. As soon as the old hash table is empty, it can be discarded. The density bounds for enlarging or shrinking tables are picked such that never more than two tables are in use at the same time. Hence, by doubling the space needed, the worst case time of an operation has been reduced from $\Theta(n)$ to constant. So in this case space is used to take the peaks out of the runtime.

Sub-Data Structures. More advanced data structures often include elementary ones as substructures. Here knowledge about the context in which the data structure is to be used, comes in handy. Consider for example a suffix tree [367].

A *suffix tree* T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of a suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i \ldots m]$.

Figure 3.10 shows an example of a suffix tree for the word MISSISSIPPI$. (The unique stop symbol "$" is used as otherwise there are strings with no suffix trees.) We will briefly describe the use of suffix trees when discussing preprocessing in the next section.

Constructing a suffix tree speeds up algorithms that make use of their structure. However, they clearly require significantly more space than the string itself. Hence, finding a space efficient representation is worth a thought. How should the children of an internal node be represented? In bioinformatics, where the strings are DNA sequences consisting of the four nucleotide bases A, C, G and T, an array providing room for pointers to each of the four possible children makes sense. Every child can be addressed in constant time and the space blowup is acceptable. If in principle the entire ASCII code could be the first character of

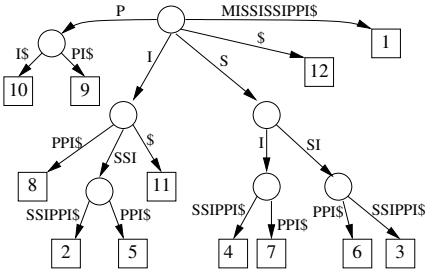

Fig. 3.10. An example suffix tree

an edge label, we will clearly not afford an array of 256 cells for every internal node. Possible alternatives are linked lists (space efficient, slows down lookup), binary trees (more space consuming than linked lists but faster in lookup times) or hash tables.

In fact, practical implementations use a mixture of these schemes. In a suffix tree internal nodes, that are higher in the tree, usually have more children than the ones located closer to the leafs. Hence, one might start with arrays for the first levels and then switch to simple lists further down.

Data Compression. The field of data compression finds ways to save space by investing calculation time. Classical algorithms make only use of statistical properties of the text to be compressed. They rely on redundancies within the text to be compressed and find ways to exploit them. Such redundancies appear in different scope depending on the nature of the compressed file: a text in a natural language has redundancy simply because letters appear with different frequency and e. g., only a small portion of all combinations of 5 letters will ever appear in a text. Is the original file a picture, redundancy arises for example from larger areas having the same color. A random binary string can hardly be compressed as it has by definition no structure, and information theoretic bounds prevent lossless compression of arbitrary input.

Usually the compression is performed to save space when the file in question is currently not used, e.g., the photos on the memory card of a digital camera should take as few space as possible, since one wants to store as many photos as possible and has no problem whatsoever with the time needed to compress and decompress the picture. A new branch of research (e.g. [738]) is dealing with compression methods that allow to execute specific tasks on the compressed file directly, hence making it superfluous to decompress and re-compress it afterwards. First methods of this kind were reported for string problems. Lately, also graph-theoretic questions have been dealt with on compressed representations [107].

3.4.4 Preprocessing

Preprocessing or input enhancement is the process of adding an additional structure to the input, such that the task can then exploit that structure to obtain a better running time. Preprocessing requires anticipating the kind of tasks that are to be performed with the stored data. Quite often the preprocessing will compute partial solutions hoping they will turn out to be useful when the real computation begins. In this case space must be invested.

What constitutes a good preprocessing, obviously depends on the task to be performed. A hint that time-space trade-offs might be obtainable by means of preprocessing is that in a given task the input breaks down into different layers, where the input components of one layer change much more frequently than the components of other layers.

An illustration is a navigation system for cars. The input consists of the *graph* letting us know which streets exist and at what speed we can expect to drive on them. Furthermore, the input consists of the requests, namely origin and destination between which the fastest route is to be found. In a middle layer the input could also include information about current traffic jams, road blocks and so on. As a change in the first layer – the graph – requires building, expanding or redirecting roads in reality, it seldomly appears. More often, traffic jams evolve or vanish, road blocks are established or lifted. Still much more frequently, a new query consisting of origin and destination is brought forward.

It would obviously be unwise if the system reacted to every query as if it had never seen the graph before, hence handling every part of the input equally. A system should at first work on the graph and enrich it with additional structures and information like e. g., the fastest connection between large cities. When a query appears, the system can then first find the closest larger cities to the origin and the destination and use the connection between these cities as a basis for its solution. If a middle layer with current traffic information is used, the system might store additional alternative connections on the graph level, such that if an anticipated scenario of traffic jams occurs, the already developed *backup plan* is immediately available. Chapter 9 has a section devoted to preprocessing techniques for shortest path queries.

It is well possible, that for one and the same problem different preprocessing steps may be useful depending on the context. Let us consider the problem of simple pattern matching: given a text T of length n and a pattern P of length m, find every occurrence of P in T. Clearly $\Omega(n + m)$ is a lower time bound as every algorithm must at least read pattern and text completely. This bound is met by several algorithms. It is possible to preprocess the pattern in time $O(m)$, such that every search for that pattern in a string of length n runs in time $O(n)$. Hence, searching this pattern in r texts of size n each is possible in time $O(rn + m)$ [367].

This approach is advisable if the pattern seldomly changes. For example in bioinformatics a certain sequence of nucleotides in a DNA string may constitute a defect and a lot of DNA samples are to checked for this defect.

Using suffix trees [333] it is even possible to preprocess the text in time $O(n)$ such that after the preprocessing every request for patterns of length m runs in time $O(m)$. Hence, a sequence of r calls with patterns of length m runs in time $O(n + rm)$.

This setup for example applies to web search engines that store huge databases of web pages. These web pages are comparably seldomly updated. More frequently, requests for web pages containing a certain word are made, and in a preprocessed web page the check for the presence of every search-phrase can be done in time linear in the size of the phrase (which is usually short) and independent of the size of the web page.

These preprocessing steps, again, require additional space. For preprocessing the text, a suffix tree of the text must be constructed. That suffix tree may require $\Omega(n^2)$ space. Hence, it should be made sure, that the number of searches to be expected before the next update of the text justifies the factor of n compared to the naive representation of the string.

Let us stick to the Internet search engine a little longer to illustrate another implicit time-space trade-off. The designers of the search engine may choose to store several web pages in one suffix tree. (This roughly corresponds to creating a suffix tree for the concatenation of the pages and deriving the page with the occurrence from the position in the concatenation). Hence, a query now delivers the results of several searches which may (depending on the degree of similarity between the pages) be faster than iterating independent searches in every single text. However the combined suffix tree may need more space than the sum of the individual ones, as fewer edge label compressions may be possible.

3.4.5 Brute Force Support

Brute force methods are mostly applied when efficient algorithms for a problem do not exist or are not acceptable even though they formally count as efficient. The recognition of a problem's NP-completeness for example does not change anything about the presence of the problem and its relevance in certain applications. If efficient approximation algorithms are not available or ruled to produce results of insufficient qualities, brute force methods may be an option.

Optimization Problems. The A^* algorithm [382] is an optimization algorithm which finds on a weighted graph $G = (V, E)$ the shortest path from a source $s \in V$ to a destination $t \in V$ (see also the case study in Chapter 9.2). Of course, this problem can be solved with Dijkstra's algorithm running for $|V|$ iterations in the worst case. The A^* algorithm is capable of exploiting a heuristic $h : V \to \mathbb{R}$ that delivers a lower bound for the lengths of the paths from a certain node to the destination. If this heuristic is weak, A^* will not perform any better than Dijkstra's algorithm. If the lower bound reflects the real distance rather well, a lot of *unpromising paths* do not have to be examined.

A^* maintains a set S of paths. Initially, the set consists of only the path $[s]$. In each iteration, the path p with minimum priority is removed from S. The priority

of a path $p = [v_1, \ldots, v_i]$ is $\sum_{j=1}^{i-1} w(v_j, v_{j+1})$, the sum of the weight of the edges of the path plus $h(v_i)$, the lower bound for the remaining path connecting v_i and t. Hence, A^* is best understood as a *most promising first search*. The possible extensions of p are included into S and it is made sure that further paths in S that end with v_i are ignored.

An obvious example is the computation of shortest routes in traffic guidance systems. The direct line distance between two points (clearly a lower bound for their road distance) serves as heuristic h. Unless the roads of the area in question are heavily distorted, the algorithm will find the fastest route without visiting the entire graph.

The A^* algorithm is not restricted to applications in which the vertices resemble distributed points in space. Another prominent example are solitaire games like the Rubik's cube or the tile game. In the $n \times n$ tile game a set of $n(n-1)$ square shaped tiles numbered 1 to $n^2 - 1$ is arranged in a square frame of side length n, leaving one empty space. Now tiles adjacent to the empty space can be moved into the empty space. The goal is to bring the tiles into a specific order. Here the sum of the distances of each tile to its destination may serve as a heuristic. As every move involves only one tile, the game cannot be won with fewer moves. More moves however may be necessary, as the tiles cannot be moved independently.

Usually the space requirement of the A^* algorithm is severe as A^* quickly proceeds into uncharted areas of the graph creating a huge set of vertices, that the algorithms has *seen* but not yet visited. Several approaches have been used to deal with this problem. One is to search simultaneously starting from the start vertex and from the goal vertex. This turns out to be helpful, if the heuristics typically is more accurate at a long distance from the goal. The straight line distance for navigation systems is clearly of that type. If one is far away from his destination, the straight line gives a good idea about the actual distance. When approaching the destination and being exposed to small labyrinthine alleys and one-way streets the straight line distances value decreases severely.

The concept to iterate A^* is also often used in order to reduce the space requirements at the cost of time. Every run of A^* is executed with a certain bound. Vertices with priorities beyond that bound are ignored and not stored. If a run of A^* fails to reach the goal, the bound is raised and A^* starts all over. This leads to an increase in calculation time, as results are recomputed in every run. This approach is driven to the extreme, if a run of A^* saves the smallest priority above the current bound that it has seen to use this value as the bound for the next run. In this case there is not even a need for a priority queue anymore, as only vertices with minimum priority are considered. This *iterated deepening* A^* approach is called IDA^*. If the length of paths from the source to the goal is extremely small in comparison to the number of vertices, this will be a preferable approach. For the Rubik's cube for example the number of configurations is greater than $4.3 \cdot 10^{19}$ but no configuration is further than 26 moves away from the solution.

Non-Optimization Problems. There is also a variety of applications where a search is not for an optimal solution with respect to a given function but just for a *solution with a certain property.* The attack of cryptographic systems is an example (meet in the middle attack or rainbow tables).

In these cases the approach can be abstracted as follows: A universe U is to be searched for an element x having a property $A(x)$. The designer decomposes the universe U to $U_1 \times U_2$, so the search is now for $x_1 \in U_1$ and $x_2 \in U_2$. If one is able to find a domain U' and a function $f : U_1 \to U'$ as well as a relation $A' \subset U' \times U_2$ such that $A'(f(x_1), x_2) \to A(x_1, x_2)$, one can speed up the process of searching the universe naively by computing as many values $f(x_1)$ as possible and storing them.

Let us demonstrate this setup with the example of the Baby Step Giant Step Algorithm to determine discrete logarithms. Let p be prime and $1 \leq a, b \leq p-1$. We want to find x so that $a^x = b \mod p$. Many cryptographic schemes are based on the hardness of constructing discrete logarithms. The Baby Step Giant Step Algorithm can be used to attack such a cryptographic system or, from the designers perspective, to reveal the systems vulnerability.

A naive algorithm checks every value from 1 to $p-1$ and hence needs exponential time in the number of bits of p. The idea is to split x into two components $x = x_2 \cdot m + x_1$ with $x_1 < m$. A good selection for parameter m is $\lceil \sqrt{p} \rceil$. We get $U_2 = \{0, 1, \ldots, \lceil \frac{p-1}{m} \rceil\}$ and $U_1 = \{0, 1, \ldots, m-1\}$.

The function $f : U_1 \to U'$ in this case is $f(x_1) = a^{x_1} \mod p$ for $x_1 \in U_1$. We compute these values and store them. Now $x_2 \in U_2$ matches $x_1 \in U_1$ if they combine to the solution we seek. Hence they must fulfil $a^{x_2 \cdot m + x_1} = b \mod p$ which we can write as $f(x_1) = \frac{b}{a^{x_2 \cdot m}} \mod p$. So this is the relation A' of the general description.

Using the extended Euclid algorithm we compute $a^{-m} \mod p$ and set $\beta := b$. For $0 \leq x_2 \leq m-1$ we do the following: If β is stored in our table of results (say as $a^{x_1} \mod p$) we have $x = x_2 \cdot m + x_1$ and are done. Otherwise, we set $\beta := \beta \cdot a^{-m} \mod p$ and continue. Hence, by iteratively dividing the target value β by powers of a^m we search for a x_2 that, paired with one of the $f(x_1)$ in store, constitutes the solution.

Choosing $m := \lceil \sqrt{p} \rceil$ we gain a factor of $\sqrt{p}$ in time as we first calculate $\lceil \sqrt{p} \rceil$ values of the function f and later divide β by a^m at most $\lceil \sqrt{p} \rceil$ times. We assume that the values of $f(U_1)$ are stored in a data structure that allows quick lookup. On the other hand we invest the space necessary to store $\lceil \sqrt{p} \rceil$ values of the function f. The naive algorithm only needs constant space.

The search scenarios described in this section allow smooth time-space transitions, as every register available for storing a value in principal shortens the search time. Every register saved for other purposes increases the runtime.

3.5 Robustness

Depending on the subject of discourse, the term *robustness* is assigned quite different meanings in pertinent literature. At the bottom line, these various denotations

may be subsumed as the degree to which some system shows insusceptible to abnormal conditions in its operational environment.

In computer science, these systems of interest are algorithmic components, the defining units of computational processes. Robustness here is usually defined with respect to a given specification of the desired behavior. Still, even in this restricted field concepts are varying and sometimes rather vague. In the following we will employ the terminology due to Meyer [568, p. 5]. He defines *correctness* as the ability of a software component to perform according to its specification, whereas *robustness* denotes its ability to react appropriately to abnormal conditions not covered by this specification. Following this definition, it must be stressed that correctness and robustness are both relative notions. A software component may thus neither called (in)correct nor (non)robust per se, but always only with respect to a given specification.

Note that the above definition has been still uncommitted as to what the precise meaning of *reacting appropriately* would be. Yet, this is just consequential, as any precise definition of some concrete behavior for certain conditions would ultimately become part of the specification. Robustness in turn by intention concerns those conditions for which the concrete behavior is *not* specified after all. Informally, the overall objective is to maintain the component's usefulness despite possible adverse situations; no condition shall make the algorithm crash, run infinitely or return absolute garbage. In other words, we seek for the algorithm in any case to behave reasonable and compute something meaningful.

Throughout this section we want to provide a survey on most relevant non-robustness issues as well as techniques and tools to deal with them. The following subsection discusses robustness from a more software engineering point of view. Sections 3.5.2 and 3.5.3 in turn address robustness issues that arise due to numerical inaccuracy during computation. There, the term *meaningful* will be re-rendered with regard to numerical, respectively combinatorial properties of computed solutions.

3.5.1 Software Engineering Aspects

Abnormal situations do not just appear from nowhere. Instead, they can generally be traced back to a single or more often a combination of several factors. Henceforth, we will call any such cause a *fault*. Note though that this term may be used quite differently in the literature.

The policy of robustness is twofold: anticipate the faults, the causes of abnormal conditions, or limit their adverse consequences. We will soon go into detail on these two notions, called *fault avoidance* and *fault acceptance*. But first let us have a closer look on the adverse momenta themselves.

Fault Types. Depending on the primary causer, faults may be grouped into two classes: *interaction faults* and *design faults*. The following passage is meant to provide a brief overview on most common faults for each category. The list will certainly be incomplete and the faults mentioned may not always be classified unambiguously.

Interaction faults originate from interaction with other systems, which in turn can be the user, some hardware resource or yet another software component. In case of the user, we can identify *accidental interaction faults*, such as operator mistakes or invalid inputs, and *intentionally malicious interaction faults* like penetration attempts or crashing aimed attacks.

As far as hardware is concerned, we have to deal with *service deliverance faults* in the first line. Typical representatives of this kind are that a given resource is not available at all, still busy in serving further requests or it has reached other limits like memory or storage capacity, for example. The second category are *data flaws*, which may arise in (at least) three different scenarios: the hardware serving for data generation, data storage/transmission, or data processing. To name just one example of each kind, imagine a 3d laser scanner producing extremely noise-polluted measured data, an ill-functioning hard disk or network connection causing data corruptions, or a GPU inducing artifacts in a crucial (e. g., medical) visualization.

The same two subcategories can be observed for the software component domain. In this scope, a service deliverance fault can be a communication to another software resource that could not be established, a service that terminated unexpectedly, or a software component that failed to provide time-critical results duly. Concrete examples are a database connection that cannot be established, a shared library that fails being loaded dynamically, or a tardy nested computation in a real-time application. As before, the second category of issues are again data flaws. In fact, any interaction with another software component bears the risk of adverse data to be transferred in one of both directions. Such a fault can usually be seen in two different ways. From the viewpoint of the component receiving adverse input it may be considered an interaction fault. Yet, regarding the component that produced adverse output despite actually benign input, one may considered it a design fault.

Design faults are faults unintentionally caused by man during the modeling, the design but also the implementation phase. In the modeling phase you try to capture the core of the problem and derive a notion of the desired input-output relation, thereby incorporating fundamental assumptions. Clearly, if one of these assumptions is in actual fact unfounded, the formal or informal specification obtained may not suit all problem instances, making the algorithm run into trouble in case of their incidence.

However, in general the modeling phase very well manages to come up with a proper specification. Instead, it more often happens to be the algorithm designer not playing by the rules. For the sake of easing correctness proofs and human understanding, *simplifying assumptions* are drawn and intricate details are omitted. Not filling the voids at a later point in time, this policy ultimately boils down to in an *incomplete design* at the very end. In the very same way also *hidden assumptions*, unintentionally incorporated at some intermediate step, can give rise to a design that does not fully meet the specification. Every so often we can even encounter algorithms being published that, although a "correctness

proof" was provided, eventually turn out to be *incorrect*. Yet, it is sometimes only thanks to the implementation that such a fact gets revealed [373].

Of course, also the implementer can be source of faults. His task is to put the algorithm into practice, thereby filling up all details that have been left open so far. This in particular covers things like *dynamic management of runtime resources* (e. g., memory, (I/O-)streams, locks, sockets, devices and so on), each of which imposing its very own potential for faults. Another problem zone arising in this phase is the realization of *dynamic data structures* and their manipulation, which needs to be crafted carefully to avoid runtime faults. On the other hand, the implementer is given the freedom to locate and make reuse of already existing implementations, e. g., as provided by libraries and frameworks. At this point, however, it is imperative to verify their suitability and to know about their possible restrictions. In both scenarios, own implementation and reuse, the pitfall is to unintentionally impose a *mismatch between the models and specifications* the design is based on, and their counterparts (re)used in the implementation. It is certainly beyond the scope of the algorithm designer to avoid all these potential faults – yet, it is not beyond his scope to treat some of them.

Fault acceptance regards the existence of faults as actually not completely avoidable. It therefore focuses on the resolution of abnormal conditions in case of their incidence, each time seeking for reestablishing some normal condition again. We can distinguish three types of approaches: *detection and recovery*, *fault masking* and *fault containment*.

Approach 1: Detection and recovery is the most common practice. As the term suggests, it consists of an initial fault detection followed by a subsequent recovery procedure. The detection of faults can be attained by the following mechanisms:

Design diversity relies on several alternative versions of a given component, expected to be of dissimilar design. Derived independently from the same specification, these so-called *variants* allow the detection of design faults that cause the diversified copies to produce distinct results on the same given input. The approach is based on the assumption that sufficiently dissimilar designs may hardly suffer from the very same design fault. It is closely related to the *duplication and comparison* technique in the field of fault-tolerant hardware architectures, which makes use of two or more functional identical (hardware) components as a means against physical faults.

Validity checks are used to test whether the given input, the requested operation or the current internal state is actually valid, i. e., covered by the component's specification. They check for the presence of an observable abnormal condition, yet they do not confirm any correctness of the computation performed so far.

Reasonableness checks assess the current internal state or some computed (intermediate) result w. r. t. plausibility. As opposed to the previous scenario, we do not only check for present abnormality. Instead, additional constraints

intrinsic to the specification are exploited, rendering necessary or sufficient conditions entities can be tested against.

Redundancy in representation is introduced to detect integrity faults in the course of data manipulation or exchange. Most well-known realizations are *error detecting codes* such as parities and checksums.

Timing and execution checks are used to detect timing faults or to monitor some component's activity. They are usually implemented by so-called "watchdog" timers and facilitate a mechanism for interception of tasks which fail to provide the result duly or are likely to suffer from an infinite loop.

Once a fault has been detected in the course of performing a requested task, there are basically the following main strategies for a software component to deal with that situation:

Backward recovery tries to return the component from the reached abnormal state back to a previous one, known or supposed to be sane. Afterwards normal service is resumed, proceeding with the next operation.

Forward recovery basically searches for a new state from which the component will be able to resume or restart the requested task.

Graceful degradation can be regarded as a variant of forward recovery where after finding the new state, only that single operation is performed at reduced capability, or the component permanently switches to some degraded operating mode.

Omission of the moot operation is also generally worth considering. The idea is to check first whether the operation would possibly lead to an abnormal state, and simply skip it in that case. Clearly, this technique is only applicable for faults that can actually be detected prior to the execution of the operation. Moreover, omission must be feasible, i. e., we cannot skip operations that are actually vital.

Fault compensation requires sufficient redundancy, either in terms of the internal state's representation or by means of design diversity. Exploiting this additional information provides means to transform the abnormal state into a suitable (usually uniquely corresponding) sane state. It should be noted that in fact more redundancy is required to compensate a fault than to just detect one.

Fault propagation is a matter of releasing competence. The component detects an abnormal condition which it is effectively unable to handle itself. Using some fault notification mechanism, it informs a competent authority, which can also be the calling component or the user, and temporarily or permanently transfers control without conducting any further changes first.

Fail-safe return terminates the execution under control after detecting a fault the component observes to being unable to handle. Based on the assumption that no other component may be capable either, some local or global fail-safe state is entered and possibly a dummy result is returned.

Increasing verbosity is no approach actually aimed for *handling* abnormal situations, yet the minimum to accomplish when concerned about robustness

on the long run. The policy is to report or record any abnormality, at least those that cannot be handled – the more severe, the more verbosely. The basic aim is, if ever an abnormal condition occurs, then to be able to locate its cause based on the recordings and to fix the design appropriately.

Approach 2: Fault masking can basically be circumscribed as "fault compensation without prior detection". It can take the following three forms:

Functional redundancy based masking relies on the design diversity principle described above. The individual outcomes of a component's different variants are merged together, e. g., using some weighted or unweighted majority voting model, thereby obtaining a common result.

Representation redundancy based masking exploits the redundancy incorporated into the internal state's representation in an implicit way. That is, the fault-induced abnormal fraction of the present state is inherently being superseded by the dominant redundant deal. Using an *error correcting code* for the state representation, simply refreshing (i. e., decoding and encoding) the current state represented automatically removes the supported number of bit errors – a common technique to enhancing integrity of data exchange.

Normalization based masking is another way of returning abnormal states or inputs back into normal domain. It usually involves a more or less simple total mapping that maps every normal element to itself and and a deal of the abnormal states to a corresponding normal element each. A trivial example is, when expecting positive inputs, to accept whatever is passed and simply turn it into its absolute value.

Note the way in which the role of redundancy differs between this *fault masking* and the previous *detection and recovery* scenario. Here, redundancy is used to directly override possible faults without explicitly checking for inconsistency first. In contrast, detection and recovery exploits redundancy to detect abnormal situations first of all. Only in case of incidence action is taken, which may then possibly but not necessarily involve recourse to redundancy again.

Approach 3: Fault containment basically neither tries to prevent faults a priori, nor to recover from abnormal conditions. Instead, it aims at restraining the evolution and propagation of abnormal conditions within a so-called containment area. The overall objective is to prevent any further components to become affected. Two obvious paradigms can be distinguished:

Self containment commits to the detection and containment of abnormal conditions eventuating in the component's internal context. In contrast to the detection and recovery principle, we are not too much interested into maintaining or re-establishing service. That is, even unplugging or shutting down is considered tolerable in the border case, as long as a safe state was established first.

Defensive design follows the inverse idea. Every component shall be designed in a way that it defends itself against its outside. By no means shall a fault of external origin be able to infect the component, if there is any chance to detect it beforehand. We will go into more detail on this strategy later.

Fault avoidance is aimed at designing algorithmic components with effectively less fault potential. Faults shall in the best case never be introduced, or get eliminated before the component or system goes into live operation.

Fault removal concentrates on reducing the number or severity of already existing faults. This clearly involves locating these present faults first in the currently established design.

Inspection of the current design (or implementation) should be the most obvious approach. The (pseudo-)code is being reviewed thoroughly by human hand to verify the correspondence between design and specification, thereby challenging once again any assumptions and conclusions made during the design phase. Yet, this is just half of the picture, in that one would have only checked for correctness then. Seeking for robustness in turn amounts to additionally asking the "*But what if ...?*" question over and over again, and requires the algorithm designer to systematically think beyond the borders of the specification.

Formal verification is the method of choice in fields of inevitably high-reliable software construction like aviation, space flight and medicine. Specifications are expressed by means of some description formalism with well-defined semantic [777]. Based on these descriptions, formal methods allow to some degree to verify correctness or other specifiable properties of a software component. Apart from human-directed proofs, two (semi-)automatic approaches can be distinguished: *Model checking*, which basically consists of an exhaustive exploration of the underlying mathematical model, and *automated theorem proving*, which based on a set of given axioms uses logic inference to produce a formal proof from scratch. Some approaches directly verify the code itself, instead of an abstract model. For functional programming languages, verification is usually done by equational reasoning together with induction. For imperative languages in turn, Hoare logic is used in general. This so-called *program verification* will be discussed in Chapter 6.

Testing is the preferred approach applied in medium- and large-scale design scenarios. It basically relies on a dual design strategy. Parallel to designing the component itself, one develops appropriate test scenarios and documents their expected I/O relation. These test scenarios are intended to capture a representative set of normal and abnormal conditions, as well as border cases. *Black-box testing* thereby only uses the given formal specification of the component, whereas *white-box testing* additionally incorporates knowledge on its design or implementation into the development of the test scenarios. Binder [101] explains in detail how to design for testability, how to generate suitable test patterns and how to finally do the tests. It should yet be noted

that, although software testing having proved to be quite successful, not observing any abnormalities does not imply having no faults – this is the curse of testing, as pointed out by Dijkstra [245].

Runtime fault detection should be used to report and/or record any abnormalities caused by faults that were still present as the component went into live operation. The overall goal w. r. t. fault avoidance is that any fault that becomes detected, can be located and removed by making suitable adjustments. The detection should be as rigorous as possible and reporting as verbose as necessary. Basically, the art is to design both fault detection and reporting in a way that renders debugging superfluous.

Fault prevention is concerned about preventing the introduction of faults, or at least limiting their occurrence, from the very beginning. This is usually achieved by starting the design phase with some clear and preferably formal specification of requirements, and by following proven design methodologies. The remaining part of this section tries to give some suggestions on basic rules and methodologies one may follow to achieve empirically more robust designs. Yet, there will certainly be no such thing as an "ultimate answer".

The Role of Specification. The probably most essential ingredient to constructing a software component is a proper specification. This description of the desired behavior is the starting point for any efforts of *validation* ("Am I building the right product?") and *verification* ("Am I building the product right?"). In fact, as pointed out by Meyer [568, p. 334]: Just writing the specification is a precious first step towards *ensuring* that the software actually meets it.

As a specification builds the basis for all subsequent steps of software construction, the resulting design is always as weak as its underlying specification. It is obvious, that latter one should therefore be precise and unambiguous. To achieve this, the best way to express a specification is to make use of some formal systems like Abstract State Machines [366, 123], Hoare logic [397], the B-method [2, 707] or Z-notation [3, 842]. The second of these formalisms will be described into more detail in Section 6.2 of Chapter 6. For a more general introduction on how to create specifications, see [777].

As mentioned before, a formal specification of a component is the basis of its formal verification. At the same time, a specification draws the line between normal and abnormal states. Everything that is not covered by the specification definitely needs to be included into the considerations of how to attain robustness. Moreover, this borderline has coining influence on the design of suitable test scenarios. Last, but not least, specifications are the pivotal elements for reuse. In fact, two designs or implementations can only be exchanged with each other if their corresponding specifications match.

Expressing Expectations. The component's specification is not the only thing that asks for being documented for later purpose. In fact, any assumption and expectation made during the design phase can become crucial at a later point

of time. Therefore, it is also a task of the designer to express any such statement in some formal or informal way. Opting for a formal representation has two advantages: First, one clearly avoids ambiguity. And second, these formal statements can later be turned into so-called *assertions*, or can already be designed that way. Assertions are Boolean expressions defined over the values available in the local context of the considered design fragment. They reflect assumptions or expectations on which the fragment is based and are therefore intended to be true. Assertions can be used to check these expectations at runtime. In fact, if an assertion ever proves false, this indicates a possible fault in the overall design.

However, as Meyer [568, p. 334] sums up, assertions convey even further relevance: They force the designer to think more closely and in formal dimensions, thus getting a much better understanding of the problem and its eventual solutions. They provide a mechanism to equip the software, at the time you write it, with the arguments showing its correctness. They document assumptions and expectations drawn throughout the design phase, thus facilitating later understanding and inspection of the design. Finally, they provide a basis for runtime fault detection and for systematic testing and debugging.

Decompose what is Decomposable. According to Meyer [568, p. 332], the probably single biggest enemy of robustness is complexity. In fact, there is just too much monolithic software construction nowadays [803]. The most obvious way out of complexity is proper decomposition. First of all, decomposition usually leads to a much simpler design. And simplicity in general reduces potential error sources considerably (cf. Section 3.2). Accordingly, Raymond [666, p. 13] outlines the *Rule of Robustness*: Robustness is the child of transparency and simplicity. Breaking the design down allows to focus on single parts at a time. Thus, a top-down analysis of the problem followed by a bottom-up synthesis of the algorithmic solution encourages the use of "building blocks", which in turn can much easier be handled (i. e., examined, tested, replaced, etc.).

Apart from the reduction of potential error sources itself, decomposition might support robustness also from another point of view. If applied in a clever way, the resulting design may pay off, in terms of the robustness of the *whole* software component to be implied in a bottom-up-fashion by robustness of its building blocks on the one hand and their interaction on the other.

But how do we guarantee, or at least increase, robustness in these two areas? Are we back again at the same problem as before? Actually not, since firstly, robustness for smaller components may certainly be assumed to be much easier to achieve than for larger ones. And secondly, robustness of sub-components on the one hand and of their interaction on the other may be treated separately and independently.

Reuse what is Reusable. Another basic rule in modern software construction is: Avoid to reinvent the wheel! The past decades gave raise to copious quantities of algorithms and data structures, ranging from most fundamental to highly specialized ones. So, instead of designing from scratch, try to (re)use building

blocks for which well-tested implementations already exist. In fact, it is certainly not unwise to let the process of decomposition be guided to some extend by the knowledge about such implementations.

Clearly, this might actually mean a trade-off to be weight out between the risk of inventing faults when designing from scratch and the risk of importing faults from these "well-tested" implementations. Apart from that, the principle of reuse asks for more: Similarly to using what is already available, try to design for reusability in the first place – you may want to reuse parts of the design and/or the implementation once again later on, probably much earlier than expected.

However, it must be stressed that designing for simplicity and for reusability is not straight forward. There is usually no such thing as a "best way of decomposition", and in particular reusability is difficult to achieve [803].

Generating Trust. One all too common backdoor for non-robustness to come into play is too much confidence in computed solutions. In general, when requesting an external algorithm for performing some desired task, the returned output is usually not checked for correctness, or at least plausibility [112]. Certainly, such checks might be performed after receiving the result computed by some external algorithm by treating it like unreliable input. Yet, from the external algorithm's point of view fairness dictates to either verify your results before returning them – or to provide some mechanism that makes it relatively easy for your clients to verify your output.

Regarding the former of both approaches, Weihe [826] suggests letting the software component apply runtime checkers that test its (given) input and its (self-produced) output for conformance with the specification. In case of a negative checking result, the component is obligated to handle the abnormal situation itself, instead of returning the adverse result. Depending on how restrictive the runtime checker is, Weihe [826] distinguishes two types of robustness in his own terminology: *Complete robustness* is achieved, when success of the runtime checkers is both, necessary and sufficient for the computation to satisfy the specification. If computed results are just checked for necessary conditions, Weihe speaks only of *partial robustness*. Certainly, the first of the two options should be expected the better choice with regard to robustness. Unfortunately, complete robustness checking may very well increase the expected asymptotic complexity of the initial computation itself.

Of course, runtime checking may in the very same way also be used to verify results obtained from any auxiliary software component invoked for performing some desired subtask. However, in general external results do not have to pass any but at the most a simple plausibility check. First, complete robustness checking can sometimes turn out to be quite expensive. Second, it can not always be achieved easily. But most of all, the task of verification actually appears rather responsibility of the callee than of the caller. The second promising approach towards result verification is therefore concerned with the provision of a mechanism for externally verifying the correctness of computed solutions. The so-called *certification* policy dictates that an algorithm for solving an instance I

of a problem $\mathcal{P}$ does not solely produce some output O, claiming that $\mathcal{P}(I) = O$. Instead, the algorithm additionally produces some *certificate* or *witness* C that (a) is easy to be verified and (b) whose validity implies that indeed $\mathcal{P}(I) = O$.

The topic of checking the correctness of computed solutions is discussed in more detail in Section 6.2 of Chapter 6.

Defensive Design. The idea of *defensive design* is simple: Take precautions to defend your software component against all external sources. On global program level, such provisions can be for instance: Explicit checking of the values of input parameters, overflow and underflow protection during numerical computations, plausibility checks for intermediate results, or redirection of data transfers in case of hardware breakdown.

However, defensive design calls for not only protecting the program as a whole against the outside, but rather every single entity against any other. In its extreme, there were no such thing as too much checking or precaution. Basically, defensive design advocates the attitude of not trusting clients and demands for the protection of any kind of interface – even internally. What you want is to completely isolate failures from one module to the next, so that a failure in module A cannot propagate and break a second module B [803]. Two examples of this kind are *information hiding*, i. e., delimiting access to any internal data solely to the use of access methods; and *defensive copying*, i. e., not sharing any data with other (untrustable) modules, but instead creating copies, both when receiving input as well as when returning output. Of course, this strategy will often be in conflict with efficiency.

In their book, dedicated to teaching how to construct large programs, Liskov and Guttag [529] emphasize the need to "program defensively". A *robust* program, so they conclude, is one that "continues to behave reasonably even in the presence of errors". After all, this is also (and maybe even in particular) a defense against intentional failures, such as hacking.

Design by Contract. The central idea of this systematic approach, developed by Bertrand Meyer [566], is the metaphor of a business contract. The way in which modules interdepend and collaborate is viewed as a kind of a formal agreement between a *client* and the *supplier* of a service, stating mutual rights and obligations. By demanding both parties to go by the contract, the obligations for one party make up the benefit for the other.

In the terminology of design by contract, the two most important elements in a contract are *preconditions*, expressing constraints under which a routine will function properly, and *postconditions*, expressing properties of the state resulting from a routine's execution [568, p. 340]. With a contract at hand, responsibilities are firmly distributed. The client is responsible for fulfilling the suppliers precondition. The supplier, in turn, is responsible for fulfilling its own postcondition. However, the supplier is bound to the contract only inasmuch as the precondition was being adhered. In fact, if the client (the caller) fails to observe

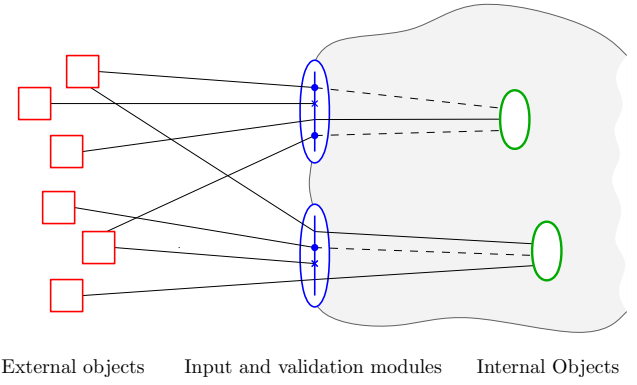

Fig. 3.11. The principle of *filter modules* (following [568, p. 345 et seq.])

its part of the deal, the supplier (the routine) is left free to similarly do as it pleases [568, p. 343].

Basically, *design by contract* even preaches a so-called *non-redundancy principle*, stating that under no circumstances the routine's body should ever test its own precondition. This is contrary to what *defensive design* advocates, which calls for modules to always check incoming messages and reject those violating its precondition [101, p. 845]. Yet, the non-redundancy principle by no means prohibits consistency checks within the body entirely. It rather postulates assigning the enforcement of those conditions to solely one of the two partners. "Either a requirement is part of the precondition and must be guaranteed by the client, or it is not part of the precondition and must be handled by the supplier" [567]. In this regard, it is up to the designer to choose either for a *demanding* or a *tolerating* type of contract.

Basically, this also answers the question of how to ensure a protection of internal components from, e. g., invalid inputs. Note, that we certainly cannot contract the user. To solve this and related problems, Meyer suggests the principle of *filter modules*. The idea is to let internal and external modules be separated by a layer of specifically designed input and validation modules, featuring tolerant contracts with the external modules and strict contracts with the internal modules (cf. Figure 3.11). It is the task of these filter modules to prohibit all external calls that do not fulfill the precondition of the respective internal module, by handling them in an appropriate way.

Being placed as close to the source of the objects in question as possible, such filter modules go in line with what Meyer calls the principle of *modular protection* [568, p. 45]. A method satisfying modular protection ensures that the effect of an abnormal condition in one module will remain confined to that module, or at worst will only propagate to a few neighboring modules [568, p. 45]. Note that this principle differs from *defensive design* in two points: First, Meyer does not ask for a maximum possible protection for each and every single module. And second, modular protection is not aimed at necessarily letting the modules

protect themselves against the outside, but leaves it open how to achieve the desired protection.

Concerning the issue of (non)robustness, it is often claimed that this notion of a contract is so powerful that many well-known failures would certainly have never been caused if design by contract had been applied in the first place. In particular, the Ariane-5 disaster is regularly quoted as an example [434].

Dealing with Adverse Input. There are a couple of reasons why input should not be assumed to be generally good-natured. First, the user should not be expected to be aware of every single aspect that makes up the difference between a valid and an invalid input, not to mention intentional malignity by means of purposeful attacks. Also, the input may not be adverse due to the user's fault, but due to some other origin it results from. In fact, nowadays a multitude of algorithms is actually designed for processing data known to originate from measurements (e. g., in GIS, medicine or bio-science) or previous computations (e. g., in numerical computing, mechanical engineering or computational geometry). Such data cannot be ruled out of being noise-prone, inaccurate, contradictory or even corrupted. Finally, even if the input is in fact valid, it may still be ill-conditioned due to other reasons, e. g., an exceeding complexity of the data or the task requested.

Hence, the question arises, how to deal with an input or request that turns out to be not handleable, based on the current design. To come to the point, there are four obvious answers to this question: *reject it*, *tolerate it*, *fix it* or simply *handle it*.

The first and probably easiest solution, of course, is to reject the request by announcing to be incapable of handling the input data or performing the requested task. Although this does not actually solve the user's problem, it is certainly still preferable to the alternative of a crash or garbage to be computed. In fact, this measure reflects a basic principle: If you are not able to handle it, then it is better to stop right away rather than continuing in spite of being aware of the problem. Just a few years ago, Yap and Mehlhorn [849] as well as Du et al. [259] still criticized the instability of modern CAD software, lacking any robustness guarantees whatsoever and crashing even on suitable choices of inputs. Nowadays, current CAD software still sometimes shows unable to perform the user's request on the given data. Yet, they do not crash anymore, but simply notify the user if an operation could not be performed.

The second option is (trying) to be tolerant. One may, for instance, dynamically decide to revert to a different variant of the algorithm that features less strict prerequisites. Or, in particular, if the input is known beforehand to possibly be subject to noise or inaccuracy, one may use a tolerance-based approach from the very beginning. Such approaches will be discussed more extensively in Section 3.5.3 in the context of geometric robustness issues.

If input turns out to be invalid, it might very well be possible to actually correct it by fixing the points of invalidity. This technique is sometimes employed

in the field of terrain modeling where meshes are checked for their adjacency and incidence information not reflecting a regular mesh (e. g., being incomplete or inconsistent), and are remeshed where necessary. Correcting input requires substantial knowledge about the domain of interest. Missing information demands for being added and contradictory information needs to be replaced, both in a way that guarantees consistency of the resulting patched version of the input. In fact, the problem amounts to answering the two questions: First, which properties do constitute a "valid input" at all? And second, given some invalid input, how exactly does a corrected version look like that, by some means, "corresponds" to that input?

Last, but not least: If the algorithm is, due to its design or its implementation, yet not capable of handling specific inputs or requests, then one can try to make it capable of doing so. Although this may sound slightly absurd, it is not that out of place actually. After all, the cause for the incapability may very well be remediable. The Sections 3.5.2 and 3.5.3, as for instance, will discuss more closely how to proceed in case numerical inaccuracy is the problem.

3.5.2 Numerical Robustness Issues

The previous section was concerned with robustness in a rather general sense and discussed issues from a software-engineering point of view. The following two sections are meant to complete the picture by considering non-robustness arising due to numerical inaccuracy. In mathematics and in theoretical computer science, one commonly assumes being able to *compute exactly* within the field of all real numbers. It is this assumption based on which algorithms are generally proven to be correct. In implementations however, this exact real arithmetic is usually replaced by some fast but inherently imprecise hardware arithmetic, as provided by the computing device.

This hardware arithmetic is generally based on some specific kind of *finite number system*, intended to mimic its infinite counterpart as good as possible. The finiteness of the co-domain of this mapping, however, inevitably results in an approximation, which in turn involves two inconvenient side-effects: discretization errors (round-offs) and range errors (overflows and underflows). They apply to both, the input representation as well as the following computation. Such numerical errors are basically fully expected and in general considered benign [848]. However, the consequence of using a finite number system is crucial: Basic mathematical laws just do not hold anymore in hardware arithmetic (cf. Chapter 6), which in turn used to form the theoretical basis used for proving relevant algorithm properties, like correctness, convergence, termination and other quality guarantees. In fact, these formerly benign errors may turn into serious ones as soon as one of the following two situations eventuates:

- the numerical error accumulates, causing the (numerical) result to be far off from correct, or
- the numerical error, involved in the computation that determines a branch in the control flow, entails a wrong decision with respect to the program logic, thus leading to an inconsistent state of the algorithm.

The remainder of this section will address the first issue, whereas Section 3.5.3 will discuss the second one in detail.

Numerical Problems and their Sensitivity to Inaccuracy. The influence of numerical errors on the quality of computed solutions has been studied extensively in the field of numerical analysis for a couple of decades already. The overriding concern thereby has been to minimize such errors by studying how they propagate, to determine the sensitivity of problems to minor perturbations in the input, and to prove rigorous error bounds on the computed solutions.

Thereby, two types of errors are generally distinguished. The *absolute error* that results from using an approximation s^* to represent some numerical data s is given by $e(s, s^*) := \|s - s^*\|$. Similarly, for $s \neq 0$ the *relative error* of this approximation is given by $\widetilde{e}(s, s^*) := \|s - s^*\| / \|s\| = e(s, s^*)/\|s\|$. A numeric *problem type* $\mathcal{T}$ may generally be expressed by some function $\mathcal{T} : X \to Y$ from an input space $X \subseteq \mathbb{R}^n$ to an output space $Y \subseteq \mathbb{R}^m$. In this sense, a *problem (instance)* $\mathcal{P}$ can be considered as a pair $\mathcal{P} = (\mathcal{T}, x)$ of a problem type $\mathcal{T}$ and a specific input $x \in X$. Solving this problem instance amounts to determining $y = \mathcal{T}(x) \in Y$.

Imagine now, the input x is subject to error such that (due to whatever reason) only an approximation x^* is at hand. This error in the input will imply a corresponding error in the output, whose size depends on both, x and $\mathcal{T}$. Let's say, someone may guarantee that the absolute error in the input will be definitely less than δ, or that the relative error will be definitely less than ε. Then one may be willing to ask, how much influence such an (absolute or relative) inaccuracy might actually have on the result. This sensitivity to minor perturbations in the input is generally referred to as the *condition of a problem*. We define, the *absolute δ-condition* $\kappa_\delta(\mathcal{P})$ of a problem $\mathcal{P} = (\mathcal{T}, x)$ as

$$\kappa_\delta(\mathcal{P}) := \sup_{e(x,x^*)\leq\delta} \frac{e(\mathcal{T}(x), \mathcal{T}(x^*))}{e(x, x^*)}.$$

There are actually different ways to measure a problem's sensitivity to minor perturbations in the input. In fact, Rice [671] and Geurts [330] as well as Trefethen and Bau [783], choose for an asymptotic version. This *asymptotic condition*, which is sometimes just referred to as the *condition* or the *condition number*, represents the limit of the absolute δ-condition for δ approaching zero.

Historically, the term *condition* was first introduced by Turing [786] in the context of systems of linear equations $Ax = b$. To quantify the *benignity* of such a system, he defined the *condition number of a matrix* with respect to inversion $\kappa(A) := \|A\| \cdot \left\|A^{-1}\right\|$. Although Turing's definition seems somewhat different, Geurts [330] showed that, when choosing the matrix norm, κ actually corresponds to the absolute (asymptotic) condition.

Let us briefly illustrate the meaning of condition. Assume, we want to solve the following linear system of equations $A \cdot x = b$

$$\begin{pmatrix} 99 & 98 \\ 100 & 99 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 197 \\ 199 \end{pmatrix} \tag{2}$$

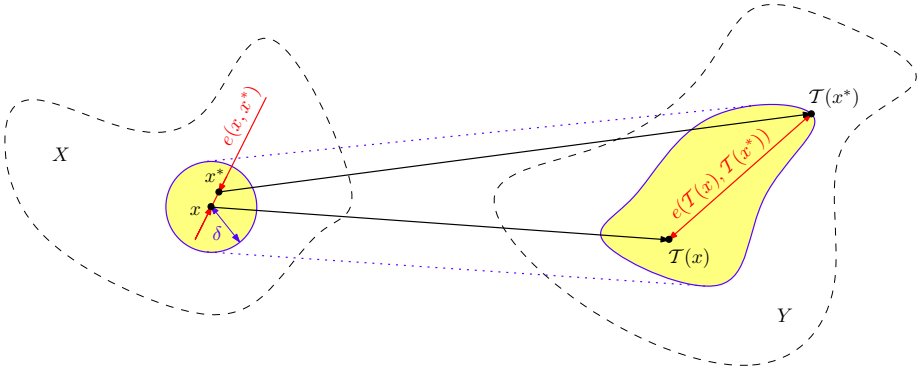

Fig. 3.12. The δ-condition of a problem $\mathcal{P} = (\mathcal{T}, x)$, based on the problem type $\mathcal{T} : X \to Y$ and a problem instance $x \in X$. It is the maximum of the ratio of $e(x, x^*)$ to $e(\mathcal{T}(x), \mathcal{T}(x^*))$ over all approximations x^* of x within an absolute distance of δ to x. As the δ-condition here is large, this problem is considered *ill-conditioned*.

This system features the solution $x_1 = x_2 = 1$, whereas the slightly perturbed version $A' \cdot x = b$

$$\begin{pmatrix} 98.99 & 98 \\ 100 & 99 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 197 \\ 199 \end{pmatrix} \tag{3}$$

has the somewhat completely different solution $x_1 = 100$, $x_2 = -99$. The reason is easily revealed: Applying Turing's formula, we obtain

$$\kappa(A) := \|A\| \cdot \left\|A^{-1}\right\| = \left\| \begin{matrix} 99 & 98 \\ 100 & 99 \end{matrix} \right\| \cdot \left\| \begin{matrix} 99 & -98 \\ -100 & 99 \end{matrix} \right\| = 199 \cdot 199 \approx 4 \cdot 10^4$$

This condition number now tells us that if we were to be faced with some specific minor error or variance in the input, we could not get around accepting a variance in the output of up to four orders of magnitude times as much.

Indeed, the condition number for a problem can be seen as some kind of a magnification factor stating the amplification or dilution of variances from the input towards the output space. It is a measure for a given problem's benignity, i. e., a property that is *inherent to that problem*, imposing an inevitable vagueness in computed solutions when dealing with (e. g., due to discretization) perturbed input. A problem instance that exhibits a small condition is generally referred to as *well-conditioned*, whereas a high condition number gives rise to the term *ill-conditioned*. By extending this notion over the whole input space, we may consequently call a problem (type) *well-conditioned*, if all valid input instances are actually well-conditioned, and similarly *ill-conditioned*, if at least one input-instance is ill-conditioned. In the above example, the problem instance given by Equation (2) is obviously ill-conditioned. Hence, the general problem of solving a system of linear equations or determining the inverse of a given matrix, should be considered ill-conditioned.

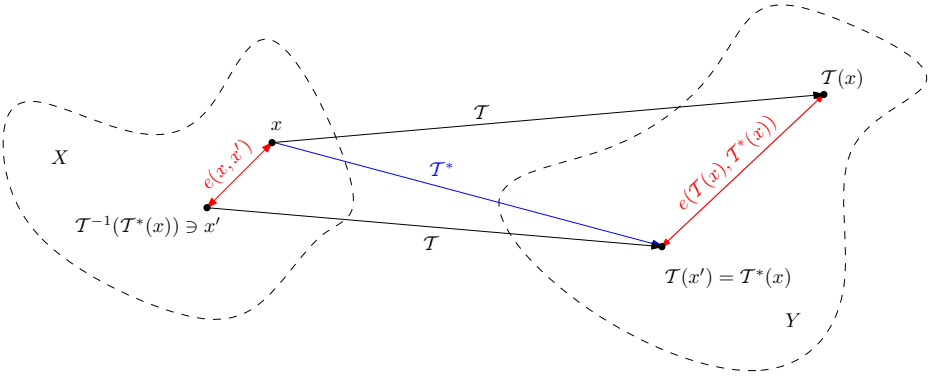

Fig. 3.13. Forward errors (right) and backward errors (left)

Algorithms and Numerical Stability. The quality of computed solutions for a problem does not solely depend on its condition. In fact, it is additionally impaired due to rounding errors that occur during computation. Even for exactly represented input, these round-off errors therefore entail the effective function induced by an algorithm for computing $\mathcal{T}$ to deviate from this ideal function by means of an approximation $\mathcal{T}^*$. The error that is introduced due to this approximation is usually measured in one of the two following ways. The *(absolute) forward error* $\underrightarrow{e}(x)$ with respect to a given input x shall be defined as $\underrightarrow{e}(x) = \underrightarrow{e}(\mathcal{T}, \mathcal{T}^*, x) := e(\mathcal{T}(x), \mathcal{T}^*(x))$. Assuming the existence of a preimage of $\mathcal{T}(x)$ with regard to $\mathcal{T}^*$, the *(absolute) backward error* with respect to a given input x shall be defined as $\underleftarrow{e}(x) = \underleftarrow{e}(\mathcal{T}, \mathcal{T}^*, x) := \inf_{\{x' | \mathcal{T}(x') = \mathcal{T}^*(x)\}} e(x, x')$. The corresponding *relative* notions are defined similarly by substituting $\widetilde{e}$ for e.

Basically, the *forward error* tells us how close the computed solution is to the exact solution, whereas the *backward error* tells us how well the computed solution satisfies the problem to be solved — in other words, how close to the initial problem x there is a similar problem x' for which the exact solution is equal to the computed result. The latter of the two notions allows us to introduce a property for classifying algorithms with regard to their computational quality: An algorithm is called *(numerically) stable*, if it guarantees the backward error to be small for all feasible inputs x. Similarly, an algorithm is called *(numerically) unstable*, if there is at least one feasible input x for which the backward error is large. A (numerically) stable algorithm thus guarantees us that the computed (approximate) solution for a given problem is at least equal to the true solution of a nearby problem. For a more detailed introduction into *numerical stability*, refer to Trefethen and Bau [783] who devote several chapters to this topic and offer the most explicit definition of stability.

Unfortunately, having an algorithm at hand, the exact function $\mathcal{T}^*$ induced by this algorithm is usually not easily determined. However, by viewing an algorithm as a finite sequence of elementary operations, one can stepwise determine a bound

for the overall computation, if for each of the operations such a bound is known. And indeed, depending on the underlying arithmetic, such bounds are at hand – for fixed precision arithmetic in their *absolute*, for floating point arithmetic in their *relative* notion. In fact, standards such as IEEE 754 [420], for instance, dictate an upper bound on the relative forward error per elementary operation in floating point arithmetic, usually referred to as *machine epsilon*, which can be taken for granted in case the environment fully conforms to that standard.

Based on such guarantees, *forward error analysis* tries to bound the forward error of the whole computation, whereas the *backward error analysis* tries to bound the backward error. The analysis of round-off goes back to the work of von Neumann and Goldstine [813, 346]. Historically, forward error analysis was developed first, but regrettably led to quite pessimistic predictions regarding how numerical algorithms would actually perform, once confronted with larger problems. Backward error analysis is heavily to be credited to Wilkinson who did pioneering work in this field (see [831,832], but also the classical books [833,834]).

Coping With Numerical Inaccuracy. As mentioned, the quality of the computed solution basically depends on three parameters: The *input error*, reflecting the initial quality of the input; the *condition* of the problem instance, reflecting the severity inherent to the problem; and finally the *numerical stability* of the algorithm, reflecting the additional deviation between the round-off error affected implementation and its corresponding model, i. e., the problem type. In fact, seeking for $\mathcal{T}(x)$, actually results in finally computing $\mathcal{T}^*(x^*)$. However, regarding the input, it is most common to assume it to be either error-free or beyond our control; and the condition of the problem can even less be biased. Hence, it remains to address the problem of inaccuracy of computer arithmetic, with the objective of avoiding the overall round-off error to blow up.

Facing the fact, that naively implementing a numerical algorithm in a straightforward way based on standard hardware arithmetic may easily result in the resulting program to be prone to numerical inaccuracy, the question arises how to cope with this issue. There are two not necessarily mutually exclusive choices: focus on minimizing or controlling the error, or adapt the underlying arithmetic.

When choosing for sticking to the given arithmetic, the overriding challenge is to get a grip on the round-off errors. This objective basically amounts to determining good bounds for numerical errors, locating numerically critical computations, and finding alternative ways for computing the same value that may show less prone to round-off. (Note, however, that a better bound does not necessarily impose a guarantee for better results.) Numerical analysis offers both static and dynamic techniques. For example, given two equivalent one-line expressions, one may statically assess bounds on the corresponding errors and choose for the more accurate version for the implementation. However, only in rare cases it will be possible to show that one way of computing a value will always yield a better bound than any other known one. In most cases, it will be necessary to react dynamically to the values of the arguments passed. Common strategies are, for

instance, to reorder the data values, to consciously select the next out of a set of permitted alternatives, to transform the data into a more pleasant equivalent, or to choose between different equivalent implementations – always seeking for the least maximum error for the given input.

For example, when computing the sum of a set of numbers, one may dynamically re-order the sequence of values in a way such that cancellation effects will be minimized. Another example can be found in the context of solving a set of linear equations $Ax = b$ via Gaussian elimination. In each step i the current matrix $A_{n\times n}$ of coefficients is pivoted by one of the entries of the still unprocessed submatrix $A[i..n;\ i..n]$. In theory, this choice does not make any difference on the computed solution. In practice, however, it turns out that selecting a small pivot may introduce large numerical errors. Simply choosing the first non-zero coefficient in the current row as pivot (*trivial pivoting*) appears thus not advisable. In consequence, other pivoting strategies have been developed: In *partial pivoting* one chooses the largest magnitude in the current column as pivot. *Scaled partial pivoting* also chooses the largest magnitude in the current column, but always relative to the maximum entry in its row. Finally, in *total pivoting* one always chooses the absolutely largest coefficient in $A[i..n;\ i..n]$. Although out of these three strategies total pivoting involves the best bound on the numerical error, partial pivoting is usually applied in practice since it is much less computationally expensive.

Apart from delimiting the negative effects of round-off by controlling the accumulation of numerical errors, numerical analysis also helps us in assessing their actual magnitude at runtime. In doing so it allows us to check at runtime whether computed solutions are reliable or not. This is an important part of the basis of reliable computing and will be discussed in Chapter 6. Another question, that chapter deals with, is how to get away from (fast, but) inaccurate hardware arithmetic in those cases where it turns out insufficient.

For a detailed introduction into the field of error analysis, the reader is referred to the classic books [833, 834]. Moreover, the topic of stability is extensively treated in [783], devoting several chapters to this issue. Last, but not least, when looking into the subject of floating-point programming, [819] and [343] should be consulted.

3.5.3 Robustness in Computational Geometry

As mentioned in Section 3.5.2, numerical errors, initially considered benign, may well turn into serious ones as soon as they start changing the control flow in a crucial way. Conditional tests delivering the wrong result impose erring branches the program runs through during computation. Whereas some algorithms are actually immune against such wrong decisions, other algorithms may be highly sensitive to them. This sensitivity in particular arises in the field of computational geometry, as we will see shortly. Afterwards, we will discuss different approaches to cope with the problem of inaccuracy.

Problems that are considered to be of *geometric* nature generally have one property in common. The given input and the desired output are supposed to

consist of a combinatorial and a numerical part each. A geometric algorithm solves a given geometric problem if for any given input it computes the output as specified by the problem definition. The algorithm involves so-called *geometric primitives* to progressively transform the given input into the desired output. These fundamental operations are basically each of one of two kinds: *Geometric constructions* create single basic geometric objects out of a constant number of given defining basic objects. *Geometric predicates* in turn test a specific relationship between, again, a constant number of given basic geometric objects. In doing so, they provide a mechanism for querying decisive properties in conditional tests that direct the control flow in geometric algorithms.

Each geometric predicate can be considered to perform in its very last step a comparison of two numbers, which in turn are determined based on the numerical data of the involved geometric object and possibly additional constants. Without loss of generality, one may assume that the second of both numbers is actually zero, which means that a geometric predicate at the bottom line amounts to determining the sign of the value of some arithmetic expression.

The crucial point now is the following. Geometric algorithms are usually designed and proven to be correct in the context of a model that assumes exact computation over the set of all real numbers. In implementations however, this exact real arithmetic is mostly replaced by some fast but finite precision arithmetic as provided by the hardware of the computing device. For some few types of problems with restricted inputs this approach works out well. However, for most of the geometric algorithms, if simply implemented this way, one would have to face adverse effects caused by this finite approximation, which for some critical input could finally result in catastrophic errors in practice.

The reason is that due to the lack of an exact arithmetic, the predicates do not necessarily always deliver the correct answer, but may err if the computed approximation happens to yield the wrong sign. In consequence, the algorithm will branch incorrectly, which in the most lucky case may be masked by some later computation. If not, the algorithm will in the best case compute some combinatorially incorrect or even topologically impossible result. In the worst case, however, an inconsistent state will be entered that causes the algorithm to crash or loop forever.

A simple example of geometrically impossible situation can be observed in the context of computing the intersection point p of two lines l_1 and l_2. Computing $p = l_1 \cap l_2$ and subsequently testing whether p lies on l_i, $i = 1, 2$, both with limited precision floating-point arithmetic, will most of the times result in at least one of the two tests to fail. Schirra [705] points out that even for obviously well-conditioned constellations the intersection point only rarely verifies to actually lie on both lines. Of course, one may argue that these failing tests may just result from p not being exactly representable within the limited precision in most of the cases. However, as it turns out, a direct floating point implementation even fails to always determine correctly for each of the two defining lines, on which side the computed point p actually lies w. r. t. line. Kettner et al. [471] explain in detail, but so that it can be readily understood, how and why an erring sideness

test may cause even the most simple convex hull algorithm to fail in various ways.

Similar robustness problems apply for practically most of the algorithms in Computational Geometry. They arise because the approximate substitute for real number arithmetic used in practice just does not behave exactly like its counterpart in theory, i. e., the real RAM model based on which algorithms and data structures were initially designed and proven to be correct. However, not all input data needs to be considered critical. For most of the configurations representable in the available finite precision format, a direct implementation of a geometric predicate will indeed deliver the correct sign. In fact, the approximate arithmetic may fail to yield the correct sign basically only in those situations, where the given configuration is somewhat close to a configuration for which an exact evaluation of the predicate would report zero. The latter configurations are commonly called *degenerate*. Hence, it is the true and near-degenerate configurations that make up the critical inputs.

These scenarios are the ones that call for approaches to deal with numerical inaccuracy. Recalling that the root of the whole issue was the assumption in theory to being able to compute exactly, but the insufficiency of plain hardware arithmetic to do so in practice, there are two obvious ways out: (a) adapt practice to fit the theory, i. e., compute exactly in practice; or (b) adapt theory to fit the practice, i. e., take imprecision into account during design.

Adapt Practice: The Exact Geometric Computation Paradigm. If we are asked for some way to guarantee that our geometric algorithm in practice will always deliver correct results, the most obvious solution would be to ensure that any numerical computation ever performed is actually (numerically) exact. When resorting for some kind of exact arithmetic, correct results are thus automatically achieved. In fact, in such a case, robustness is actually a non-issue. Chapter 6 discusses such approaches in detail, how to compute exactly within the field of rational numbers or algebraic numbers, respectively. However, it should be noted that, in general, off-the-shelf use of exact arithmetic packages may become quite expensive and should therefore be employed cautiously.

Now, in order to make the algorithm behave in practice just as in theory, we do not necessarily need numerical exactness all the time. Instead, all we need to assure is that the program flows are identical in both cases. In fact, this will guarantee a correct combinatorial part of our output. To achieve this, we request that any decision is always made in the same way as if it was done on a real RAM. As mentioned above, the predicates that are evaluated during the branching steps of the algorithm may w. l. o. g. be assumed to deliver just the sign of some arithmetic expression over the numerical values of the geometric objects involved (and possibly additional constants). In effect, what it takes to always guarantee correct decisions is to (a) have a suitable representation for any object which is always sufficient to (b) compute the correct sign of the expression for the inquired predicate. Since compliance with these two requirements enables

the geometric algorithm to always compute the correct geometric result, this postulation is called the *exact geometric computation* paradigm.

The term *suitable representation* already suggests that we do not necessarily seek for numerical exactness. So, (a) and (b) can be achieved in further ways, other than exact arithmetic. The applicability of these approaches, however, depends on the class of problem to be solved – more precisely, on the kind of arithmetic operations that are required and the maximum *depth of derivation* for any value that may occur during computation.

In short, a problem that can be solved by using solely the four basic arithmetic operations $+, -, *, /$ is called *rational*. Similarly, one that does not require more than algebraic primitives is called *algebraic*. Moreover, following Yap [845], one may inductively define the *depth* d of a value: given a set U of numbers, x is of depth 0, if $x \in U$; and x is of depth at most $d+1$ if it is obtained by applying one of the rational operations to numbers of depth at most d, or by root extraction from a degree k polynomial with coefficients of degree at most $d - k + 1$. An algebraic problem now is said to be *bounded-depth*, if there exists an algorithm that does not impose any value of more than some fixed depth d, otherwise it is called *unbounded-depth*.

Unbounded-depth problems cannot reliably be solved without employing exact arithmetic. However, such problems are rather rare in traditional Computational Geometry [846]. One example of this kind is a solid polyhedral modeler allowing us to perform rational transformations and Boolean operations on solids. There, each such transformation and operation inherently may increase the depth of derivation. But due to the lack of a well-defined input-output-relation, this kind of problem is often not considered a "computational problem" in algorithmics.

Problems that are bounded-depth but not rational may go beyond an off-the-shelf use of standard (arbitrary precision) rational arithmetic. They require techniques for determining the root of a polynomial of bounded degree, which will be explained in Chapter 6. In contrast, *rational bounded-depth* problems, in short *RBD*, can be solved without arbitrary precision arithmetic. In fact, for any given RBD algorithm there is a constant D such that as long as the input is known to involve only rational numbers of size (at most) s, all intermediate computations involve only rational numbers of size at most $D \cdot s + O(1)$. This fact allows us to limit the needed precision in the context of specific applications if the input precision is known in advance. And indeed, in most of the applications, the input is given as either integers of fixed maximum length or in floating-point format with fixed-precision. A few examples shall be given now.

Many geometric predicates used in prevalent geometric algorithms can be expressed by computing the sign of some determinant. Common representatives include the orientation test in two- or three-dimensional space, the in-circle test in 2D as well as the in-sphere test in 3D, the intersection test of lines in 2D, etc. Not surprisingly, a lot of effort has been done on computing the exact sign of determinants.

Concerning matrices with integral entries of bounded bit length, different authors have proposed algorithms for computing the sign of a determinant. The approaches vary from specializations for 2×2 and 3×3 matrices [52] to general $n \times n$ matrices [177,137] and cover standard integer as well as modular arithmetic [138]. In each case, a bound on the required bit length for the arithmetic is given.

Besides these algorithms for integral instances, further algorithms were suggested for computing the exact sign of a determinant for a matrix given in floating-point format. The ESSA algorithm ("exact sign of sum algorithm") due to Ratschek and Rokne [665] computes the sign of a finite sum for double precision floating-point values. Since the determinant of a 2D orientation test is representable as a sum $\Sigma x_k y_l$ over the coordinates of the three points involved, ESSA can compute the orientation of three points if their coordinates are given in single-precision floating-point format (which simply guarantees that all $x_k y_l$ are exactly computable in double-precision). Shewchuk [731] offers a method for adaptively computing exact signs of matrices of size up to 4×4 with entries in double precision floating-point format, provided that neither overflow nor underflow occurs.

Apart from the sign of a determinant, also its actual value may be of interest sometimes. Whereas testing two lines in 2D for intersection only amounts to determining just the sign of 3×3 determinants, computing the intersection itself amounts to determining their actual values. Hoffmann [401] shows how to compute such an intersection point for two lines in parametric form based on the *exact inner product* and derives a bound on the bit length of the homogeneous coordinates of this point. Sugihara and Iri [763] introduced an algorithm for exactly computing polyhedral intersections. In this method, geometric elements are represented without redundancy, giving only the coefficients of the parametric plane equations of the faces. The key property of Sugihara and Iri's method is that neither edges nor vertices are computed explicitly. Instead, all primitives are represented topologically. Vertices are represented as intersections of three planes, edges by their two endpoints and finally faces by delimiting edge loops. Two important observations shall not stay unmentioned: First of all, the representable polyhedra are not restricted to convex polyhedra only. And second, no digit proliferation takes place when intersecting the polyhedra, since the resulting polyhedron always inherits its surfaces from the two input polyhedra. However, as noted by Hoffmann [401], Sugihara and Iri's proposed approach unfortunately lacks support for exactly representing rotations, which is due to the fact that plane coefficients are not being exactly representable anymore. However, Sugihara and Iri represent their polyhedra in a dual form, based on a CSG (constructive solid geometry) tree of trihedral polyhedra and a history recording the boundary structure for the Boolean operations. Then, rotation of a complex polyhedron is performed by first rotating the trihedral primitives and then reconstructing the rotated polyhedron from the CSG representation.

Adapt Theory: Design for Inaccuracy. When having to rely on potentially inaccurate computations, one has to resign from the assumption of getting exact

results to base the decisions on. However, an algorithm may still be quite suitable as long as it guarantees to deliver the exact result for a problem that is (in some sense) close to the original one. This motivates the following definition of robustness which was omitted in this section until now, since in the scope of the exact geometric computation paradigm, robustness was actually a non-issue.

Following Fortune [295], a geometric algorithm shall be called *robust*, if it produces the correct result for some perturbation of the input. In close connection to Section 3.5.2, we want to call the algorithm *stable* if this perturbation is small. Moreover, Shewchuk [731] suggests calling an algorithm *quasi-robust* if it computes some useful information, even though not necessarily a correct output for any perturbation of the input.

Interval Geometry. These approaches can be seen as the geometric counterpart to interval arithmetic (cf. Chapter 6). Numerical inaccuracy is treated by systematically thickening the geometry of processed objects, or by adjusting the geometric meaning of the predicates. The *tolerance-based approach* due to Segal and Sequin [720, 719] associates tolerance regions to geometric objects. The challenge now is to always keep the data in a consistent state. To achieve this consistency and to obtain correct predicates on these "toleranced objects" they enforce a *minimum feature separation*. Features that are too close to each other (i. e., have overlapping tolerance regions) must either be shrunken (by re-computation with higher precision), merged or split. Each of these actions might require backtracking if the new tolerance region of one object happens to start or stop overlapping the tolerance region of an object that has already been processed. In order to enable this kind of consistency checking, tolerance-based approaches usually maintain additional neighborhood information.

Epsilon Geometry due to Guibas, Salesin and Stolfi [688] treats the problem of uncertainty due to numerical inaccuracy from the other side, namely the geometric predicates. An epsilon-predicate returns a real number ε that reflects, how much the input satisfies the predicate. A non-positive outcome states that the input could successfully be verified to satisfy the predicate and, moreover, that the predicate would even be satisfied when perturbing the input by not more than ε. A positive ε in turn states that the input could not be verified to satisfy the predicate. However, ε is the size of the smallest perturbation that would actually produce an input satisfying the predicate. Unfortunately, reasoning in this framework seems to be difficult [704], and until now epsilon geometry has been applied successfully only to a few basic geometric problems, cf. [688, 364].

Axiomatic Approach. Another quite tempting approach was proposed by Schorn [708, 709]. The key idea of what he calls the *axiomatic approach* is to determine properties of primitive operations that are sufficient for performing a correctness proof of an algorithm, and to find invariants that solely base on these properties. Schorn applies his axiomatic approach to the problem of computing a closest pair within a set of points in 2D, but also to the problem of finding pairs of intersecting line segments.

In the former case, he introduces some abstract functions d, d_x, d_y, d'_y of type $(\mathbb{R}^2 \times \mathbb{R}^2) \to \mathbb{R}$ as substitutes for $\|p - q\|$, $p_x - q_x$, $p_y - q_y$, and $q_y - p_y$, respectively. Then he lists some properties that these functions shall fulfill: First, d is to be symmetric and furthermore an upper bound for each of the functions d_x, d_y, d'_y. And second, the functions d_x, d_y, d'_y need to feature some monotonicity properties. Schorn proves that based on these *axioms* his sweep algorithm is guaranteed to compute $\min_{s,t \in P} d(s,t)$ for the given point set P – no matter what d, d_x, d_y, d'_y are, as long as they satisfy the postulated axioms. He also shows that floating-point implementations of the substituted exact distance functions from above would yield a guaranteed relative forward error of at most 8ε, for ε being the machine epsilon.

Consistency Approach. The consistency approach implements robustness by simply demanding that no decisions are contradictory. The requirement of *correct* decisions is weakened towards *consistent* ones. In fact, as long as they are consistent with all other decisions, also incorrect decisions are tolerable. Of course, in the lucky case that an algorithm performs only tests that are completely independent of previous results, it would always deliver consistent results, even for random outcomes for the predicates. Fortune [295] calls such algorithms *parsimonious*. He presumes that – in principle – many algorithms should be capable of being made parsimonious.

An algorithm that is not parsimonious needs to assure consistency in another way. The *topology-oriented approach* due to Sugihara and Iri [765] puts highest priority on topology and combinatorial structure. Whenever a numerical computation would entail a decision violating the current topology, this decision is substituted by a consistent one that actually conforms to the topology. Degeneracies are not treated explicitly. If the sign of a predicate evaluates to zero, it is replaced by a positive or negative one, whatever is consistent. Sugihara's approach ensures topological consistency throughout the whole computation, in particular for the final result. However, there is no guarantee for obtaining topologically *correct* results, and the numerical values computed may be noticeably far away from correct. Usually it is argued at this point that computing with higher precision will make the output getting closer to the correct result – finally being equal to it, once the precision is sufficient. Yet, this argument only holds as long as no true degeneracies are involved. Irrespective thereof, the topology-oriented approach has still proven capable of leading to amazingly robust algorithms. In fact, Sugihara et al. presented several algorithms for polyhedral modeling problems (see p. 117, but also [762]), for computing Voronoi diagrams [764, 619, 765] and for determining the convex hull in 3D [575]. The reader is also referred to Chapter 9, devoting a whole section to Voronoi diagrams, including a topology oriented implementation due to Held [385].

Milenkovic proposes an approach, called the *hidden variable method* [572], which is based on two components: a structure with topological properties and a finite approximation of the numerical values. The topological structure is chosen in such a way that there exist infinite precision numerical values (close to the given finite precision parameter values), for which the problem has the

chosen structure. The name *hidden variable method* derives from the fact that the topology of the infinite precision version is known but not its numerical values. In [572], the approach is applied to the computation of line arrangements.

Adapting Input. Milenkovic [572] also proposes a second approach to deal with numerical inaccuracy. His technique called *data normalization* modifies the input in such a way that it will finally be able to be processed with approximate arithmetic. It basically constitutes a preprocessing procedure, in which incidence is enforced for features that are too close together and the subsequent algorithm may not be able to tell apart. After eliminating any near coincidence this way, it is guaranteed that all the finite precision operations performed by the algorithm will yield correct results based on this *normalized* input. Milenkovic exemplifies his approach on the problem of polygonal modeling, based on the two operations permitted to be performed on the input: *vertex shifting*, which merges two vertices that are closer than ε, and *edge cracking*, which subdivides an edge into a poly-edge at all vertices that are too close to that edge. Clearly, introducing incidences definitely changes the local topological structure of the input in terms of connectivity. Yet, Milenkovic proves bounds on the maximum positional displacement introduced by his method.

Controlled perturbation is another approach that adapts the input so as to make it pleasant enough for being processed with approximate arithmetic. Again, one concedes the problem at hand not necessarily to be solved for the given input but for some nearby input. The basic procedure is yet somewhat different to the *data normalization* technique. The idea is to run the algorithm in a supervised manner, and to monitor if any of the predicate invocations is once not guaranteed to deliver a reliable result. In the latter case, the current pass is interrupted, the original input is perturbed and the algorithm is restarted again on the perturbed input. This protection is achieved by augmenting each of the predicate calls with a so-called *guard*. A guard G_E for a geometric predicate E is a Boolean predicate that, if evaluating to true in the given approximate arithmetic, guarantees E to yield the correct sign when evaluated in the same arithmetic. By increasing the size of the perturbation after each unsuccessful run, one increases the probability that none of the guards evaluates to false. However, the larger the perturbation, the less nearby is the finally solved input instance to the original one. Controlled perturbation, as proposed by Halperin et al. [378], therefore calls for controlling over the size of perturbation by means of choosing the perturbed input in a careful manner. Halperin et al. use this technique to compute arrangements of spheres [378], arrangements of polyhedral surfaces [654], and arrangements of circles [376], each in floating point arithmetic. Klein [474] applies the paradigm to the computation of Voronoi diagrams. Finally, Funke et al. [311] compute Delaunay triangulations using controlled perturbation. In each case, the authors derive a relation between the perturbation amount and the quality guarantee of the approximate arithmetic, i. e., the precision of the floating point system intended to be used.

Funke et al. point out that controlled perturbation (as opposed to data normalization) is actually a general conversion strategy for idealistic algorithms

designed for the real RAM model and some general position assumption. In addition to this original approach, Klein [474] and Funke et al. [311] also consider so-called *lazy controlled perturbations*, which perturb only those sites that during their incremental insertion caused one of the involved guards to fail. Unfortunately, neither the perturbation bound nor the expected running time could be shown to carry over from the standard scenario.

Representation and Model Approach. This approach is probably the most abstract one to deal with numerical inaccuracy. Basically, an explicit distinction is drawn between mathematical objects, the *models*, on the one hand and their corresponding computer *representations* on the other. Based on this distinction, a geometric problem $\mathcal{P}$ is considered to define a mapping $\mathcal{P} : \mathcal{I} \to \mathcal{O}$ from a set $\mathcal{I}$ of input models into a set $\mathcal{O}$ of output models. In comparison, a computer program $\mathcal{A}$ imposes a mapping $\mathcal{A} : \mathcal{I}_{rep} \to \mathcal{O}_{rep}$ on corresponding sets $\mathcal{I}_{rep} \supseteq \{rep_{\mathcal{I}}(i) | i \in \mathcal{I}\}$ and $\mathcal{O}_{rep} \supseteq \{rep_{\mathcal{O}}(o) | o \in \mathcal{O}\}$ of computer representations.

A computer program $\mathcal{A}$ for a problem $\mathcal{P}$ will be called *correct*, if it holds that $rep_{\mathcal{I}} \circ \mathcal{A} \circ rep_{\mathcal{O}}^{-1} = \mathcal{P}$, i. e., if for any $i \in \mathcal{I}$ we have $rep_{\mathcal{O}}^{-1}(\mathcal{A}(rep_{\mathcal{I}}(i)) = \mathcal{P}(i)$. Obviously, this requires $rep_{\mathcal{I}}$ and $rep_{\mathcal{O}}$ both to be bijections. In other words, it would take a one-to-one correspondence between representations and models. However, because of the infinite character of most mathematical models on the one hand and the finite nature of computer representations on the other, the correspondence between the two is normally not one-to-one.

Taking this issue into account, the term correctness is therefore replaced by robustness as follows: A computer program $\mathcal{A} : \mathcal{I}_{rep} \to \mathcal{O}_{rep}$ for a problem $\mathcal{P} : \mathcal{I} \to \mathcal{O}$ will be called *robust*, if for every computer representation $i_{rep} \in \mathcal{I}_{rep}$ there is a corresponding model $i \in \mathcal{I}$ such that $\mathcal{P}(i) \in \mathcal{O}$ is among the models corresponding to $\mathcal{A}(i_{rep}) \in \mathcal{O}_{rep}$ – or formally, if for any $i_{rep} \in \mathcal{I}_{rep}$ we have $\{\mathcal{P}(i) | i \in \mathcal{I},\ rep_{\mathcal{I}}(i) = i_{rep}\} \cap \{o \in \mathcal{O} | rep_{\mathcal{O}}(o) = \mathcal{A}(i_{rep})\} \neq \emptyset$. So, in order to prove that a given computer program is robust in this terminology, one basically has to show that for any given representation there always exists a model for which the computer program takes the correct decisions.

Admittedly, this definition of robustness allows a fairly generous interpretation of the term "correspondence". In particular, if we were to define only a single input representation x and a single output representation y and let $\mathcal{A}$ simply return y for the only possible input x, then $\mathcal{A}$ would be a robust algorithm for any problem $\mathcal{P}$, as all input models are mapped to x and all output models are mapped to y. In fact, this definition of robustness basically rather reflects what Shewchuk [731] suggests to be called quasi-robust.

Hoffmann, Hopcroft, and Karasick [402] introduced this formalization. They also gave an algorithm for intersection of polygons and proved its robustness with respect to their formalism. Hopcroft and Kahn [411] considered robust intersection of polyhedron with a half-space. However, in both cases the interpretation of "correspondence" was actually quite generous, leading to fairly loose relationship between computer representation and its "corresponding" model.

Related Issues. In the remaining part of this subsection, we will look at three issues that are actually closely related to precision-caused robustness problems in computational geometry, namely: inaccurate data, degeneracies and geometric rounding. Each of these issues shall be briefly discussed now in closing.

Inaccurate Data. Approximate arithmetic employed for computation is not the only type of inaccuracy relevant in computational geometry. As Schirra [704] points out, many geometric data arising in practice is actually known or supposed to be inaccurate. Since both types of inaccuracy basically impose a similar kind of uncertainty, there is actually not much of a difference between processing geometric objects that result from inaccurate computations and processing real-world data that is potentially inaccurate by nature.

Now, if we were to directly employ exact geometric computation on this kind of input, we would implicitly treat inaccurate data as exact. This way we would determine the correct result for some possibly inaccurately represented problem instance. Yet, this procedure only works out as long as the given data are actually consistent. Otherwise, we were to face similar problems as discussed in the course of this subsection: The algorithm may happen to enter a state that it was never supposed to be confronted with. In fact, it got actually launched in such a state already. This close relationship was a sufficient natural reason for researchers to address both kinds of inconsistencies in a uniform way. An immediate consequence however is, that any error in the output cannot be identified whether to be caused by inaccuracy of input or during computation.

In order to finally achieve an error-free output, there are two ways to choose between. When sticking to exact geometric computation, one will have to fix the relevant deal of the data – either in advance or on the fly, if possible. Basically, the input needs to be considered as non-benign and asks for being handled in one of the ways discussed in Section 3.5.1. Geometric rounding, as discussed shortly, may turn out one of these possible answers. The second way is to follow one of the approaches of consciously designing for inaccuracy discussed earlier in this section. In particular, tolerance-based and consistency-driven approaches appear naturally promising in this respect.

Degeneracy. As already mentioned in the beginning of this subsection, precision-caused non-robustness is closely related to degeneracy in computational geometry. The nearly degenerate and true degenerate instances are the critical scenarios in computing with numerical inaccuracy. Roughly speaking, degeneracies may be deemed to be points of discontinuity of the input-output-function induced by an algorithm – usually configurations of the input data where one of the predicates involved in the overall computation evaluates to zero, thus entailing a switch-over in the algorithm's control flow.

When designing a geometric algorithm it is common practice to assume the absence of such degeneracies. In fact, in scientific publications authors on a regular base tend to declare them as negligible details "left to the reader". The assumption is usually justified, for in most cases the details can indeed more or less easily be filled up. Yet, the more the phase of implementation approaches,

the less pleasant it becomes from the point of robustness to keep hold of any such void in the algorithm's specification.

Following Yap [848], an algorithm is called *generic*, if it is only guaranteed to be correct on generic (i. e., non-degenerate) input. A *general* algorithm in turn is one that works for all (legal) inputs. In order to avoid the final implementation to crash due to degenerate input that could not be handled, it is desirable for the implementer to being delivered a general algorithm. So, if the initial description did not cover all degeneracies, then at one point in time a generic algorithm asks for being turned into a general one. There are basically two (not mutually exclusive) options for doing so: Adapting the algorithm or modifying the input.

Certainly, when following one of the approaches for computing with inaccuracy discussed earlier in this section, true degeneracy is actually kind of a non-issue. Since they cannot reliably determine true degeneracy anyway, these approaches treat any nearly degenerate case like an untrustworthy outcome of a predicate. For example, tolerance-based approaches simply adapt their tolerance information for any (true or) near degeneracy they hit upon. Consistency approaches in turn do not treat degeneracies explicitly at all. If the sign of a predicate evaluates to zero, it is replaced by a positive or negative one, whatever is consistent. In contrast, the controlled perturbation approach perturbs the input until it does not entail any predicate anymore that evaluates too close to zero. In general, approaches adapting the input (like also Milenkovic's data normalization technique) explicitly remove possible degeneracies.

In contrast to that, when applying exact geometric computation true degeneracy becomes an explicit issue. In fact, exact geometric computation guarantees to always yield the correct sign for each of the involved geometric predicates. However, it does not help us any further, once a predicate happens to correctly evaluate to zero, but the algorithm by design doesn't have an answer at hand how to handle this degenerate situation. Again, one could adapt the algorithm, i. e., extend it in a way that allows for degeneracies to being handled. However, apart from really treating them, there is again the second option, namely bypassing.

Edelsbrunner and Mücke [268] were the first ones to introduce the notion of so-called *symbolic perturbation schemes* into the field of computational geometry. The idea of this concept is to perturb the input in a symbolic way in order to remove degeneracies, but at the same time to obtain a result as close as possible to the real solution. In fact, perturbing only in a symbolic manner is the way to ensure that the perturbation does not change the sign of any non-zero predicate result. Edelsbrunner and Mücke [268] introduced a scheme, called *Simulation of Simplicity* (SoS). This technique, that was already known in the context of the simplex method, amounts to adding powers of some indeterminate ε to each input parameter. Emiris and Canny [273] reduced the computational complexity by applying *linear perturbations* only: to each input parameter x_i they add a perturbations $\pi_i \cdot \varepsilon$ where $\pi_i \in \mathbb{Z}$ and ε infinitesimal.

Yap [845] proposes a slightly more generalized concept called *blackbox sign evaluation schemes*. In this approach, every call to a predicate is generally replaced by a call to a sign blackbox which (a) always returns a non-zero sign and

(b) guarantees to preserve any non-zero sign concluded by the original predicate. Yap [845, 844] shows how to formulate a consistency property for the blackbox and offers a whole family of admissible schemes applicable for polynomial functions. This approach is in particular important, as SoS can be applied to determinants only.

Finally, Seidel [722] proposes an approach, based on the following idea. Given a problem $P(x)$ and just a single non-degenerate input x^*, then every other input x can be made non-degenerate by perturbing it in the direction of x^*, leaving us with computing $P(x + \varepsilon x^*)$. He shows that the other perturbation schemes mentioned above are each special cases of his general approach.

Geometric Rounding. Computing geometrically exact results is only worth as much as the computed result can actually be successfully passed on to subsequent stages. In fact, it does not help much, for example, to determine the exact solution of a geometric problem if the output format is yet not capable of representing it without information loss. There may also be a couple of other reasons (e. g., the computational cost) that call for a reduction of the numerical and/or structural complexity of geometric data.

The goal of *geometric rounding* now is to find a *simplification* of a given geometric structure, i. e., a geometric structure of lower complexity that does not deviate too much from its original with respect to some specific geometric or topological criteria. Thereby, two different objectives may be the driving forces, namely combinatorial versus precision simplification [848]. Whereas the former one aims at reducing the number of primitives and their combinatorial relations, the latter objective seeks for reducing the (bit-)complexity of the numerical values themselves.

Greene and Yao [359] were the first authors to introduce the topic of geometric rounding in the field of computational geometry. They considered the problem of rounding line segments consistently to a regular grid. They suggested to break the line segments into polygonal chains by moving any vertex of the subdivision (i. e., every endpoint of a line segment or intersection point of two segments) to its nearest grid point. Their approach moves edges only by a distance of at most half a grid cell's diameter. It may introduce new incidences, but no additional crossings, which is the property that is meant by "consistently". Unfortunately, it may produce a large number of new vertices along the polygonal chains. Since n segments may show $\Theta(n^2)$ intersections, Greene/Yao rounding may end up with quadratically many additional vertices, compared to initially $O(n)$.

Snap rounding, usually attributed to Hobby [398] and Greene [358], overcomes this issue. The general idea is based on the notion of a *hot pixel*, which also gave raise to the name *hot pixel rounding*. A pixel of the grid is called *hot*, if it either contains an endpoint of an original line segment, or an intersection point of two original line segments. The rounding procedure consists of snapping all segments intersecting a hot pixel to the pixel center. As with Greene/Yao rounding, snap rounding guarantees that the resulting arrangement will be contained within the Minkowski sum of the original arrangement and a unit grid cell centered at the origin.

However, Halperin and Packer [377] showed that a vertex of the output computed that way may be very close to an actually non-incident edge. Since this might induce new potential near-degeneracies, they proposed an augmented procedure, called *iterated snap rounding*, aimed to eliminate the undesirable property. Their rounding basically consists of two stages. In a preprocessing stage they compute hot pixels defined by the vertices of the arrangement. Additionally, they prepare a segment intersection search structure that allows to query for all hot pixels that a given segment s intersects. In a second stage they perform a procedure the call *reroute* on each input segment. This recursive procedure produces a polygonal chain s^* as an approximation for a given segment s, such that when s^* passes through a hot pixel, it passes through its center. Halperin and Packer show that their rounding procedure guarantees that any vertex is at least half the width of a pixel away from any non-incident edge.

Milenkovic [573] proposes a scheme called *shortest path rounding*, which introduces even fewer bends than snap rounding. He defines a *deformation* to be a continuous mapping $\pi : [0,1] \times \mathbb{R}^2 \to \mathbb{R}^2$ such that $\pi(0,p) = p$ for all $p \in \mathbb{R}^2$, and for any fixed $t \in [0,1)$ the function $\pi_t(p) := \pi(t,p)$ is a bijection. (Note that π_1 not necessarily needs to be a bijection – distinct points may collapse at time $t = 1$. However, π_1 is clearly the limit of a series of bijections.) π_t represents the state of the deformation at time $t \in [0,1]$. In comparison, $\gamma_p(t) := \pi(t,p)$ reflects the path that p travels through during the whole process of deformation, starting at p and ending at the target position $\rho(p) := \pi(1,p) = \pi_1(p) = \gamma_p(1)$. A geometric rounding of a straight line embedding $G = (V,E)$ to a lattice S is then a deformation of the plane such that the following two properties hold:

(a) For any $v \in V$, γ_v is completely contained in $CELL(S,v)$, i. e., the deformation path of any vertex $v \in V$ always stays within the lattice cell corresponding to v.
(b) Each $(u,v) \in E$ is deformed into a polygonal chain having its vertices in lattice points of vertices of V only.

Such a geometric rounding is called a *shortest path rounding* if every rounded edge results in a polygonal chain with shortest possible paths among all (feasible) roundings. Milenkovic shows that the result of a shortest path rounding is always unique.

Apart from 2-dimensional arrangements and planar subdivisions, geometric rounding has also been studied in 3D. As already mentioned in Section 3.5.3, Sugihara and Iri [763] apply what may be called *CSG rounding* to a geometric object by first rounding all involved CSG primitives and subsequently reconstructing the tree. Fortune [296] in turn rounds geometric objects given in manifold representation. This *manifold rounding* works by first rounding the equations or faces and afterwards, in case the rounded solid is self-intersecting, retaining only the "unburied" portion of the boundary.

In general, rounding geometric data is far more than just rounding numbers, and doing it properly can be very difficult. In fact, the quest for reducing complexity significantly while always keeping the numerical and combinatorial data consistent may turn out highly complicated.

Final Remarks. As we have seen in the second part of this section, computing with inaccuracy obviously seems to impair problems for geometric algorithms. In fact, as pointed out by Fortune [297], it is in general "notoriously difficult to obtain a practical implementation of an abstractly described geometric algorithm". In effect, the number of problems that have been successfully attacked this way is still small (see [524, 401, 704] for surveys on robustness issues in geometric computation). In fact, these approaches entail two major disadvantages: First of all, the respective techniques are highly specific to the considered problem and do hardly generalize to other geometric problems. And second, they do not permit off-the-shelf use of all the various geometric algorithms already available, but require a redesign of practically every single algorithm intended to be used. In short: They force us to redo Computational Geometry [141].