

Algorithm Engineering – An Attempt at a Definition

Peter Sanders*

Universität Karlsruhe, 76128 Karlsruhe, Germany
sanders@ira.uka.de

Abstract. This paper defines algorithm engineering as a general methodology for algorithmic research. The main process in this methodology is a cycle consisting of algorithm design, analysis, implementation and experimental evaluation that resembles Popper’s scientific method. Important additional issues are realistic models, algorithm libraries, benchmarks with real-world problem instances, and a strong coupling to applications. Algorithm theory with its process of subsequent modelling, design, and analysis is not a competing approach to algorithmics but an important ingredient of algorithm engineering.

1 Introduction

Algorithms and data structures are at the heart of every computer application and thus of decisive importance for permanently growing areas of engineering, economy, science, and daily life. The subject of *Algorithmics* is the systematic development of efficient algorithms and therefore has pivotal influence on the effective development of reliable and resource-conserving technology. We only mention a few spectacular examples.

Fast search in the huge data space of the internet (e.g. using Google) has changed the way we handle knowledge. This was made possible with full-text search algorithms that are harvesting matching information out of petabytes of data within fractions of a second and by ranking algorithms that process graphs with billions of nodes in order to filter *relevant information* out of heaps of result data. Less visible yet similarly important are algorithms for the efficient distribution and caching of frequently accessed data under massive load fluctuations or even distributed denial of service attacks.

One of the most far-reaching results of the last years was the ability to read the human genome. Algorithmics was decisive for the early success of this project [1]. Rather than just processing the data coming out of the lab, algorithmic considerations shaped the implementation of the applied shotgun sequencing process.

The list of areas where sophisticated algorithms play a key role could be arbitrarily continued: computer graphics, image processing, geographic information systems, cryptography, planning in production, logistics and transportation,...

* Partially supported by DFG grant SA 933/4-1.

How is algorithmic innovation transferred to applications? Traditionally, algorithmics used the methodology of *algorithm theory* which stems from mathematics: algorithms are designed using simple models of problem and machine. Main results are provable performance guarantees for all possible inputs. This approach often leads to elegant, timeless solutions that can be adapted to many applications. The hard performance guarantees lead to reliably high efficiency even for types of inputs that were unknown at implementation time. From the point of view of algorithm theory, taking up and implementing an algorithmic idea is part of application development. Unfortunately, it can be universally observed that this mode of transferring results is a slow process. With growing requirements for innovative algorithms, this causes growing gaps between theory and practice: Realistic hardware with its parallelism, memory hierarchies etc. is diverging from traditional machine models. Applications grow more and more complex. At the same time, algorithm theory develops more and more elaborate algorithms that may contain important ideas but are usually not directly implementable. Furthermore, real-world inputs are often far away from the worst case scenarios of the theoretical analysis. In extreme cases, promising algorithmic approaches are neglected because a mathematical analysis would be difficult.

Since the early 1990s it therefore became more and more apparent that algorithmics cannot restrict itself to theory. So, what else should algorithmicists do? *Experiments* play a pivotal role here. Algorithm engineering (AE) is therefore sometimes equated with *experimental algorithmics*. However, in this paper we argue that this view is too limited. First of all, to do experiments, you also have to *implement* algorithms. This is often equally interesting and revealing as the experiments themselves, needs its own set of techniques, and is an important interface to software engineering. Furthermore, it makes little sense to view design and analysis on the one hand and implementation and experimentation on the other hand as separate activities. Rather, a feedback loop of design, analysis, implementation, and experimentation that leads to new design ideas materializes as the central process of algorithmics.

This cycle is quite similar to the cycle of theory building and experimental validation in Popper's scientific method [2]. We can learn several things from this comparison. First, this cycle is driven by *falsifiable hypotheses* validated by experiments – an experiment cannot prove a hypothesis but it can support it. However, such support is only meaningful if there are conceivable outcomes of experiments that prove the hypothesis wrong. Hypotheses can come from creative ideas or result from *inductive reasoning* stemming from previous experiments. Thus we see a fundamental difference to the *deductive reasoning* predominant in algorithm theory. Experiments have to be *reproducible*, i.e., other researchers have to be able to repeat an experiment to the extent that they draw the same conclusions or uncover mistakes in the previous experimental setup.

There are further aspects of AE as a methodology for algorithmics, outside the main cycle. Design, analysis and evaluation of algorithms are based on some *model* of the problem and the underlying machine. Since gaps between theory and practice often relate to these models, they are an important aspect of AE.

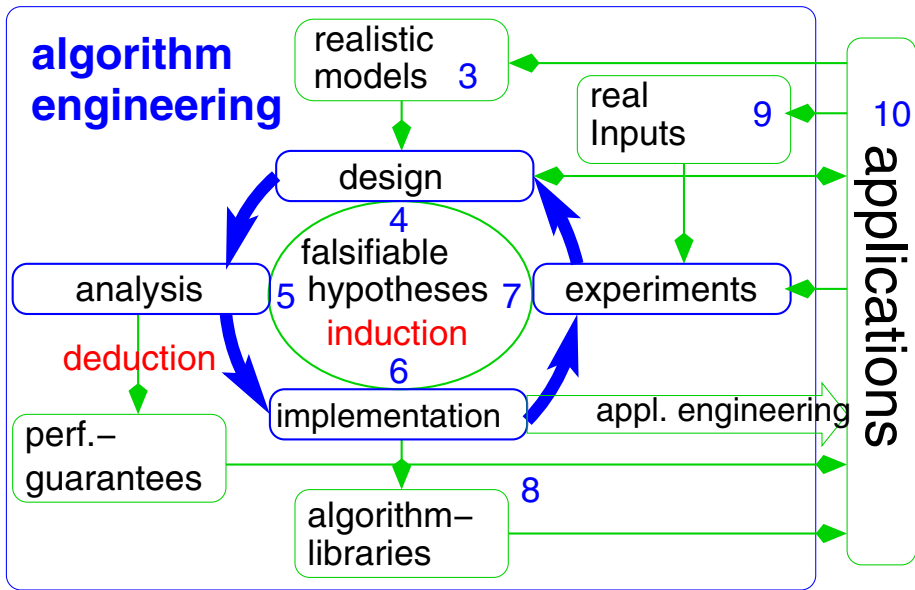


Fig. 1. Algorithm engineering as a cycle of design, analysis, implementation, and experimental evaluation driven by falsifiable hypotheses. The numbers refer to sections.

Since we aim at practicality, *applications* are an important aspect. However we choose to view applications as being outside the methodology of AE since it would otherwise become too open ended and because often one algorithm can be used for quite diverse applications. Also, every new application will have its own requirements and techniques some of which may be abstracted away for algorithmic treatment. Still, in order to reduce gaps between theory and practice, as many interactions as possible between the application and the activities of AE should be taken into account: Applications are the basis for *realistic models*, they influence the kind of analysis we do, they put constraints on useful implementations, and they supply *realistic inputs* and other design parameters for experiments. On the other hand, the results of analysis and experiments influence the way an algorithm is used (fast enough for real time or interactive use?, ...) and implementations may be the basis for software used in applications. Indeed, we may view *application engineering* as a separate process living in both AE and a concrete application domain where methods from both areas are used to adapt an algorithm to a particular application. Applications engineering bridges remaining unavoidable gaps between experimental implementations and production quality code. Note that there are important differences between these two kinds of code: fast development, efficiency, and instrumentation for experiments are very important for AE, while thorough testing, maintainability, simplicity, and tuning for particular classes of inputs are more important for the applications. Furthermore, the algorithm engineers may not even know all the applications for which their algorithms will be used. Hence, *algorithm libraries*

of highly tested codes with clear simple user interfaces are an important link between AE and applications.

Figure 1 summarizes the resulting schema for AE as a methodology for algorithmics. The following sections will describe the activities in more detail. We give examples of challenges and results that are a more or less random sample biased to results we know well. Throughout this paper, we will demonstrate the methodology using the external minimum spanning tree (MST) algorithm from [3] as an example. This example was chosen because it is at the same time simple and illustrates the methodology in most of its aspects.

2 A Brief “History” of Algorithm Engineering

The methodology described here is not intended as a revolution but as a description of observed practices in algorithmic research being compiled into a consistent methodology. Basically, all the activities in algorithm development described here have probably been used as long as there are computers. However, in the 1970s and 1980s algorithm theory had become a subdiscipline of computer science that was almost exclusively devoted to “paper and pencil” work. Except for a few papers around D. Johnson, the other activities were mostly visible in application papers, in operations research, or J. Bentley’s programming pearls column in *Communications of the ACM*. In the late 1980s, people within algorithm theory began to notice increasing gaps between theory and practice leading to important activities such as the Library of Efficient Data Types and Algorithms (LEDA, since 1988) by K. Mehlhorn and S. Näher and the DIMACS implementation challenges (<http://dimacs.rutgers.edu/Challenges/>). It was not before the end of the 1990s that several workshops series on experimental algorithmics and algorithm engineering were started.¹ There was a Dagstuhl workshop in 2000 [4], and several overview papers on the subject were published [5, 6, 7, 8, 9].

The term “algorithm engineering” already appears 1986 in the Foreword of [10] and 1989 in the title of [11]. No discussion of the term is given. At the same time T. Beth started an initiative to move the CS department of the University of Karlsruhe more into the direction of an engineering discipline. For example, a new compulsory graduate-level course on algorithms was called “Algorithmentechnik” which can be translated as “algorithm engineering”. Note that the term “engineering” like in “mechanical engineering” means the *application* oriented use of science whereas our current interpretation of algorithm engineering has applications not as its sole objective but equally strives for general scientific insight as in the natural sciences. However, in daily work the difference will not matter much.

P. Italiano organized the “Workshop on Algorithm Engineering” in 1997 and also uses “algorithm engineering” as the title for the algorithms column of EATCS

¹ The Workshop on Algorithm Engineering (WAE) is not the engineering track of ESA. The Alex workshop first held in Italy in 1998 is now the ALENEX workshop held in conjunction with SODA. WEA, now SEA was first organized in 2002.

in 2003 [12] with the following short abstract: “Algorithm Engineering is concerned with the design, analysis, implementation, tuning, debugging and experimental evaluation of computer programs for solving algorithmic problems. It provides methodologies and tools for developing and engineering efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures.” Independently but with the same basic meaning, the term was used in the influential policy paper [5]. The present paper basically follows the same line of argumentation attempting to work out the methodology in more detail and providing a number of hopefully interesting examples.

3 Models

A big difficulty for defining models for problems and machines is that (apparently) only complex models are adequate images of reality whereas only simple models lead to simple, widely usable, portable, and analyzable algorithms. Therefore, AE must simultaneously and carefully abstract from application problems and refine theoretical models.

A successful example for a machine model is the external memory model (or I/O model) [13, 14, 15] which is a careful refinement of the von Neumann model [16]. Instead of a uniform memory, there are two levels of memory. A fast memory of limited size M and a slow memory that is accessed in blocks of size B . While only counting I/O steps in this model can become a highly theoretical game, we get an abstraction useful for AE if we additionally take internal work into account and if we are careful to use the right values for the parameters M and B^2 . Algorithms good in the I/O model are often good in practice although the model oversimplifies aspects like rotational delays, seek time, disk data density depending on the track use, cache replacement strategies [17], flexible block sizes, etc. Sometimes it would even be counterproductive to be too clever. For example, a program carefully tuned to minimize rotational delays and seek time might experience severe performance degradation as soon as another application accesses the disk.

An practical modelling issue that will be important for our MST example is the maximal reasonable size for the external memory. In the last decades, the cost ratio between disk memory and RAM has remained at around 200. This ratio is not likely to increase dramatically as long as RAM and hard disk capacities improve at a similar pace. Hence, in a *balanced* system with similar investments for both levels of memory, the ratio between input size and internal memory size is not huge. In particular, the *logarithm* of this ratio is bounded by a fairly small constant.

² A common pitfall when applying the I/O model to disks is to look for a natural *physical* block size. This can lead to values (e.g. the size of 512 byte for a decoding unit) that are four orders of magnitude from the value that should be chosen – a value where data transfer takes about as long as the average latency for a small block.

The I/O model has been successfully generalized by adding parameters for the number of disks D , number of processors P , or by looking at the cache-oblivious case [18] where the parameters M and B are not known to the program.

An example for application modelling is the simulation of traffic flows. While microscopic simulations that take the actual behavior of every car into account are currently limited to fairly small subnetworks, it may soon become possible to simulate an entire country by only looking at the paths taken by each car.

4 Design

As in algorithm theory, we are interested in efficient algorithms. However, in AE, it is equally important to look for simplicity, implementability, and possibilities for code reuse. Furthermore, efficiency means not just asymptotic worst case efficiency, but we also have to look at the constant factors involved and at the performance for real-world inputs. In particular, some theoretically efficient algorithms have similar best case and worse case behavior whereas the algorithms used in practice perform much better on all but contrived examples. An interesting example are maximum flow algorithms where the asymptotically best algorithm [19] is much worse than theoretically inferior algorithms [20, 21].

We now present a similar, yet simpler example. Consider an undirected connected graph G with n nodes and m edges. Edges have nonnegative weights. An MST of G is a subset of edges with minimum total weight that forms a spanning tree of G . The MST problem can be solved in $\mathcal{O}(\text{sort}(m))$ expected I/O steps [22] where $\text{sort}(N) = \mathcal{O}(N/B \log_{M/B} N/B)$ denotes the number of I/O steps required for external sorting [13]. There is also a deterministic algorithm that requires $\mathcal{O}(\text{sort}(m) \lceil \log \log(nB/m) \rceil)$ I/Os [23].

However, before [3] there was no actual implementation of an external MST algorithm (or for any other nontrivial external graph problem). The reason was that previous algorithms were complicated to implement and have large constant factors that have never been exposed in the analysis. We therefore designed a new algorithm.

The base case of our algorithm is a simple *semiexternal* variant of Kruskal's algorithm [22] (A *semiexternal* graph algorithm is allowed $\mathcal{O}(n)$ words of fast memory): The edges are sorted (externally) by weight and scanned in sorted order. An edge is accepted into the MST if it connects two components of the forest defined by the previously found MST edges. This decision is supported by an internal memory union-find data structure. Even this simple algorithm is a good example for algorithm reuse since it can call highly tuned external sorting codes such as the routine in the external implementation of the STL, STXXL [24]. The *pipelining* facility of the STXXL saves up to 2/5 of the I/Os by directly feeding the sorted output into the final scan. For semiexternal algorithms, constant factors are particularly important for the space consumption in fast memory. Therefore, we developed a variant of the union-find data structure with path compression and union-by-rank [25] that needs only $\lceil \log(n + 1 + \log n) \rceil \approx \log n$

bits for each node. The trick is that root nodes of the data structure need rank information while only non-root nodes need parent information. Since ranks are at most $\log n$, the values $n..n + \log n$ can be reserved for rank information.

If $n > M$, all known external MST algorithms rely on a method for reducing the number of nodes. Our algorithmically most interesting contribution is *Sibeyn's* algorithm for node reduction based on the technique of *time forward processing*. The most abstract form of Sibeyn's algorithm is very simple. In each iteration, we remove a random node u from the graph. We find the lightest edge $\{u, v\}$ incident to u . By the well known cut-property that underlies most MST algorithms, $\{u, v\}$ must be an MST edge. So, we output $\{u, v\}$, remove it from E , and *contract* it, i.e., all other edges $\{u, w\}$ incident to u are replaced by edges $\{v, w\}$. If we store the original identity of each edge, we can reconstruct the MST from the edges that are output.

We transform the algorithm into a *sweeping algorithm* by renumbering the nodes using a random permutation π and then removing the nodes in the order $n..M$. When this is finished, the remaining problem can be solved using the semiexternal algorithm.

There is a very simple external realization of Sibeyn's algorithm based on priority queues of edges. Edges are stored in the form $((u, v), c, e_{\text{old}})$ where (u, v) is the edge in the current graph, c is the edge weight, and e_{old} identifies the edge in the original graph. The queue normalizes edges (u, v) in such a way that $u \geq v$. We define a priority order $((u, v), c, e_{\text{old}}) < ((u', v'), c', e'_{\text{old}})$ iff $u > u'$ or $u = u'$ and $c < c'$. With these conventions in place, the algorithm can be described using the simple pseudocode in Figure 2. This algorithm is not only conceptually simple but also easy to implement because it can again reuse software by relying on the external priority queue in STXXL [24]. Note that while a sophisticated external priority queue needs thousands of lines of code, the actual implementation of Figure 2 is not much longer than the pseudo code.

ExternalPriorityQueue: Q

```

foreach  $(e = (u, v), c) \in E$  do  $Q.insert(((\pi(u), \pi(v)), c, e))$            -- rename
currentNode := -1                                           -- node currently being removed
 $i := n$                                                      -- number of remaining nodes
while  $i > n'$  do
   $((u, v), c, e_{\text{old}}) := Q.deleteMin()$ 
  if  $u \neq \text{currentNode}$  then                                -- lightest edge out of a new node
    currentNode :=  $u$                                          -- node  $u$  is removed
     $i--$ 
    relinkTo :=  $v$ 
    output  $e_{\text{old}}$                                              -- MST edge
  elseif  $v \neq \text{relinkTo}$  then  $Q.insert((v, \text{relinkTo}), c, e_{\text{old}})$  -- relink non-self-loops

```

Fig. 2. An external implementation of Sibeyn's algorithm using a priority queue

5 Analysis

Even simple and proven practical algorithms are often difficult to analyze and this is one of the main reasons for gaps between theory and practice. Thus, the analysis of such algorithms is an important aspect of AE. For example, randomized algorithms are often simpler and faster than their best deterministic competitors but even simple randomized algorithms are often difficult to analyze.

Many complex optimization problems are attacked using *meta heuristics* like (randomized) local search or evolutionary algorithms. Algorithms of this type are simple and easily adaptable to the problem at hand. However, only very few such algorithms have been successfully analyzed (e.g. [26]) although performance guarantees would be of great theoretical and practical value.

An important open problem is partitioning of graphs into approximately equal sized blocks such that few edges are cut. This problem has many applications, e.g., in scientific computing. Currently available algorithms with performance guarantees are too slow for practical use. Practical methods first contract the graph while preserving its basic structure until only few nodes are left, compute an initial solution on this coarse representation, and then improve by local search. These algorithms, e.g., [27] are very fast and yield good solutions in many situations yet no performance guarantees are known.

An even more famous example for local search is the simplex algorithm for linear programming. Simple variants of the simplex algorithm need exponential time for specially constructed inputs. However, in practice, a linear number of iterations suffices. So far, only subexponential expected runtime bounds are known – for intractable variants. However, Spielmann and Teng were able to show that even small random perturbations of the coefficients of a linear program suffice to make the expected run time of the simplex algorithm polynomial [28]. This concept of *smoothed analysis* is a generalization of *average case analysis* and an interesting tool of AE also outside the simplex algorithm. Beier and Vöcking were able to show polynomial smoothed complexity for an important family of NP-hard problems [29]. For example, this result explains why the knapsack problem can be efficiently solved in practice and has also helped to improve the best knapsack solvers. There are interesting interrelations between smoothed complexity, approximation algorithms, and pseudopolynomial algorithms that is also an interesting approach to practical solutions of NP-hard problems.

Our randomized MST edge reduction algorithm is actually quite easy to analyze.

Theorem 1. *The expected number of edges inspected by the abstract algorithm until the number of nodes is reduced to n' is bounded by $2m \ln \frac{n}{n'}$.*

Proof. In the iteration when i nodes are left (note that $i = n$ in the first iteration), the expected degree of a random node is at most $2m/i$. Hence, the expected number of edges, X_i , inspected in iteration i is at most $2m/i$. By the linearity of expectation, the total expected number of edges processed is

$$\begin{aligned}
 \sum_{n' < i \leq n} \mathbb{E}[X_i] &\leq \sum_{n' < i \leq n} \frac{2m}{i} = 2m \sum_{n' < i \leq n} \frac{1}{i} = 2m \left(\sum_{1 \leq i \leq n} \frac{1}{i} - \sum_{1 \leq i \leq n'} \frac{1}{i} \right) \\
 &= 2m(H_n - H_{n'}) \leq 2m(\ln n - \ln n') = 2m \ln \frac{n}{n'}
 \end{aligned}$$

where $H_n = \ln n + 0.577 \dots + \mathcal{O}(1/n)$ is the n -th harmonic number. ■

Plugging in the complexity of the priority queue used [30] we obtain an I/O complexity of $\mathcal{O}(\text{sort}(m) \lceil \log(n/M) \rceil)$. This is actually asymptotically *worse* than the previous theoretical algorithms by a factor up to $\log(n/M)$. However, recall from Section 4 that $\log(n/M)$ is a *constant* in balanced machines. A close analysis of the constant factors involved [3] in the theoretical algorithms reveals that all things considered, Sibeyn’s algorithm needs a factor at least *four less* I/Os in realistic situations. Hence, our MST example exemplifies that closer looks at constant factors are an important aspect of algorithm analysis in AE and that constant factors can beat asymptotic behavior.

Sibeyn’s algorithm is also a good example for the importance of looking at non-worst case instances. It turns out that for planar graphs the factor $\log(n/M)$ is not needed since planar graphs remain planar under edge contraction and thus we always have constant average degree if we are careful enough to collapse parallel edges.

An MST algorithm can also be used to find connected components [31]. Since edges have no weights now, we are free to choose any edge. Choosing the edge leading to the node with smallest index actually looks like a good idea since this measure delays reconsidering the relinked edges. It looks like this should reduce the “suboptimality” of the algorithm to $\log \log(n/M)$. However, a full analysis remains an open problem³. This is an example for a simple randomized algorithm that is difficult to analyze because there are subtle dependencies to be taken into account.

6 Implementation

Implementation only appears to be the most clearly prescribed and boring activity in the cycle of AE. One reason is that there are huge semantic gaps between abstractly formulated algorithms, imperative programming languages, and real hardware. A typical example for this semantic gap is the implementation of an $\mathcal{O}(nm \log n)$ matching algorithm in [32]. Its abstract description requires a sophisticated data structure whose efficient implementation only succeeded in [32].

An extreme example for the semantic gap are geometric algorithms which are often designed assuming exact arithmetics with real numbers and without considering degenerate cases. The robustness of geometric algorithms has therefore become an important branch of AE [33, 34, 35].

Even the implementation of relatively simple basic algorithms can be challenging. You often have to compare several candidates based on small constant

³ In [31] there is no proof of the stated bounds.

factors in their execution time. Since even small implementation details can make a big difference, the only reliable way is to highly tune all competitors and run them on several architectures. It can even be advisable to compare the generated machine code (e.g. [30, 36], [37]).

Often only implementations give convincing evidence of the correctness and result quality of an algorithm. For example, an algorithm for planar embedding [38] was the standard reference for 20 years although this paper only contains a vague description how an algorithm for planarity testing can be generalized. Several attempts at a more detailed description contained errors (e.g. [39]). This was only noticed during the first correct implementation [40]. Similarly, for a long time nobody succeeded in implementing the famous algorithm for computing three-connected components from [41]. Only an implementation in 2000 [42] uncovered and corrected an error. For the related problem of computing a maximal planar subgraph there was a series of publications in prominent conferences uncovering errors in the previous paper and introducing new ones – until it turned out that the proposed underlying data structure is inadequate for the problem [43].

An important consequence for planning AE projects is that important implementations cannot usually be done as bachelor or master theses but require the very best students or long term attendance by full time researchers or scientific programmers.

Our MST code was implemented as a Bachelor thesis [44], however by one of the best programmers I have seen and reusing tens of thousands of lines of code from the STXXL that were the basis for a PhD thesis [45]. A particular challenge was the exploitation of parallel disks since it turned out that the code was compute bound. We obtained a considerable speedup by implementing a special purpose bucket priority queue that exploits the properties the problem: We only sort by the node ID of the larger endpoint. The minimum weight incident edge is found by extracting all incident edges. This is no actual overhead since those edges will later be relinked anyway.

7 Experiments

Meaningful experiments are the key to closing the cycle of the AE process. For example, experiments on crossing minimization in [46] showed that previous theoretical results were too optimistic so that new algorithms became interesting.

Experiments can also have a direct influence on the analysis. For example, reconstructing a curve from a set of measured points is a fundamental variant of an important family of image processing problems. In [47] an apparently expensive method based on the travelling salesman problem is investigated. Experiments indicated that “reasonable” inputs lead to easy instances of the travelling salesman problem. This observation was later formalized and proven. A quite different example of the same effect is the astonishing observation that arbitrary access patterns to data blocks on disk arrays can be almost perfectly balanced when two redundant copies of each block are placed on random disks [48].

Compared to the natural sciences, AE is in the privileged situation that it can perform many experiments with relatively little effort. However, the other

side of the coin is highly nontrivial planning, evaluation, archiving, postprocessing, and interpretation of results. The starting point should always be falsifiable hypotheses on the behavior of the investigated algorithms which stem from the design, analysis, implementation, or from previous experiments. The result is a confirmation, falsification, or refinement of the hypothesis. The results complement the analytic performance guarantees, lead to a better understanding of the algorithms, and provide ideas for improved algorithms, more accurate analysis, or more efficient implementation.

Successful experimentation involves a lot of software engineering. Modular implementations allow flexible experiments. Clever use of tools simplifies the evaluation. Careful documentation and version management help with reproducibility – a central requirement of scientific experiments, that is challenging due to the frequent new versions of software and hardware.

Experiments with external memory algorithms are challenging because they require huge inputs and execution times measuring in hours. In particular, when you compare against a *bad* algorithm, running times can easily reach months. Perhaps this is the reason why [3] was the first actual implementations of an external graph algorithm. Many previous implementations of external algorithms relied on artificially restricted main memory sizes to achieve small running times. We believed that this is unacceptable for results intended to convince practitioners to use external algorithms. Our solution was to use carefully configured yet relatively cheap machines that can be dedicated to the experiments for weeks, high performance implementations, and careful planning of experiments.

Our starting point for designing experiments was the study by Moret and Shapiro [49]. We have adopted the instance families for *random* graphs with random edge weights and random *geometric* graphs where random points in the unit square are connected to their d closest neighbors. In order to obtain a simple family of planar graphs, we have added *grid* graphs with random edge weights where the nodes are arranged in a grid and are connected to their (up to) four direct neighbors. We have not considered the remaining instance families in [49] because they define rather dense graphs that would be easy to handle semiexternally or they are specifically designed to fool particular algorithms or heuristics. We have chosen the parameters of the graphs so that m is between $2n$ and $8n$. Considerably denser graphs would be either solvable semiexternally or too big for our machine.

The experiments have been performed on a low cost PC-server (around 3 000 Euro in July 2002) with two 2 GHz Intel Xeon processors, 1 GByte RAM and 4×80 GByte disks (IBM 120GXP) that are connected to the machine in a bottleneck-free way (see [50] for more details on the hardware). This machine runs Linux 2.4.20 using the XFS file system. Swapping was disabled. All programs were compiled with g++ version 3.2 and optimization level -O6. The total computer time spent on the experiments was about 25 days producing a total I/O volume of several dozen Terabytes.

Figure 3 summarizes the results using bucket priority queues. The internal implementations were provided by Irit Katriel [51]. The curves only show the

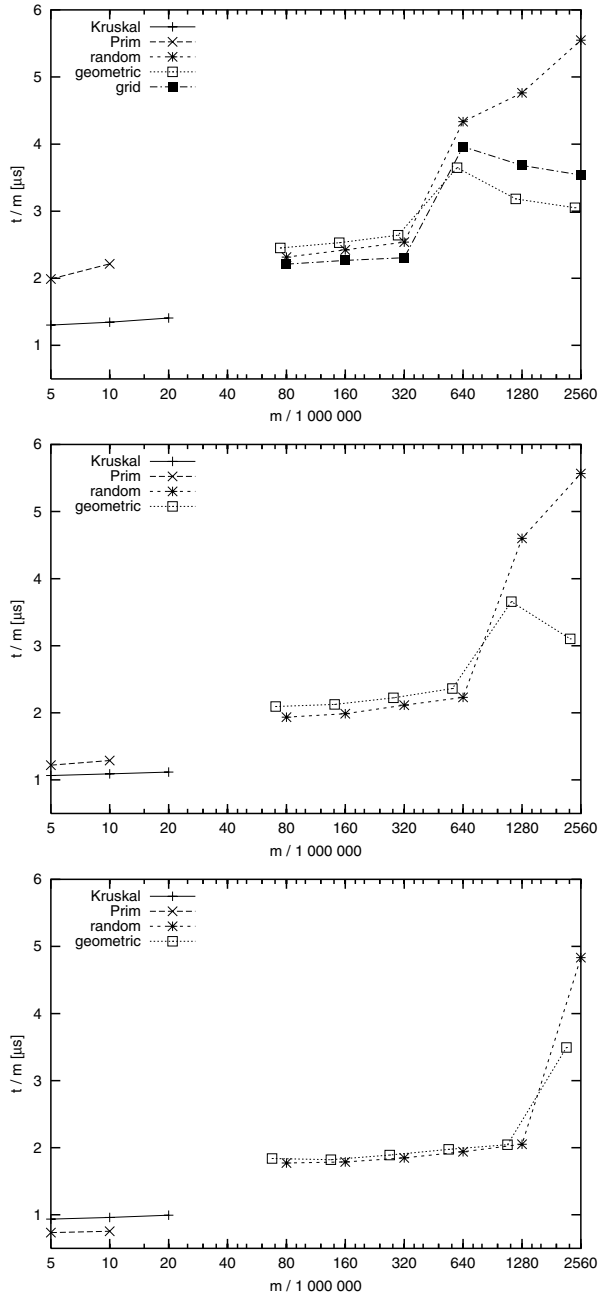


Fig. 3. Execution time per edge for $m \approx 2 \cdot n$ (top), $m \approx 4 \cdot n$ (center), $m \approx 8 \cdot n$ (bottom)

internal results for random graphs — at least Kruskal’s algorithm shows very similar behavior for the other graph classes. Our implementation can handle up to 20 million edges. Kruskal’s algorithm is best for very sparse graphs ($m \leq 4n$) whereas the Jarník-Prim algorithm (with a fast implementation of pairing heaps) is fastest for denser graphs but requires more memory. For $n \leq 160\,000\,000$, we can run the semiexternal algorithm and get execution times within a factor of two of the internal algorithm. The curves are almost flat and very similar for all three graph families. This is not astonishing since Kruskal’s algorithm is not very dependent on the structure of the graph. Beyond 160 000 000 nodes, the full external algorithm is needed. This immediately costs us another factor of two in execution time: We have additional costs for random renaming, node reduction, and increasing the size of an edge from 12 bytes to 20 bytes (for renamed nodes). For random graphs, the execution time keeps growing with n/M as predicted by the upper bound from Theorem 1.

The behavior for grid graphs is much better than predicted by Theorem 1 because planar graphs remain sparse under edge contraction. It is interesting that similar effects can be observed for geometric graphs. This is an indication that it is worth removing parallel edges for many nonplanar graphs. Interestingly, the time per edge *decreases* with m for grid graphs and geometric graphs. The reason is that the time for the semiexternal base case does not increase proportionally to the number of input edges. For example, $5.6 \cdot 10^8$ edges of a grid graph with $640 \cdot 10^6$ nodes survive the node reduction, vs. $6.3 \cdot 10^8$ edges of a grid graph with twice the number of edges.

Another observation is that for $m = 2560 \cdot 10^6$ and random or geometric graphs we get the worst time per edge for $m \approx 4n$. For $m \approx 8n$, we do not need to run the node reduction very long. For $m \approx 2n$ we process less edges than predicted by Theorem 1 even for random graphs simply because one MST edge is removed for each node.

8 Algorithm Libraries

Algorithm libraries are made by assembling implementations of a number of algorithms using the methods of software engineering. The result should be efficient, easy to use, well documented, and portable. Algorithm libraries accelerate the transfer of know-how into applications. Within algorithmics, libraries simplify comparisons of algorithms and the construction of software that builds on them. The software engineering involved is particularly challenging, since the applications to be supported are *unknown* at library implementation time and because the separation of interface and (often highly complicated) implementation is very important. Compared to applications-specific reimplementations, using a library should save development time without leading to inferior performance. Compared to simple, easy to implement algorithms, libraries should improve performance. In particular for basic data structures with their fine-grained coupling between applications and library this can be very difficult. To summarize, the triangle between generality, efficiency, and ease of use leads to challenging

tradeoffs because often optimizing one of these aspects will deteriorate the others. It is also worth mentioning that *correctness* of algorithm libraries is even more important than for other software because it is extremely difficult for a user to debug library code that has not been written by his team. Sometimes it is not even sufficient for a library to be correct as long as the user does not *trust* it sufficiently to first look for bugs outside the library. This is one reason why result checking, certifying algorithms, or even formal verification are an important aspect of algorithm libraries. All these difficulties imply that implementing algorithms for use in a library is several times more difficult / expensive / time consuming / frustrating / ... than implementations for experimental evaluation. On the other hand, a good library implementation might be *used* orders of magnitude more frequently. Thus, in AE there is a natural mechanism leading to many exploratory implementations and a few selected library codes that build on previous experimental experience.

Let us now look at a few successful examples of algorithm libraries. The Library of Efficient Data Types and Algorithms LEDA [21] has played an important part in the development of AE. LEDA has an easy to use object-oriented C++ interfaces. Besides basic algorithms and data structures, LEDA offers a variety of graph algorithms and geometric algorithms.

Programming languages come with a run-time library that usually offers a few algorithmic ingredients like sorting and various collection data structures (lists, queues, sets, ...). For example, the C++ standard template library (STL) has a very flexible interface based on templates. Since so many things are resolved at compile time, programs that use the STL are often equally efficient as hand-written C-style code even with the very fine-grained interfaces of collection classes. This is one of the reasons why our group is looking at implementations of the STL for advanced models of computation like external computing (STXXL [24]) or multicore parallelism (MCSTL, GNU C++ standard library [52]). We should also mention disadvantages of template based libraries: The more flexible their offered functionality, the more cumbersome it is to use (the upcoming new C++ standard might slightly improve the situation). Perhaps the worst aspect is coping with extra long error messages and debugging code with thousands of tiny inlined functions. Writing the library can be frustrating for an algorithmicist since the code tends to consist mostly of trivial but lengthy declarations while the algorithm itself is shredded into many isolated fragments.

The Boost C++ libraries (www.boost.org) are an interesting concept since they offer a forum for library designers that ensures certain quality standards and offers the possibility of a library to become part of the C++ standard.

The Computational Geometry Algorithms Library (CGAL) www.cgal.org that is a joined effort of several AE groups is perhaps one of the most sophisticated examples of C++ template programming. In particular, it offers many *robust* implementations of geometric algorithms that are also efficient. This is achieved for example by using floating point interval arithmetics most of the time and switching to exact arithmetics only when a (near)-degenerate situation is detected. The mechanisms of template programming make it possible to hide

much of these complicated mechanisms behind special number types that can be used in a similar way as floating point numbers.

As already mentioned, our external MST algorithm successfully uses the STXXL thus importing much of its code from a library. This is particularly true for the priority queue based implementation that has acceptable performance when using a single disk.

9 Instances

Collections of realistic problem instances for benchmarking have proven crucial for improving algorithms. There are interesting collections for a number of NP-hard problems like the travelling salesman problem ⁴, the Steiner tree problem, satisfiability, set covering, or graph partitioning. In particular for the first three problems the benchmarks have helped enable astonishing breakthroughs. Using deep mathematical insights into the structure of the problems one can now compute optimal solutions even for large, realistic instances of the travelling salesman problem [53] and of the Steiner tree problem [54]. It is a bit odd that similar benchmarks for problems that are polynomially solvable are sometimes more difficult to obtain. For route planning in road networks, realistic inputs have become available in 2005 [55] enabling a revolution with speedups of up to six orders of magnitude over Dijkstra’s algorithm and a perspective for many applications [56]. In string algorithms and data compression, real-world data is also no problem. But for many typical graph problems like flows, random inputs are still common practice. We suspect that this often leads to unrealistic results in experimental studies. Naively generated random instances are likely to be either much easier or more difficult than realistic inputs. With more care and competition, such as for the DIMACS implementation challenges, generators emerge that drive naive algorithms into bad performance. While this process can lead to robust solutions, it may overemphasize difficult inputs. Another area with lack of realistic input collections are data structures. Apart from some specialized scenarios like IP address lookup, few inputs are available for hash tables, search trees, or priority queues.

Our MST example lives on the dark side of the world of problem instance collections. This is a (moderate) risk for evaluating Sibeyn’s algorithm since it is not clear how the density of the graph behaves in practice and whether nodes with very high degree might emerge during the computation. But even simple internal algorithms have such input dependencies: Kruksal’s algorithm is much faster if we can use bucket sorting and the Jarník–Prim algorithm suffers when a lot of decrease-key operations are needed. Some literature/web search revealed that there is no lack of actual applications of the MST problem. Interestingly, clustering by removing MST edges seems to be more important than the classical network design motivation. However, it was difficult to find applications where a) huge inputs look important, b) the inputs are sparse (otherwise semiexternal algorithms are fine), and, c) *generating* the input is faster than finding the MST.

⁴ <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>

Later, our code turned out to be a useful tool for implementing external breadth-first-search and shortest path computations [57].

10 Applications

We could discuss many important applications where algorithms play a major role and a lot of interesting work remains to be done. Since this would go beyond the scope of this paper, we only want to mention a few: Bioinformatics (e.g. sequencing, folding, docking, phylogenetic trees, DNA chip evaluations, reaction networks); information retrieval (indexing, ranking); algorithmic game theory; traffic information, simulation and planning for cars, busses, trains, and air traffic; geographic information systems; communication networks; machine learning; real time scheduling.

An example for application engineering is recently started work on MSTs for image segmentation where satellite images define huge grid graphs. Here, aggressive exploitation of the special structure of the problems leads away from Sibeyn's algorithm. We exploit the simple 2D structure of the inputs and their small integer edge weights and also use parallelism. Furthermore, processing the edges in sorted order allows identifying the segments in a single pass. All this led to a highly specialized parallel variant of Kruskal's algorithm.

The effort for implementing algorithms for a particular application usually lies somewhere between the effort for experimental evaluation and for algorithm libraries depending on the context.

An important goal for AE should be to help shaping the applications (as in the example for genome sequencing mentioned in the introduction) rather than act as an ancillary science for other disciplines like physics, biology, mechanical engineering,...

11 Conclusions

We hope to have demonstrated that AE is a "round" methodology for the development of efficient algorithms which simplifies their practical use. We want to stress, however, that it is not our intention to abolish algorithm theory. The saying that there is nothing as practical as good theory remains true for algorithmics because an algorithm with proven performance guarantees has a degree of generality, reliability, and predictability that cannot be obtained with any number of experiments. However, this does not contradict the proposed methodology since it views algorithm theory as a subset of AE, making it even more rich by asking additional interesting kinds of questions (e.g. simplicity of algorithms, care for constant factors, smoothed analysis,...). We also have no intention of criticizing some highly interesting research in algorithm theory that is less motivated from applications than by fundamental questions of theoretical computer science such as computability or complexity theory. However, we do want to criticize those papers that begin with a vague claim of relevance to some fashionable application area before diving deep into theoretical constructions that look completely

irrelevant for the claimed application. Often this is not intentionally misleading but more like a game of “Chinese whispers” where a research area starts as a sensible abstraction of an application area but then develops a life of itself, mutating into a mathematical game with its own rules. Even this can be interesting but researchers should constantly ask themselves why they are working on an area, whether there are perhaps other areas where they can have larger impact on the world, and how false claims for practicality can damage the reputation of algorithmics in practical computer science.

Acknowledgements

I would like to thank the coinicators of the DFG SPP 1307, Kurt Mehlhorn, Rolf Möhring, Burkhard Monien, and Petra Mutzel for their advice and fruitful discussions that led to the definition presented here. I would like to thank Jop Sibeyn for his elegant MST algorithm and many other insights, Roman Dementiev for his excellent work on STXXL, and Dominik Schultes for the very good implementation of Sibeyn’s algorithm. Discussions with many other colleagues, in particular with Rudolf Fleischer have helped to shape my view of algorithm engineering.

References

1. Venter, J.C., Adams, M.D., Mayers, E.W., et al.: The sequence of the human genome. *Science* 291(5507), 1304–1351 (2001)
2. Popper, K.R.: *Logik der Forschung*. Springer (1934) English Translation: *The Logic of Scientific Discovery*, Hutchinson (1959)
3. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: *IFIP TCS*, Toulouse (2004)
4. Fleischer, R., Moret, B., Schmidt, E.M.: *Experimental Algorithmics*. LNCS, vol. 2547. Springer, Heidelberg (2002)
5. Aho, A.V., Johnson, D.S., Karp, R.M., Kosaraju, S.R., McGeoch, C.C., Papadimitriou, C.H., Pevzner, P.: Emerging opportunities for theoretical computer science. *SIGACT News* 28(3), 65–74 (1997)
6. Moret, B.M.E.: Towards a discipline of experimental algorithmics. In: *5th DIMACS Challenge*. DIMACS Monograph Series (2000) (to appear)
7. McGeoch, C.C., Precup, D., Cohen, P.R.: How to find big-oh in your data set (and how not to). In: Liu, X., Cohen, P.R., Berthold, M.R. (eds.) *IDA 1997*. LNCS, vol. 1280, pp. 41–52. Springer, Heidelberg (1997)
8. McGeoch, C., Moret, B.M.E.: How to present a paper on experimental work with algorithms. *SIGACT News* 30(4), 85–90 (1999)
9. Johnson, D.S.: A theoretician’s guide to the experimental analysis of algorithms. In: Goldwasser, M., Johnson, D.S., McGeoch, C.C. (eds.) *Proceedings of the 5th and 6th DIMACS Implementation Challenges*. American Mathematical Society (2002)
10. Beth, T., Clausen, M. (eds.): *AAECC 1986*. LNCS, vol. 307. Springer, Heidelberg (1988)
11. Beth, T., Gollman, D.: Algorithm engineering for public key algorithms. *IEEE Journal on Selected Areas in Communications* 7(4), 458–466 (1989)

12. Demetrescu, C., Finocchi, I., Italiano, G.F.: Algorithm engineering, algorithmics column. *Bulletin of the EATCS* 79, 48–63 (2003)
13. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. *Communications of the ACM* 31(9), 1116–1127 (1988)
14. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory, I: Two level memories. *Algorithmica* 12(2/3), 110–147 (1994)
15. Meyer, U., Sanders, P., Sibeyn, J. (eds.): *Algorithms for Memory Hierarchies*. LNCS, vol. 2625. Springer, Heidelberg (2003)
16. von Neumann, J.: First draft of a report on the EDVAC. Technical report, University of Pennsylvania (1945)
17. Mehlhorn, K., Sanders, P.: Scanning multiple sequences via cache memory. *Algorithmica* 35(1), 75–93 (2003)
18. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: 40th Symposium on Foundations of Computer Science, pp. 285–298 (1999)
19. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. *Journal of the ACM* 45(5), 1–15 (1998)
20. Cherkassky, B.V., Goldberg, A.V.: On implementing push-relabel method for the maximum flow problem. In: Balas, E., Clausen, J. (eds.) *IPCO 1995*. LNCS, vol. 920. Springer, Heidelberg (1995)
21. Mehlhorn, K., Näher, S.: *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge (1999)
22. Abello, J., Buchsbaum, A., Westbrook, J.: A functional approach to external graph algorithms. *Algorithmica* 32(3), 437–458 (2002)
23. Arge, L., Brodal, G., Toma, L.: On external memory MST, SSSP and multi-way planar graph separation. In: Halldórsson, M.M. (ed.) *SWAT 2000*. LNCS, vol. 1851, pp. 433–447. Springer, Heidelberg (2000)
24. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL data sets. *Software Practice & Experience* 38(6), 589–637 (2008)
25. Tarjan, R.E.: Efficiency of a good but not linear set merging algorithm. *Journal of the ACM* 22, 215–225 (1975)
26. Wegener, I.: Simulated annealing beats metropolis in combinatorial optimization. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) *ICALP 2005*. LNCS, vol. 3580, pp. 589–601. Springer, Heidelberg (2005)
27. Karypis, G., Kumar, V.: Multilevel k -way partitioning scheme for irregular graph. *J. Parallel Distrib. Comput.* 48(1) (1998)
28. Spielman, D., Teng, S.H.: Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In: 33rd ACM Symposium on Theory of Computing, pp. 296–305 (2001)
29. Beier, R., Vöcking, B.: Typical properties of winners and losers in discrete optimization. In: 36th ACM Symposium on the Theory of Computing, pp. 343–352 (2004)
30. Sanders, P.: Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics* 5 (2000)
31. Sibeyn, J.F.: External connected components. In: Hagerup, T., Katajainen, J. (eds.) *SWAT 2004*. LNCS, vol. 3111, pp. 468–479. Springer, Heidelberg (2004)
32. Mehlhorn, K., Schäfer, G.: Implementation of weighted matchings in general graphs: The power of data structure. *ACM Journal of Experimental Algorithmics* 7 (2002)

33. Burnikel, C., Könnemann, J., Mehlhorn, K., Näher, S., Schirra, S., Uhrig, C.: Exact geometric computation in leda. In: SCG 1995: 11th annual symposium on Computational geometry, pp. 418–419. ACM, New York (1995)
34. Berberich, E., Eigenwillig, A., Hemmer, M., Hert, S., Kettner, L., Mehlhorn, K., Reichel, J., Schmitt, S., Schömer, E., Wolpert, N.: EXACUS: Efficient and exact algorithms for curves and surfaces. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 155–166. Springer, Heidelberg (2005)
35. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. *Science* 316(5824), 566 (2007)
36. Sanders, P., Winkel, S.: Super scalar sample sort. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 784–796. Springer, Heidelberg (2004)
37. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. In: 6th Workshop on Algorithm Engineering and Experiments (2004)
38. Hopcroft, J., Tarjan, R.E.: Efficient planarity testing. *J. of the ACM* 21(4), 549–568 (1974)
39. Mehlhorn, K.: Data Structures and Algorithms. EATCS Monographs on Theoretical CS, vol. I — Sorting and Searching. Springer, Heidelberg (1984)
40. Mehlhorn, K., Mutzel, P.: On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica* 16(2), 233–242 (1996)
41. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. *SIAM J. Comput.* 2(3), 135–158 (1973)
42. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR-trees. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)
43. Jünger, M., Leipert, S., Mutzel, P.: A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design* 17(7), 609–612 (1998)
44. Schultes, D.: External memory minimum spanning trees. Bachelor thesis, Max-Planck-Institut f. Informatik and Saarland University (2003), <http://algo2.iti.uni-karlsruhe.de/schultes/emmst/>
45. Dementiev, R.: Algorithm Engineering for Large Data Sets. PhD thesis, Saarland University (2006)
46. Jünger, M., Mutzel, P.: 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications* (JGAA) 1(1), 1–25 (1997)
47. Althaus, E., Mehlhorn, K.: Traveling salesman-based curve reconstruction in polynomial time. *SIAM Journal on Computing* 31(1), 27–66 (2002)
48. Sanders, P., Egner, S., Korst, J.: Fast concurrent access to parallel disks. In: 11th ACM-SIAM Symposium on Discrete Algorithms, pp. 849–858 (2000)
49. Moret, B.M.E., Shapiro, H.D.: An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 15, 99–117 (1994)
50. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: 15th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, pp. 138–148 (2003)
51. Katriel, I., Sanders, P., Träff, J.L.: A practical minimum spanning tree algorithm using the cycle property. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 679–690. Springer, Heidelberg (2003)
52. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 682–694. Springer, Heidelberg (2007)

53. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Math. Programming* 97(1-2), 91–153 (2003)
54. Polzin, T., Daneshmand, S.V.: Extending reduction techniques for the Steiner tree problem. In: Möhring, R.H., Raman, R. (eds.) *ESA 2002*. LNCS, vol. 2461, pp. 795–807. Springer, Heidelberg (2002)
55. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Brodal, G.S., Leonardi, S. (eds.) *ESA 2005*. LNCS, vol. 3669, pp. 568–579. Springer, Heidelberg (2005)
56. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering route planning algorithms (2008) (submitted for publication),
<http://i11www.ira.uka.de/extra/publications/dssw-erpa-09.pdf>
57. Ajwani, D., Dementiev, R., Meyer, U.: A computational study of external-memory BFS algorithms. In: *ACM-SIAM Symposium on Discrete Algorithms*, pp. 601–610 (2007)