

Algorithm Engineering: Concepts and Practice*

Markus Chimani and Karsten Klein

Chair of Algorithm Engineering, TU Dortmund, Germany
markus.chimani@uni-jena.de[†] karsten.klein@tu-dortmund.de

Abstract

Over the last years the term *algorithm engineering* has become wide spread synonym for experimental evaluation in the context of algorithm development. Yet it implies even more. We discuss the major weaknesses of traditional “pen and paper” algorithmics and the ever-growing gap between theory and practice in the context of modern computer hardware and real-world problem instances. We present the key ideas and concepts of the central *algorithm engineering cycle* that is based on a full feedback loop: It starts with the design of the algorithm, followed by the analysis, implementation, and experimental evaluation. The results of the latter can then be reused for modifications to the algorithmic design, stronger or input-specific theoretic performance guarantees, etc. We describe the individual steps of the cycle, explaining the rationale behind them and giving examples of how to conduct these steps thoughtfully. Thereby we give an introduction to current algorithmic key issues like I/O-efficient or parallel algorithms, succinct data structures, hardware-aware implementations, and others. We conclude with two especially insightful success stories—shortest path problems and text search—where the application of algorithm engineering techniques led to tremendous performance improvements compared with previous state-of-the-art approaches.

***Preprint!** For the authoritative version see: M. Chimani and K. Klein. *Algorithm Engineering: Concepts and Practice*. Chapter in: *Experimental Methods for the Analysis of Optimization Algorithms*. T. Bartz-Beielstein, M. Chiarandini, L. Paquete, M. Preuss, Eds. (to appear 2010)

[†]Current affiliation: Algorithm Engineering, Friedrich-Schiller-University Jena, Germany. Juniorprofessorship funded by the Carl-Zeiss-Foundation.

1 Why Algorithm Engineering?

“Efforts must be made to ensure that promising algorithms discovered by the theory community are implemented, tested and refined to the point where they can be usefully applied in practice. [...] to increase the impact of theory on key application areas.”

[Aho et al. [1997], *Emerging Opportunities for Theoretical Computer Science*]

For a long time in classical algorithmics, the analysis of algorithms for combinatorial problems focused on the theoretical analysis of asymptotic worst-case runtimes. As a consequence, the development of asymptotically faster algorithms—or the improvement of existing ones—was a major aim. Sophisticated algorithms and data structures have been developed and new theoretical results were achieved for many problems.

However, asymptotically fast algorithms need not be efficient in practice: The asymptotic analysis may hide huge constants, the algorithm may perform poorly on typical real-world instances, where alternative methods may be much faster, or the algorithm may be too complex to be implemented for a specific task. Furthermore, modern computer hardware differs significantly from the *von Neumann* model (1993 republication) that often forms the basis of theoretical analyses. For use in real-world applications we therefore need robust algorithms that do not only offer good asymptotic performance but are also designed and experimentally evaluated to meet practical demands.

The research field of *algorithm engineering* copes with these problems and intends to bridge the gap between the efficient algorithms developed in algorithmic theory and the algorithms used by practitioners. In the best case, this may lead to algorithms that are asymptotically optimal and at the same time have excellent practical behavior.

An important goal of algorithm engineering is also to speed up the transfer of algorithmic knowledge into applications. This may be achieved by developing algorithms and data structures that have competitive performance but are still simple enough to be understood and implemented by practitioners. Additionally, free availability of such implementations in well-documented algorithm libraries can foster the use of state-of-the-art methods in real-world applications.

1.1 Early Days and the Pen-and-Paper Era

In the early days of computer algorithms, during the 1950s and 1960s, many pioneers also provided corresponding code for new algorithms. The classic and timeless pioneering work by Donald Knuth [1997], first published in 1968, gives a systematic approach to the analysis of algorithms and covers the aspect of algorithm implementation issues in detail without the restriction to a specific

high-level language. Knuth once condensed the problems that arise due to the gap between theory and practice in the famous phrase

“Beware of bugs in the above code; I have only proved it correct, not tried it.”
[D. E. Knuth]

During the following two decades, which are often referred to as the “pen-and-paper era” of algorithmics, the focus shifted more towards abstract high-level description of new algorithms. The algorithmic field saw many advances regarding new and improved algorithms and sophisticated underlying data structures. However, implementation issues were only rarely discussed; no implementations were provided or evaluated.

This led to a situation where on the one hand algorithms that may have been successful in practice were not published, as they did not improve on existing ones in terms of asymptotic runtime while on the other hand, the theoretically best algorithms were not used in practice as they were too complex to be implemented.

1.2 Errors

This situation had another side-effect: When algorithms are theoretically complex and never get implemented, errors in their design may remain undetected. However, the appearance of errors is inevitable in scientific research, as stated in the well-known quote

“If you don’t make mistakes, you’re not working on hard enough problems.”
[F. Wikek]

Indeed there are a number of prominent examples: In 1973, Hopcroft and Tarjan [1973] presented the first linear-time algorithm to decompose a graph into its tri-connected components, an algorithmic step crucial for graph-theoretic problems that build upon it. However, their description was flawed and it was not until 2001 [Gutwenger and Mutzel, 2001]—when the highly complex algorithm was first implemented—that this was detected and fixed.

Another prominent example that we will revisit later in a different context is *planarity testing*, i.e., given a graph, we ask if it can be drawn in the plane without any crossings. Already the first algorithm [Auslander and Parter, 1961] (requiring cubic time) was flawed and fixed 2 years later [Goldstein, 1963]. It was open for a long time whether a linear-time algorithm exists, before again Hopcroft and Tarjan [1974] presented their seminal algorithm. If we indeed have a planar graph, we are usually—in particular in most practical applications—interested in an embedding realizing a planar drawing. Hopcroft and Tarjan only sketched how to extract such an embedding from the data structures after the execution of the test, which Mehlhorn [1984] tried to clarify with a more detailed

description. Overall, it took 22 years for a crucial flaw—that becomes apparent when one tries to implement the algorithm—to be detected (and fixed) in this scheme [Mehlhorn and Mutzel, 1996].

Runtime complexity versus runtime

The running time is the most commonly used criterion when evaluating algorithms and data structure operations. Here, asymptotically better algorithms are typically preferred over asymptotically inferior ones. However, usual performance guarantees are valid for all possible input instances, including pathological cases that do not appear in real-world applications. A classic example that shows how theoretically inferior algorithms may outperform their asymptotically stronger competitors in practice can be found in the field of mathematical programming: The most popular algorithm for solving linear programming problems—the simplex algorithm introduced by Dantzig in 1947—has been shown to exhibit exponential worst-case time complexity [Klee and Minty, 1972], but provides very good performance for most input instances that occur in practice. In the 1970s, the polynomial-time solvability of linear programs was shown using the ellipsoid method [Khachiyan, 1979], and promising polynomial alternatives have been developed since then. However, the simplex method and its variants dominated for decades due to their superior performance and are still the ones most widely used.

In practice, the involved constants—that are ignored in asymptotic analyses—play an important role. Consider again the history of planarity testing algorithms: the first linear time algorithm was presented in the 1970s [Hopcroft and Tarjan, 1974]; since this is the best possible asymptotical runtime, one could assume that the topic is therefore closed for further research. However, still nowadays new algorithms are developed for this problem. While the first algorithms were highly complex and required special sophisticated data structures and subalgorithms, the state-of-the-art algorithms [de Fraysseix and Ossona de Mendez, 2003, Boyer and Myrvold, 2004] are surprisingly simple and require nothing more than *depth first search* (DFS) traversals and ordered adjacency lists. Additionally, and also since they are comparably easy to implement, these new algorithms are orders of magnitude faster than the first approaches [Boyer et al., 2004].

An even more extreme example can be found in algorithms based upon the seminal graph minor theorem by Robertson and Seymour, discussed at length in the series *Graph Minors I–XX*. The theorem states that any minor-closed family of graphs is characterized by a finite *obstruction set*, i.e., a set of forbidden graph minors. In particular, the nonconstructive proof tells us that we can decide whether a given graph contains a fixed minor in cubic time. There are several

graph problems, in particular many *fixed parameter tractable* (FPT) problems, that can be formulated adequately to use this machinery and obtain a polynomial algorithm. However, observe that the obstruction set, even when considered fixed, may in fact be very large and nontrivial to obtain. This leads to conceptual algorithms that, although formally polynomial, are of little to no use in practice.

An example is the FPT algorithm for the graph genus—i.e., to decide whether a given graph can be embedded on a surface of fixed genus g . The theorem tells us that this can be tested in polynomial time. However, even for the case $g = 1$ —i.e. whether a graph can be drawn on the torus without crossings—the exact obstruction set is still unknown and has at least 16,629 elements, probably many more [Gagarin et al., 2005].

Generally, even though better asymptotic running time will typically pay off with growing input size, there are certain thresholds up to which a simpler algorithm may outperform a more complicated one with better asymptotic behavior. The selection of the best algorithm has to be made with respect to the characteristics of the input instances in the considered practical applications.

Traditionally, algorithmic analysis focuses on the required running times, while space consumption plays a second-order role. With the advent of mass data analysis, e.g., whole genome sequencing, the latter concept gains importance. This is further aggravated by the fact that the increasing number of levels in the memory hierarchy leads to drastically different running times compared with what would be assumed based on the von Neumann model, see Sect. 3.2.

2 The Algorithm Engineering Cycle

There are multiple competing models that describe how software should be designed and analyzed. The most traditional one is the *waterfall model*, which describes the succession of design, implementation, verification, and maintenance as a sequential process. The algorithm engineering cycle—originally proposed by Sanders et al. [2005]—diverges from this view as it proposes multiple iterations over its substeps, cf. Fig. 1. It focuses not so much on software engineering aspects but rather on algorithmic development.

We usually start with some specific *application* (1) in mind and try to find a *realistic model* (2) for it such that the solutions we will obtain match the requirements as well as possible. The main cycle starts with an initial algorithmic *design* (3) on how to solve this model. Based on this design we *analyze* (4) the algorithm from the theoretical point of view, e.g., to obtain *performance guarantees* (5) such as asymptotic runtime, approximation ratios, etc. These latter two steps—to find and analyze an algorithm for a given model—are essentially the steps traditionally performed by algorithmic theoreticians in the pen-and-paper

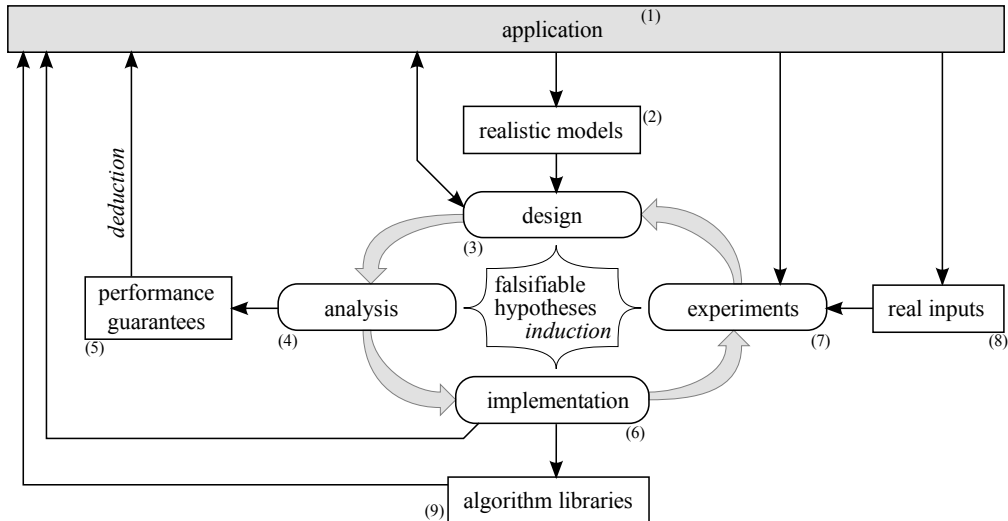


Figure 1: The algorithm engineering cycle

era.

We proceed with our algorithmic development by *implementing* (6) the proposed algorithm. We should never underestimate the usefulness of this often time-consuming process. It forms probably the most important step of algorithm engineering. First of all, we can only succeed in this step if the algorithm is reasonable in its implementation complexity. As noted before, often purely theoretical algorithms have been developed in the past, where it is unclear how to actually transform the theoretical ideas into actual code. Secondly, and in stark contrast to the assumption that theoretical proofs for algorithms are sufficient, it has often been the case that during this step certain flaws and omissions became obvious. As noted before, there have been papers which take theoretical algorithms and show that the runtime analysis is either wrong, because of some oversight in estimating the complexity of look-ups, or that the algorithmic description has to be extended in order to achieve the suggested running time.

Furthermore, algorithms may assume some static, probably preprocessed, data structure as its input, where certain queries can be performed asymptotically fast. However, in many real-world scenarios such input has to be stored in a more flexible format, e.g., in a linked list instead of an array. When thinking about graphs, this is even more common, as we will often store them as a dynamic graph with adjacency lists, or as compressed or packed adjacency matrices. In such data structures edge look-ups cannot be easily performed in constant time, as for a full adjacency matrix. However, the full unpacked matrix, which may

be required by an algorithm to achieve optimal runtime performance, may simply be too large to be stored in practice. In such cases, alternative algorithms, which may be asymptotically slower under the assumption of constant-time edge look-ups but require fewer look-ups, may in fact be beneficial.

Such observations can also be made by corresponding *experiments* (7), which are the final major piece in our cycle. In particular, we are not interested in toy experiments, but in ones considering *real-world data* (8). Such input instances have to be at least similar to the problems the algorithm is originally designed for. The benefit of this step is manifold. It allows us to compare different algorithms on a testing ground that is relevant for the practice. We can better estimate the involved constants that have been hidden in the asymptotic runtime analysis, and thereby answer the question of which algorithm—even if asymptotically equivalent—is beneficial in practice. We may also find that certain algorithms may behave counter intuitively to the theoretical investigation: analytically slower algorithms may be faster than expected, either due to too crudely estimated runtime bounds or because the worst cases virtually never happen in practice. Such situations may not only occur in the context of running time, but are also very common for approximation algorithms, where seemingly weaker algorithms—probably even without any formal approximation guarantee—may find better solutions than more sophisticated algorithms with tight approximation bounds.

It remains to close the main algorithm engineering cycle. Our aim is to use the knowledge obtained during the steps of analysis, implementation, and experimentation to find improved algorithmic designs. E.g., when our experiments show a linear runtime curve for an algorithm with a theoretically quadratic runtime, this may give a hint for improving the theoretical analysis or that it is worthwhile to theoretically investigate the algorithm’s average running time. Another approach is to identify bottlenecks of the algorithm during the experimentation, probably by profiling. We can then try to improve its real-world applicability by modifying our implementation or—even better—the algorithmic design.

Generally, during the whole cycle of design, analysis, implementation, and experimentation, we should allow ourself to also work on the basis of hypotheses, which can later be falsified in the subsequent steps.

A final by-product of the algorithm engineering cycle should be some kind of *algorithm library* (9), i.e., the code should be written in a reusable manner such that the results can be later used for further algorithmic development that builds upon the obtained results. A (freely available) library allows simpler verification of the academic findings and allows other research groups to compare their new results against preceding algorithms. Finally, such a library facilitates one of the main aspects of algorithm engineering, i.e., to bridge the gap between

academic theory and practitioners in the nonacademic field. New algorithmic developments are much more likely to be used by the latter, if there exists a well-structured, documented reference implementation.

During the steps of this cycle, there are several issues that come up regularly, in particular because modern computers are only very roughly equivalent to the von Neumann machine. In the following section we will discuss some of these most common issues, which by now have often formed their own research fields.

3 Current Topics and Issues

When we move away from simple, cleanly defined problems and step into the realm of real-world problems, we are faced with certain inconsistencies and surprises compared with what theoretic analysis has predicted. This does not mean that the theory itself is flawed but rather that the considered underlying models do not exactly reflect the problem instances and the actual computer architecture. The inherent abstraction of the models may lead to inaccuracies that can only be detected using experimentation.

Generally, there are multiple issues that arise, rather independent of the specific problem or algorithm under investigation. In this section we try to group these into general categories and topics (cf. Fig. 2), which have by now defined worthwhile research fields of their own. Clearly, this categorization is not meant to be exhaustive but only representative. In the field of algorithm engineering we are always looking out for further interesting issues being revealed that influence algorithmic behavior in practice and are worth systematic investigation.

In the following we will concentrate on the algorithmic aspects that are involved when developing theoretical algorithms for real-world problems. Due to the algorithmic focus of this chapter, we will not discuss other modeling issues based on the fact that our underlying algorithmic problem might be too crude a simplification of the actual real-world problem. We will also not discuss the issues arising from noisy or partially faulty input data with which our algorithms may have to deal. However, bear in mind that these are in fact also critical steps in the overall algorithmic development. They may even influence the most fundamental algorithmic decisions, e.g., it will usually not be reasonable to concentrate on finding provably optimal solutions if the given input is too noisy.

3.1 Properties of and Structures in the Input

Often certain considerations regarding the expected input data are not regarded highly enough in purely theoretic analysis and evaluation of algorithms. Most analyses focus on worst-case scenarios, considering all inputs possible within the model's bounds. Average-case analyses are relatively rare, not only because

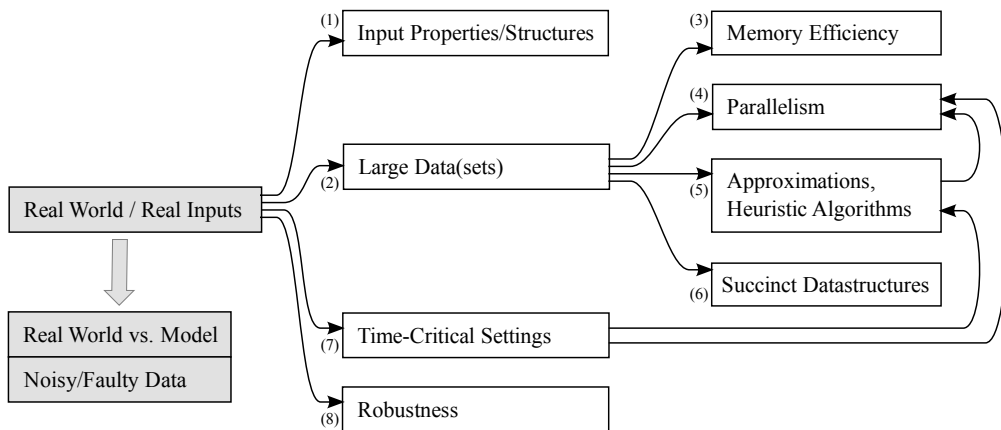


Figure 2: Major topics and issues in current algorithm engineering

they are mathematically much harder. It is hard to grasp what the “average case” should look like. Often—e.g., for the quicksort algorithm in the probably best known average-case analysis in algorithmics—it is assumed that all possible inputs are equally likely.

However, in most real-world scenarios the span of different inputs usually turns out to be only a fraction of all the possible inputs. In particular, the input data often has certain properties that are not problem but application specific. Failure to take these into account can lead to selecting inadequate algorithms for the problem at hand.

Consider problems on graphs. Many such problems become easier for graphs that are, e.g., planar or sparse, or have fixed tree width, fixed maximum degree, etc. Sometimes such properties even change the problem’s complexity from *NP*-completeness to polynomially solvable. Alternatively, an algorithm may, e.g., find an optimal solution in such a restricted case, but only have a weak approximation ratio (if any) for the general case.

Clearly, when we know that our real-world data consists only of graphs of such a simplifying type, we ought to use algorithms specifically tuned for these kind of instances. However, algorithm engineering goes one step further: we ask the question of which algorithm will perform best, if we consider only graphs that are “close to,” e.g., being planar. Due to the origin of the problem instances, such “close to” properties are often hard to grasp formally. Following the algorithm engineering cycle above we can either try to generalize special-purpose algorithms to be able to deal with such graphs, or to specialize general-purpose algorithms to deal with these graphs more efficiently. Often hybridization of these two approaches gives the best results in practice. After finding such ef-

efficient algorithms and demonstrating their performance experimentally, we can of course go back to theory and try to find some formal metric to describe this “close to” property and perhaps show some performance guarantees with respect to it.

In Sect. 4.1 we will showcase an example of such a development, regarding finding shortest paths in street maps. Even though the considered map graphs are not necessarily planar, we can still embed them in the plane with few crossings; although the edge weights do not satisfy the triangle inequality, the graphs are somehow close to being Euclidean graphs. This background knowledge allows for much more efficient algorithms in practice.

Interestingly, moving from the theoretical model to the practical one with respect to input instances does not always simplify or speed up the considered algorithms. In algorithmic geometry, there are many algorithms which assume that the given points are in *general position*, i.e., no three points may lie on a common line. It is often argued that, given there are some points which do not satisfy this property, one can slightly perturb their positions to achieve it. The other possibility is to consider such situations as special cases which can be taken care of by a careful implementation. The main algorithmic description will then ignore such situations and the implementer has to take care of them himself. Even though these additional cases may not influence the asymptotic runtime, they do affect the implementability and practical performance of the algorithm.

If we choose to perturb the points first—even disregarding any numerical problems that may arise from this step—we may lose certain important results which are in fact based on the collinearity of certain points. This is further amplified by the observation that real-world geometric data is often obtained by measuring points on some regular grid. Such input data can in fact exhibit the worst case for algorithms assuming general positions, but may on the other hand allow very simple alternative approaches. This shows that knowing the source and property of the input data can play a very important role in practice. However, no general rules can be applied or suggested, as the best approach is highly application specific.

3.2 Large Datasets

Huge data sets arise in many different research fields and are challenging not only because of the mere storage requirements, but more importantly because it is usually nontrivial to efficiently access, analyze, and process them.

“One of the few resources increasing faster than the speed of computer hardware is the amount of data to be processed.”

[IEEE InfoVis 2003 Call-For-Papers]

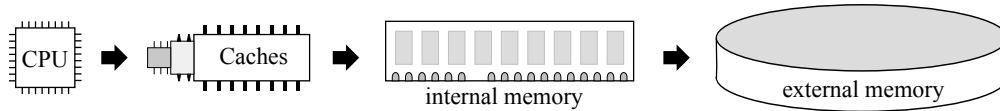


Figure 3: Typical memory hierarchy in current PCs. Each component is orders of magnitude larger but also slower than the one on the previous layer

In bioinformatics, a single high-through put experiment can result in thousands of datapoints; since thousands of these experiments can be conducted per day, data sets with millions of entries can be quickly accumulated for further analysis. For example, the Stanford Microarray Database contains around 2.5 billion spot data from about 75,000 experiments [Demeter et al., 2007]. Other fields of increasing importance are, among others, the analysis of the dynamics in telecommunication or computer networks, social statistics and criminalistics, and geographical information systems (GIS). The latter can, e.g., be used for computationally highly complex algorithms that analyze flooding or predict weather phenomena due to climate change.

The sheer size of such data sets can render traditional methods for data processing infeasible. New methodologies are therefore needed for developing faster algorithms when, e.g., even a simple look-up of the full data set is already inacceptably slow. This ranges from clever overall strategies to special-purpose heuristics, if the amount of data does not allow exact algorithms. When the data set is stable for a number of queries, such problems can often be tackled by preprocessing strategies whose computation costs are amortized over a number of optimization tasks. Also careless access to external memory storage such as hard disks may render the task at hand infeasible. For a realistic performance analysis in these cases, memory models that consider these costs have to be applied and algorithms have to be tuned to optimize their behavior respectively.

In the following, we will discuss some of these aspects in more detail.

3.3 Memory Efficiency

For a long time the von Neumann model was the dominant design model for computer architecture in the theoretical analysis of algorithms. An important aspect of this model—in contrast to current real-world computer hardware—is the assumption of a single structure storing both data and programs, allowing uniform constant memory access costs. In modern hardware architectures, the memory is organized in a hierarchy instead (cf. Fig. 3), where the access costs may differ between different levels by several orders of magnitude. Such memory hierarchies are mainly the result of a trade-off between speed and cost of available

hardware.

Processors usually have a small number of internal registers that allow fast and parallel access but are expensive in terms of chip area. The next hierarchy layer is then made up of several (typically 2–3) levels of cache memory, currently multiple kilobytes to some megabytes in size. The main idea behind the use of cache memory is based on the assumption that memory access shows some temporal or spatial locality, i.e., objects are fetched from the same memory location within a short period of time or from locations that are close together. This allows copies of the data to be stored in fast memory for subsequent access, resulting in a *cache hit* if the data requested is stored in cache memory, and in a *cache miss* otherwise. The cache management, including which memory locations to store, where to store them, and which data to remove again, is controlled by the cache hardware and is usually outside the programmer’s influence. When a cache miss occurs, the data has to be retrieved from the next memory level, the *internal (main) memory*, which is typically made up of relatively fast integrated circuits that are *volatile*, i.e., they lose the data information when powered off. The expensive access costs of lower memory levels are typically amortized by fetching a *block* of consecutive memory locations including the addressed one. The lowest level in the hierarchy, the *external memory*, provides large capacities, currently with gigabytes to terabytes of storage, and consists of nonvolatile memory that is much cheaper per byte than main memory. Current techniques include hard disks and more recently also so-called solid-state memory typically based on flash memory. The access to the external memory is called *input/output (I/O) operation* and may require blockwise or even sequential access. The type of access allowed and the access speed is often limited by the architecture of the memory hardware, e.g., hard disks require moving a mechanical read–write head, slowing down data retrieval significantly. In contrast the internal memory allows direct access to any memory locations in arbitrary order and is therefore also called *random access memory (RAM)*.

Performance guarantees built on the uniform access cost assumption therefore may not reflect the practical performance of the analyzed algorithms as the memory hierarchy may have a large impact. In order to approach a realistic performance rating, the analysis consequently needs to take the real memory organization into account. Clearly, it would not only be a very difficult task to analyze algorithms with respect to the exact architecture of existing systems, but also a futile attempt due to the constant change in this architecture. Hence, an abstraction of the existing memory hierarchies needs to be the basis of our memory model. This model should represent the main characteristics of real-world architectures and therefore allow the analysis to be close to the real behavior, but still be simple enough to be used with acceptable effort. The same argument can be used when designing algorithms: From an algorithm engineering perspective,

a memory hierarchy model should already be applied in the algorithm design phase so that the real memory access costs can be minimized.

Several models have been proposed to allow performance analysis without having to consider particular hardware parameters. The conceptually most simple one is the *external memory model* (EMM) [Aggarwal and Vitter, 1988], which extends the von Neumann model by adding a second, external, memory level. The M internal memory locations are used for the computation, whereas the external memory has to be accessed over expensive I/O operations that move a block of B contiguous words. Goals for the development of efficient external memory algorithms are to keep the number of internal memory operations comparable to the best internal memory algorithms while limiting the number of I/O operations. This can be done by implementing specific data replacement strategies to process as much data as possible in internal memory before writing them back and to maximize the amount of processable data in blocks that are read.

The field of external memory algorithms attracted growing attention in the late 1980s and the beginning of the 1990s, when increasingly large data sets had to be processed. A main contribution to the field was the work by Aggarwal and Vitter [1988] and Vitter and Shriver [1994a,b]; an overview on external memory algorithms can be found in [Vitter, 2001]. Even though the external memory model is a strong abstraction of the real situation, it is a very successful model both in the amount of scientific research concerned with it as well as in the successful application of external memory algorithms in practice due to satisfactory performance estimation.

One could argue that the best way to consider, e.g., a cache memory level, would be to know exactly the hardware characteristics such as memory level size, block transfer size, and access cost, and then to consider these values during the algorithm's execution. Algorithms and data structures that are tuned in this way are called *cache-aware*. This approach not only needs permanent adaption and hinders an implementation on different hardware, but also complicates the analysis. In the late 1990s the theoretical concept of *cache-oblivious* algorithms was introduced [Frigo et al., 1999]; it proposes better portable algorithms without having to tune them to specific memory parameters. Within this model, the parameters M and B are unknown to the algorithm, and in contrast to the EMM, block fetching is managed automatically by the hardware and the operating system.

It should be noted that, even though the term *cache* is used as a synonym for a specific level of the memory hierarchy, every level of the memory hierarchy can take the role of the cache for a slower level, and it is therefore sufficient to analyze cache-oblivious algorithms using a two-level hierarchy without compromising their performance on a deeper hierarchy. For multiple problems, including

sorting, fast Fourier transformation, and priority queues, asymptotically optimal cache-oblivious algorithms have been found [Frigo et al., 1999].

Regarding effects on the performance induced by real hardware memory architecture, the actual picture is even more complicated than the discussion in this section. Not only is cache memory organized in multiple levels, but in multicore architectures that are mainstream today, caches on some of the levels might be shared between multiple cores whereas there are dedicated caches for each core on other levels. Also the internal cache design, the way the cache is operated, and hard-disk internal caches can have an impact on the resulting performance. Other influencing factors that we do not discuss here are effects related to physical organization and properties of the hardware, and to software such as the operating system and the compiler used.

A thorough discussion of memory hierarchies and the corresponding topics with respect to algorithm development is given by Meyer et al. [2003]. The *Standard Template Library for Extra Large Data Sets* (STXXL) provides an implementation of the C++ standard template library for external memory computations [Dementiev et al., 2008b].

3.4 Distributed Systems and Parallelism

Over the past few years we have seen remarkable advances regarding the availability of computing power. The main reason was not new advanced processor technology or increased clock speeds, but rather the availability of relatively cheap computers and the introduction of multicore processors that combine multiple processor cores on a single chip. Large numbers of computers can be clustered in farms at relatively low financial expense to tackle computational problems too complex to be solved with a single high-performance computer. Typical examples are render farms for *computer generated-imagery* (CGI), server farms for web searches, or computer clusters used for weather prediction.

In order to take advantage of the processing power these hardware environments provide, algorithm and data structure implementations need to be adapted. Programs have to be distributed over multiple processing units such that tasks, threads or instructions can be executed simultaneously to speed up computation. These processing units may be virtual processors allowing multithreading on a single processor, multiple cores on a chip, or multiple computers connected over a network, also in heterogeneous environments.

Parallelism can be exploited by *data decomposition* (i.e., distributing the computation among multiple threads processing different parts of data), *task decomposition* (i.e., operating on a set of tasks that can run in parallel), or a combination of both.

Typical issues regarding the performance of parallel processing are load bal-

ancing, i.e., distributing the workload to keep all processing units busy, data decomposition, task synchronization, and shared resource access, e.g., how to effectively use the shared bus bandwidth in multicore systems. When modeling parallel algorithms, also the communication costs between processors for synchronization etc. have to be taken into account. Considering distributed computing, we also have to cope with reliability issues, both of the hardware and the connection. This includes possible outages of processing units and consequently the tasks assigned to them, and connection management regarding latency times, information loss, etc. Together, all these factors comprise the parallelization overhead that limits the speed-up achieved by adding processing units, compared with the speed-up obtained for example by an increase of the clock rate.

A number of libraries exist that allow to exploit parallelism with low computational and programming overhead. The Open Multi Processing interface¹ supports multiplatform shared-memory parallel programming in C, C++, and Fortran and implements multithreading, where multiple sequential parts of a process, the threads, may share resources and are distributed among the processing units. Recently, a parallel implementation of the standard C++ library, the Multi-Core Standard Template Library (MCSTL), has been integrated into the GNU Compiler Collection (GCC) as the `libstdc++` parallel mode. Internally, it uses OpenMP for multithreading and does not require specific source code changes to profit from parallelism. The open cross-platform parallel programming standard Open Computing Language² provides an API for the coordination of parallel computation and a C-based programming language with extensions to support parallelism to specify these computations.

Due to recent advances in the performance and programmability of modern graphics processing units (GPUs), the use of these for general-purpose computation is gaining increasing importance. Even when employed for graphics acceleration in standard PCs, their computational power is typically used only to a small extent most of the time. The availability of new interfaces, standardized instruction sets, and programming platforms such as OpenCL or the Compute Unified Device Architecture³ to address graphics hardware allows to exploit this computational power and has led to a new field focused on the adaptation of algorithm implementations in order to optimize their performance when processed on GPUs.

¹Open Multi Processing API, <http://openmp.org/>

²Open Computing Language standard, <http://www.khronos.org/opencl/>

³Compute Unified Device Architecture, <http://www.nvidia.com/CUDA>

3.5 Approximations and Heuristic Algorithms

In traditional algorithmics, we tend to use the complexity classes of P and NP to decide what kind of algorithms to develop: if a problem is polynomially solvable, we try to find the (asymptotically or practically) fastest algorithm to solve the problem. If the problem is NP -hard, there cannot exist such algorithms (unless $P = NP$), and hence our efforts are divided into exact and inexact approaches. For the former, we allow that our runtime may become exponential in certain cases, but try to find algorithms which are “usually” much faster. For inexact approaches, we require a polynomial running time (probably also depending on parameters such as number of iterations etc.) but allow the solutions to be sub-optimal. Approximation algorithms form a special subclass of such algorithms, as they guarantee that their solutions are mathematically close to the optimal (e.g., at most by a factor of 2 larger).

Recently, progress in computer hardware, modeling, and mathematical methods has allowed exact algorithms, based on integer linear programs, to run fast enough for practical applications [Polzin and Daneshmand, 2001]. In particular, there are even problem areas of combinatorial optimization where such approaches outperform state-of-the-art metaheuristics in terms of running time; see, e.g., [Chimani et al., 2009]. Generally, such approaches typically are strong when the problem allows to include sophisticated problem-specific knowledge in the program formulation.

Identifying the best approach that fits the specific task at hand is clearly a critical decision and highly application dependent. Recent research advances show that traditional rules of thumb—e.g., “The problem is NP -hard, hence we have to resort to heuristics”—for this decision are often outdated.

Concerning the problem field of large datasets, we can even see a reverse situation than the one described above. Even though certain problems are polynomially solvable, the required runtime may be too large, regarding the amount of data to consider. Hence, we may need to develop heuristics or approximation algorithms with smaller runtime requirements, simply to be able to cope with the data in reasonable time at all.

A well-known example of such an approach is a method often used for *sequence alignment* in bioinformatics. Generally, we can compute the similarity of two sequences—textstrings representing, e.g., parts of genomes—in $\mathcal{O}(n \cdot m)$ time, whereby n and m are the lengths of the sequences. When investigating some unknown (usually short) sequence S of length n , the researcher wants to check it against a database of k (usually long) sequences, say of length m . Although we could find sequences similar to S in $\mathcal{O}(n \cdot m \cdot k)$ time, this becomes impractical when considering multigigabyte databases. Hence, usually only heuristic methods such as the well-known FASTA (FASTA stands for “FAST-All”, because

it works with any alphabet) [Lipman and Pearson, 1985, Pearson and Lipman, 1988] or the *basic local alignment search tool* (BLAST) family [Korf et al., 2003] are used. Thereby the aim is to quickly discard sequences which have only a low chance of being similar to S (e.g., by requiring that at least a certain number of characters in the strings have to be perfect matches). The alignment itself can then also be approximated by only considering promising areas of the long sequences in the database.

3.6 Succinct Datastructures

In recent years, exact analysis of space requirements for data structures is gaining increasing attention due to the importance of processing large data sets. There is a strong need for space-efficient data structures in application areas that have to query huge data sets with very short response time, such as text indexing or storage of semistructured data.

This leads to the natural question how much space is needed not only to store the data but also to support the operations needed. In order to emphasize the focus on space efficiency, the term *succinct data structures* was coined for structures that represent the data with a space requirement close to the information-theoretic lower bound. Succinct data structures were introduced by Jacobson [1989] to encode bit vectors, trees, and planar graphs yet support queries efficiently. There has been a growing interest in this topic since then, leading to a wealth of publications that deal with both the practical and theoretical aspects of succinct data representations, see for example [Jansson et al., 2007, Geary et al., 2004, Golynski et al., 2007, Gupta, 2007].

3.7 Time-Critical Settings

Above, we used the grand topic of large data sets to motivate the research topics regarding parallelism and also inexact approaches with approximate or heuristic solutions. Similar to this topic there is the large area of *time-critical* applications.

There are often cases where, although the amount of considered data is not too large to cause side-effects w.r.t. the memory hierarchy, it is impracticable to run traditional, probably optimal, algorithms on the full data set. E.g., a quadratic asymptotic runtime, even though not distressing on first sight, might turn out to be too much to solve a problem within tight time bounds. Such cases often arise when algorithms are used as subroutines within grander computing schemes, such that they are called a large number of times and even small performance benefits can add up to large speed-ups.

In other situations—in particular in real-time environments such as car systems, power-plant control systems, etc.—the situation may be even more critical:

A real-time system sends a request to an algorithm and requires an answer within a certain timeframe. It can be a much larger problem if the algorithm misses this deadline, than if it would give a suboptimal—i.e., heuristic or approximative—answer.

We can view the research topics of parallelism (Sect. 3.4) and approximations (Sect. 3.5) also in the light of this problem field. While the overall concepts are similar, the developed algorithms have to cope with very different input data and external scenarios. In practice, this can lead to different algorithmic decisions, as certain aspects may become more relevant than others. In particular with parallelism, we have to keep a close look on the overhead introduced by many state-of-the-art techniques. E.g., on modern computer architectures, parallelization of sorting only becomes beneficial for a large number of elements (multiple millions); for fewer elements, traditional sorting is more efficient.

3.8 Robustness

As a final topic we want to touch on the area of algorithmic *robustness*, i.e., we aim for algorithms that behave stably in real-world scenarios. We differentiate between the two orthogonal problem fields regarding robustness: hardware failures and robustness with respect to input-data.

For the first category, we concentrate on the robust executability of algorithms. Even if our hardware exhibits failures—be it connection failures in distributed algorithms or random flips of bits in RAM modules—we would like to be able to detect and possibly recover from them. Especially the latter problem, though sounding fictitious at first, becomes a statistically important problem when considering large server farms, e.g., for web search [Henzinger, 2004]. It lead to the development of *resilient* algorithms where the aim is to find algorithms that are as efficient as the traditional ones, but can cope with up to k random bit flips (for some fixed k). Therefore we are not allowed to increase the memory requirement by the order of k , but only by at most $\omega(k)$; i.e., it is not an option to simply duplicate the memory sufficiently. See Finocchi and Italiano [2004] for a more detailed introduction to this topic, and first algorithms achieving this aim, e.g., for sorting data.

The second category of robustness deals with errors or slight modifications of the input data. Often, such input data will be a measured value with certain intrinsic inaccuracy. When we find a solution to our modeled problem, it is important to be able to apply it to the real-world scenario. Furthermore we would like that the found solution still resembles a good solution in this latter case, even if it turns out to be slightly different to the input used for the computation. Consider a shortest path problem where each edge length is only known within a certain margin of error. Assume there are two shortest paths with virtually

the same lengths, one of which uses an edge with a high probability of being underestimated. Clearly, we would like our algorithm to find the other path that promises a shorter path with higher probability. For such scenarios, there are only a few results regarding purely combinatorial algorithms. However the problem of finding robust solutions with respect to small modifications to the input has been considered under the term *sensitivity analysis* in mathematical programming. Within mathematical programming, there are even dedicated research fields of *stochastic programming* and *robust optimization*, the first of which optimizes the average over the error distributions (or, in most cases, multiple *scenarios*), whereas the second optimizes the worst case. See for example Birge and Louveaux [2000] for an introduction to this topic.

4 Success Stories

Algorithm engineering has led to an improvement of algorithm and data structure performance in many different fields of application. We give two exemplary descriptions of the development of advanced approaches using algorithm engineering.

4.1 Shortest Path Computation

The history and still ongoing research on shortest path algorithms is probably the most prominent showcase of the successes of algorithm engineering. Interest in it has also been fueled by the 9th DIMACS implementation challenge⁴ that asked for the currently best shortest path algorithms. We consider the traditional problem of finding a point-to-point (P2P) shortest path, i.e., given a graph with specified edge lengths, find the shortest edge sequence to connect two given nodes s and t of the graph. The problem is well known for decades, and often occurs as a subproblem within other complex problems. The presumably best-known application is in route-finding on street maps, railroad networks, etc.

In this realm, we can assume that all edge lengths are nonnegative, in which case we can use the famous algorithm Dijkstra presented already in [1959]. Given a start node s , the algorithm in fact computes the shortest paths to all other nodes, but can be terminated early when the given target node t is scanned for the first time. The conceptual idea of the algorithm is as follows: during the course of the algorithm, we will label the nodes with upper bounds on their distances to s . Initially, all nodes except for s , which has distance 0, are labeled with $+\infty$. We hold all the nodes in a priority queue, using these labels as keys.

⁴DIMACS: Center for Discrete Mathematics and Theoretical Computer Science, <http://dimacs.rutgers.edu/>, Shortest Path Challenge: <http://www.dis.uniroma1.it/~challenge9/>

We iteratively perform a *scanning* step, i.e., we extract the node v with smallest bound b . This bound then in fact gives the minimum distance from s to v . Now we update the bounds on the neighbors of v via *edge relaxation*: We decrease a label of a neighbor u to $b + d(v, u)$ —where the latter term gives the length of the edge between v and u —if this value is less than the current label on u . Thereby we move u up in the priority queue correspondingly.

Hence the choice of the data structure realizing the priority queue becomes a major issue when bounding the algorithm’s overall running time. Using a k -ary heap we can achieve $\mathcal{O}(m \log n)$, where n and m are the numbers of nodes and edges, respectively. By using a Fibonacci heap we can even guarantee $\mathcal{O}(m + n \log n)$ time.

Most interestingly, theoretic analysis suggests that the binary heap would constitute a bottleneck for sparse graphs. Since the considered graphs (street maps, railroad networks) often have this property, research has long concentrated on finding more efficient heap data structures. However, in practice, the really essential bottleneck is in fact not so much the complexity of the queue operations but the number of edge relaxations.

This insight and the investigation of the properties of the considered real-world networks led to algorithms far superior to Dijkstra’s algorithm, although their theoretic runtime bounds are at best identical if not inferior. State-of-the-art P2P algorithms are *tens of thousands* of times faster than the standard Dijkstra approach. They usually require some seemingly expensive preprocessing before the first query, but these costs can be amortized over multiple fast queries. We can differentiate between a number of different conceptual acceleration ideas. Mostly, these concepts can be seen as orthogonal, such that we can combine acceleration techniques of different categories to obtain an even faster overall speed-up.

4.1.1 Bi- and Goal-Directed Algorithms

Dijkstra’s algorithm scans all nodes that are closer to s than t . Although this property can come in handy in certain applications, it is the main reason why the algorithm is slow for P2P queries. The first improvements thereto were already proposed in the 1960s (bidirectional search) and the 1970s (goal-directed search). The former variant starts two Dijkstra algorithms from the start and the end node simultaneously. The search terminates when the two search frontiers touch, and we therefore have to scan fewer nodes, except for pathological cases.

The goal-directed (called A^*) search reduces the number of scanned nodes by modifying the keys in the priority queue: instead of only considering the distance between s and some node v for the key of v , it also takes estimates (in fact lower bounds) for the remaining distance between v and t into account. This allows

us to extract nodes that are closer to t earlier in the algorithm and overall scan fewer nodes.

For graphs embedded in the Euclidean plane, we can simply obtain such lower bounds on the remaining distance geometrically. For general graphs, the concept of *landmarks* turned out to be useful [Goldberg and Harrelson, 2005, Goldberg et al., 2005]: for each node ℓ of some fixed subset of nodes, we compute the shortest distances to all other nodes in the preprocessing. These distances can then be used to obtain bounds for any other node pair.

4.1.2 Pruning

This acceleration technique tries to identify nodes that will not lie on the shortest paths and prune them early during the algorithm. This pruning may be performed either when scanning or when labeling a node. Multiple different concepts were developed, e.g., geometric containers [Wagner and Willhalm, 2003], arc flags [Köhler et al., 2004], and reach-based pruning [Gutman, 2004, Goldberg et al., 2005]. Although they are all different, they use the common idea of preprocessing the graph such that, when considering a node or edge, we can—based on supplementing data structures—quickly decide whether to prune the (target) node and not consider it in the remaining algorithm.

4.1.3 Multilevel Approaches

The newest and in conjunction with the above techniques, most successful acceleration technique is based on extracting a series of graph layers from the input graph. The initial idea is to mimic how we manually would try to find an optimal route: when traveling between two cities, we will usually use city roads only in the beginning and at the end of our route, and use highways in between. The aim is to automatically extract such a hierarchy from the input graph, use it to accelerate the query times—by mainly using the smallest graph only considering the “fast” streets or subroutes—but still guarantee an overall optimal route. Multiple techniques arose to achieve this goal. While they are all beneficial to the more traditional approaches, the *highway hierarchies* as presented in Sanders and Schultes [2005] currently offer the best speed-ups.

4.1.4 State-of-the-Art and Outlook

Overall, for each such acceleration technique we can construct instances where they are not beneficial. However when applied to real-world instances such as the full road-networks of the USA or Europe, which contain multiple millions of nodes and edges, these techniques allow us to compute provable optimal shortest paths in a couple of microseconds (μs , a millionth part of a second). See, e.g.,

the repository of Schulz⁵ for a comprehensive compilation of current algorithms and results.

Based on the successes for static route finding, current research now also focuses on adapting these algorithms for dynamic settings, taking into account situations where travel times on certain streets are dependent on the current time (e.g., rush hour versus 3 am) or on certain traffic conditions due to road blocks, accidents, etc.

4.2 Full-Text Indexes

Full-text indexes are data structures that store a (potentially long) text T and allow to search for a short substring, called the *pattern* p , within it. Finding all occurrences of p in T is an important task in many applications, e.g., for text search in large document collections or the Internet, data compression, and sequence matching in biological sequence databases.

The naive way to perform this task is to scan through the text, trying to match the pattern at each starting position, leading to a worst-case running time of $\mathcal{O}(nk)$, where $n = |T|$ and $k = |p|$. There are a number of direct search algorithms, such as those by Knuth–Morris–Pratt and Boyer–Moore, that try to decrease the necessary effort by skipping parts of the text, typically by preprocessing the pattern. Using this preprocessed information, these algorithms manage to achieve a search time of $\mathcal{O}(n+k)$, i.e., linear in the size of the text and the pattern. However, with today’s huge data sets arising from high-throughput methods in biology or web indexing, even search time linear in the text size is inacceptably slow. Additionally, when the text is used for several searches of different patterns—e.g., when searching for peptide sequences in a protein database—it can pay off to build a data structure that uses information derived from the text instead, such that the construction time can be amortized over the number of searches. *Suffix trees* and *suffix arrays* are prominent examples of such data structures; after construction for a given input text they allow to answer queries in time linear in the length of the pattern. Their concept is based on the fact that each substring of a text is also the prefix of a suffix of the text.

Depending on the task at hand and the available hard- and software environment, the time and space requirements for the construction of such a data structure may be critical. For many decades now, there has been an ongoing research into time- and space-efficient construction algorithms and implementations for suffix trees and arrays, especially motivated by the increasing needs in application areas such as bioinformatics.

⁵Frank Schulz, website collecting shortest path research, <http://i11www.iti.uni-karlsruhe.de/~fschulz/shortest-paths/>

In the 1970s, Weiner [1973] proposed a data structure to allow fast string searching—the *suffix tree*—that can be constructed in linear time and space in the length of the text string. The data is organized as a tree where paths from the root to the leaves are in a one-to-one correspondence to the suffixes of T . The suffix tree became a widely used string-matching data structure for decades. McCreight later improved the space consumption of the tree construction [1976], using about 25 % less than Weiner’s original algorithm. Ukkonen [1995] improved on the existing algorithms by giving an online algorithm that allows the data structure to be easily updated.

With today’s huge data collections, the memory consumption of suffix trees became a major drawback. Even though there have been many advances in reducing the memory requirements by developing space-efficient representations, see, e.g., Kurtz [1999], the use of 10–15 bytes on average per input character is still too high for many practical applications. The human genome, for example, contains about 3×10^9 base pairs, leading to a string representation of about 750 megabytes, whereas a suffix tree implementation would need around 45 gigabytes of internal memory. The performance of suffix trees is also dependent on the size of the alphabet Σ . When searching for DNA sequences, where $|\Sigma| = 4$, this is not a problematic restriction, but already for protein sequence searches with $|\Sigma| = 20$ —and certainly when processing large alphabet texts such as, e.g., Chinese texts using an alphabet with several thousand characters—they are less efficient.

In the 1990s, Manber and Myers proposed a space-efficient data structure that is independent of the alphabet size—the *suffix array* [Manber and Myers, 1993]. It is constructed by sorting all suffixes of the text in lexicographic order and then storing the starting indices of the suffixes in this order in an array. This means that the look-up operation that returns a pointer to the i -th suffix in lexicographical order can be performed in constant time. The search for a pattern then can be done by using binary search in the array, leading to $\mathcal{O}(k \log n)$ runtime for a straight-forward implementation where a character-by-character comparison is done for each new search interval boundary. Using an additional array of longest common prefixes, the search can be sped up to $\mathcal{O}(k + \log n)$. Abouelhoda et al. showed how to use suffix arrays as a replacement for suffix trees in algorithms that rely on the tree structure and also discussed their application for genome analysis [Abouelhoda et al., 2002, 2004].

In recent years, there was a rally for the development of better, i.e., faster and more space-efficient, construction algorithms [Ko and Aluru, 2005, Kim et al., 2005, Manzini and Ferragina, 2004, Kärkkäinen et al., 2006]. Today’s best algorithms are fast in practice, some but not all of them have asymptotically optimal linear running time, they have low space consumption, and often can be easily implemented with only a small amount of code; see also Puglisi et al.

[2007] for a classification and comparison of numerous construction algorithms.

4.2.1 Compressed and External Memory Indexes

Looking at the simplicity of the data structure and the great improvement in space consumption, the suffix array may appear to be the endpoint of the development. However, even the small space requirements of an integer array may be too large for practical purposes, e.g., when indexing biological databases, the data structure may not fit into main memory and in particular the construction algorithm may need far more space than is accessible. This motivated further research in two different directions: a further decrease in memory consumption and the development of external memory index construction algorithms. The first path led to so-called *compressed* data structures, whereas the second one led to algorithms with decreased I/O complexity.

Compressed data structures typically exhibit a trade-off between decreased space consumption for the representation and increased running time for the operations. When the compression leads to a size that allows to keep the whole data structure in internal memory, the increase in running time does not necessarily need to lead to slower performance in practice. Compressed indexes are especially well suited to reduce space complexity when small alphabet sizes, such as in genomic information, allow good compression rates. Compressed indexes that also allow to reproduce any substring without separate storage of the text, so-called *self-indexes*, can lead to additional space savings and are gaining increasing attention in theory and practice.

The first compressed suffix array, proposed by Grossi and Vitter [2005], reduced the space requirement of suffix arrays from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n \log |\Sigma|)$. Sadakane extended its functionality to self-indexing [2003] and also introduced *compressed suffix trees* [2007], using only $\mathcal{O}(n \log |\Sigma|)$ bits but still providing the full functionality of suffix tree operations with a slowdown factor of $\text{polylog}(n)$. Independently of the development of compressed suffix arrays, Mäkinen proposed the so-called *compact suffix array* [2002], which uses a conceptually different approach and is significantly larger than compressed suffix arrays, but is much faster to report pattern occurrences and text contexts.

In recent years, a number of improvements have been proposed that either decrease memory consumption or try to find a good practical setting regarding the space and time trade-off, e.g., the FM index [Ferragina and Manzini, 2002, Ferragina et al., 2007], and compressed compact suffix arrays [Mäkinen and Navarro, 2004, Navarro and Mäkinen, 2007]. Compressed indexes for the human genome can be constructed with a space requirement of less than 2 gigabytes of memory.

Arge et al. [1997] were the first to address the I/O complexity of the string

sorting problem, showing that it is dependent on the relative lengths of the strings compared with the block size. Farach-Colton et al. [2000] presented the first I/O optimal construction algorithm, which is rather complex and hides large factors in the \mathcal{O} notation. Crauser and Ferragina [2002] gave an experimental study on external memory construction algorithms, where they also described a discrepancy between worst-case complexity and practical performance of one of the algorithms and used their findings to develop a new algorithm that combines good practical and efficient worst-case performance. Dementiev et al. [2008a] engineered an external version of the algorithm by Kärkkäinen et al. [2006] that is theoretically optimal and outperforms all other algorithms in practice.

4.2.2 State-of-the-Art and Outlook

The research on text indexes led to many publicly available implementations, and the Pizza&Chili Corpus ⁶ provides a multitude of implementations of different indexes together with a collection of texts that represent a variety of applications. The available implementations support a common API and additional information. Software is provided to support easy experimentation and comparison of different index types and implementations.

Further improvements on the practical time and space consumption of the construction as well as the implementations of index structures can be expected, eventually for specific alphabets. Regarding the size of the texts and the availability of cheap medium-performance hardware, also the aspects of efficient use of external memory and of parallelization have to be further considered in the future. In order to speed up construction of text indexes for huge inputs, a number of parallel algorithms have been given [Kulla and Sanders, 2007, Futamura et al., 2001]. There are also parallel algorithms that convert suffix arrays into suffix trees [Iliopoulos and Rytter, 2004].

5 Summary and Outlook

Algorithm engineering has come a long way from its first beginnings via simple experimental evaluations to become an independent research field. It is concerned with the process of designing, analyzing, implementing, and experimentally evaluating algorithms. In this chapter we have given an overview of some of the fundamental topics in algorithm engineering to present the main underlying principles. We did not touch too much on the topic of *designing* experiments, where the goal is to make them most expressive and significant. Though beyond the scope of this chapter, this is a critical part of the full algorithm engineering

⁶Pizza&Chili Corpus, <http://pizzachili.dcc.uchile.cl/>

cycle; see, e.g., Johnson [2002] for a comprehensive discussion of this topic. More in-depth treatment of various other issues can be found in works by Mehlhorn and Sanders [2008], Dementiev [2006], and Meyer et al. [2003].

An increasing number of conferences and workshops are concerned with the different aspects of algorithm engineering, e.g., the International Symposium on Experimental Algorithms (SEA), the Workshop on Algorithm Engineering and Experiments (ALENEX), and the European Symposium on Algorithms (ESA). There are many libraries publicly available that provide efficient implementations of state-of-the-art algorithms and data structures in a variety of fields, amongst others the Computational Geometry Algorithms Library,⁷ the Open Graph Drawing Framework,⁸ and the STXXL for external memory computations. The Stony Brook Algorithm Repository⁹ is a collection of algorithm implementations for over 70 of the most fundamental problems in combinatorial algorithmics.

Due to increasingly large and complex data sets and the ongoing trend towards sophisticated new hardware solutions, efficient algorithms and data structures will play an important role in the future. Experience in recent years, e.g., in the sequencing of the human genome, show that improvements along the lines of algorithm engineering—i.e., coupling theoretical and practical research—can lead to breakthroughs in application areas.

Furthermore, as demonstrated in the above sections, algorithm engineering is not a one-way street. By applying the according concepts, new and interesting questions and research directions for theoretical research arise.

References

- M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the Second International Workshop on Algorithms in Bioinformatics (WABI 02)*, pages 449–463. Springer, 2002.
- M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
- A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

⁷Computational Geometry Algorithms Library, <http://www.cgal.org/>

⁸Open Graph Drawing Framework, <http://www.ogdf.net/>

⁹Stony Brook Algorithm Repository, <http://www.cs.sunysb.edu/~algorithm/>

- A. Aho, D. Johnson, R. M. Karp, R. Kosaraju, C. McGeoch, and C. Papadimitriou. Emerging opportunities for theoretical computer science. *SIGACT News*, 28(3), 1997.
- L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC 97)*, pages 540–548. ACM, 1997.
- L. Auslander and S. V. Parter. On imbedding graphs in the plane. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.
- J. R. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer Series in Operations Research and Financial Engineering. 2000.
- J. M. Boyer and W. J. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.
- J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P’s and Q’s: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In *Proceedings of the 11th International Symposium on Graph Drawing (GD 03)*, volume 2912 of *LNCS*, pages 25–36. Springer, 2004.
- M. Chimani, M. Kandyba, I. Ljubić, and P. Mutzel. Obtaining optimal k -cardinality trees fast. *Journal of Experimental Algorithmics*, 2009. (Accepted). A preliminary version appeared in: Proceedings of the 9th Workshop on Algorithm Engineering and Algorithms (ALENEX 08), pages 27–36, SIAM, 2008.
- A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.
- H. de Fraysseix and P. Ossona de Mendez. On cotree-critical and DFS cotree-critical graphs. *Journal of Graph Algorithms and Applications*, 7(4):411–427, 2003.
- R. Dementiev. *Algorithm Engineering for Large Data Sets*. PhD thesis, Universität des Saarlandes, December 2006.
- R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12: 1–24, 2008a.
- R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice & Experience*, 38(6):589–637, 2008b.

- J. Demeter, C. Beauheim, J. Gollub, T. Hernandez-Boussard, H. Jin, D. Maier, J. C. Matese, M. Nitzberg, F. Wymore, Z. K. Zachariah, P. O. Brown, G. Sherlock, and C. A. Ball. The Stanford microarray database: implementation of new analysis tools and open source release of software. *Nucleic Acids Res*, 35 (Database issue), 2007.
- E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
- P. Ferragina and G. Manzini. On compressing and indexing data. Technical Report TR-02-01, 2002.
- P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans Algorithms*, 3(2):20, 2007.
- I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC 04)*, pages 101–110, 2004.
- M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS 99)*, page 285. IEEE Computer Society, 1999.
- N. Futamura, S. Aluru, and S. Kurtz. Parallel suffix sorting. In *Proceedings of the 9th International Conference on Advanced Computing and Communications*, pages 76–81. Tata McGraw-Hill, 2001.
- A. Gagarin, W. Myrvold, and J. Chambers. Forbidden minors and subdivisions for toroidal graphs with no $K_{3,3}$'s. *Electronic Notes in Discrete Mathematics*, 22:151–156, 2005.
- R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 04)*, pages 1–10. SIAM, 2004.
- A. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 05)*. SIAM, 2005.

- A. V. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. Technical Report MSR-TR-2005-132, Microsoft Research (MSR), October 2005.
- A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Proceedings of the Graph and Combinatorics Conference '63*, 1963.
- A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proceedings of the 15th Annual European Symposium on Algorithms (ESA 07)*, volume 4698 of *LNCS*, pages 371–382. Springer, 2007.
- R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- A. Gupta. *Succinct data structures*. PhD thesis, Durham, NC, USA, 2007.
- R. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX 04)*, pages 100–111. SIAM, 2004.
- C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Proceedings of the 8th International Symposium on Graph Drawing (GD 00)*, volume 1984 of *LNCS*, pages 77–90. Springer, 2001.
- M. Henzinger. The past, present and future of web search engines (invited talk). In *31st International Colloquium Automata, Languages and Programming, (ICALP 04)*, volume 3142 of *LNCS*, page 3, 2004.
- J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.
- J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.
- C. S. Iliopoulos and W. Rytter. On parallel transformations of suffix arrays into suffix trees. In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA'04)*, 2004.
- G. Jacobson. Space-efficient static trees and graphs. *Symposium on Foundations of Computer Science*, 0:549–554, 1989.
- J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 07)*, pages 575–584. SIAM, 2007.

- D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 215–250. AMS, 2002.
- J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.
- L. G. Khachiyan. A polynomial algorithm in linear programming. *Dokl. Akad. Nauk SSSR*, 244:1093–1096, 1979. English translation in *Soviet Math. Dokl.* 20, 191–194, 1979.
- D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.
- V. Klee and G. J. Minty. How good is the simplex algorithm? In O. Shisha, editor, *Inequalities III*, pages 159–175. Academic Press, 1972.
- D. E. Knuth. *Art of Computer Programming, Volumes 1–3*. Addison-Wesley Professional, July 1997.
- P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.
- E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of shortest path computation. Article 42, Technische Universität Berlin, Fakultät II Mathematik und Naturwissenschaften, 2004.
- I. Korf, M. Yandell, and J. Bedell. *BLAST*. O'Reilly, 2003.
- F. Kulla and P. Sanders. Scalable parallel suffix array construction. *Parallel Computing*, 33(9):605–612, 2007.
- S. Kurtz. Reducing the space requirement of suffix trees. *Software–Practice and Experience*, 29:1149–1171, 1999.
- D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.
- V. Mäkinen. Compact suffix array: a space-efficient full-text index. *Fundamenta Informaticae*, 56(1,2):191–210, 2002.
- V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 04)*, volume 3109 of *LNCS*, pages 420–433. Springer, 2004.

- U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.
- E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- K. Mehlhorn. *Graph algorithms and NP-completeness*. EATCS Monographs in Theoretical Computer Science, Vol. 2, Springer, 1984.
- K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.
- K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.
- U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.
- G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.
- W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences USA*, volume 85, pages 2444–2448, 1988.
- T. Polzin and S. V. Daneshmand. Improved algorithms for the Steiner problem in networks. *Discrete Applied Mathematics*, 112(1-3):263–300, 2001.
- S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.
- K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, 2007.
- P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings 17th European Symposium on Algorithms (ESA 05)*, volume 3669 of *LNCS*, pages 568–579. Springer, 2005.

- P. Sanders, K. Mehlhorn, R. Möhring, B. Monien, P. Mutzel, and D. Wagner. Description of the DFG algorithm engineering priority programme. Technical report, 2005. <http://www.algorithm-engineering.de/beschreibung.pdf> (in German).
- E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.
- J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994a.
- J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994b.
- J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.
- D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 03)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.
- P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 73)*, pages 1–11. IEEE Computer Society, 1973.